Team Grasp: Final Report

Deval Shah
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
devals@andrew.cmu.edu

Rohan Pandya
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
rpandya@andrew.cmu.edu

Alvin Shek
Electrical
Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
ashek@andrew.cmu.edu

Chandreyee Bhaumik
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
cbhaumik@andrew.cmu.edu

Abstract

Reinforcement learning (RL) is a powerful learning technique that enables agents to learn useful policies by interacting directly with the environment. While this training approach doesn't require labeled data, it is plagued with convergence issues and is highly sample inefficient. The learned policies are often very specific to the task at hand and are not generalizable to similar task spaces. Meta-Reinforcement Learning is one strategy that can mitigate these issues, enabling robots to acquire new skills much more quickly. Often described as "learning how to learn", it allows agents to leverage prior experience much more effectively. Recently published papers in Meta Learning show impressive speed and sample efficiency improvements over traditional methods of relearning the task for slight variations in the task objectives. A concern with these Meta Learning methods was that their success was only achieved on relatively small modifications to the initial task. Another concern in RL, as highlighted by [1], is reproducibility and lack of standardization of the metrics and approaches, which can lead to wide variations in reported vs observed performances. To alleviate that, benchmarking frameworks such as [2] and [3] have been proposed that establish a common ground for a fair comparison between approaches. In this work, we aim to utilize the task variety proposed by [2] and compare PPO, a vanilla policy gradient algorithm, with their Meta Learning counterparts (MAML and Reptile). The objective is to verify the magnitude of success of meta-learning algorithms over the vanilla variants, and test them on a variety of complicated tasks to test their limits in responding to the size of task variations. We also introduce a new technique — Multi-headed Reptile, which proposes a novel approach to address some of the shortcomings of both these Meta Learning techniques, with some computational implementation suggestions that prevent slowdown.

Github: https://github.com/Team-Grasp/idl-project

1 Introduction

Reinforcement learning is a subset of machine learning that involves learning a task by directly interacting with the environment and attempting to maximize its received cumulative reward. An

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

agent (learner) explores the environment and collects rewards for the actions taken in the environment. This exploration helps the agent learn a policy, which is a mapping from the agent's state to some action in the action space. Deep learning provides a high-dimensional, scalable framework for learning complex policies, which is especially useful when the dimensionality of action space is large. Deep Reinforcement Learning is the amalgamation of these techniques and has proved to be useful in solving a wide range of complex decision-making tasks.

In this project, we look at the various approaches of Deep Reinforcement Learning to solve the problem of robotic manipulation. We compare a traditional policy-gradient RL algorithm, Proximal Policy Optimization (PPO) with Meta Reinforcement Learning algorithms MAML and Reptile [4], a comparatively newer approach. To make a fair assessment of their relative merits, we implement and compare the algorithms on the same set of tasks.

Model-free, deep reinforcement learning methods typically need a notoriously large number of experiences to learn a good policy. Meta-learning is a method of "learning to learn" that can speed up the process of learning a an unseen task by using previously learnt tasks which have some structural similarity with the new task. While there are many approaches to the problem of meta-learning, such as Metric-Based, Model-Based and Optimization-Based, we look primarily at optimization based approaches. These proposes have gained popularity due to the use of neural networks as policy networks in deep reinforcement learning.

For our learned task using PPO, we use a subset of the Meta-World [2] tasks to evaluate the policies. These tasks are designed such that they offer a good diversity of tasks for the agent to learn from, but also maintain a structural similarity between tasks to ensure learnt policies can be transferred to the unseen tasks. And finally, we compare the performance of PPO, MAML, Reptile and Multi-headed Reptile (our proposed algorithm) on the selected task set and critically analyse benefits and pitfalls of each.

2 Literature Review

The approach of meta-learning [5] [6] [7] is an extension of learning generalized skills through reinforcements learning. Along with recent policy gradient methods for continuous control [8], they offer an exciting approach to learning policies that can be reused in similar tasks. The standard methods for policy optimization typically maintain two networks, one for active policy learning and the other for learning the target value function. Both networks are periodically updated and the former tries to learn the latter. This makes the learning process very unstable. To improve training stability, [8] introduced TRPO which keeps parameter updates within a range. This is done by enforcing a KL divergence constraint on the size of policy update at each iteration. While it induces stability, it increases the complexity of the problem. Proximal Policy Optimization (PPO) [9] was later introduced which provides a good trade-off between complexity and stability. PPO is also mathematically guaranteed to monotonically improve the expected total reward over iterations in the policy space, but is an approximation. Their work shows promising results in high-dimensional continuous control problems. [10] compares the performance of common policy gradient algorithms for robotic manipulation tasks. PPO performs consistently well across a range of tasks. Although TRPO performs stronger guarantees on policy improvement, it computes an expensive second-order Hessian, whereas PPO does simple clipping of gradients and thus runs much faster.

We reviewed the state of the art algorithms [11] [12] [13] in meta-reinforcement learning and identified several deficiencies pertaining to application in its field of manipulation. The first deficiency was the problems that the algorithms were validated on. RL^2 [11] was evaluated on Multi-arm bandits, tabular MDPs and visual navigation in a 2D maze. MAML [12] was evaluated on Regression task, Omniglot image recognition task and Mujuco 3D quadruped. Reptile [13] was also evaluated on Mini-Imagenet and Omniglot. As observed in [2], these tasks have either been evaluated on disjoint an overly diverse tasks such as the Atari suite, or on tasks with very narrow task distributions as identified in the evaluations in the papers above. [2] also presents a benchmark for better evaluating such algorithms, Meta-World, that presents a suite of 50 diverse simulated manipulation tasks. The second deficiency is the lack of empirical evaluations of the state of the art continuous action policy-gradient methods such as PPO vs its meta learning variants on a manipulation task. For example, [2] compares the multi-task variants of a few algorithms vs their meta-learning counterparts. From a practical standpoint, the choice lies between the vanilla algorithms vs their meta-learning counterparts, with metrics such

as training time, sample efficiency and sensitivity to hyper-parameters being more relevant to their application than merely success rate. To adopt a more scalable simulation framework, we propose using RLBench [3] which has more tasks than meta-world, and also allows the use of sensors and realistic physics interactions.

3 Contribution

The goal of meta learning is to learn a pre-trained model that can adapt and generalize to new and unseen tasks using fewer training samples. The train and test tasks, although different, belong to the same family of task distribution. Some experiments from the papers include multi-armed bandit with different reward structures, navigation of mazes with different layouts and so on. The goal of the pre-trained model to generalize well over the family of tasks, learning this shared, latent structure in the the tasks that allows it to fit to the specific task distribution as compared to a random model.

In this work, we first compare two popular, optimization-based, meta-learning algorithms that aim to learn better weight initialization for the policy model. Secondly, we introduce the meta-learning technique **Multi-headed Reptile**. It draws inspiration from the previous work on MAML[12] and Reptile[13] and attempts to tackle the issues with stability and convergence. Before delving into the details of the new algorithm, let us take a look at the approaches of MAML and Reptile.

MAML

In this method we learn a model parameterized by θ (say). During training, we train a copy of the model for each task in the training task set $\mathcal{T}$. We perform k optimization steps for each model on a subset of the data and at the end of k steps the model for any task $\mathcal{T}_i$ is parameterized by θ_i. It is important to note here that each of these models start off with θ as initial parameter. θ is updated at the end of the model rollout using a standard gradient update step : $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}i \sim p(\mathcal{T})} \mathcal{L}\mathcal{T}i\,(f\theta_i)$ where $p(\mathcal{T})$ is the probability distribution over tasks and $\mathcal{L}$ is the loss function, which is the negative reward function.

Reptile

Reptile's takes a similar approach to meta-learning as MAML, seeking to learn a better initialization for the parameters of a neural network, such that the network can be fine-tuned using a small amount of data from a new task. The major difference between the two is that Reptile performs sequential stochastic gradient descent on each task instead of model unrolling and cumulative gradient update. This makes it more memory efficient and less computationally extensive. However since each update of weights are based on a single task's policy unrolling, the weight updates are less stable than MAML's updates.

Multi-headed Reptile

Despite being fast and efficient, the fast gradient updates make Reptile inherently unstable. Multi-headed Reptile borrows from both of these concepts. Similar to Reptile, it performs a model weight update at every time step, instead of a k step rollout. To stabilize the neural network policy updates, we perform asynchronous rollouts of the policy to get updates updated weights parameters, and use the mean of the batch as the update to the original policy weights. This update technique was inspired by batch gradient update algorithms used in gradient descent, with the 'batch' referring to a subset of tasks in this context.

Mathematically, we can check if the batch updates reduce variance. Let σ_i be the variance in weights for task i and ρ be the correlation between weights of two tasks. We assume this correlation to be same across any two tasks from the task set. Let N be the number of tasks sampled in a batch.

$$
\begin{aligned}
Var\left(\frac{1}{N}\sum_{i=1}^{N}\theta_i\right) &= \frac{1}{N^2}Var\left(\sum_{i=1}^{N}\theta_i\right) = \frac{1}{N^2}\left(\sum_{i=1}^{N}\sigma_i^2 + \sum_{i\neq j}\rho_{ij}\sigma_i\sigma_j\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\sigma_i^2 + \frac{1}{N^2}\sum_{i\neq j}\rho_{ij}\sigma_i\sigma_j \\
&= \frac{1}{N}Var(\theta) + \frac{1}{N^2}2\binom{N}{2}\rho\sigma^2 \\
&= \frac{1}{N}Var(\theta) + \frac{N-1}{N}\rho Var(\theta)
\end{aligned}
$$

For Reptile, the variance in the weight update equation is $Var(\theta)$. Hence our algorithm reduces variation or uncertainty in weight update as $\rho < 1$. We set $\rho \neq 0$ as the individual tasks in our batch are drawn from a similar distribution of tasks. If we sample tasks from a more diverse distribution, ρ becomes smaller and the variability reduces further. Algorithm 1 contains the pseudo-code for Multi-headed Reptile.

Algorithm 1: Multi-headed Reptile Algorithm

Initialize θ, the initial parameter vector;
for *iteration 1,2,3..* **do**
  Sample batch of N tasks $\mathcal{T}_i \sim p(\mathcal{T})$;
  for *all* $\mathcal{T}_i$ **do**
    Perform $k > 1$ steps of SGD on task $\mathcal{T}i$, starting with parameters θ, resulting in parameters θ_i;
  end
  Update: $\theta \leftarrow \theta + \beta\left(\frac{1}{N}\sum_{\mathcal{T}i\sim p(\mathcal{T})}\theta_i - \theta\right)$;
end
Result: θ

Training Asynchronously

As shown above, in theory, multi-headed Reptile holds the potential to improve the stability of the updated weights for each iteration of the learning process. However, stepping through multiple tasks before a gradient update can make the process considerably slower, especially since most simulators use cpu. Also unlike supervised learning, the RL agent, for each iteration has to sample new experiences to evaluate and improve the policy. This makes gathering experience in RL very slow, and a proposed technique using batching would be infeasible if the runtimes also scale linearly with the proposed batch-size (contrary to other deep learning applications where batching makes updates faster due to extreme parallel processing by GPUs).

To combat this, we introduce asynchronous model execution using Ray framework [14] in multi-headed reptile. We launch multiple instances of the simulator, and gather environmental interactions and perform loss computation in parallel. This speeds up the training process thus allowing for parallel exploration within the environment and more stable gradient updates. This allows our multi-headed reptile to be computationally feasible.

4 Experimental Evaluation

4.1 Dataset

As opposed to standard supervised learning with a dataset, RL does not have ground-truth, labeled data. Rather, training in RL involves running a model or policy through randomly-initialized simulation episodes for a given environment and task. The policy will try to maximize some reward function specific to each task and in attempt to learn optimal behavior. We use RLBench [3], a simulation library that contains over 100 different environments and tasks to solve. Such tasks range from simple

target-reaching with a robot arm to more complex challenges such as pushing open a sliding window. This wide diversity of tasks is necessary for the meta-learning algorithms to learn general skills that can transfer to new tasks. The wide variety of tasks will also enable us to thoroughly evaluate the performance of each algorithm and identify their strengths and weaknesses. RLBench is built on top of the popular robotics simulator Pyrep [15] and provides a standard RL environment interface.

In addition to using RLBench, we draw inspiration from another library, Meta-World [2], that has even more tasks specially designed for evaluating meta-learning. To adhere to the community standard for evaluating meta-learning, we will follow Meta-World's grouping of challenges by increasing difficulty. We select the ML1 task-set, which is a set of push tasks with different goal positions. For training, we have a fixed set of five goal points. During inference, we sample a random goal point within the workspace of the manipulator.

4.2 Experiments

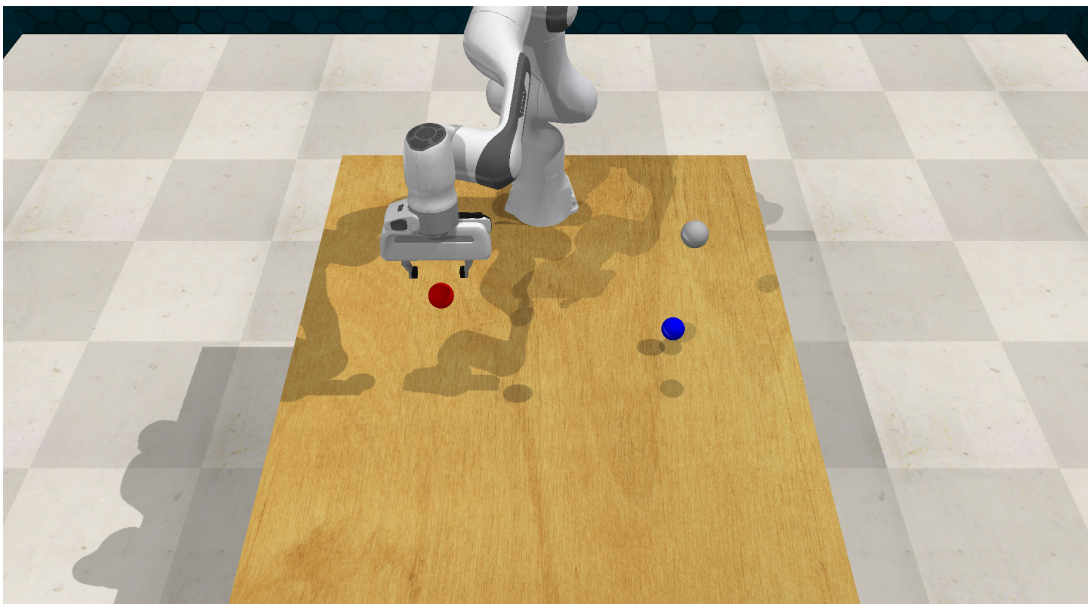

Figure 1: RLBench: Reach Task

Robot control tasks are typically difficult to learn from scratch. High-dimensionality, continuous action space compounded with the sequential nature of the predictions, the robot's own physical constraints makes it harder to achieve an optimal policy within a short time frame. The optimal policy also changes with the reward distribution, the dimensions of the robot and many other factors. This essentially means, even after extensive training of a robot on one task, one needs to train an entirely new policy for the same robot on a slightly different task or a slightly different robot on the same task. This is a redundant and inefficient method. Drawing from the way humans learn, one can easily see that much of the learnings from one of the set-ups can be used in the others, with some amount of task-specific fine-tuning. Meta-learning provides a formal framework for this kind of generalized learning and hence we explore it's applicability and effectiveness in robot manipulation.

The main goal of this project was to evaluate the effectiveness of meta learning in accelerating the speed of learning new policies for tasks in a similar task space. For the purpose of this project, we chose the reach tasks, which represents parametric task variation within the task set as described by [2]. More specifically, the goal of our experimental evaluation is to answer the following questions:

1. Can meta learning algorithms reliably outperform the typically used policy gradient algorithms in sample efficiency?
2. Can we propose novel modifications to the existing algorithms for faster convergence?
3. Can we train these algorithms asynchronously to speed up training?

To answer the questions above, we designed the experimentation and data collection on 4 different algorithms:

1. Vanilla PPO
2. Model-Agnostic Model Adapation (MAML)
3. Reptile
4. Multi-headed Reptile (Asychronous reptile)

To make a fair evaluation across the different experiment and configuration parameters, we consider the following factors for fairness:

- **Similar task variations**: Within the parametric task variations, we keep the reach goals static across different methods to prevent any stochastic unfair advantage.
- **Similar policy gradient steps for comparison**: Since each algorithm may have variations in how and when the gradient steps are undertaken, the comparison is made across these steps and not some arbitrary iterations.
- **Same policy parameters**: The initial parameters for the policy gradient method (PPO) while training the meta learning algorithms
- **Similar environment experience time across runs**: To compare across runs, we also ensure that the environment timesteps calculated for each gradient updates are consistent
- **Three random seeds**: To account for stochasticity of our learning process, we use 3 random seeds per experiment to ensure repeatability of our results.
- **Policy Hyper-parameters**: Since RL is notoriously sensitive to hyper-parameters, policy-gradient (PPO) algorithm hyper-parameters were kept consistent across different meta-learning parameters.

We started by setting up our simulator to generate the environment, define action and observation space, and create the pipeline for training (data collection, control, and resetting). We followed it by implementing from scratch two state of the art methods in meta-learning - MAML and Reptile and interfaced them with RL-Bench. Next, we formulated and implemented Multi-headed Reptile (described in Section 3) which seemed to balance stability, accuracy and complexity of gradient updates, theoretically. Finally, we added asynchronous training to speed up the process.

Table 1: Hyperparameters

Max Episode Length	200
Total Samples	1000
Num Iterations	400
Num Epochs	2
Batch Size	64
Num Tasks	10
Task Batch Size	10
Manual Termination	True
Penalize Illegal Actions	True

The above hyperparameters can be understood in the following pseudocode:

Algorithm 2: Pre-training Skeleton Algorithm

```
for iter = 1 : num_iters do
    for task in task_batch (size=task_batch_size) do
        During PPO's Adaption to a specific task:;
        for t = 1:total_timesteps do
            for e = 1:num_episodes do
                Gen + add new episode of max episode_length;
            end
            for epoch = 1:n_epochs do
                for batch (size=batch_size) in batches do
                    Calc loss over collected episodes, step gradients
                end
            end
            Post-update collection of new data and gradients:;
            for e = 1:num_episodes do
                Gen + add new episode of max episode_length
            end
            Gradients += this task's PPO Gradients
        end
    end
    Step with summed gradients
end
```

"Manual Termination" and "Penalize Illegal Actions" both describe how we handle infeasible actions taken by the policy. If the policy tries to move the robot's end-effector to a position beyond the reach of the arm or into itself, the underlying simulator will throw an error. By specifying "Manual Termination" as True, the episode will immediately stop. This is done since we empirically observe that future actions will simply reach nowhere and this exploration will be meaningless.

"Penalize Illegal Actions" means that the above early termination will incur an additional negative reward of -5, chosen as a hyperparameter. This is meant to discourage movement to infeasible regions.

4.3 Evaluation Metrics

The main goal of the project is to evaluate the effectiveness of meta-learning approaches towards generalizing to unseen tasks from a similar, yet sufficiently diverse task space. As opposed to the other approaches of evaluation discussed in the literature review, we propose evaluating the standard RL algorithm against their meta-learning variants using *goal task policy training time*.

It is important to clarify here that there are two training times that we need to be mindful of for the purposes of evaluation:

Meta-algorithm training time: This refers to the training time required to train a good model initialization. MAML, Reptile and multi-headed Reptile were all trained for equal number of iterations . For each gradient step, there are k roll-outs over N tasks for D max timesteps. This represents the total computation required to gain simulated experience.

Goal Task policy training time: Once we have a meta-learned initialized policy network, it is trained for a specific task using standard PPO. This training time is way more important since in theory you would like to learn the meta-algorithm only once for a task distribution and use those initialized weights to learn many tasks in the future.

4.4 Baseline

We will be collecting the data and executing the policies in the simulation environment. In the current simulation setup we have a 7 DoF robotic arm (Franka-Emika Panda) mounted on a table. Each episode starts with three random (sampled within the configuration space of the robot) points in the

3D space, one being the target and other two will act as distractors. After every reset these target points are sampled again. The robot has to reach the goal point by changing the end-effector position. Once it reaches there, the episode ends. The episode will also terminate after a specified number of iterations. We formally define the task parameters as follows.

Observation space The observation space consists the end-effector pose in 3D $[X_{ee}, Y_{ee}, Z_{ee}]$ and the goal position in 3D $[X_t, Y_t, Z_t]$. The pose is composed of the position (m) and orientation (quaternions) of the end effector.

Action space: We consider change in the gripper's 3D position as action. $[dX_{ee}, dY_{ee}, dZ_{ee}]$.

Goal The goal is derived by uniformly sampling joint angles within the joint's limitations. Based on these joint angles, setting the forward kinematics result (end-effector's pose) in Cartesian space as the goal can ensure a reachable target pose. The episode ends when end-effector of the arm reaches within a certain distance of the sphere.

Reward We have chosen dense reward function which is commonly used for reaching tasks. The reward is given based on the Euclidean distance between the end-effector and the goal pose. If the position distance is smaller than the required precision, we consider it a successful action. Otherwise, the reward is set to penalize the distance, which is value of the position distance itself.

We have built our end-to-end pipeline, from scratch, using RL Bench[3] and PyTorch[16].

Motivation for Baseline Our baseline algorithm is Proximal Policy Optimization (PPO). We chose this for its state-of-the-art performance as well as low implementation overhead. As a policy gradient, PPO can handle our complex and continuous action and state space using various flexible MLP architectures. As an improved version of Trust Policy Optimization Optimization (TRPO), PPO also has guarantees that incremental steps in policy parameter weights will produce equivalent or better performance. For these reasons, PPO is a popular algortihm in RL, and comparing to this SOTA method will highlight the performance of the meta-RL approaches we want to explore.

Caveats In traditional prediction tasks such as image classification, word prediction, or value estimation, there are no consequences for poor outputs from the model besides incurring more loss. In the robot control domain, however, the learned policy's actions must adhere to the robot's own physical limits. These include the 3D subspace of reachable end-effector positions, joint angle limits, and singularities that cause the robot's internal inverse kinematics solvers to fail. To handle these constraints, we needed to apply our own constraints to the raw outputs of the policy model. Since our action space is 3D translation of end effector position, translation has no theoretical constraints in real space: $[dX_{ee}, dY_{ee}, dZ_{ee}] \in \mathbb{R}^3$. However, the robot has physical limits to how far it can reach, and so any desired change to its end-effector position also has constraints. Before training our policy, we verified that the arm can indeed reach various positions within the target space.

We have two options of constraining the actions. First, the environment returns high cost and immediately terminates an episode if the policy attempts an invalid action. This works theoretically in the limit as the policy will learn to output valid actions that lie in the robot's constraints. In practice, however, its initial random policy will output many invalid actions and learn nothing from them in the beginning, so the model will initially make no progress. A second option is to scale down the model's output by some factor k as follows: $\frac{[dX_{ee}, dY_{ee}, dZ_{ee}]}{k}$. With option 2, the model can immediately begin to take valid actions and observe real reward signals. This does not bias the learning in any way since we are simply scaling actions by a constant, and the model will learn to adjust the scale of its output actions.

5 Results

To evaluate the performance of MAML, Reptile, and Multi-headed Reptile, we ran pre-training for 250 iterations. After pre-training, we then trained each model with the standard PPO algorithm on 10 different test tasks. In other words, given some model parameters θ, we spawned 10 different workers

with this same parameter set. Each worker then ran PPO to train on their particular test task, which is simply moving to one fixed location.

As a control, we also ran a set of randomly initialized weights with no pre-training on the same PPO training procedure. All models were initialized with the same random values using the three different seeds: 12345, 320, and 420.

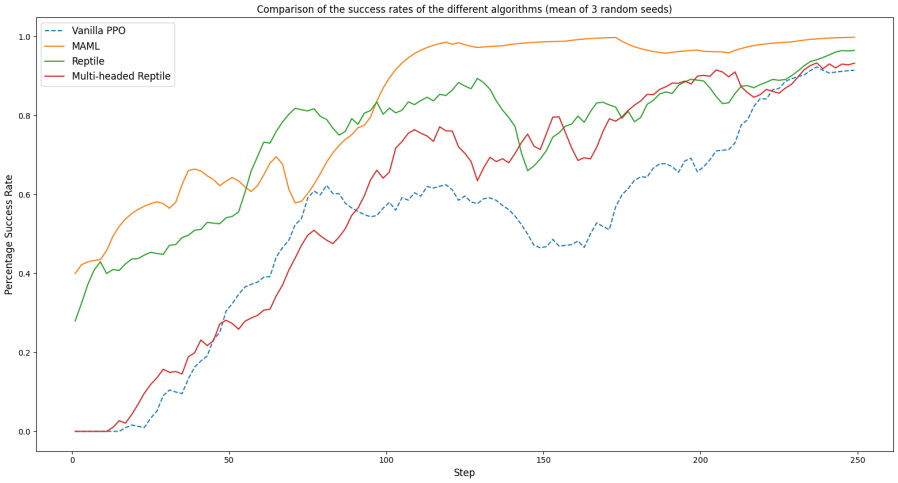

Figure 2: PPO: Success Rate Reach Task

Figure 2 compares the success rate of each model averaged over the 10 test tasks. MAML and Reptile both converge to 80% success rate within 100 iterations, whereas Multi-headed Reptile and random-initialized (Vanilla PPO) require 150 and 225 iterations respectively.

As theorized, both MAML and Reptile converge significantly more quickly than Vanilla PPO. Interestingly, even in the first iteration, these two meta-learning techniques achieve around 40% success rate, demonstrating their learned behaviour about the nature of the task distribution.

However, we found that Multi-headed Reptile does not work as well as we expected, performing just as well as Vanilla PPO. Although Multi-headed Reptile converged in fewer iterations than Vanilla PPO, its poor initial performance indicates that it failed to initialize weights optimally. As a sanity check, Vanilla PPO indeed has poor initial performance due to randomly initialized weights.

Table 2: Pre-training Run-time Comparison

MAML	13.5 mins
Reptile	2.1 mins
Multi-headed Reptile	2.6 mins

The table above also demonstrates that Reptile takes considerably less time to pre-train than MAML — approximately 6x less time — as an SGD method. Due to parallelism across asynchronous workers, Multi-headed Reptile achieves comparable performance with Reptile while still performing batch computation over multiple tasks.

6 Conclusion and Future Work

In this work we have compared the performance of new meta-learning variants of the standard RL algorithm, PPO. We have demonstrated that our baseline algorithm, PPO, can successfully learn a reasonable policy. Adding an outer loop of meta-learning to it makes the process of training for a new task considerably faster.

For the meta-learning algorithms, our results suggest that the policy learnt using MAML converges faster, however at the cost of computational complexity. This can be attributed to the heavy computational graphs for the loss functions which preserve more information for accurate gradient computation. Policy trained with Reptile takes significantly less time ($\sim$6x less) and provides comparable results with MAML.

We also explored Multi-headed Reptile, which has the following two changes,

- Batch updates of parameters
- Asynchronous training

We observed that the Multi-headed Reptile doesn't perform as expected. We currently have two hypotheses in mind to explain the results.

- Our method assumes θ_{new} - θ_{old} to be the gradient of weights. This, while simplifying computation, loses information of higher-order terms.
- Lowering variance might be preventing exploration in the parameter space as we are taking very controlled steps from the very first iteration.

One possible experiment to test out hypothesis 2 is to start with batch size (N) of 1 and gradually increase it.

Further next steps in this direction would include checking how an increase in task complexity affects the results and getting an understanding of which algorithms scale well with complexity. Meta-world provides some other, more complicated task sets like ML10 and ML45, which may be used for this. This will give us a more comprehensive understanding of the performance of the algorithms.

7 Work Division

The following table represents the key responsibilities of each of the team members, however they do not imply their only contribution in the project. All the members of the team assisted with most parts of the project to a large extent, resulting in effective collaboration.

Alvin	MAML and Reptile core code implementation
Chandreyee	Algorithm research: MAML and RL^2; Reports
Deval	Skeleton code; Code for Multi-headed reptile; Training scheduling
Rohan	Algorithm research: Reptile and Multi-headed Reptile; Slides and video

References

[1] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019.

[2] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning, 2019.

[3] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. Rlbench: The robot learning benchmark learning environment, 2019.

[4] Tianbing Xu, Qiang Liu, Liang Zhao, and Jian Peng. Learning to explore with meta-policy gradient, 2018.

[5] Pratt L. Thrun S. *Learning to Learn*, chapter Learning to Learn: Introduction and Overview, pages 201–213. Springer, Boston, MA, 1998. https://doi.org/10.1007/978-1-4615-5529-2_1.

[6] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Evolutionary principles in self-referential learning. on learning how to learn., 2019.

[7] Devang K. Naik and R. Mammone. Meta-neural networks that learn by learning. *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, 1:437–442 vol.1, 1992.

[8] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[10] A. Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. Benchmarking reinforcement learning algorithms on real-world robots, 2018.

[11] Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning, 2016.

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[13] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms, 2018.

[14] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017. URL `http://arxiv.org/abs/1712.05889`.

[15] Stephen James, Marc Freese, and Andrew J. Davison. Pyrep: Bringing v-rep to deep robot learning, 2019.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.