Using REPTILE to Learn General Strategies Across Environments

May 12, 2020

1 Introduction

Inspired by the MAML paper, in this project we aim to evaluate the effectiveness and bounds of "meta-learning" a fast-adaptation policy initialization for solving a set of tasks [3]. The goal is to evaluate whether such fast-adaptation techniques are effective in the context of current RL algorithms, can be extended beyond the task distributions that were tested in papers, and offer any benefit beyond pre-existing methods such as pretraining a network.

Methods such as MAML, FOMAML, and more recently, REPTILE, which is an easier-to-implement and provably similar method to FOMAML [4] have shown great promise in rapidly increasing the adaptation speed of neural networks in areas such as classification and regression tasks. However, although MAML has seen success at increasing the adaptation speed of a network in RL tasks, we believe that these task distributions are not representative of the task distributions that meta-learning a network initialization should be able to handle. Namely, methods such as MAML and REPTILE evaluate performance against image classification tasks in the context of having many different classes of images (the tasks in the the distribution of classification tasks), in which the meta-learning algorithm is provided k samples of each in the meta-training stage, and then must accurately classify an unseen sample from one of these classes. This is the problem called k-shot learning. However, when testing RL algorithms, the MAML paper evaluates its performance on a 2D navigation problem, and a half-cheetah problem. It seems to us as though the diversity of tasks in the task distribution is very narrow. Whilst in the classification task, the space of inputs to the MAML algorithm is incredibly large (the space of possible images of input dimensions), in the RL 2D control task distributions, the space of inputs are only the $[x, y]$ location of the agent. Meanwhile, in the half-cheetah experiment, the number of desired behaviors is just 2: move forward, or move backwards. Meanwhile, image classification has countless possible classifications of which the MAML algorithm is able to find a network initialization that lies near the manifold of solutions to each one of the classification task distributions. We seek to measure if fast adaptation algorithms can do the same for environment distributions that are broader than half-cheetah and 2D control.

1.1 Project Contributions

1. Implement an actor-critic version of REINFORCE with PPO.
2. Design and test two fast-adaptation algorithms for learning a good network initialization: Batch REPTILE for RL, inspired by [4], pretraining for RL.
3. Implement Batch REPTILE for RL, Pretraining for RL in an easy to use Jupyter Notebook, and construct a framework for testing the algorithm against baselines.
4. Replicate the MAML 2D control experiment to compare REPTILE against the performance of a pre-trained network, and evaluate how this stacks up against reported MAML results against a pre-trained network.
5. Build a new task distribution of games that is more complex in the number of task distributions than either 2D Navigation Control or Half-Cheetah, and extend and evaluate Batch REPTILE for RL against this new distribution.

2 Fast Adaption Algorithms

Algorithm 1: Batch REPTILE for RL

```
**Require:** Distribution over tasks $p(\tau)$
**Require:** $\alpha, k$ hyper-parameters, $J$ the number of tasks per meta-train loop, $B$ is the batch size per
  task, $n$ is the number of meta-train batches
*Initialization:* Random $\theta$;
**for** $i$ = *1, 2,...,n* **do**
    sample set of tasks $\{\tau_0, ..., \tau_j\} \sim p(\tau)$ for $j \in [0, J]$;
    **for** *each* $\tau_j$ **do**
        Reset any optimizers or momentum between tasks;
        $\theta_j \leftarrow$ perform $k$ steps of update on $\theta$ using $B$ episodes generated from $\tau_j$ per update;
    **end**
    $\theta \leftarrow \theta + \frac{\alpha}{k} \sum_{j=0}^{J} (\theta_j - \theta)$;
**end**
```

Algorithm 2: Pretraining for RL

```
**Require:** Distribution over tasks $p(\tau)$
**Require:** $J$ the number of tasks per pretrain loop, $n$ is the number of meta-train batches
*Initialization:* Random $\theta$;
**for** $i$ = *1, 2,...,n* **do**
    sample set of tasks $\{\tau_0, ..., \tau_j\} \sim p(\tau)$ for $j \in [0, J]$;
    Generate episodes $E = \{e_0, ..., e_j\}$ following $\theta$ on $\{\tau_0, ..., \tau_j\}$;
    $\theta \leftarrow$ update $\theta$ with one step using episodes $E$;
**end**
```

FOMAML and REPTILE are two similar meta-initialization algorithms. REPTILE is known to be easier to implement. The primary difference is that in FOMAML, the initialization network gets updated with just the k^{th} gradient update of the inner training loop on the sample task, while in REPTILE, all k gradient updates from the inner training loop are used to move the initialization network towards the optimal solution for all τ.

We use a Batch version of REPTILE modified for RL in this project, due to its simpler implementation than MAML or FOMAML. Additionally, we see in the figure below that REPTILE version that uses all 4 inner loop gradient updates ($g_1 + g_2 + g_3 + g_4$) actually outperforms the equivalent FOMAML (g_2) in learning a meta-initialization on an Omniglot test set [4]. Unless otherwise noted, in our testing we use REPTILE with $k = 4$.

To provide a comparison baseline, we additionally pretrain an initialization on the same tasks as REPTILE, using Algorithm 2.

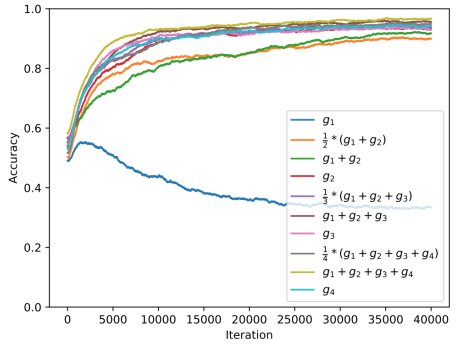


Figure 3: Different inner-loop gradient combinations on 5-shot 5-way Omniglot.

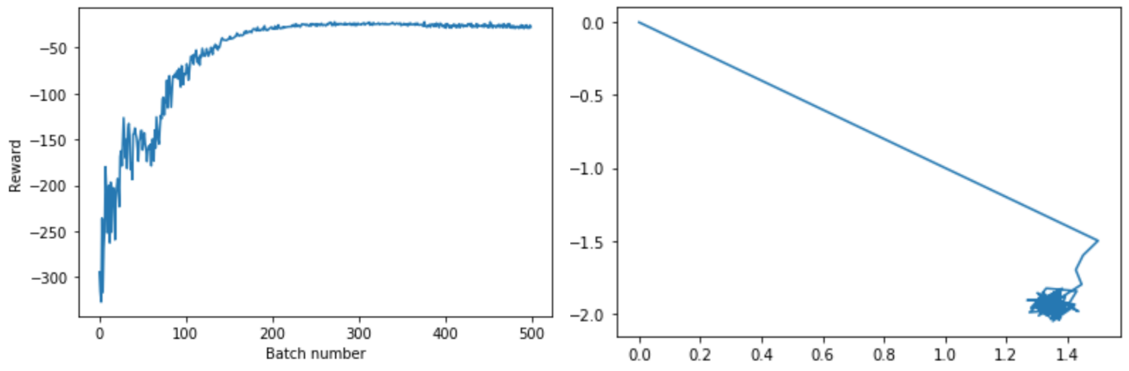
The line in blue (g_1) does not follow the trend of the rest of the lines, and is a degenerate case. In the blue situation, only using the first gradient update falls apart in environments in which the initial gradient updates are in completely different directions (such as permuted labels in classification). Thus the resulting initialization would theoretically perform poorly. Later in the paper, we provide a pair of environments that would have the potential to cause this degenerate behavior.

3 REPTILE in the 2D Control Experiment

In this environment, the agent starts at location (0, 0) and must navigate to a randomly specified location on the continuous plane. We randomly draw goal locations from the [-2, 2] box in both the x and y coordinate locations (note that this is 4x larger a sample space from the unit square that the MAML paper draws from). The agent's commands correspond to velocity commands clipped to [-0.1, 0.1]. Reward is the negative distance to the goal, and episode ends if the agent gets within 0.01 distance of the goal, or the horizon of 100 is reached. We believe that any meta-knowledge in a good initialization should include the ability to quickly adapt to any of the countless possible random goal locations. Additionally, the tendency to reduce the magnitude of velocity commands as time goes on, in the hopes of getting close enough to end the episode, should emerge in from an ideally learned meta-initialization.

3.1 Baseline Results of actor-critic REINFORCE with PPO

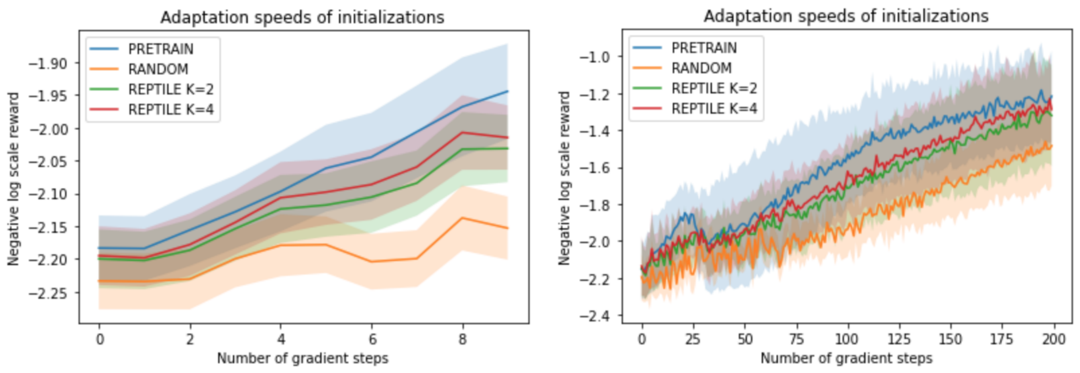
We first ensure that the policy gradient algorithm is able to solve the control task in the domain of a fixed goal. We run the algorithm with 500 batches of batch size 10, with 2 mini batches over the data generated in each complete batch.



The agent is able to achieve the near-optimal performance at around batch 200, during which the agent reaches the goal and ends the episode with high frequency. We observe the tendency of the agent to use its full velocity commands, and then slowly reduce the magnitude of the velocity for fine grained control as it is close to the goal location.

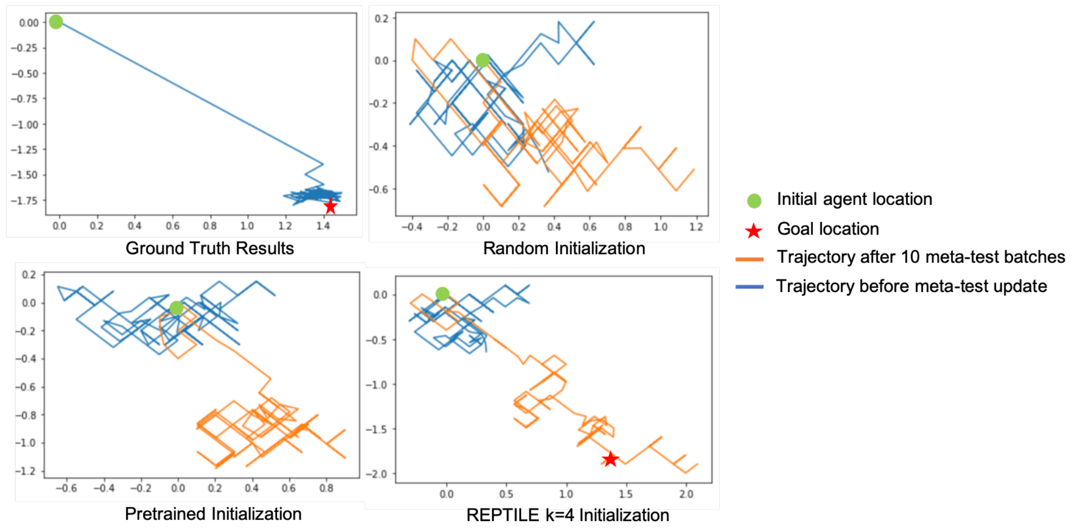
3.2 REPTILE vs. Pre-trained vs. Random initialization

The MAML paper measures how quickly a network initialization is able to increase its rewards to evaluate the effectiveness of that initialization. This makes sense, since given enough time, compute, and a theoretically sound RL algorithm, most initializations should converge to a locally optimal solution. To ensure that our comparison is fair, we set the random seed for the initial random initialization of θ to 1. Then, we run REPTILE on θ with $k = 2$ gradient updates and $k = 4$ gradient updates to produce a new meta-learned network initialization θ_{rep_2} and θ_{rep_4}. We also pre-train on θ to yield θ_{pre}. We finally compare the initialization speed of θ_{rep_2}, θ_{rep_4}, θ_{pre}, and just θ. To test initialization, we use 10 gradient updates, each with a batch size of 10 and 2 mini-batch updates over the complete data in from each batch, and do this on 30 sample tasks (i.e. goal locations) drawn from the task distribution. The same 30 sample tasks are used over the evaluation for all methods, and 95% confidence bounds are shaded.



The first thing to note is that as with the MAML paper results, both REPTILE and pretraining a network initialization produce results that have larger slopes of reward/gradient updates than a random initialization. Additionally, we confirm that REPTILE with 4 gradient steps does slightly outperform REPTILE with 2 gradient steps.

Certainly the most surprising thing is that the pre-trained network outperforms REPTILE. It is uncertain why this is the case. One stipulation of REPTILE is that we reset the optimizer for the network updates between training on sample tasks, such that there is no information leakage (such as momentum) from one sample task to another. However, while pre-training a network initialization, we do not reset the optimizer for the network updates. This is one of the few differences between what distinguishes the pre-train method from REPTILE with $k = 1$ (which supposedly is the degenerate case). Another possibility is that since update of the inner meta-train loop in REPTILE contributes $\alpha = 0.1$ to moving the initial network. However, in the pretrain method, each gradient update is directly applied to the initial network. We believe that 500 batches (same as MAML paper runs) of REPTILE and pre-train was sufficient to lead both methods to converge to their optimal meta-initializations, however, it is possible that REPTILE needs to run for longer. Lastly, we present an additional hypothesis explained by the figure below.



To get a visual sense of the behavior of these meta-initializations, we have plotted the trajectory of each initialization before and after 10 meta-test gradient updates. We randomly chose a goal location at (1.42, -1.976) to do this evaluation. Qualitatively, it does seem that the REPTILE result is able to move to the final goal location faster than the pre-trained initialization is able to do so (in fact, the pre-trained initialization does not reach the goal location in this example). Certainly, (1.42, -1.976) is a goal that is at the farthest extremes of the sampling box of goal distributions, so it is possible that while pre-training is able to quickly reach to the goal location in expectation (i.e. goals in a radius of 1 around the agent's starting position), REPTILE is actually able to reach goals that are farther away from the mean of the sampling task distribution.

4 Extending the Task Distribution

To test fast-adaptation beyond the relatively narrow task distributions provided as examples in the MAML paper, we define several new game-like environments. The experiment is formulated to have a larger set of task distributions than either the 2D Navigation Control environments and the Half-Cheetah Environments from the MAML experiment. If we think of the the 2D navigation task, REPTILE is being trained essentially on one task distribution τ_{2D}, from which different tasks each with a different goal location are sampled. However, in the realm fast-adaptation for k-shot Image Classification, we may have more task distributions; for example, $\tau_{dog}, \tau_{cat}, \tau_{bird}$, in which the REPTILE/MAML/FOMAML would see k different instances each of a dog, cat, and bird.

It makes sense to attempt to meta-learn a network initialization that is better at handling random samples from simply one task distribution. Ideally, a great meta-learning initialization could more closely mimic human behavior, with strong priors over general game mechanics ranging from 2D games controlled with a joystick, to 3D games controlled with a multi-input controller, to strategy games, and more. Such knowledge priors might look like knowing generally what an object is, and having expectations that map actions to behavior given a certain observed environment. However, training an initialization on such a diverse set of task distributions would require care in constructing inputs/outputs of the policy that could handle the various diverse tasks.

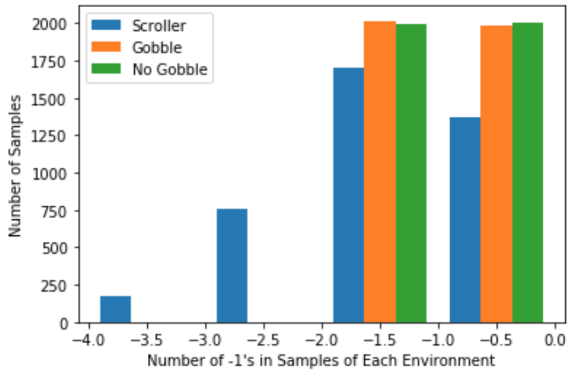
This experiment is kept simple by having all of the test game distributions have a standard 6x6 input to the policy, and be interfaced with a standard arrow-key style, 4-option, controls controls that correspond to different actions in each environment. We define 3 testing environments, that each acts as a distribution from which to draw randomly sample a different game map that uses the game rules of that distribution. In this manner, training a meta-learned RL initialization bears more similarity to the k-shot image classification problem – the meta-learning algorithm will first sample an environment distribution, and then it will get a random instance of that environment, that follows the game rules of that environment.

Test	**Set of task distributions**	**Within task variance**
2D Navigation Control*	$\{\tau_{2D}\}$	Location of goals
Half-Cheetah Experiment*	$\{\tau_{cheetah}\}$	Moving forwards vs. backwards
Our New Game Environments*	$\{\tau_{gobble}, \tau_{nogobble}, \tau_{scroller}\}$	Randomly generated maps
k-shot Dog/Cat/Bird Classification	$\{\tau_{dog}, \tau_{cat}, \tau_{bird}\}$	Different images

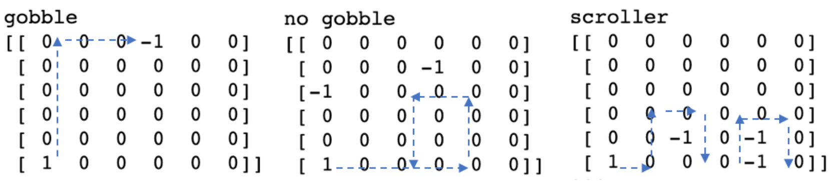
This table provides a compare and contrast between different test spaces used to evaluate the performance of meta-learning a network initialization. The asterisked tests indicate that the test is used in the RL setting.

4.1 Summary of Distributions

Here we provide a summary of the three simple games that were created to test REPTILE. Each of the games lies within an adjustable nxn grid, which we keep small at 6x6 to enable faster training, but still provide a good evaluation of the adaptation speed of a REPTILE-learned initialization. In all of the environments, the agent starts at the bottom left. Additionally, the only elements of each environment is a **1** to represent the agent, a **-1** to represent an object, and **0** to represent empty space. Thus, on the surface each game would look the same to a fast-adaptation algorithm – given the plain board state, there nearly no way of knowing which world environment an agent is in. World dynamics can generally only be figured out by engaging with the world and observing results. The only edge case to this situation is the situtation in which a Side Scroller environment generates 2 walls, where at least one of the walls is of height > 1. The histogram below shows the sample distribution of the number of **-1**s in each environment over many samplings of each environment. Most samples of each environment distribution result in a map with 1 or 2 bucket instances of **-1**. Clearly, using just the number of **-1**s, all of the time the environments are indistinguishable most of the time. Of course, the likelihood of observing a sample environment with more **-1**s in the bottom rows is higher given that the environment distribution is scroller, allowing for the ability to use bayesian inference to draw conclusions about the environment distribution – if such a feat could be accomplished by a network.



Sample game environments



```
gobble
[[ 0  0  0 -1  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 1  0  0  0  0  0]]

no gobble
[[ 0  0  0  0  0  0]
 [ 0  0  0 -1  0  0]
 [-1  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 1  0  0  0  0  0]]

scroller
[[ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0  0  0  0  0]
 [ 0  0 -1  0 -1  0]
 [ 1  0  0  0 -1  0]]
```

Movement	Up, Down, Left, Right by +1	Up, Down, Left, Right by +1	Jump +2 if on bottom row, Accelerate L +1, or R +1
Objective	Gobble either one and two −1s by moving over them	Survive as long as possible	Reach bottom right square
Starting Position	Bottom left square	Bottom left square	Bottom left square
Ending state	When all −1s are gobbled	Touching a −1, or hitting a boundary	Agent is in bottom right square
Reward	+10 at each gobbled item, -1 for each step otherwise	-100 for premature game termination, +1 for each step otherwise	-1 each step
Special rules	N/A	N/A	Jumping and gravity. Max L/R velocity is 2. Hitting a −1 sets velocity to 0.
Horizon	100	100	100
Within task variance	Either one or two −1s can spawn anywhere on the board	Either one or two −1s can spawn anywhere on the board	One or two columns of −1 walls spawn randomly at height 1 or 2 between agent and goal

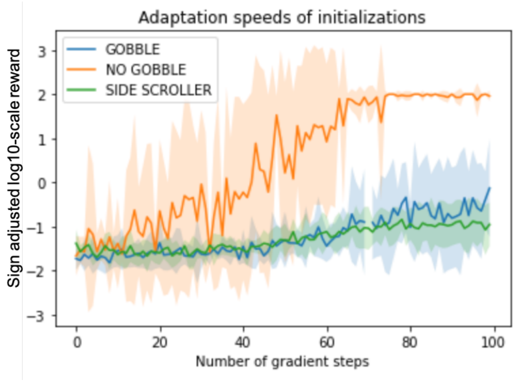
In the summary table above, the blue dotted lines represent possible action trajectories of an agent acting on that specific instance sampled from each game environment. Note that the maps shown above the table do not represent that environment in its entirety, but rather are just single samples of a possible maps in from environment.

4.2 Evaluation

For all training, the we use a two sliding window state as input to the policy. For a 6x6 maze at timestep t, we provide a 1x36 input composed of $S_t + 0.5 \cdot S_{t-1}$. This sliding window serves as one of the few indications to an agent as to what world environment it is in. Note that this actually is only helpful in distinguishing the Side Scroller environment, as in this environment an agent can move a distance of greater than one cell per step, such that the **0.5** $t-1$ agent signal and **1** t agent signal may not be adjacent when arranged in a 6x6 grid.

4.2.1 Assessing Performance from Training on Each Environment Individually

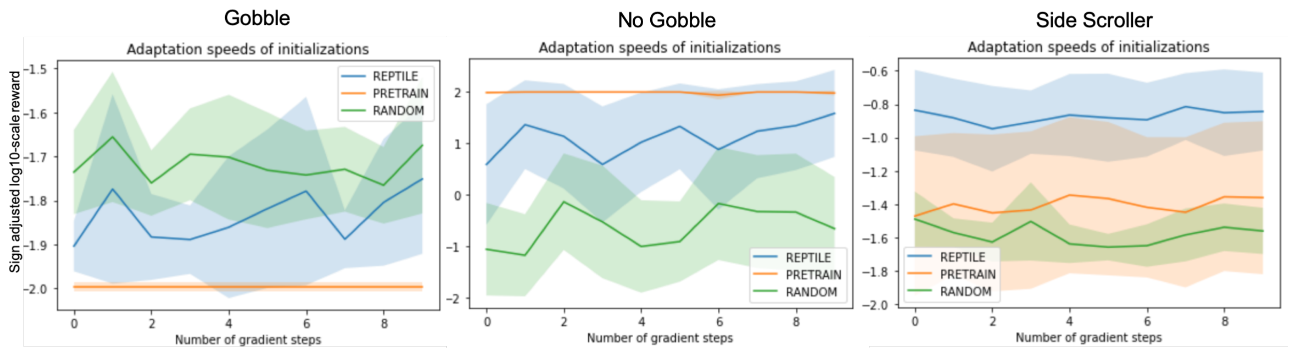
Since REPTILE evaluation will measure how quickly a network initialization can accumulate reward in the first 10 gradient updates, we verify that a random initialization cannot achieve the maximum reward for each 6x6 environment in less than 10 gradient updates. To do this, we train a random network initialization (with seed 1) on *just one instance of one environment distribution each* – i.e. we solve each problem individually, as well as it can be solved by the policy gradient. We do this 5 times to establish a confidence interval of how well the policy gradient can solve each specific instance of the environment. We did experiment with using a CNN equipped policy, however this proved to be much less effective than the plain MLP policy. Maximum reward in the Gobble and Side Scroller environments is not achieved until beyond gradient update 80, while No Gobble maximum reward reliably converges around 60. Thus, any benefit from a good initialization could reliably be observed within the first 10 gradient steps of evaluation.



4.2.2 Assessing Performance from Training on all Environments in Aggregate

In contrast to the baselines above, we now train both the pretraining and REPTILE algorithms on random samples of randomly choosen test environments. That is to say, $\tau = \{\tau_{gobble}, \tau_{nogobble}, \tau_{scroller}\}$, and there is uniform probability over sampling each of the three sub-distributions. Batch REPTILE is trained on 500 meta-train loops, with a batch size of 10 meta-train samples and 5 episodes per sample for $k = 4$ gradient updates in each meta-train loop, with a a meta-adaptation $\alpha = 0.1$ and Adam used as the optimizer within the meta-train loop with a learning rate of 3e-4. Pretraining uses a batch size of 10 random samples of environments with one episode per sampled environment, for 500 batches also with Adam and a learning rate of 3e-4.

Given the resulting REPTILE, PRETRAIN, and RANDOM initializations, we test the ability of each initilization to adapt to each environment distribution separately from the other environment distributions. The results are seen below.



As seen in the figure, REPTILE has the best overall performance. REPTILE outperforms a random initialization in both the No Gobble and Side Scroller environments. Additionally, REPTILE has a stellar performance in the Side Scroller environment, in which it off-the-bat meets the max reward seen in the baseline results from the figure above.

REPTILE does not fare as well in the Gobble environment. One hypothesis is that this is because the Gobble and No Gobble environments have directly opposite goals from one another, and are perfectly indistinguishable in terms of environment appearance and movement dynamics. It is actually surprising that a network pretrained on both of them learns anything at all, since one would expect that gradient updates in opposite directions would lead to network initialization degeneracy.

We observe the effect that these opposite goals have on the PRETRAIN network. As observed, the pretrain network achieves maximal performance on No Gobble, but makes no reward at all on Gobble. This is certainly due to the fact that the PRETRAIN network has fallen into the degenerate case of solely optimizing for No Gobble, which is the easier environment to win.

Overall, it is quite awesome that REPTILE is able to nearly match or beat the next best network initialization option across the board. [4] stated that REPTILE did not yet work in the RL setting, but from these experiments, we believe that REPTILE shows great promise in learning a good network initialization in RL environments.

5 Conclusion

Given REPTILE's performance on the Gobble, No Gobble, and Side Scroller environments, we believe it shows that REPTILE takes good steps towards learning a solid foundational policy initialization. Hopefully one day, a policy initialization suitably trained on a wide array of tasks may be able to match or beat human performance at quickly adapting and learning from new, previously unseen, games. Future work may include expanding the tasks to learn from, or improving the policy structure – such as more successfully incorporating CNN architecture, and determining how to properly design an input and output set that is flexible across a diverse range of tasks.

To try out the project yourself, or view the code that generated these experiments, navigate to `https://github.com/kennyderek/reptile-rl`, and try running Jupyter Notebooks configured with REPTILE and pretrain settings.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning : an Introduction / Richard S. Sutton and Andrew G. Barto.* Second ed., The MIT Press, 2018, *Richard Sutton's Home Page*, incompleteideas.net/book/RLbook2020.pdf.

[2] Olah, Christopher. "Understanding LSTM Networks." *Understanding LSTM Networks – Colah's Blog*, 17 Aug. 2015, colah.github.io/posts/2015-08-Understanding-LSTMs/.

[3] C. Finn, P. Abbeel, S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks,“ *arXiv:1703.03400*, 2017.

[4] A. Nichol, J. Achiam, J. Schulman, "On First-Order Meta-Learning Algorithms,“ *arXiv:1703.03400*, 2018.

[5] D. Wierstra, A. Förster, J. Peters and J. Schmidhuber, ”Recurrent policy gradients," in *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 620-634, Oct. 2010, doi: 10.1093/jigpal/jzp049.

[6] "An Introduction to Policy Gradients for Sequence Tasks." *An Introduction to Policy Gradients for Sequence Tasks - Lunit Tech Blog*, 7 Aug. 2017, blog.lunit.io/2017/08/07/an-introduction-to-policy-gradients-for-sequence-tasks/.

[7] Falcon, William. “Taming LSTMs: Variable-Sized Mini-Batches and Why PyTorch Is Good for Your Health.” *Medium*, Towards Data Science, 4 June 2018, towardsdatascience.com/taming-lstms-variable-sized-mini-batches-and-why-pytorch-is-good-for-your-health-61d35642972e.

[8] “Making a Maze.” *Making a Maze*, 13 Apr. 2017, scipython.com/blog/making-a-maze/.