From Multi-Task RL to Meta-RL: Fine-Tuning and Experience Replay

Chris Guthrie
Computer Science
University of Colorado Boulder
christopher.guthrie@colorado.edu

Thanushraam Suresh Kumar
Robotics
University of Colorado Boulder
thanushraam.sureshkumar@colorado.edu

***Abstract*—This report investigates different approaches to training a reinforcement learning algorithm to perform well across multiple tasks and to generalize to unseen task types. We aim to reproduce various results from across the multi-task and meta-learning RL literature, and to understand the effect of different replay buffer strategies on the effectiveness of multi-task generalization.**

I. Introduction

An important challenge for autonomous learning systems is generalization. To generalize successfully, a system must be able to learn both similarities and differences across the situations it encounters. That is, a general autonomous system must be able to remember and exploit the "laws of the universe" in the environment in which it operates, while also understanding that some patterns are not law-like, and only apply in specific situations.

Generalization challenges in reinforcement learning are often framed in terms of *multi-task* environments, which require simultaneously mastering multiple related tasks. A learning system which is able to handle multi-task environments well is a sample-efficient learning system, since it doesn't need to re-learn the common structure of the environment for each new task.

The *meta-RL* setting challenges systems to generalize well, not simply across tasks, but to new unseen tasks. A system which can handle meta-RL settings well is not only more sample-efficient, but also potentially deployable into more unpredictable situations, since it is able to adapt to novel situations more quickly.

In this report, we evaluate extensions to the Soft Actor-Critic algorithm against a multi-task and meta-RL robotic simulator benchmark. We show a positive transfer effect across skills when skills are trained simultaneously in the multi-task setting. For the meta-RL setting, we find that online training against unseen skills is effective, and can be helped by pre-training against multiple other skills. Our experiments with prioritized experience replay do not show positive results.

II. Background and Related Work

A. Multi-Task RL

Multi-Task Reinforcement Learning refers to training a single agent to perform multiple tasks, rather than trying to learn each task separately. In multi-task RL, at the start of each environment episode, a specific "task" is selected from a distribution of tasks. Each task will have distinct rewards and distinct initial states. However, within each task, the rewards and dynamics typically don't change over time or with new instances of the task.

There are a few broad approaches to multi-task RL, as outlined in [1]. The simplest is the *multi-task model*, in which the different tasks share all their model parameters, and the tasks are differentiated from one another via an explicit task ID signal in the model input. Another approach is the *multi-head multi-task model*, in which all tasks share a common representation model, but each has its own "head" model which takes the common representation as input. The most involved approach is *soft-modularization*, which modularizes neural network layers and learns task-specific combinations of these modules to produce a final output. [1] proceeds to propose a novel mechanism for multi-task RL, which learns a unique weighting of the same model parameters for each task.

For this paper, we will focus on the multi-task model for its simplicity.

B. Meta-RL

The meta-RL setting is similar to the multi-task setting in that each new episode of the environment resets to a different task. However, in meta-RL, new tasks are introduced at test-time that cannot be seen during training.

[2] describes two broad approaches to meta-RL. The first approach, called *parameterized policy gradient*, involves running a meta-gradient-descent algorithm (e.g. MAML) on a policy gradient. The second, called the *black box* approach, involves treating the meta-learning problem itself as a POMDP, in which the hidden state is the current task (defined as an MDP) along with its state. Meta-learning then proceeds via general POMDP techniques.

However, some research suggests that these approaches may be more sophisticated than necessary to achieve adequate performance. In particular, [3] finds that (at least in some vision-based applications), simple fine-tuning of multi-task models during train-time suffices to achieve good meta-learning performance. Since this is easier to implement, it will be our choice of "meta-learning" algorithm.

C. Experience Replay

Experience replay is a technique in reinforcement learning where the agent stores past interactions in the form of *(state, action, reward, next state)* within a replay buffer. The training algorithm then samples batches from this buffer, instead of relying solely on the most recently gathered data. Re-sampling past data improves sample efficiency and helps prevent *catastrophic forgetting*, in which previously well-learned patterns are "overridden" by newer samples biased toward recent experiences.

Previous works, notably [4], have shown that experience replay helps mitigate forgetting in multi-task learning, particularly when tasks are trained sequentially. Since the fine-tuning approach to meta-RL involves, effectively, sequential training, we expect this approach to be the most sensitive to the particulars of how experience replay is implemented.

A well-studied enhancement is *Prioritized Experience Replay* (PER) [5], which addresses a key inefficiency in uniform sampling: treating all transitions as equally important. PER assigns higher sampling probabilities to transitions with large temporal-difference (TD) errors—transitions where the model's prediction significantly diverges from the actual outcome. These transitions typically carry stronger learning signals.

By focusing updates on high-TD-error transitions, PER improves sample efficiency and accelerates convergence. In environments with sparse or deceptive rewards, PER also surfaces rare but informative experiences that would otherwise be under-sampled in a uniform replay scheme.

III. Problem Formulation: The Metaworld Environment

Metaworld [6] is a benchmarking library published in 2019 by leading multi-task RL and meta-RL researchers. Metaworld provides a variety of Gymnasium environments with shared structure. Specifically, each Metaworld environment is built on top of the MuJoCo physics simulator, and models a robotic "Sawyer" arm on a surface. There are two levels of variation within Metaworld environments. The coarser level of variation, referred to in the original Metaworld paper as non-parametric variability, comes from distinct skills which the arm can learn — for example, opening a door or placing a ball into a basketball hoop. The finer level of variation, or parametric variability, comes from different initial positions and/or goal positions within each skill — e.g. the door may be placed in a different location.

Fig. 1, Fig. 2, and Fig. 3 illustrate some of the skills the simulated robot is trying to learn.

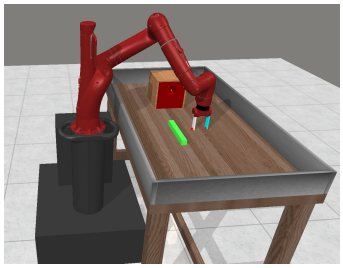

Fig. 1. Peg Insert Side Skill

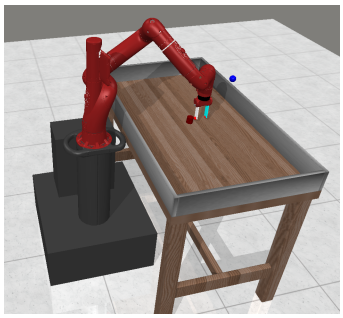

Fig. 2. Pick and Place Skill

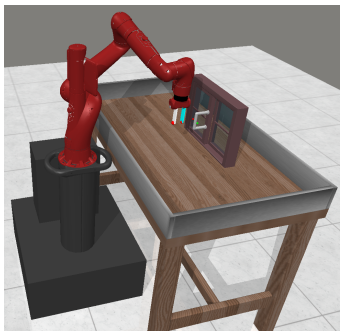

Fig. 3. Window Open Skill

A. States, Actions, Observations, Transitions, Rewards, and Discount

The authors of the Metaworld paper define the multi-task and meta-learning problems in terms of simple MDPs, but their environment gives the agent access only to observations rather than to states, making it technically a POMDP. As we will see, the observations of the given POMDP do give a nearly-complete picture of the actual underlying state space, so we chose not to use POMDP methods like recurrent NNs, and to treat the problem as if the observation space were a state space.

1) States: The shared state space of this problem, $\mathcal{S}$, is not made explicit in the original Metaworld paper, but is implemented in and dependent on the MuJoCo physics engine. It's thus a rich and complex state space. It includes, in part, the robot's end-effector position and velocity, as well as the object pose (if present).

The state is also augmented with a task identity $\mathcal{T}$, which is different for different tasks and skills, and influences the rewards. So, the state of each timestep is represented by a point in $\mathcal{S} \times \mathcal{T}$.

2) Actions: Each task has a 4-dimensional action space $\mathcal{A} \subset \mathbb{R}^4$. The first three dimensions correspond to the 3D Cartesian control of the end-effector of the robot, and the last dimension is a torque that the robot arm's grippers should

apply.

3) Transitions: The dynamics $\mathcal{P}(s' \mid s, a)$ of each skill are determined deterministically by the MuJoCo physics engine, based on the shared state space $\mathcal{S}$.

4) Observations: The observation space $\mathcal{O} \subset \mathbb{R}^{39}$ includes:

- The 3D Cartesian positions of the two sides of the end-effector.
- A measurement of how open the gripper is.
- The 3D Cartesian positions of up to two objects.
- The rotations of up to two objects.
- Some summary of observation history (the specifics of this are unclear).
- The 3D Cartesian position of a goal, if present.

5) Rewards: The rewards were shaped by the Metaworld designers to be dense and easy to learn. Reward functions potentially include multiple components to help hint the agent to solve important sub-tasks. The rewards are also normalized across tasks. The reward of a successful termination is 10 in all tasks.

6) Discount factor: We use a discount factor of $\gamma = 0.99$.

7) Initial Position: The initial state varies widely depending on the skill involved and also with the specific task instantiation of that skill. However, it will be deterministic in the task identity $\mathcal{T}$.

B. The Benchmarks

Metaworld includes environments for a total of 50 distinct skills, and each skill has a total of 50 distinct task instances for each skill.

In this paper, we focus on three benchmark environments from the Metaworld suite: **MT1**, **ML1**, and **MT10**.

- The **MT1 (Multi-task 1)** benchmark focuses on a single manipulation skill such as `reach-v2` or `pick-place-v2`. Within each skill, there are 50 separate task instances (e.g. multiple goal locations in `reach-v2`). At the start of each environment episode, one of these 50 task instances are randomly selected. The 50 task instances are shared between training and testing environments.
- The **ML1 (Meta-learning 1)** is a *meta-learning* setup, where the agent is trained on a single-skill environment indistinguishable from the corresponding MT1 environment. However, the agent is tested against different task instances that were not seen during training. The objective is to enable fast adaptation to new unseen variations of the same skill at test time, using a few examples (few-shot learning).
- **MT10 (Multi-task 10)** is a multi-task learning benchmark spanning 10 diverse manipulation skills. This setup evaluates the agent's ability to generalize across heterogeneous skills and share knowledge effectively.
- **ML10 (Meta-learning 10)** includes distinct skills in the training set and the test set. Every skill has a task distribution of 50 different starting position. This benchmark tests an algorithm's ability to generalize not only to new instances of the same skill, but to entirely new skills.

C. Evaluation of Success

The Metaworld paper has a notion of task "success" which it uses to evaluate algorithmic performance, and we follow that paper. Each task has its own unique definition of success (documented in the Metaworld paper), but it typically involves getting an object within some distance ϵ of a goal position. The Metaworld environment helpfully indicates whether a given action resulted in a success via side-channel information (the `info` return value from the Gymnasium `step` function).

IV. Solution Approach

A. Baseline: SAC with one-hot encoding

We used the stable-baselines3 package [7] for our SAC implementation. As is standard (according to the Metaworld paper), we added a one-hot encoding of the task ID to the observation space to enable the algorithm to distinguish between different task instances. Additionally, we added a one-hot encoding of a "skill" ID to enable the algorithm to easily distinguish between different skills. We did this by writing from scratch a Gymnasium environment wrapper around the Metaworld library, referring at times to the Garage Metaworld wrapper[1].

The SAC agent was trained to solve all tasks and skills within each benchmark using a single policy, with the task identity provided as input.

We then did some hyperparameter tuning, finding that typically the hyperparameters given in the Metaworld paper for multi-task SAC were already well-tuned.

We found that many tasks were difficult to learn and getting good performance on the MT-10 benchmark took a decent amount of training time. So, in the interest of efficiency, we focused just on the ten skills contained in the MT-10 benchmark and ignored the other forty skills.

We first trained and evaluated this basic implementation against all ten skills individually, to establish a performance baseline for our SAC extensions.

B. Fine-tuning for Meta-RL

We implemented a fine-tuning strategy for meta-RL. This strategy involved training on the train set in an identical manner to the baseline. During test time, however, we took training steps in the model while we were interacting with the test environment. For each environment interaction, we added the interaction to the replay buffer and then ran one gradient descent step.[2]

Implementing this required going slightly deeper into the stable-baselines3 SAC API, so that instead of simply calling the `learn` method with appropriate hyperparameters, we had to manually update the replay buffer and initiate a gradient descent step. However, we still used public stable-baselines3 APIs to accomplish this.

[1] https://github.com/rlworkgroup/garage/blob/master/src/garage/envs/multi_env_wrapper.py

[2] Future work could include tuning the ratio of gradient descent steps to test environment interactions.

C. Including Prioritized Experience Replay

Our baseline is based on the default behavior of SAC in stable-baselines3, which uses a replay buffer with uniform random sampling. From there, we implemented prioritized experience replay (PER), a variant that samples important experiences preferentially.

TD Error-Based Sampling: In Prioritized Experience Replay (PER), the probability of sampling a transition i from the replay buffer is based on the magnitude of its temporal-difference (TD) error. The sampling probability is defined as:

$$P(i) = \frac{|\delta_i|^\alpha}{\sum_k |\delta_k|^\alpha}$$

where:

- δ_i is the TD error of the i-th transition, computed as the absolute difference between predicted and target Q-values.
- $\alpha \in [0, 1]$ controls the degree of prioritization. $\alpha = 0$ corresponds to uniform sampling.

This ensures transitions with higher TD error, which are likely to contain more learning signal, are sampled more frequently.

Bias Correction via Importance Sampling: Because prioritized replay introduces a bias in the gradient estimation (by over-sampling transitions with large TD errors), we apply importance sampling (IS) weights to correct for this:

$$w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta$$

where:

- w_i is the IS weight for transition i.
- N is the total number of transitions in the buffer.
- $P(i)$ is the sampling probability of transition i.
- $\beta \in [0, 1]$ controls how much IS is used; $\beta = 1$ fully compensates for non-uniform probabilities.

In practice, β is annealed from a small value (e.g., 0.4) towards 1 over the course of training. These weights are normalized and used to scale the TD loss during gradient updates.

Replay Buffer Implementation: We wrote our own implementation of `PrioritizedReplayBuffer` which extends stablebaseline3's `ReplayBuffer` and overrides the `add()` and `sample()` methods. Features include:

- TD errors are stored per transition.
- The `sample()` method returns indices and IS weights.
- After each training step, the TD errors are recomputed using updated critic networks and used to refresh priorities via the `update_priorities()` method.

We also needed to fork the stable-baselines3 SAC code, so we could extract the TD error for use in our replay buffer, and also so we could apply importance-sampling weights to gradient updates.

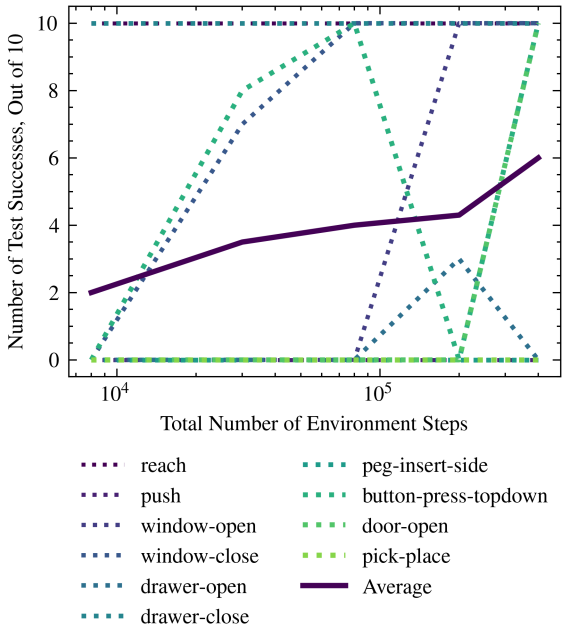

Fig. 4. Individual baseline: learning curves of skills under MT-1

V. Results

First, a quick note on the learning curves we've plotted. Many are noisy and don't include a lot of measurements. This was because we gave the tasks many environment steps (1000) to attempt to solve the problems during test-time, meaning that evaluating the models was fairly expensive. We also trained models separately rather than tracking the success of a single model as it was trained over time, an approach which I think we would change in the future for the multi-task benchmarks.

A. Baseline Results

1) MT-1: Here we show the test results of the ten MT-10 skills trained against their respective MT-1 benchmarks (that is, individually). For each skill, we trained 5 models separately, with differing numbers of training steps, ranging from 8,000 to 400,000. Each trained model was given tested against 10 task instances (each of which was already seen during training), and the number of successes is plotted in Fig. 4.

We found that even with 400,000 training steps, three skills (push, peg-insert-side, and pick-place) could not be successfully trained at all. We'll call these skills "difficult". Two skills (reach and drawer-close) were successfully trained in fewer than 8,000 steps. We'll call these skills "easy". The other five skills were at least partially trainable, or sometimes trainable, in-between. We'll call these skills (window-open, window-close, drawer-open, button-press-topdown, and door-open) "intermediate" in difficulty.

2) MT-10: When we trained the MT-10 skills jointly (that is, as part of the same model with one-hot encoding of the task IDs), we found that the learning happens less consistently but generally with much higher sample-efficiency. See Fig. 5.

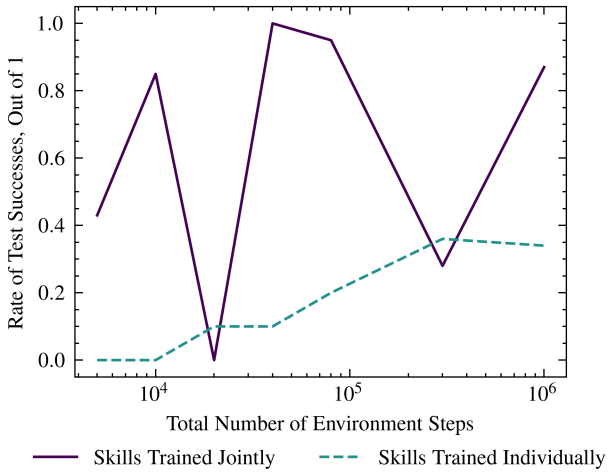

Fig. 5. Combined baseline: learning curves of skills trained individually vs. jointly

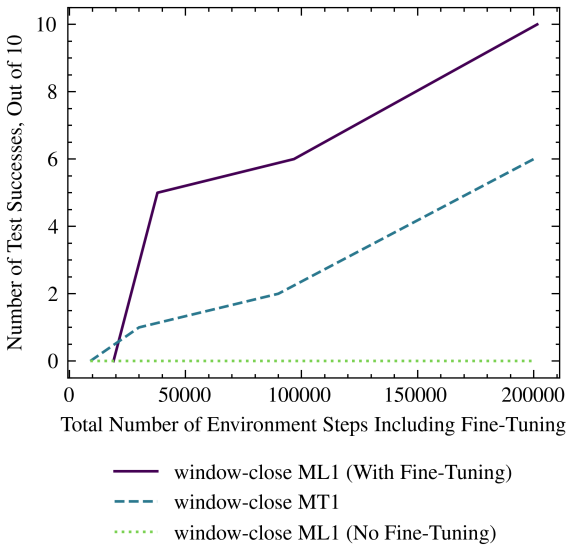

Fig. 6. Fine-tuning shows significant improvement in sample-efficiency.

This is consistent with the hypothesis that we would see some positive transfer from training all the different skills together.

B. Effectiveness of Online Fine-tuning

Our first experiment with online fine-tuning for meta-learning was on the ML1 benchmark. We compared this against the same benchmark without fine-tuning, and against the performance of the same algorithm without fine-tuning on the MT1 benchmark. Fig 6 shows the effectiveness of fine-tuning the model online against the meta-learning benchmark for the "window-close" skill.

Our initial experiment found that online fine-tuning performed dramatically better, requiring many fewer overall training samples than the (easier) MT1 benchmark for the same performance level. Without fine-tuning, a model trained with 200,000 environment steps could not solve a single task instance on the ML1 test set, since that particular starting configuration had not been seen during training. However, with online fine-tuning, a model pre-trained with approximately 38,000 total environment steps (30,000 of which were pre-training) could solve 5 out of 10 tasks in the test set.

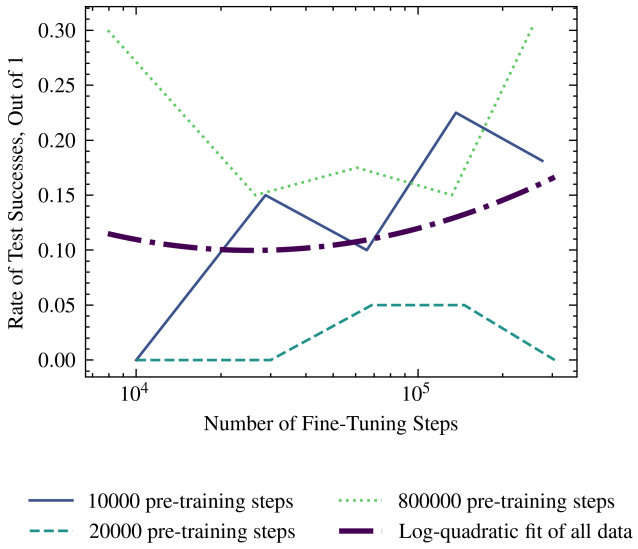

Fig. 7. Attempting meta-learning with new skills (on the ML10 benchmark) suggests a subtle forgetting effect.

C. Fine-tuning and Forgetting

We tested our fine-tuning meta-learning algorithm (with default random replay buffer) against the ML10 benchmark, over a grid of hyperparameters which varied the number of pre-training steps and the number of fine-tuning episodes.

Although we did not analyze the statistical significance or the stability of our findings across many runs, we found anecdotal data suggesting fine-tuning is susceptible to forgetting. We see in Fig. 7 that for curves with a large enough number of pre-training steps, test performance is first dips very slightly then recovers with more fine-tuning steps. This points to the fine-tuning "overwriting" some of the original learned sub-skills. [3] This seems a more probable explanation for a dip in test performance than negative transfer, since with minimal fine-tuning, the models with more pre-training performed better on the unseen tasks.

D. Adding PER

We ran the same fine-tuning experiments with prioritized experience replay enabled.[4]

[3]We'll see in the next section that another run against the benchmark doesn't show the same dip, but does show an overall convex learning curve. So, learning slows down before it gets better. This is still consistent with a "forgetting" effect, albeit a weaker one.

[4]We noticed soon before turning in the report that our PER implementation had some potentially significant implementation differences from the original PER paper. So, the results here are scoped specifically to our PER implementation, and do not necessarily reflect the behavior of canonical PER.

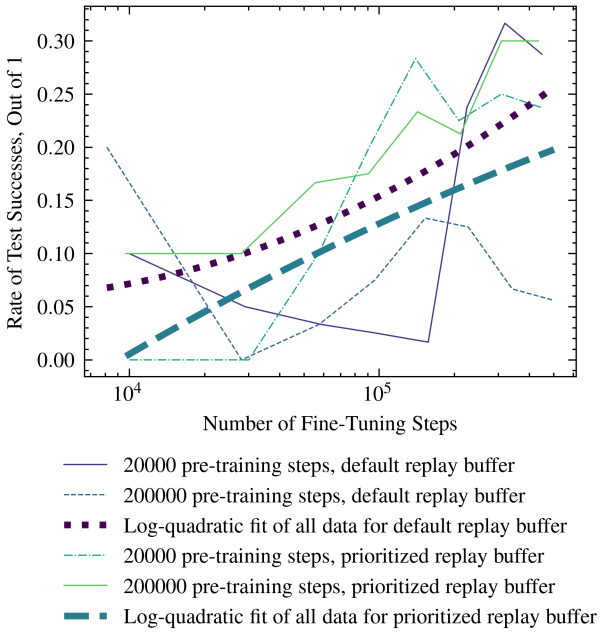

Fig. 8. Another trial on the ML10 dataset doesn't show forgetting, but does show a convex learning curve for the default uniform replay buffer. Adding PER appears to make the learning curve less convex, but at the cost of performance. Note here, that the log-quadratic fits account for underlying data that aren't shown on the graph.

With the same caveat about the data being anecdotal, adding PER does seem to mitigate the convexity of the learning curve during fine-tuning. See Fig. 8.

However, PER on the whole performs worse than the default replay buffer behavior, which remains unexplained.

E. Priority and Sampling Behaviour

Here, we show some results analyzing the behavior of the replay buffer and the distribution of TD errors of SAC against the Metaworld environment.

We plotted the behavior of priority replay along two axes:

1) how priorities are distributed in the buffer, and
2) how often individual transitions are actually replayed.

The plots were generated after running 100_000 training steps on the MT1 *peg-insert-side* task.

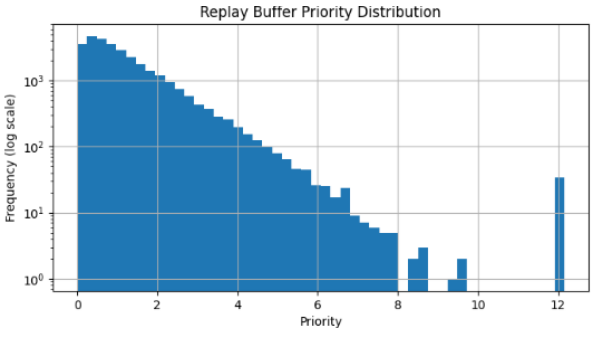

Fig. 9. Distribution of priorities in the PER buffer. A heavy peak at low values and a sparse tail of high-priority outliers are characteristic of prioritized replay.

1) Priority histogram: Fig. 9 shows a long-tailed distribution for PER: the vast majority of transitions accumulate at low TD-error values (0–2), while a thin tail extending to 8–12 highlights rare, high-error experiences.

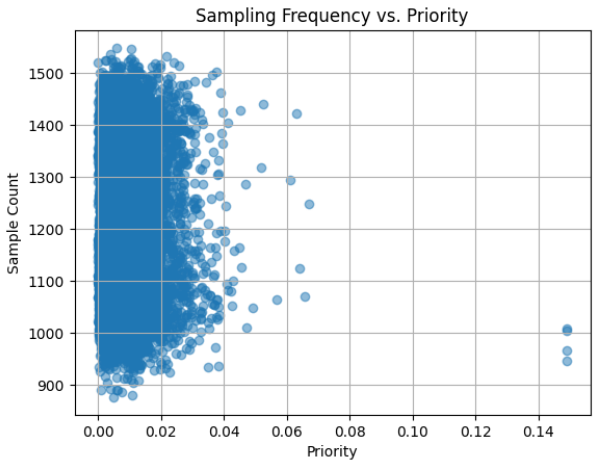

Fig. 10. PER: sampling frequency versus priority. Higher-priority transitions are replayed noticeably more often, confirming the intended bias of prioritized replay.

2) Priority vs. sampling frequency: Fig. 10 plots every stored transition as a single point, positioning it horizontally by its current priority and vertically by how many times that transition has been replayed. Although low priority ones are sampled more frequently but high priority ones i.e those who have large errors (0.14) are also sampled around 1000 times, which is better than what uniform sampling would sample.

VI. Conclusion

Indeed, jointly training different skills results in positive transfer and increased sample-efficiency in the Metaworld environment. We show that online learning of tasks in the Metaworld environment is extremely feasible. With caveats, online learning is improved by pre-training against other tasks before proceeding to fine-tune. We also show some anecdotal evidence suggesting that catastrophic forgetting is a problem in this fine-tuning paradigm with uniform replay sampling. Our implementation of PER was unable to achieve better performance than the default uniform replay buffer, but may have been able to mitigate the catastrophic forgetting effect to some extent.

VII. Contributions and Release

Chris wrote the online fine-tuning implementation and most of the plumbing to enable experimentation and hyperparameter tuning. Chris also ran most of the experiments and generated the learning curves.

Thanush wrote the original environment wrapper and got things running to begin with. Thanush also wrote the PER implementation and the custom SAC implementation, and performed the priority histogram analysis.

Both authors contributed to the report.

The authors grant permission for this report to be posted publicly.

REFERENCES

[1] L. Sun, H. Zhang, W. Xu, and M. Tomizuka, "Paco: Parameter-compositional multi-task reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 495–21 507, 2022.

[2] J. Beck *et al.*, "A survey of meta-reinforcement learning," 2024. arXiv: 2301.08028 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2301.08028.

[3] Z. Mandi, P. Abbeel, and S. James, "On the effectiveness of fine-tuning versus meta-reinforcement learning," 2023. arXiv: 2206.03271 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2206.03271.

[4] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, and G. Wayne, "Experience replay for continual learning," *Advances in neural information processing systems*, vol. 32, 2019.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2016. DOI: 10.48550/arXiv.1511.05952. [Online]. Available: https://arxiv.org/abs/1511.05952.

[6] T. Yu *et al.*, "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning," 2021. arXiv: 1910.10897 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1910.10897.

[7] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.