Extended Abstract

Motivation In many RL applications, collecting real-world data can be inherently costly due to reward-independent operating costs (e.g. robotics, human feedback) or real harm from low-reward trajectories (e.g. medicine, robotics). This project aims to see if deterministic control tasks over a continuous observation space can be solved with less real data by using a non-model-specific simulator, initialized solely from past data.

Method The simulator groups together "nearby" states, discretizing the state space. Acting in the discretized space entails identifying the closest action vector actually executed, then going to the discretized state into which that actions destination state was grouped. In the control case, we perform generic on-policy actor-critic training on the true environment. In the experiment case, we intersperse each rollout on the true environment with several rollouts in a simulator, still using the actor-critic algorithm.

Implementation All experimentation was done with the Hopper environment. Two different implementations of actor-critic with environment switching were ultimately made. The first involved bare-bones actor-critic being written from scratch. The second involved the preexisting `stable-baselines3` `A2C` implementation being modified (via the creation of a subclass) to allow it to store the full list of all past results (instead of just the most recent collected data), and to implement environment switching. This entailed redefining several of `A2C`'s methods to be essentially the same as before, but with additional lines added to implement data storage and environment switching.

Results and Discussion Unfortunately, both implementations of actor-critic failed to perform adequately on the Hopper environment, despite it seeming reasonable that they should. In particular, the basic `A2C` instance without the simulator failed to replicate the results claimed by `stable-baselines3` even with the hyperparameters taken from `rl_zoo3`. That being said, `A2C` did show clear signs of improving, albeit with many hours of training.

There was too much training noise during the tests using `A2C` to directly assess whether the simulator led to increased performance, in terms of gaining a better policy with the same number of true environment steps. However, the simulator did not cause clear failure, which is a positive sign. More experiments, averaged together, would no doubt be able to answer the question.

Curiously, one experiment with the simulator (Apparatus 1 Experiment 5) seems to have resulted in a sort of stable behavior cloning, in that the returns became flat. This appears to be due to the discretization being fine enough that many discretized states only had one observation (and thus one available action).

Conclusion The simulator shows promise, but due to the aforementioned noise, the Apparatus 2 experiments show no clear results, positive or negative. More experiments are required.

There were many features that were planned but unable to be implemented due to ongoing failure of the first apparatus and the time taken afterwards to implement the second apparatus. The result is that the simulator idea has a lot of still-untested potential.

Simulation-Based Policy Training for Costly-Data Scenarios

Ethan Bogle
Department of Computer Science
Stanford University
ebogle@stanford.edu

1 Introduction

The biggest obstacle to machine learning in most domains is currently data acquisition. Any machine learning policy will only be accurate at test time if it is trained on data that adequately reflects the distribution of data to which it will be applied. The minimum size of the training set will therefore of necessity grow with the size of the expected range of input.

The success of LLMs has made it clear that a sufficient quantity of reasonable-quality expert demonstration data is sufficient to achieve objectives in an RL environment. Unfortunately, in almost all RL applications the size of the space of trajectories is exponential in the size of the action space and the trajectory length, and the vast majority of RL applications have little or no prior expert data.

The reason why there is usually so little data in most RL environments is because there is some sort of cost associated to collecting that data. Most commonly, the cost is either:

1. a flat operating cost (e.g. operating a machine, paying humans to perform execution or evaluation), or
2. a suboptimality cost (e.g. avoidably misdiagnosing a patient, a robot injuring a human during interaction, a robot falling over and breaking).

Even when an environment is simulated on a computer, and has neither of these costs, collecting data using a policy is slow simply due to the sequential nature of stepping through a simulated environment. This has led to a proliferation of methods that attempt to use collected data more efficiently, which can generally be categorized into on-policy and off-policy methods. However, both have weaknesses: on-policy methods can be made more data-efficient through tricks, but still fundamentally cannot use data from policies that aren't similar to the current policy. On the other hand, off-policy methods have a harder time getting accurate value estimates and thus discovering optimal policies in environments with high sensitivity.

This project proposes an alternative approach that attempts to achieve the best of both worlds. The key pieces are:

1. Use an on-policy learning algorithm, to enable discovery of high-return policies in sensitive environments
2. Use all available data on the environment to build a simulator that approximates the true environment
3. Alternate training on the true environment and the simulated environment to save on training, updating the simulator with all new data from the true environment.

The use of the on-policy algorithm allows for more precise exploitation, while the use of the less-precise simulator allows the policy and critic to achieve a certain level of accuracy while reducing

utilization of the true environment. The simulator also allows some high-level uncertainty to be reflected in training, hopefully allowing both the policy and the critic to be more accurate in the long run.

2 Related Work

2.1 Actor-Critic Algorithms

The primary obstacle that all RL methods must overcome in low-data applications is that less data makes it harder to estimate the utility of an action. That is, it is very hard to estimate the function $Q^*(s, a)$, the best expected returns that can be acquired starting in state s and taking action a.

A common class of approaches, known as "actor-critic" approaches, forgoes estimating Q^* and instead opts to estimate $Q_{\pi_\theta}(s, a)$: Instead of estimating the value of an action under an optimal policy, we instead estimate the value of an action under the current policy π_θ (parameterized by the weights θ). We can also define (and estimate) the related function $V_{\pi_\theta}(s)$, the expected return from following policy π_θ starting in state s. If $Q_{\pi_\theta}(s, a)$ is greater than $V_\pi(s)$, that means that action a followed by policy π_θ has a better outcome on average than just following π_θ, and we should modify the policy weights θ to increase the log probability of action a in state s. Put another way, given a set of (state, action) tuples (s_j, a_j), we wish to maximize the quantity:

$$\sum_j \log \pi_\theta(a_j, s_j) \left(Q_{\pi_\theta}(s_j, a_j) - V_{\pi_\theta}(s_j)\right)$$

where if the coefficient on the right side is positive we want to increase the probability of action a and if it is negative we want to decrease the probability of action a. Thus, instead of trying to directly learn an optimal policy, we instead slowly improve the policy until we (theoretically) reach an optimal policy.

While there are several ways to estimate this difference expression (using the observed rewards), the inherent problem all of them share is that the policy π_θ which these functions estimate the value of is constantly changing with θ. In particular, they only remain accurate on data that was collected using the original policy. After even one gradient step, the policy π_θ is no longer the same policy whose value these functions are trying to estimate, and the estimates are therefore incorrect.

However, due to keeping the value functions (Q and V) constantly updated to π, on-policy methods can reliably continue to optimize provided that the estimates are accurate.

2.2 Uses of Off-Policy Data

The question remains how to continue making use of data as it becomes outdated *and* how to make use of preexisting data that has already been collected (and is therefore "free" from our standpoint).

There are three well-known uses for off-policy data: Taking multiple gradient steps per batch (e.g. PPO), direct imitation of expert data (behavior cloning), and direct off-policy adaptations (e.g. SAC).

Multiple Gradient Steps. Taking multiple gradient steps on the same collected data. This can give a multiplicative factor of data efficiency, but can cause problems because even if the parameters θ haven't changed very much, the overall policy π_θ can change drastically. Importance sampling and gradient clipping can help. The most well-known example of this type of improvement is PPO. Ultimately, however, we only increase utilization by a small factor, and problems can still arise even with PPO's safeguards.

Behavior Cloning: If it is possible to obtain data from an expert, then behavior cloning can sometimes be used, either initially, or periodically throughout training. However, if you have too little expert data, you don't get good coverage and your initial policy may overfit. In addition, performance may be overly sensitive to how you initialize the critic. Finally, if the expert data comes from multiple experts who use different strategies, mixing the two datasets may actually result in a worse starting policy and critic.

Off-policy adaptation: Algorithms like actor-critic can be adapted to use a replay buffer over all prior data, updating on randomly sampled batches, in such a way that the value estimates remain approximately correct to the actual current policy without overfitting. An example of this is "Soft

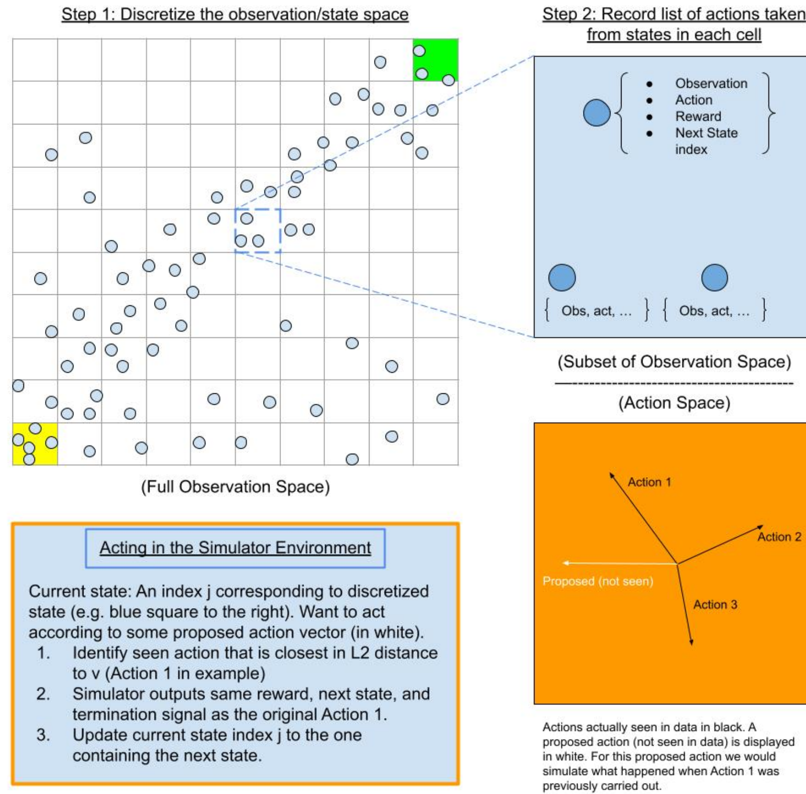

Figure 1: Simulator Illustration.

Actor-Critic" (SAC). However, this approach still has drawbacks: The estimates have substantially higher variance (meaning that training is less stable unless you have a lot of data), and it can be tricky to tune hyperparameters (as detailed in lecture 5).

2.3 Simulation

Simulation is an extremely popular method of solving some RL problems. Its most famous use is no doubt in game-playing programs (e.g. AlphaGo). However, most applications of simulation focus on planning (that is, going forward in the computation tree to get a more precise estimate of future value) or model-specific approximate simulation (such as using a virtual environment to train a dog-robot program before deploying it in the real world). My focus differs in that I am trying to do model-free simulation (building the simulator from the data) to *reduce* the amount of (real-world) data collected rather than using simulation to directly enhance prediction or to fully substitute for real-environment data.

3 Method

For this project, I implemented a simulator apparatus, as well as two different actor-critic apparatuses to test it with, one based on the homeworks, and one based on `stable-baselines3` (Raffin et al. (2021)). All experiments were done using the Hopper environment (Brockman et al. (2016); Towers et al. (2024)). Unfortunately, neither of the actor-critic apparatuses were effective at learning the Hopper environment, even without the simulator. The second apparatus was built after the first one failed.

3.1 Simulator Apparatus

The simulator is built around the idea of *mental rehearsal*. Mental rehearsal is a human learning tactic: When about to face a high-stakes challenge, we may rehearse our actions in our mind in anticipation of the event, priming ourselves to act appropriately in the moment without having to think too much. For example, you may mentally rehearse a series of actions that you are about to take, or you may mentally rehearse ways to divert a conversation from a topic you don't want to talk about if you know someone might ask about it.

In particular, mental rehearsal is different from planning: planning is systematic. Mental rehearsal instead attempts to predict the outcome of your existing mental policy in a mental simulation of the world, and may possibly modify your mental policy if needed. In planning, you prime yourself to answer a specific circumstance with a certain action, usually with a good deal of knowledge of the situation. In mental rehearsal, you update your general mental policy, in a way that is not overly specific to the immediate circumstances you are facing, because you are relying on an imperfect model of the world.

This simulator mimics mental rehearsal: It builds an imperfect model of the world from all available data, on which the policy can be trained in a similar manner to how it trains on the true environment. The simulator's world model is updated every time new data is collected. Ultimately, the trained policy is used to act in evaluation cases exactly as it would without the simulator (that is, the simulator is not used for planning).

The simulator was designed for deterministic environments with continuous observation spaces and continuous or discrete action spaces, and uses a gym-style API. (A version for nondeterministic environments is relegated to future works.)

The operation of the simulator is detailed in Figure 1. In Step 1, all of the available (s, a, r, s') pairs are assembled. Then the observation space is discretised: The observation states are grouped into clusters of similar states. Each cluster is associated with a state id j that functionally acts as the simulator's internal state. The principle is that similar states, under the same action, should lead to similar output states.

In Step 2, we list out, for each cluster of observations, all of the observed actions from states in that cluster, along with the corresponding rewards, and next states. We identify the state ids j' corresponding to the next states.

Finally, to act in the environment, we take our current state cluster id j, the proposed action to be taken v, and identify among the stored actions the action which is closest (in L_2 distance) to v. The reward and next state are then the stored values corresponding to that action when it was originally taken in the dataset.

The simulator's advantages are:

- It is model-independent: It only requires some mechanism for discretizing the states, and otherwise does not care what those states represent.
- It is able to make full and continuous use of all available data, regardless of what policy was used to capture that previous data, since more data will always improve its accuracy in simulating the dynamics and reward model of the true environment.

3.2 Actor-Critic Apparatus Attempt 1 (based on homework)

This was the first of the two actor-critic apparatuses built to test the simulator, and its results were used for both the milestone and the poster. It was ultimately discontinued because the resultant actor-critic algorithm could not be made to function well on Hopper, even before involving the simulator.

The basic setup of the Hopper environment, the shape of the policy network for Hopper, and an initial set of expert rollouts were pulled directly from Homework 1. Some components of basic actor-critic were pulled from Homework 2, but some of those components (e.g. multiple critics) were left out in order to keep the setup as simple as possible, since logging and plotting graphs also had to be written by hand (this apparatus did not use `tensorboard`).

The intent was to build this apparatus mostly from scratch, using preexisting code as a guideline, to reduce the amount of debugging required, and to examine a simple case without trying to augment actor-critic in any way.

The apparatus initializes the policy network using behavior cloning on the expert rollouts, and initializes the critic network (same shape as mean network from policy network) by matching the discounted sum of future rewards from the expert rollouts. It then performs basic actor-critic training, with the critic predicting the value function. The critic was updated using TD0 (bootstrapping). The actor advantage was computed as

$$A(s, a) = r(s, a) + \gamma V(s') - V(s)$$

Due to the finite horizon of the hopper task, $\gamma = 1$ was used.

Due to the memoryless nature of this algorithm, switching between the true environment and the simulated environment is trivial.

Unfortunately, this algorithm failed to perform well on Hopper without the simulator, even after numerous hyperparameter explorations. The hyperparameters explored included: Total number of iterations, the learning rates for the critic and the policy, the number of gradient steps taken on the critic and policy per batch. For consistency, all later experiments were done from the same starting policy and critic models from the initial behavior cloning step.

Hyperparameters for the simulator that were played with (though ultimately not significant), were: How many true and simulator environment batches, respectively, were done before switching environments, and the granularity of the observation discretization.

The observation discretization was set by hand after looking at a histogram of the observation distributions for each dimension in the provided expert rollouts.

3.3 Actor-Critic Apparatus Attempt 2 (based on `stable-baselines3`)

After the first actor-critic apparatus proved unable to perform well on Hopper, I determined after consulting with TAs in office hours that Hopper was a difficult environment for actor-critic, and that the various augmentations of actor-critic would likely be necessary to succeed. I therefore elected to attempt to use the fully-online actor-critic algorithm available from `stable-baselines3`, `A2C`. All of the work on this apparatus occurred after the poster session.

My first task was to determine the appropriate hyperparameters to use for training on Hopper. An attempt to reproduce the listed Hopper-v5 using `rl_zoo3` (Raffin (2020)) failed do to unresolvable installation conflicts. I then attempted to directly extract the hyperparameters that were used by inspecting the `rl_zoo3` code and running `A2C` with those parameters. However, I was unable to reproduce reasonable Hopper performance. Because it was not clear if I had actually found all of the needed hyperparameters for Hopper, I continued and implemented environment switching for `A2C`.

Where the implementation of apparatus 1 was straightforward but time-consuming since it was being programed from scratch, the implementation of apparatus 2 was quicker (though only because of my prior experience with apparatus 1), but much more convoluted. I had originally elected to code from scratch precisely because I anticipated that trying to implement mid-train environment switching would require extensive reading into the code. I ultimately implemented environment switching by defining a subclass of `A2C` which redefined the `collect_rollouts()`, `dump_logs()`, and `learn()` and had to write a new buffer class based on the `RolloutBuffer` class that would be able to store all collected data, and specifically the data that I need to build the simulator each time the simulator is initialized, instead of just the most recent rollout. Most of the redefinitions involved copying the original code and adding lines.

It is quite possible there are major bugs in my code, which may throw off any experimental results.

This apparatus did not use behavior cloning. The hyperparameters that were actively varied in experiments included: The total number of timesteps, the number of timesteps on the true environment and simulated environment, respectively, before switching, and the level of discretization of the observation space.

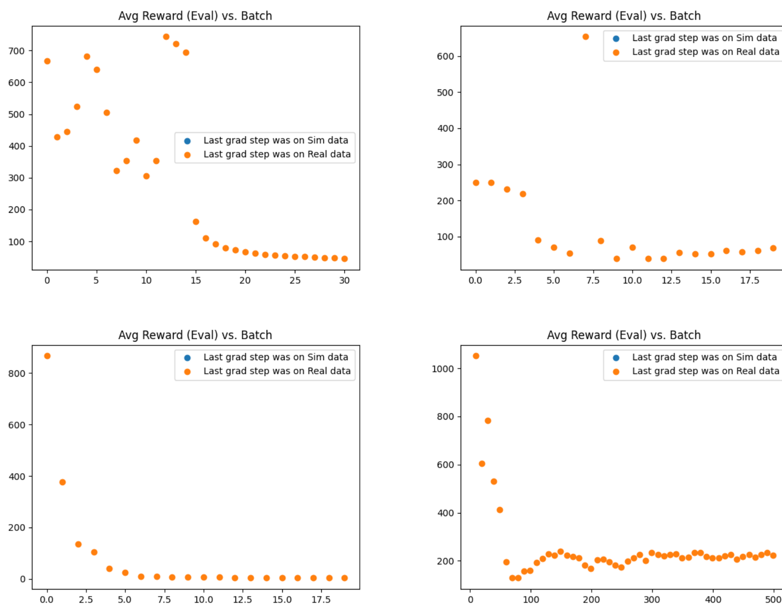

Figure 2: Eval Average Reward, plotted vs. number of batches, for each of experiments 1, 2, 3, 4 (left to right, top to bottom). Note that the x- and y-axes are not aligned between plots.

4 Experiments, Results, and Discussion

4.1 Apparatus 1

Here is an ablation of the different hyperparameters used for basic actor-critic without the simulator. Actor LR and Critic LR are the learning rates of the actor and critic, respectively. Actor Grad Steps and Critic Grad Steps are the number of gradient steps taken on the actor and critic, respectively, per batch. In all four instances, the minimum batch size was 1000. The plots of average eval return are plotted in Figure 2

	Actor LR	**Critic LR**	**Actor Grad Steps**	**Critic Grad Steps**
Exp. 1	5e-4	5e-3	5	5
Exp. 2	5e-3	5e-3	5	1
Exp. 3	1e-3	5e-3	5	3
Exp. 4	1e-4	5e-3	5	3

Table 1: Training Hyperparameters for Apparatus 1 Actor-Critic (no simulator)

All of the experiments decay to terrible performance, though some decay to lower plateaus than others.

In experiments 5 and 6, 10 batches of the true environment are alternated with 10 batches of the simulator. All non-simulator hyperparameters are the same as Experiment 4 above. The only differnece between experiments 5 and 6 is that the width of the discretization window was doubled in experiment 6 compared to experiment 5. The graphs for Experiments 5 and 6 can be found in Figure 3.

It appears that the tighter discretization in Experiment 5 may have had the effect of locking in the performance at a higher level without allowing it to decay. This can occur if the majority of discretized states visited have only one action to choose.[1] The final performance of Experiment 6 is also better than that of Experiment 4, but the reasons are not clear.

[1]The simulator's code prints the percentages of the discretized observations with at least n actions for $n = 1, 2, 3, 4$ each time that data is loaded, but I neglected to make graphs of these.

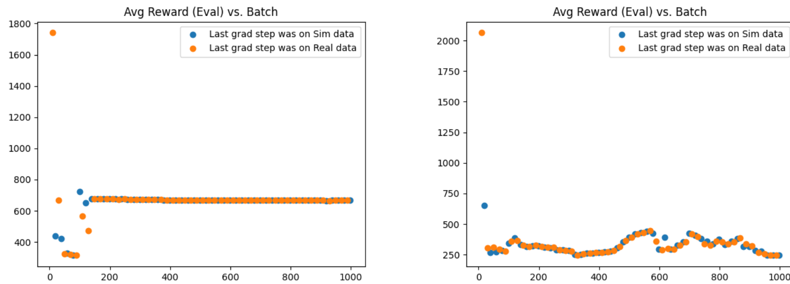

Figure 3: Eval Average Reward, plotted vs. number of batches, for experiments 5 and 6. Note that the y-axes are not aligned between plots. Compare to the graph for experiment 4 in Figure 2.

4.2 Apparatus 2

The primary hyperparameter that was varied for apparatus 2 was the relative amount of true environment steps and simulator steps. True Env Duration represents the number of timesteps spent in the true environment before switching to the simulator, and Simulator Duration is analogous. (Note that all experiments were done with 4 copies of Hopper running in parallel, so a single timestep corresponds to four unique observations). Experiments 1 and 2 did not use the simulator. Experiments 1 and 2 are the same, and Experiments 4 and 5 are the same, except for the length of time they were allowed to run.

To accommodate for the fact that we were not starting with behavioral cloning data as part of the prior data (due to lack of time to implement it), all experiments were run with a discretization of 4 times that used in Apparatus 1, Experiment 6.

The graphs of eval mean rewards for the trials are found in Figure 4.

	True Env Duration	**Simulator Duration**
Exp. 1	N/A	N/A
Exp. 2	N/A	N/A
Exp. 3	50000	10000
Exp. 4	50000	50000
Exp. 5	50000	50000

Table 2: Training Hyperparameters for Apparatus 2 Actor-Critic

Based on experiments 1 and 2, it appears that `A2C` is in fact capable of learning Hopper, although it seems like it will take many more timesteps than were available (experiment 2 took between 3 and 4 hours to run).

When comparing runs 1 and 2 and runs 4 and 5, it is clear there is a lot of noise in the training process, so drawing concrete conclusions about the simulator's effect is difficult. In particular, Experiment 5 does better than experiment 4 in the timestep range that they both overlap. A comarison of experiments 1 and 3 (which both have the same number of total true environment steps) seems to indicate that using the simulator less frequently (10,000 steps to 50,000 real env steps) doesn't seem to help.

At the very least, the simulator does not seem to be causing critical drops in performance, which is a good sign.

5 Conclusion and Future Work

The simulator shows some promise, but there is ultimately too much noise in the training process to be sure at this point. Ultimately, more experiments are required.

There were a number of planned experiments and features that I was unable to get to due to the troubles with getting actor-critic to work on Hopper. Among the most relevant are:

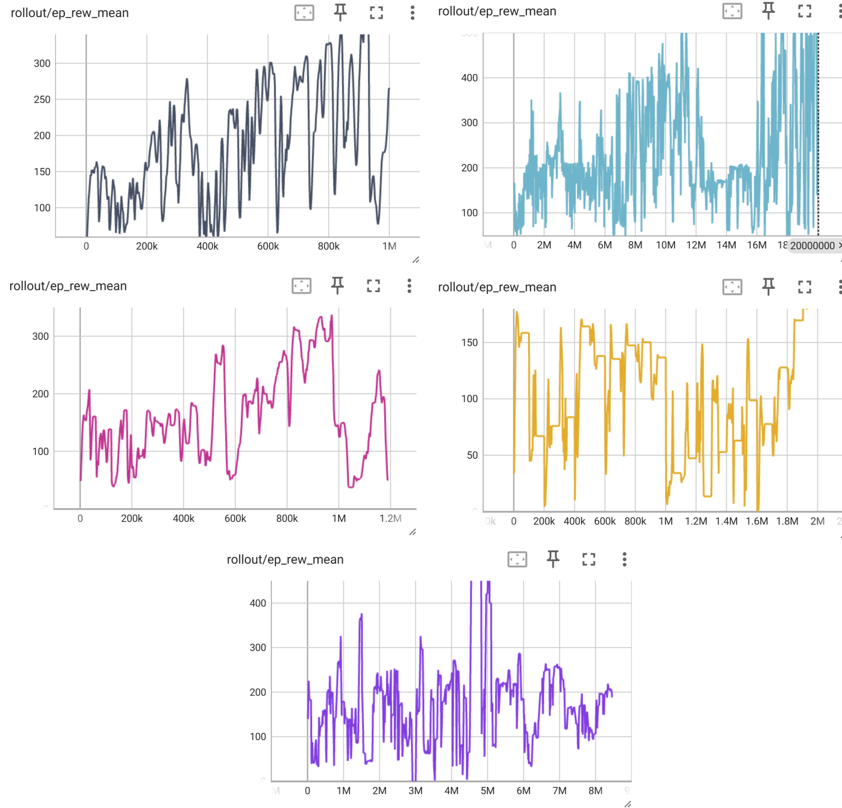

Figure 4: Eval Average Reward, plotted vs. number of training timesteps, for Apparatus 2 Experiments 1 through 5. Note that the x-axis includes both simulated and real timesteps, and comparisons should be made with this in mind.

- The concrete metrics that we eventually want to measure in this setup is how good of a policy can be achieved under a fixed operating cost. This requires ablating over different learning schedules, and the answer is also likely to be different for different budgets, but that was the intended end goal of this project.
- Currently, the way discretization is performed is static throughout training, even though it might presumably be helpful to make discretization more granular as more data is acquired. Ideally, it would be desireable to make discretization an automatic process that can ensure an appropriate tradeoff between granularity and the number of available actions at each state cluster, to make the choice of action always meaningful.
- Current discretization also relies on L_2 distance in the action space without considering that the scale of different dimensions may be different (the same would apply to the observations space for automatic discretization). A way to measure relative density or sensitivity in each action and observation dimension would potentially improve performance.
- Currently, we are only able to alternate between a constant number of true environment steps/rollouts and a constant number of simulated environment steps/rollouts. A system that allowed more dynamic control over when to use the simulator vs. when to use the true environment could be useful in practice (e.g. training on the simulator until performance starts to peak, and only then returning to the true environment). Even without this level of intelligence, there may be advantages to using the simulator at specific points in training.
- Can we modify the exploration/exploitation tradeoff? Would exploration in the simulated environment (especially if past data is available) help performance in the long run?
- A direct comparison with other off-policy methods would also be essential.

- The simulator is built to function in environments with discrete action spaces, but this was never showcased. Interestingly, this adds randomness in the simulator that is not present in the original environment.
- And finally, of course, an adaptation of the simulator to a non-deterministic environment.

Changes from Proposal As described in the body of the paper, the failure of the baseline model to work from Apparatus 1 required the building of an entirely new Apparatus for testing, meaning that twice as much time was spent writing setup code for this assignment as would have been expected. Sadly, the scope of the results obtained is, as a consequence, substantially narrower than I had ever thought it would be.

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540

Antonin Raffin. 2020. RL Baselines3 Zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* 22, 268 (2021), 1–8. `http://jmlr.org/papers/v22/20-1364.html`

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. 2024. Gymnasium: A Standard Interface for Reinforcement Learning Environments. arXiv:2407.17032 [cs.LG] `https://arxiv.org/abs/2407.17032`