Investigating the Utility of Off-Policy Data in PPO Algorithm

Yufeng Yuan & Dhawal Gupta

***Abstract*— This projects is to investigate some subtle modification on the PPO algorithm to increase the sample efficiency for real time physical robots. This project will specifically focus on the UR5 robot, where we test our algorithm on simulation as well the real robot and report their performances.**

I. INTRODUCTION

Recent advances in the class of policy gradients algorithm have shown promising results in the domain of continuous control. They have reinvigorated an interest in these methods over the traditional value-based approaches.

TRPO [6] and it's more straightforward and lighter version PPO [7] have shown really promising results on the simulated control environments which encompass robotics as well as games environments. Some example of these environments [1] include `Acrobot-v1` and `Pendulum-v0` in classical control, Mujoco Environments like `Reacher-v2`, and robotics environment based on Mujoco such as `FetchReach-v1`. These environments range from simple 1-dimensional control to sophisticated 4-dimensional control, and PPO can solve all these environments to a satisfactory level.

Both PPO and TRPO are on-policy algorithms and require vast amount of simulation runs to learn and achieve excellent performance. Effectively they use the collected batch of data only once to update their policy and then discard it. This approach though is suitable for simulated environments, immediately starts to look ineffective and unpractical for real-time physical systems where data is collected in real-time. For comparison, a batch collection in simulation (without parallelization) can be done in under 5 seconds, whereas the same batch collection on the UR5 robot arm takes well over 80 seconds. [3] this paper presents a systematic approach of learning on a real-time physical system with PPO and TRPO. They report that learning a 3 dimensional reaching task takes around 3 hours of wall time to learn. Which when compared to simulation is around 9 times and also cannot be parallelized.

In this project, we try to explore some off policy approaches where we wish to reuse the collected off policy data and investigate whether reusing this data can help improve the sample efficiency of the PPO algorithm.

The report will be structured as follows, Section II will present the hypothesis that we wish to investigate in the report. Section III introduces techniques we used in our modification to the original PPO algorithm. Section IV explains our modification on PPO and gives the algorithm that we followed. [4] focuses on the importance of specifying implementation details for repeatably, hence Section V gives a detailed summarization of the implementation details and specifies our experimental design both for simulation and for the real robot. Section VI presents the results for our modification. Section VII presents the conclusions for our findings and the future work that can be incorporated in the same.

II. HYPOTHESIS

We propose the following hypothesis and wish to test the same : Storing data in replay buffer and modifying the priority with importance sampling ratios, will allow to use and reuse data more efficiently eventually bringing the training time down substantially in context to the current performance of the algorithm.

III. BACKGROUND

A. Proximal Policy Optimization (PPO)

The base for our implementation is the Proximal Policy Optimization Algorithm [7], which is based upon essentially a 9 step modification on REINFRORCE [8]. We will be listing the 9 steps over here in detail. Appendix contains some additional steps that have been adapted from the baselines implementation of PPO and are essentially tricks that help in stabilise the learning. Appendix also presents a scenario where this methods fails and the extra modification helps the model to learn.

The REINFORCE update can be written as $\theta_{k+1} = \theta_k + \alpha_k \sum_{t=0}^{T_k-1} \gamma^t \nabla log \pi_{\theta_k}(A_t^k|S_t^k) G_t^k$. Where k indexes the episode that we update **9 Steps**

1) Drop γ^t from G_t.
2) Batch Updates : Modify the updates from begin episodic to collecting batch of multiples episodes and then update your policy.
3) Divide the batch into mini batch updates with shuffling.
4) Multiple Epoch over the batch : Do multiple epochs of update on the batch.
5) Define/Use a surrogate objective : Objective for deriving policy gradients are often written as : $E[\nabla log \pi_\theta(A_t|s) q_{\pi_\theta}(s, A_t)]$. As samples become off policy right after the first update the update devolves into $E_{\pi_{old}}[\nabla \pi_\theta(A_t|s) \pi_{\theta old}(A_t|s) q_{\pi_\theta}(s, A_t)]$. Hence the surrogate function that we use is as follows $\pi_\theta(A_t^k|S_t^k \pi_{\theta old}(A_t^k|S_t^k G_t^k$.
6) Add Baseline : Baseline should be a term that can be added or negated from the G_t which is not dependent on the action. This produces our output Advantage.
7) Use λ Return
8) Normalize Advantage by the batches mean and variance.
9) Use proximity constraint on θ_{old}.

B. Importance Resampling

To reuse the transitions from previous policies, we can employ a couple of approaches, first approach known as importance ratio correction corrects a transition sampled from the buffer by multiplying the update with the ratio of probability of sampling that transition / trajectory under the current policy. This often ends up being a very high variance update for long trajectories and can prevent the agent from learning. The second method that we use i.e. **Importance Resampling** uses a replay buffer, the replay buffer will be used to store previous transitions. To ensure the transitions sampled from the replay buffer are not too different from current policy, a technique called importance resampling [5] will be used: Consider a setting where a buffer of data is stored, generated by a behavior policy. Samples for policy π can be obtained by resampling from this buffer, proportionally to $\pi(a|s)/\mu(a|s)$ for state-action pairs (s, a) in the buffer. In this way, the sampled transitions will follow current policy π. Shiftinf the importance ratio to sampling can eessentially allow us to avoid very high variance updates.

C. Value Correction

As importance resampling only corrects actions instead of trajectories, to correct for the λ-return from the replay buffer, we used two techniques: V-trace [2] and λ-decay

1) V-trace: Consider a trajectory $(x_t, a_t, r_t)_{t=s}^{t=s+n}$ generated by the actor following some policy μ. We define the n-steps V-trace target for $V(x_s)$, our value approximation at state x_s, as:

$$v_s = V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \Big(\prod_{i=s}^{t-1} c_i \Big) \delta_t V,$$

where $\delta_t V \rho_t \big(r_t + \gamma V(x_{t+1}) - V(x_t)\big)$ is a temporal difference for V, and $\rho_t \min\left(\bar{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)}\right)$ and $c_i \min\left(\bar{c}, \frac{\pi(a_i|x_i)}{\mu(a_i|x_i)}\right)$ are truncated importance sampling (IS) weights (we make use of the notation $\prod_{i=s}^{t-1} c_i = 1$ for $s = t$). In addition we assume that the truncation levels are such that $\bar{\rho} \geq \bar{c}$.

The above formula can be rewritten as the on-policy n-steps Bellman target. Thus in the on-policy case, V-trace reduces to the on-policy n-steps Bellman update. This property allows one to use the same algorithm for off- and on-policy data. Details of the above proofs and steps can be found in [2].

2) λ-decay: For the actor part we use λ returns to correct our value estimates where λ is often closer to the value of 1. Now as the batches collected get older w.r.t. the current policy there value estimates become more and more off policy and importance re-sampling transitions with these importance sampled corrected trajectories can be very biased and high variance. So to counter this we recompute all these returns with a modified λ' where the λ' for older batches have a decreasing regime. This shrinks the value of λ for older batches effectively making them more shortsighted with respected the rewards, this reduces the variance due to long trajectory and we use our latest value function to evaluate these λ returns so as to reduce the bias and variance. We will follow the exponential decay regime for λ over here.

Figure 1 shows the return calculation for a given value of λ, and in Figure 2 we show the value of λ for different sections of the replay buffer.

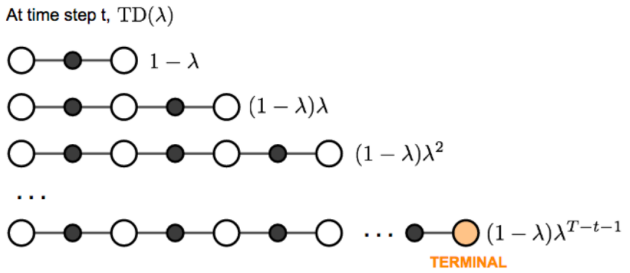

Fig. 1. Calculation of the λ return for a given time step for a given value of λ

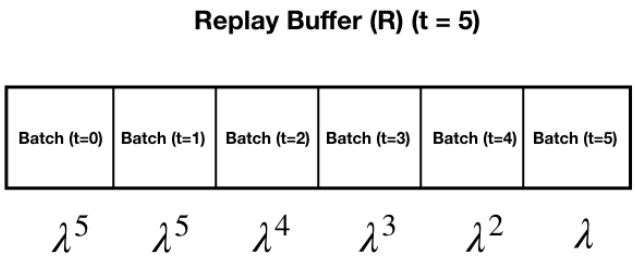

Fig. 2. Exponential decay regime for λ. So for a batch at time $t - i$, we will calculate the new returns with λ^{i+1} this will results in the values being more myopic for near returns reducing variance.

IV. ALGORITHM

Our modification of the PPO algorithm is based on the original algorithms and we separate the policy and value update into two stages: on-policy and off-policy. The on-policy stage remains the same as in the original PPO algorithm and the off-policy stage is placed after the on-policy stage. After on-policy update, the batch of transition will be put into the replay buffer for off-policy update. In off-policy stage, value function and policy function will be trained separately. For value function, transitions will be sampled with importance resampling with the importance ratios of their following trajectories while for policy function, transition are sampled with importance resampling based the importance ratios of that state-action pair. Here, the target of the value function is the v-trace corrected target.

Result: Updated policy and value
Initialize : $\theta_{old} \leftarrow \theta$;
Initialize : H_t, G_t, π_{old};
scramble the batch;
for *m minibatches* **do**
  draw minibatch ;
  optimize.step() ;
end

Algorithm 1: Update()

Result: Learned policy $\pi(a|s,\theta)$ and value function $\hat{v}(s,\mathbf{w})$
Initialize : $R \leftarrow \{\}$;
while *True* **do**
  $B \leftarrow$ a batch of transitions (s_t, a_t, r_t, s_{t+1}) collected;
  for *e epochs* **do**
    PPO Update with B
  end
  $R \leftarrow R \cup B$;
  Update ρ for B with $\pi(a|s,\theta)$;
  Correct λ-return for B with $\hat{v}(s,\mathbf{w})$;
  for *t epochs* **do**
    Sample B_{off} from R based on the trajectory importance sampling ratios;
    PPO value update with B_{off};
    Sample B_{off} from R based on the state-action pairs importance sampling ratios;
    PPO policy Update with B_{off};
  end
end

Algorithm 2: PPO with Replay

V. EXPERIMENT

This section will introduce both the simulated and real environments that we'll be testing our algorithms on as well as our methodology to compare the performance.

A. Environment setup of the Mujoco task

Although this report focuses around implementing the algorithms for real robots, it is really important to debug our code so as to not waste precious real robotics runs, hence we initially debug our code on the Robotic Platform that is provided by the OpenAI Gym. We will be using `FetchReach-v1` [1] environment as it closely resembles the task that we want to achieve with our real robot. Here we provide the description of the simulated environment for the reference of the user. The observation space of the original robot is broken down into 3 subcategories, i.e. `observation` (Size : 11), `achieved_goal` (Size : 3) & `desired_goal` (Size : 3). Where we need to have the desired goal and the current robot configuration to actually learn in the environment, we can just compose our state space of `observation` and `desired_goal` which makes our observation effectively to be 13 Dimensional. The action space for the robot is 4 dimensional. Reading from the code it seems as if the control type is specified for torque , where the first 3 sets of action controls the 3 joints and the 4th action corresponds to the gripper. See Figure 3 for the simulated environment setup.

B. Environment setup of the UR5-2D task

Now we move onto experimenting our algorithm with physical hardware on the UR5 Robot. The task we will be training for is UR5 2D. The environment has a state space which contains joint angles, the joint velocities, and the vector difference between the target and the fingertip coordinates. The tasks involved controlling 2 joints which allow the end effector of the robot to span a 2 dimensional space. The actions can be set to control the speed, torque and position of the joints. The tasks involved resetting the hand to a default starting position and randomly generates a point that the robot has to reach. More details about the reward and environment can be found in the Senseact Library [4]. See Figure 4

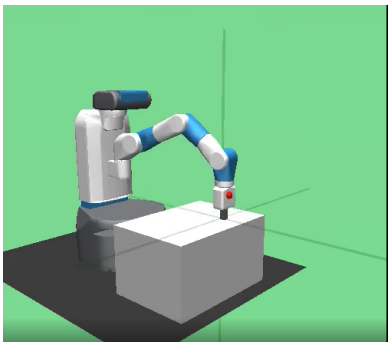

Fig. 3. Simulated Setup for the FetchReach-v1 environment

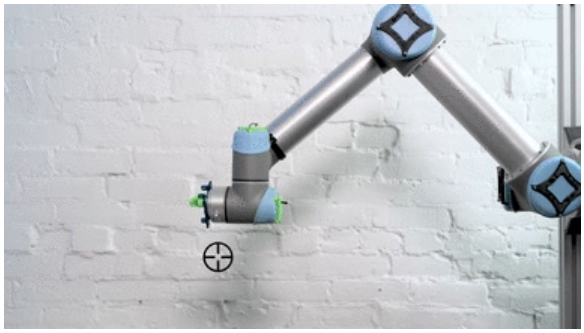

Fig. 4. The Physical setup of the robot.

C. Experiment methodology

- **Multiple runs** Every algorithms will be trained for multiple runs with random initialization to reduce the variance in performance introduced by different random seed. **Details of runs** : We tried to do multiple numbers of the runs overnight unassisted by a human by making a bash script. But what we saw was that the robot often stopped working after 1-2 runs which we assume was because of improper restart. To remedy that we used to force kill the complete process after doing one run, and then run the next experiment.
- **Average return of multiple episodes** As the return of a single episode still has high variance, the performance will be compared in terms of the average return of 100 episode.
- **Mean and standard deviation of the performance** For the information collected above, the mean, standard deviation will be computed to plot, and this will give

[1] https://gym.openai.com/envs/FetchReach-v1/

us a more sense of the expected performance and the variability of performance.

Appendix mentions the Environment specification to run the UR5 robot.

D. Experiment objective and implementation

As an obvious and straightforward way to use samples more efficiently in on-policy methods is using more optimization epochs, we will compare our algorithms with original PPO with default number of optimization epochs and larger number of optimization epochs. Hopefully, with the same amount of experiences observed, our algorithms will achieve better eventual performance as it's capable to use the off-policy data.

Our implementation is based on OpenAI Baselines[2] and our code are open-sourced on Github [link] [3] .

VI. RESULTS & ANALYSIS

For the simulated task, learning curve of each algorithms is the average of 5 runs with random initialization with the shaded area showing the standard deviation. For the UR5 task, due to the long training time, learning curves are average of 3 runs with random initialization with shaded area as the standard deviation.

A. Results on simulation tasks

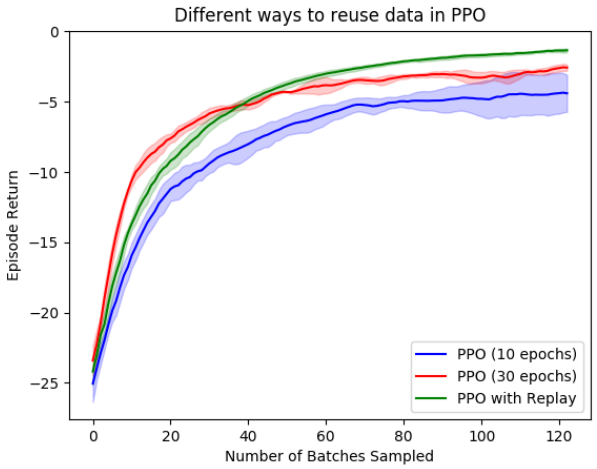

Fig. 5. Comparison of proposed algorithms with baseline PPO with different optimization epochs

Plot showing above are original PPO with 10 on-policy optimization epochs, original PPO with 30 on-policy optimization epochs and our proposed PPO with 30 on-policy optimization epochs and 60 off-policy optimization epochs. It should be mention here, we also tested original PPO with 50 on-policy optimization epochs, but it turned out to diverge in early stage of training which we didn't show here.

From the result above, it's obvious to see that increasing the on-policy optimization epochs has a significant influence on sample efficiency as the PPO with 30 epochs achieve better and more stable performance across the training phase. For our algorithms, though the initial learning speed is between PPO with 10 epochs and PPO with 30 epochs but it soon surpassed the other two algorithms and achieved the best performance eventually. It should also be noted here that the standard deviation of our algorithms is significantly smaller than original PPO, which potentially implies that using previous transitions in a proper way might stabilize performance.

[2]https://github.com/openai/baselines

[3]https://github.com/YufengYuan/PPO_Project

B. Hyper-parameters for the Mujoco task

Hyperparameter	Value
Learning Rate	3e-4
Value Loss Coefficient	0.5
Entropy Coefficient	0.0
Batch Size	1600
Mini Batch Size	40
On-policy Epochs	10 & 30
Off-policy Epochs	30
Replay Buffer Size	8000
Clip Range	.2
Gradient Norm Clip	.5
Optimizer	Adam
Gamma	0.99
Lambda	0.95

TABLE I

HYPER-PARAMETERS FOR PPO ALGORITHMS ON THE MUJOCO TASK

C. Results on UR5 tasks

In the UR5 task, we compared the original PPO with 10 on-policy optimization epochs, original PPO with 30 on-policy optimization epochs and our proposed PPO with 20 on-policy optimization epochs and 40 off-policy optimization epochs. The total time steps for all three algorithms are all 150000 steps.

Figure 6 shows the comparison of the baselines PPO alongside the modified PPO with replay. As it is clearly evident from the figure that PPO modified with replay learns much faster than the normal PPO and reaches peak performance in 20 steps (around 40K transitions) as compared to PPO which takes around 70 (around 140-150K transitions) steps to reach the same performance. Even PPO with 30 epochs performs significantly better than just 10 epochs showing us that the number of epochs is a really important hyper-parameter. Now we can see that there is sudden drop in the performance at around 30-40 update step which can be attributed to forgetting in Neural Networks due to big updates. This can be more clearly seen in Figure 7, which shows all the different runs for the robot. Run 2 seemed to have a huge dip in the performance but also cannot be ruled out as an outlier because the number of runs is not that significant. But there is some amount of degradation in the learning for all the runs. But the results are really encouraging and this bolsters our hypothesis about the utility of using replay in PPO.

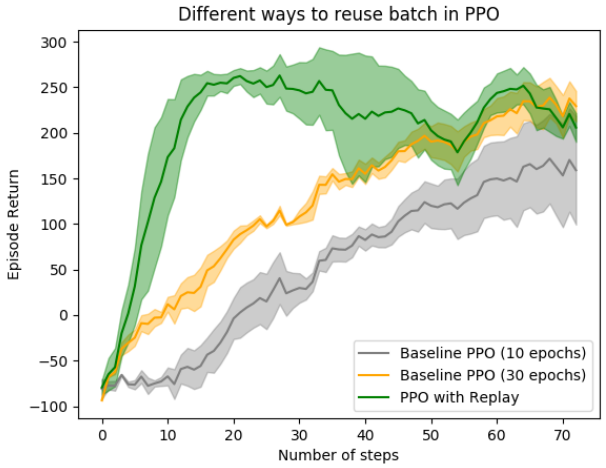

Fig. 6. Comparison of proposed algorithms with baseline PPO on UR5-2D task. Grey curve shows the learning for PPO with 10 epochs. Yellow curve shows the learning for PPO with 30 epochs. Green curve shows the learning for our algorithm with replay buffer, with 20 On Policy Epochs and 40 Off policy epochs. All experiments lasted 150K transition and around 3 hours of wall time.

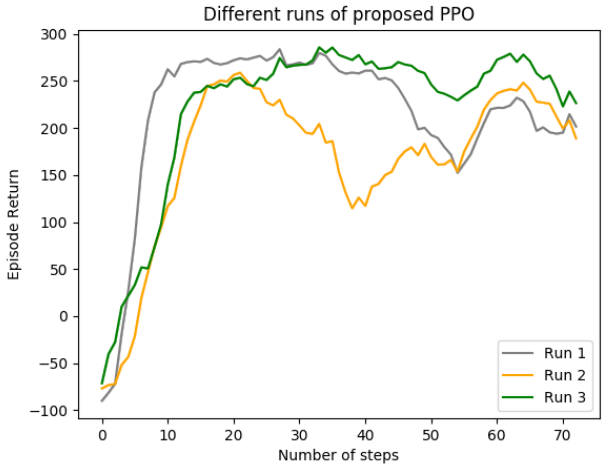

Fig. 7. Three different runs on the UR5-2D task

D. Hyper-parameters for the UR5 task

Table II mentions the hyper-parameter that were used for the PPO on the UR5. Performing a parameters sensitivity test is utterly difficult given the time it takes to conduct one experiment. Hence we have not been able to provide a sensitivity analysis on the same. We can see that hyper-parameters don't want vary a lot between the simulation and the real robot which is a desired property for our algorithms.

VII. CONCLUSIONS

Our experiment result shows that PPO algorithms can yield some performance gain when the off-policy data is utilized. However, there're some limitations in our approaches. First, the replay buffer size should not be too large, otherwise, the bias of λ-return after correction will be large and impede learning. Second, for every iteration, the log-likelihood and value of each sample in the replay buffer need to be updated which leads to high computational complexity. We hope those problems can be resolved in future investigation.

Hyperparameter	**Value**
Learning Rate	3e-4
Value Loss Coefficient	0.5
Entropy Coefficient	0.0
Batch Size	2000
Mini Batch Size	50
On-policy Epochs	10 & 30
Off-policy Epochs	40
Replay Buffer Size	8000
Clip Range	.2
Gradient Norm Clip	.5
Optimizer	Adam
Gamma	0.99
Lambda	0.95

TABLE II
HYPER-PARAMETERS FOR PPO ALGORITHMS ON THE UR5 TASK

APPENDIX

Here we will comparing our implementation of the PPO with respect to the baselines and try to identify the reasons for difference in performance. The PPO originally is a extension of 9 extra steps on top of the REINFORCE Algorithm. But completing those 9 steps doesn't allow the agent to learn to on the FetchReach environment. Reading the OpenAI Baselines code, they have used extra tricks on top of this implementation. The following paragraph describes the these extra modification. Though this allows the agent to converge on FetchReach, but it doesn't seem to improve the learning on the real Robot. The comparison is show in the Figure 8.

Extra modifications on top of the 9 step PPO:

1) Value Clipping : The loss of the value function is clipped so as to not cause extra large changes in the network
2) Gradient Norm Clipping : The **Norm** of the gradients in both the networks are clipped.
3) State Dependent Standard Deviation : Though this is not entirely from the OpenAI , but this is another modification that can be used.
4) Value Loss Coefficient : The coefficient that is to be given to the value loss for the combined loss, this parameters won't affect when the actor and critic have separate networks. Hence this is not really used.
5) Normalization of Observation Vector : We can normalize the observation by maintaining a running mean and standard deviation. We can run a few dummy episodes at first to get a good initial estimate of the mean and standard deviation. But in practice we observed that this didn't help the learning, so this modification has not been used in our experiments.

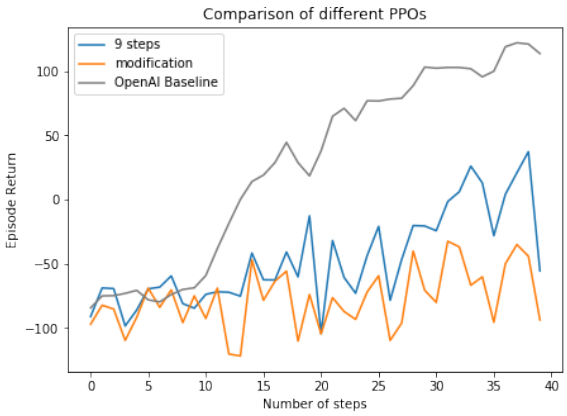

Fig. 8. Comparison of different PPO implementations. Blue line represents the original 9 steps that were taught in the class. Orange line represent the modification to replicate OpenAI Baselines PPO. Grey line indicates the OpenAI Baselines PPO

Figure 9 shows the results of the extra modification on top of the 9 steps on the PPO implemetnation for the `FetchReach-v1` environment. We can see that the changes enable the agents to achieve really good performance on the simulation but somehow fails on the real robot.

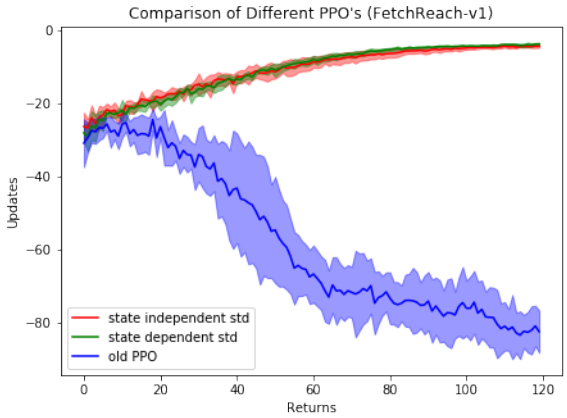

Fig. 9. Comparison of different versions of PPO implemented on the FetchReach-v1 environment

Table III presents the hyperaparameters that were used for 9 steps and modified step PPO.

Table IV provides the enviornment settings that were used to conduct the experiments.

ACKNOWLEDGMENT

We would like to thank Dr Rupam Mahmood for his continual guidance and support throughout the project. We would also like to thank the course TA, Craig Sherstan in helping us out with the initial setup of the robot.

Hyperparameter	**Value**
Learning Rate	3e-4
Value Loss Coeffecient	1
Batch Size	2000
Mini Batch Size	250
Epoch Count	10
Value Difference Clip	.5
Gradient Norm Clip	.5
Optimizer	Adam (Default)
Gamma	0.99
Lambda	0.95
Clip Action	False

TABLE III

THIS TABLE PRESENTS THE HYPERPARAMTERS THAT WERE USED FOR THE FETCH AND THE UR5 ROBOT FOR THE 9 STEPS AND MODIFIED STEP PPO.

Environment Variable	**Value**
Setup	UR5_default
Degree of Freedom	2
Control Type	velocity
Target Type	position
Reset Type	zero
Reward Type	precision
Derivative Type	none
deriv_action_max	5
First Derivate Max	2
Acceleration Max	1.4
Speed Max	0.3
Speedj_a	1.4
Episode Time Length	4 seconds
Episode Length Step	none
Actuation_sync_period	1
dt	0.04
run_mode	multiprocess
movej_t	2.0
Delay	0.0

TABLE IV

UR5 SETTINGS SPECIFICATION

REFERENCES

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[2] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.

[3] A. R. Mahmood, D. Korenkevych, B. J. Komer, and J. Bergstra. Setting up a reinforcement learning task with a real-world robot. *CoRR*, abs/1803.07067, 2018.

[4] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. *CoRR*, abs/1809.07731, 2018.

[5] M. Schlegel, W. Chung, D. Graves, J. Qian, and M. White. Importance resampling for off-policy prediction. *CoRR*, abs/1906.04328, 2019.

[6] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[8] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.