COMPSCI 687 - Final Project Report

Kamalesh Kumar K, Andrew Lin

December 10, 2024

Intro

Deep RL is an area of RL exploring the use of neural networks in RL algorithms. As neural networks are universal approximators that can interpret inputs of various shapes and sizes, much more complex states and policies can be represented. For example, convolutional networks can be used to efficiently interpret image state representations. This has been greatly useful in training agents to play video games such as classic Atari games, and recently much more visually and strategically complex games. For example, in 2019 Google DeepMind released a high-performing StarCraft agent that was trained using multi-agent deep RL[3]. In recent years, Deep RL has also successfully been used in practical applications such as robotics and self-driving cars.

Although image-based environments are impressive and possible, for the scope and purposes of this project we stuck to vector state API environments accessed through the Gymnasium interface. In particular, we implemented and tested DDQN and PPO. We tested DDQN on CartPole and Lunar Lander and we tested PPO on Lunar Lander and Bipedal Walker.

DDQN

DDQN or Double Deep Q Networks is an algorithm that extends upon tabular Q learning. Instead of representing our Q function as a table, we represent it as a neural network, which allows us to work with a continuous state space. Like Q learning we iteratively improve our Q function estimate by taking an action in our domain using a policy derived from our Q function and then comparing the value of this action with the bootstrapped estimate of the optimal action value using the observed reward. In DDQN, we use SGD to minimize the squared difference between the bootstrapping and the current estimate. By minimizing this difference we iteratively improve our Q function estimate until it represents the Q function of the optimal policy.

Additionally, with DDQN, there are a few further differences, one of which is called experience replay. This method stores transitions and randomly samples batches to update our Q function. As more transitions are recorded, older ones are discarded from memory such that we only remember the most recent or most relevant transitions. By batching our updates we improve the efficiency of our updates and by reusing past experiences we improve learning stability.

The other addition is a second Q estimate called the target. We use the target to create our bootstrapped optimal action value estimate and compare it to the main Q function. We choose to update the target less frequently than Q. By using two Q networks instead of one, we can further stabilize learning and reduce overestimation. The rate at which the target and Q are updated can then be adjusted via their respective hyperparameters, which we represented as time steps within each episode. This addition is what gives it the name Double DQN. Lastly, to incentivize exploration, we use an exponentially decaying epsilon value which dictates the probability that a random action is selected instead of an action chosen by our policy.[2]

The Pseudocode for DDQN is as follows:

```
Algorithm Double DQN
1: initialize network $Q$ with random parameters $\theta$ and stepsize $\alpha$
2: initialize network $Q_t$ with parameters $\Phi = \theta$
3: let $f$ be the $Q$ update frequency, and let $C$ be the $Q_t$ update frequency
4: for each episode do
5:    observe $S_t$ = initial state
6:    $D \in \{0, 1\} = 0$, the done flag which signals if the episode has ended
7:    while not done ($D \neq 1$) do
8:       with probability $\epsilon$, $a$ =random action, otherwise $a = \text{argmax}_a(Q(S_t, a))$
9:       take action $a$, observe $r$ = reward, and $S_{t+1}$ = the next state
10:      update $D$ depending on whether episode has ended
11:      store transition $(S_t, a, r, S_{t+1}, D)$ in memory
12:      sample $N$ transitions
13:      for each transition $i$ $(S_t, a, r, S_{t+1}, D)$ in batch do
14:         $q(s, a) = Q(S_t, a)$
15:         $q_t(s', a') = \max_a(Q_t(S_{t+1}, a))$
16:         $y_i = r + (1 - D)\gamma q_t(s', a')$
17:         $\text{loss}_i = (q(s, a) - y_i)^2$
18:      loss= $\frac{1}{N} \sum_i^N \text{loss}_i$
19:      every $f$ steps set $\theta = \theta - \alpha\nabla_\theta\text{loss}$
20:      every $C$ steps set $\Phi = \theta$
21:      $S_t = S_{t+1}$
22:   decay $\epsilon$
   =0
```

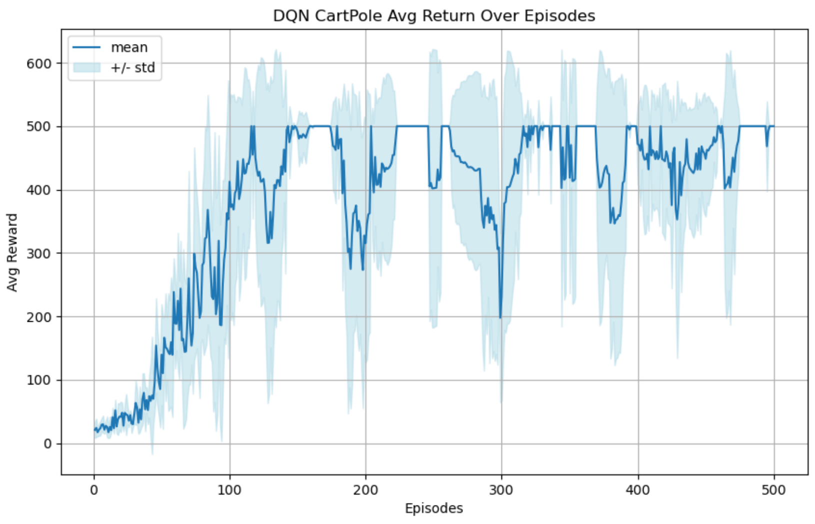
Hyperparameter Tuning

For DDQN our hyperparameters were learning rate (α), gamma (λ), epsilon start, epsilon min, epsilon decay, target update frequency (C), Q update frequency (f), memory capacity, and batch size. As for fixed values we did not tune, we kept a fixed max number of steps per environment: 2000 for lunar lander and 500 for cart pole. We chose to train across 1000 episodes for lunar lander and 500 for cart pole and we used $\lambda = 0.99$ for both environments.

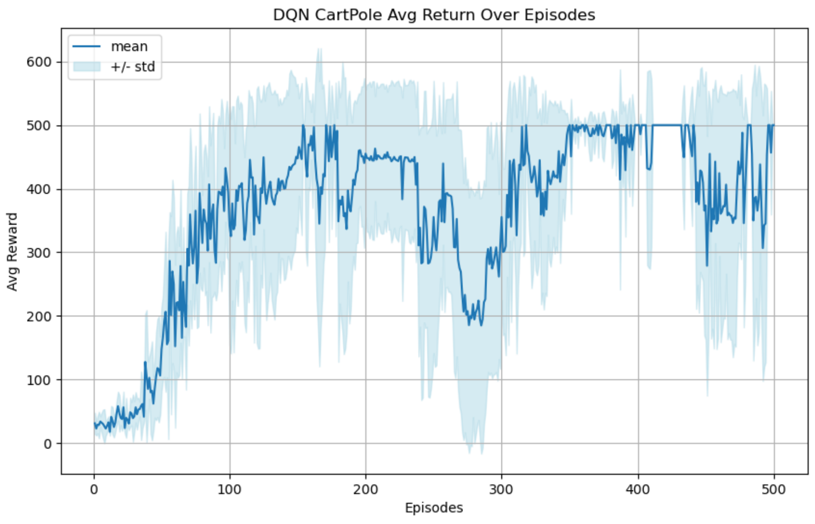
For our network architecture, we used a simple network with one hidden layer using layer normalization to help with divergence in our outputs. The input and output dimensions varied based on the environment, for Cart Pole, states were vectors of size 4 and there were two actions, making an input dimension of 4 and an output dimension of 2. For Lunar Lander, states are vectors of size 8 and there are 4 actions making an input dimension of 8 and an output of 4. We varied the size of the hidden layer by factors of 2 but mainly stuck with 64 as most variations didn't seem to have too much impact.

In terms of tuning strategies, we started arbitrarily, monitored results across initial episodes, and restarted with adjustments if it seemed like the model wasn't making sufficient progress or could be improved. Learning rates were adjusted by factors of 10. Epsilon was decayed from 1 to 0 or 0.05. Decay rates were varied from 0.9 to 0.999 by increments of 0.01 and 0.001. Capacity was varied in increments of 1000. Batch size was adjusted by factors of 2 (e.g. $16, 32, 64$, etc).f was adjusted by values of 10, and C was adjusted by values of 4. Once a set of parameters was identified to learn well, we ran it for the full range of episodes for 5 runs and plotted the return over episodes. From this identified set of parameters, we made adjustments to several particularly impactful parameters and plotted the averaged results for those to visualize their impacts on performance.

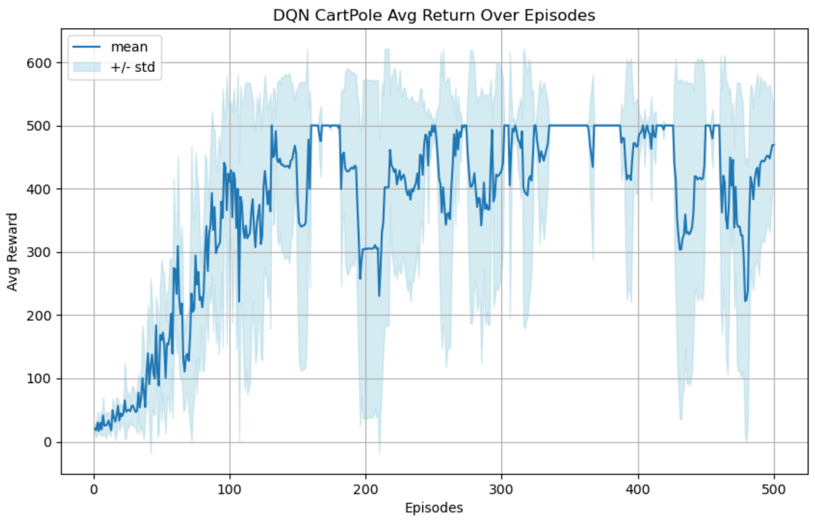
DDQN Cart Pole Results



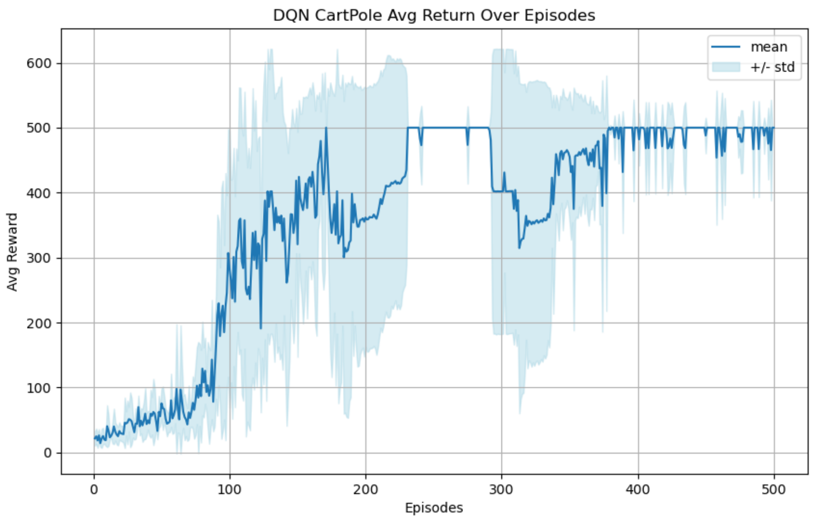
Here we see the reward over training episodes of DQN on the Cart Pole environment. We had epsilon decay from 1 to 0 with a decay value of 0.98. We had $C = 10$, $f = 4$, a batch size of 16, a learning rate of $1e-3$, and a capacity of 1000. For the most part, we see the agent learns to balance the pole for the maximum length of each episode (500 steps), however, there are still many places where performance drops. Given the capacity of 1000, the memory fully resets every 2 episodes, so there is quite a bit of "forgetting" happening across 500 episodes. This means if the agent overestimates some action values and leads to a less than optimal policy, it may take some time for it to relearn and refill the memory buffer with good experiences. Still, this set of hyperparameters seems to reduce the variation and size of these dips over time as the (mean - std) minimum values roughly trend upwards over 500 episodes.



Here we keep the same hyperparameters, but increase capacity from 1000 to 3000. We notice that although the agent still maxes out the reward at several points, there are still dips in performance that are now even larger. This may be a sign that too much memory can be an issue. It is possible that the algorithm is training itself on outdated transitions, leading to suboptimal updates which also fill the memory and take longer to replace, leading to longer dips in performance

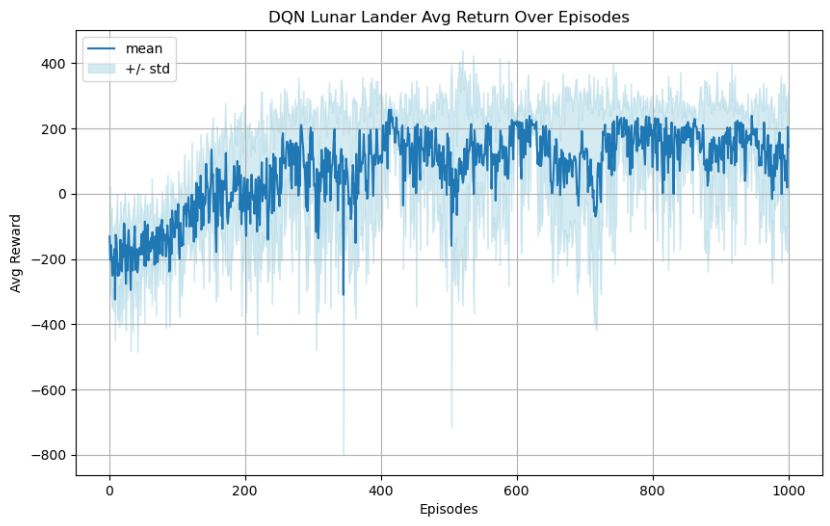


Here, from our first set of hyperparameters on the Cart Pole environment, instead of increasing the capacity, we decrease the capacity to 500. We get similar results to our original hyperparameters, but worse performance towards the end of the 500 episodes. This may mean that in contrast to the first set of hyperparameters, there is now no general trend towards more stable performance in the long term. If this is the case it is safe to assume it is because the capacity is too small and samples are not diverse enough leading to unstable performance.

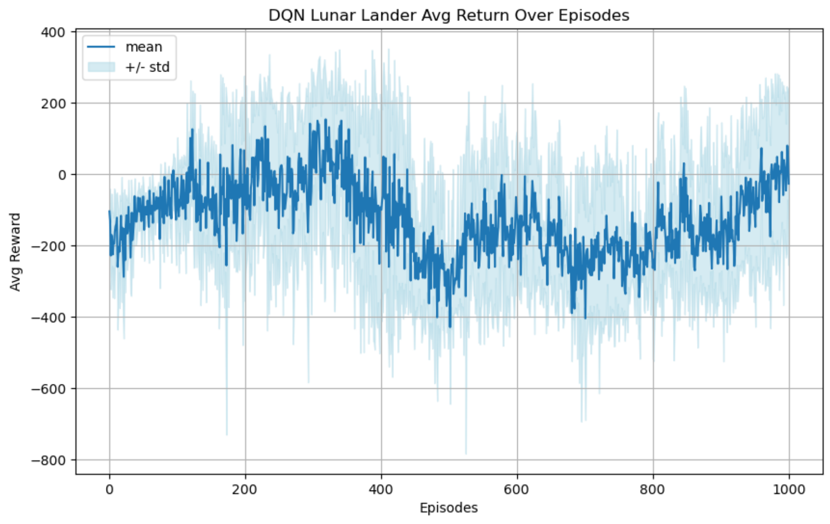


Finally, instead of changing capacity, we decrease our update frequency f and update only every 10 time steps instead of 4. Here, although there is a huge variance in return up until 380 epochs or so, after this point, there seems to be a very tight convergence on the optimal policy with minimal deviations. Updating less frequently seems to have allowed for increased stability after a certain point as the memory can fill with more relevant experiences to the current Q function before updating. If judging by the last 120 epochs, this is the best performance achieved.

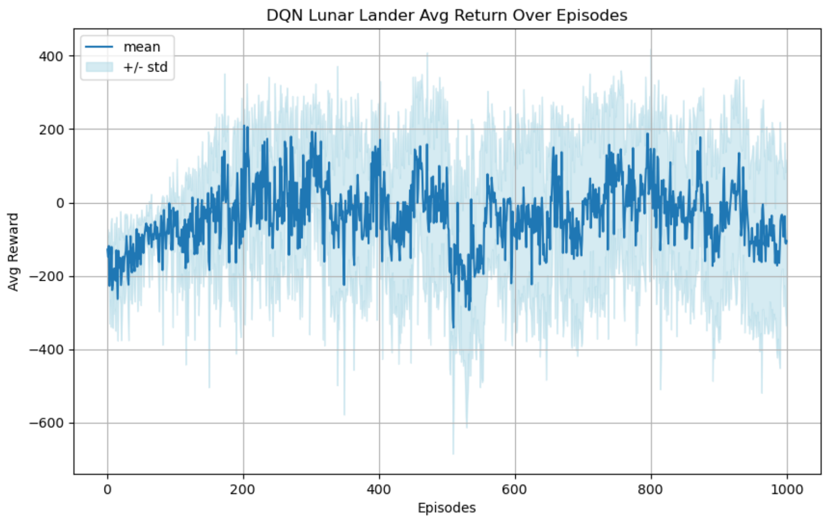
DDQN Lunar Lander Results



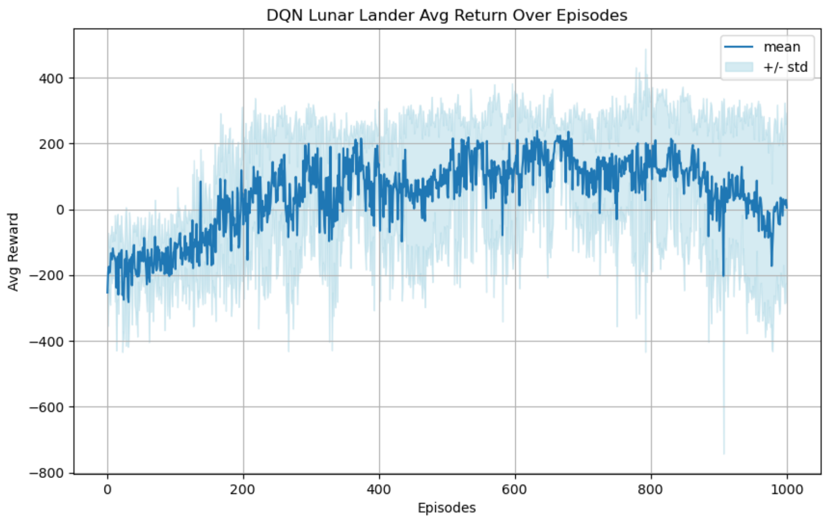
Here we see the DDQN results for lunar lander. Here we decayed epsilon from one to 0 with a decay value of 0.99, we have $C = 40$, $f = 4$, a capacity of 2000, a batch size of 64 and a learning rate of 1e-4. Across the board, we also used a learning delay (called learning_buffer) in the code, where updates don't happen until after 10000 total time steps. This was done somewhat arbitrarily, the buffer was to allow the memory to fill up fully. Although 10000 is overkill for a capacity of 2000, to keep things consistent and be able to experiment with larger capacities we kept it fixed at this amount. In the end, there was little impact on performance. We see here that with these settings, the model is able to go from -200 reward to barely touching and average of 200 reward, which is the stated standard for solving the environment. There is a good amount of dipping, however, overall it is safe to say the model learned successfully. In practice, the model was able to land frequently.



Here we fix all parameter settings we had previously and only increase capacity to 5000. We can see this had the effect of drastically decreasing performance. Although the mean return is relatively consistent, it wavers greatly and never reaches 200. As in the cart pole environment, it seems as though this capacity is too large and holds outdated transitions, misleading the agent and leading to widely varying sub-optimal performance.



Here, we take our original parameters and decrease capacity to 1000. We can see that although the global trend of the average reward is more consistent, there is a much larger variance, and performance is still poor, hovering above and below 0. This suggests that 1000 is too low, and the pool of trajectories to sample from lacks diversity. This means the agent will over-train on recent transitions, and perform poorly on new states it encounters, leading to the massively varying performance over time.



Here, instead of capacity, we decrease our update frequency from $f = 4$ to $f = 12$. This seems to have similar variance to our original parameter settings, however, overall performance was worse, and trended downwards in the last 200 episodes. More infrequent updates, although possibly good for more stable performance, also decrease learning speed. This may mean that our target gets updated to worse Q estimates which are then updated based on worse targets, thus hindering overall performance.

PPO

Proximal Policy Optimization (PPO) [1] is a RL algorithm introduced by OpenAI to address some of the limitations of earlier methods like Trust Region Policy Optimization (TRPO). PPO is an on-policy algorithm that belongs to the family of policy gradient methods, directly optimizing the policy that determines an agent's behaviour in a given environment. PPO is known for its efficiency and robustness, making it one of the widely adopted RL algorithms.

The core idea of PPO revolves around stablizing policy updates. In policy gradient methods, the policy is parametrized by a neural network with parameters θ. Training involves sampling trajectories from the environment using the current policy, estimating rewards, and updating the policy to maximize expected cumulative rewards. However, large updates to the policy can lead to instability or even catastrophic drop in performance. While TRPO introduces a constraint on the size of the policy update to mitigate this, it requires complex computations involving second-order optimization, which can be computationally expensive and challenging to implement.

PPO simplifies this by replacing the complex constraint of TRPO with a clipping mechanism that prevents the policy from diverging too far from the current one. PPO uses a surrogate objective function that compares the new policy π_θ with the old policy π_{old}. Specifically it computes the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)}$$

where a_t is the action taken at state s_t. The surrogate objective maximizes the advantage weighted probability ratio but clips the ratio within range $[1-\epsilon, 1+\epsilon]$ to limit the size of the update:

$$L^{CLIP}(\theta) = \mathbb{E}_t\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big]$$

Here $\hat{A}_t$ is the advantage function which quantifies the relative benefit of taking action a_t compared to the average action at s_t. This clipped objective ensures that the policy updates are constrained while still allowing improvement thereby maintaingn a balance between exploration and exploitation. PPO also employs an actor-critic framework where the policy network (actor) determines the action probabilties, and a separate value network (critic) estimates the value of the states. The value network is trained to minimize the squared error between its predictions and the actual returns observed:

$$L^{VALUE}(\phi) = \mathbb{E}_t\Big[(R_t - V_\phi(s_t))^2\Big]$$

where R_t is the cumulative reward from time step t onward. This value network helps compute the advantage function $\hat{A}_t = R_t - V_\phi(s_t)$.

ANother important component of PPO is entropy regularization, which encourages exploration by penalizing policies that become overly deterministic. This is achieved by adding an entropy term to the objective:

$$L^{ENTROPY} = -\beta.\text{Entropy}(\pi_\theta)$$

where β is a hyperparameter controlling the strength of the penalty. The total loss for the policy is then the sum of the clipped surrogate objective and the entropy term. The PPO training process alternates between collecting trajectories by interacting with the environment and updating the policy and value netowrks using mini-batch optimization. The psuedocode for PPO is as follows:

Algorithm Proximal Policy Optimization (PPO)

```
1: Initialize policy network $\pi_\theta$ and value network $V_\phi$
2: for iteration = 1, 2, ... do
3:     Collect trajectories $\{(s_t, a_t, r_t)\}$ by running policy $\pi_\theta$ in the environment
4:     Compute rewards-to-go $R_t$ and advantages $\hat{A}_t = R_t - V_\phi(s_t)$
5:     for epoch = 1, 2, ... do
6:         for each mini-batch of trajectories do
7:             Compute the policy loss using sampled data:
                   $$L_{\text{policy}} \approx \frac{1}{N}\sum_{t=1}^{N} \min\left(r_t\hat{A}_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon)\hat{A}_t\right) - \beta \cdot \text{Entropy}(\pi_\theta)$$
8:             Compute the value loss using sampled data:
                   $$L_{\text{value}} \approx \frac{1}{N}\sum_{t=1}^{N} (R_t - V_\phi(s_t))^2$$
9:             Update the policy network:
                   $$\theta \leftarrow \theta + \alpha\frac{1}{N}\sum_{t=1}^{N} \nabla_\theta L_{\text{policy}}$$
10:            Update the value network:
                   $$\phi \leftarrow \phi - \alpha_v\frac{1}{N}\sum_{t=1}^{N} \nabla_\phi L_{\text{value}}$$
    =0
```

Hyperparameters

For our implementation of PPO, we mention a number of hyperparameters that were used in the training process. These include the learning rate for the actor and critic networks, the discount factor (γ), the clipping parameter (ϵ), the number of epochs for updating the policy (K), and the update frequency (timesteps per policy update). For continuous action spaces, we also tuned the initial standard deviation of the action distribution (σ), its decay rate, and the minimum standard deviation to balance exploration and exploitation. We tested the algorithm on two environments on the Gymnasium library, namely Lunar Lander and Bipedal Walker.

Fixed parameters such as the maximum episode length and the total training timesteps were set to ensure consistent evaluation across environments. For example, for lunar lander we used a maximum of 300 timesteps per episode, and similarly 1500 timesteps for Bipedal Walker. The maximum number of training steps for Bipedal Walker and Lunar Lander were set to 3×10^6, and 1×10^6 respectively.

Our neural network architecture for both the actor and critic networks consists of two fully connected hidden layers with 64 units each, using *tanh* activations. The actor's output varied depending on the action space: for discrete actions, it produced probabilities through a softmax layer, while for continuous actions, it provided the mean of a multivariate normal distribution with a fixed or decayed standard deviation. The critic outputs a single scalar value representing the expected return for a given state. Input dimensions were determined by the environment's state space, while the actor's output dimensions were determined by the action space. The Lunar Lander variant had a discrete action space, whereas the Bipedal Walker had a continuous action space.

In terms of tuning strategies, we fixed certain hyperaparameters as per the values used in the experiments section of the PPO paper [1] . In both of the environments tested, the actor and critic learning rates were fixed to 3×10^{-4} and 1×10^{-3} respectively, while the clipping parameter (ϵ) was set to 0.2. $\gamma = 0.99$ was used in all the experiments. In case of Bipedal Walker, the standard deviation for was decayed linearly starting from 0.6 after every 250,000 timesteps until reaching a minimum threshold of 0.1. In our experiments shown in the results section, we show the influence of two hyperparameters, the update frequency of the policy, and the number of iterations the policy is updated for. In each setting of the hyperparameters explored, 5 independent trials were conducted, and the mean and standard deviation of the rewards obtained are shown in the following plots in the results section.

PPO results for Bipedal Walker

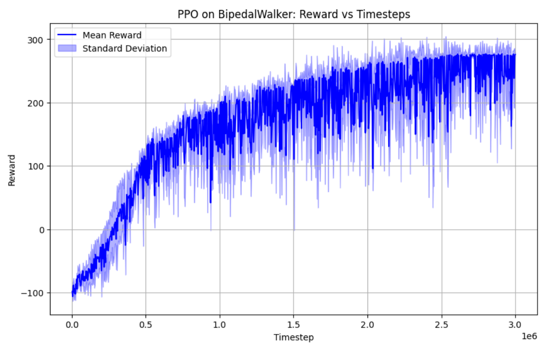


Figure 1: With $K = 80$

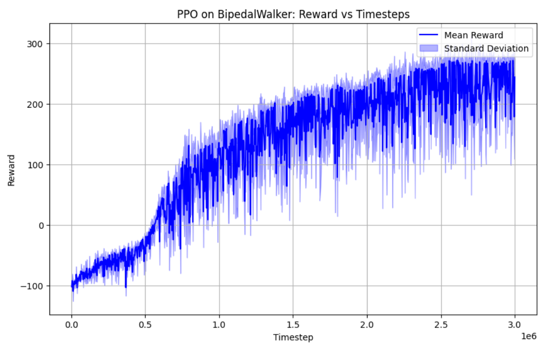


Figure 2: With $K = 10$

In the above plots, Figure 1 was for the best combination of hyperparameters tested. In addition to the fixed values mentioned before, in both the above experiments, the update frequency for for policy was set to every 6000 timesteps, and the number of epochs K for the policy update, was set to 80 to left figure (best setting), and 10 for the figure on the right. Clearly, it can be inferred that a higher value of K gave rise to slightly more stable and fast learning with reduced variance.

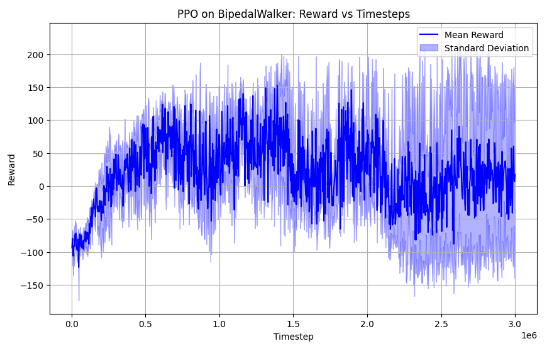


Figure 3: With update frequency as 1500 timesteps

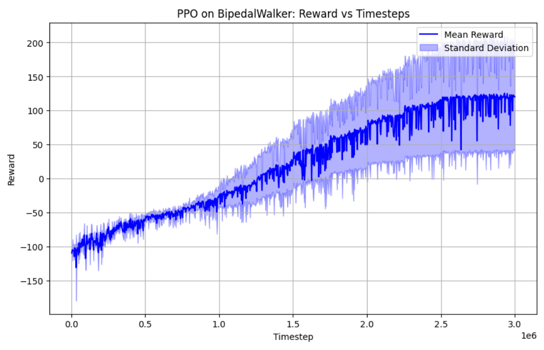


Figure 4: With update frequency as 30000 timesteps

In the next set of experiments, we fix all the other hyperparameters as per the best setting, but now only vary the update frequency. Remember from the best setting as in Figure 1, that there the update frequency for the policy was every 6000 timesteps. From Figure 3, we can infer that updating the policy way too frequently is not beneficial as it never ends up learning, whereas updating the policy very infrequently as in Figure 4 does gradually help the agent learn but greatly reduces the speed of learning.

In the final experiment, the only hyperparameter we vary with respect to the best setting is the learning rate of the actor. The best setting has a learning rate of 3×10^{-4} for the actor, and the plot below shows for a ten fold increase in same. It's also crucial to note that this new learning rate for the actor is now higher than that of the critic (1×10^{-3}), which was not the case in the best setting. And clearly from the below plot, such a scenario is not beneficial, as the critic in general should learn faster to predict better estimates of the value for the actor to rely on while updating it's policy.

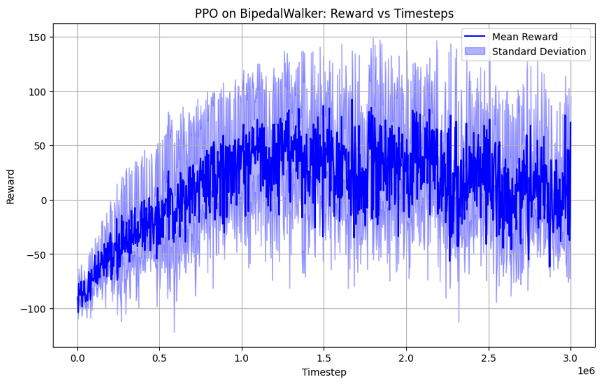


Figure 5: With actor learning rate = 3×10^{-3}

PPO results for Lunar Lander

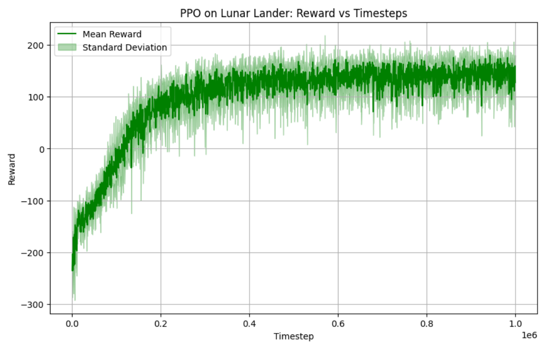


Figure 6: With $K = 30$

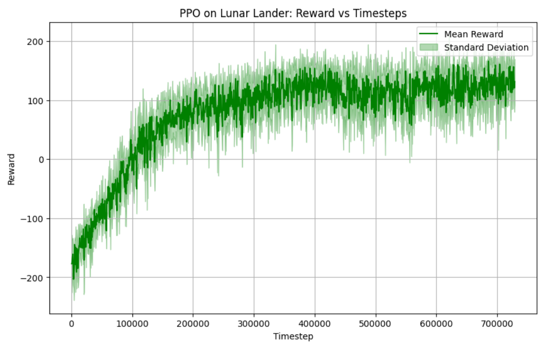


Figure 7: With $K = 5$

In the above plots, similar to experiments on Bipedal Walker, Figure 6 was for the best combination of hyperparameters tested. In addition to the fixed values mentioned before, in both the above experiments, the update frequency for for policy was set to every 900 timesteps (best setting), and the number of epochs K for the policy update, was set to 30 to left figure (best setting), and 5 for the figure on the right. Clearly, it can be inferred that a higher value of K gave rise to better and faster learning, and higher mean reward towards the end of training. Note that in both the above figures, the y-axis are not of the same scale. The second converges close to 100, whereas the best setting close to 200.

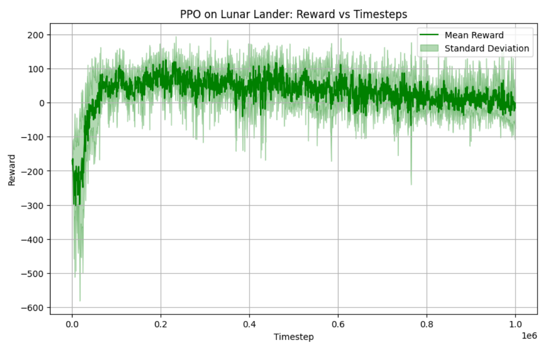


Figure 8: With update frequency as 150 timesteps

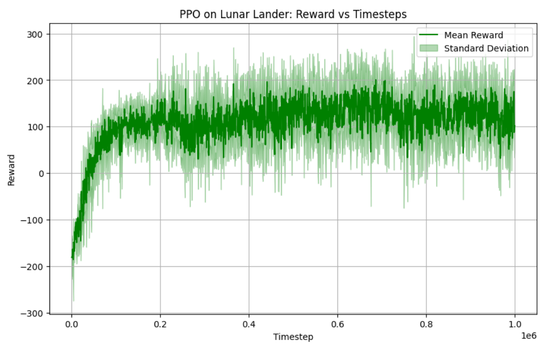


Figure 9: With update frequency as 6000 timesteps

In the next set of experiments for Lunar Lander, we fix all the other hyperparameters as per the best setting, but now only vary the update frequency. Remember from the best setting as in Figure 6, that there the update frequency for the policy was every 300 timesteps. From Figure 8, we can infer similar to the case of Bipedal Walker that updating the policy way too frequently is not beneficial and in fact jeopardizes the final performance, whereas updating the policy very infrequently as in Figure 9 does help the agent learn but has higher variance and lower final mean reward compared to the best setting.

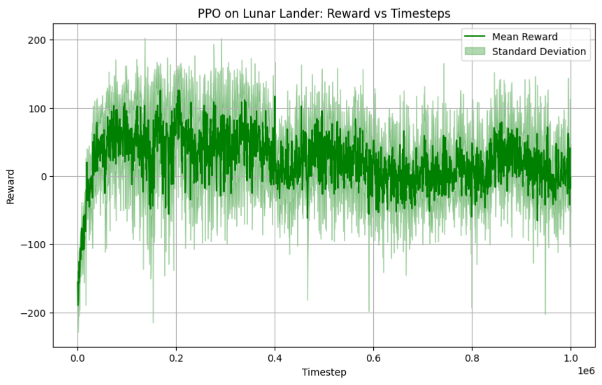


Figure 10: With actor learning rate $= 3 \times 10^{-3}$

In the final experiment, similar to Bipedal Walker, the only hyperparameter we vary with respect to the best setting is the learning rate of the actor. The best setting has a learning rate of 3×10^{-4} for the actor, and the plot above shows for a ten fold increase in same. Thus extending the conclusion for this from Bipedal Walker, we can again infer that the critic in general should learn faster to predict better estimates of the value for the actor to rely on while updating it's policy, and hence the critic's learning rate should be higher than that of the actor.

Conclusion

In this final project, we successfully implemented the DDQN and PPO algorithms from scratch, and showed our results on a range of environments from Gymansium interface such as, CartPole, Lunar Lander, and Bipedal Walker. We have also conducted an extensive hyperparameter study for each of these algorithms to thoroughly understand how each parameter plays a role in the agent's learning process. Apart from that, this project has also paved way to our detailed understanding of the setting of Deep RL, and policy-gradient methods, and the implementational details of DDQN, and PPO, combined with the motivations for the framework of the algorithms, such as for example how PPO's unified clipping objective tries to tackle a more complex second-order optimization problem posed by TRPO. The code and other details of our implementation is attached along with the zip file for this report.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[3] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575:350 – 354, 2019.