Comparing Reinforcement Learning Algorithms on OpenAI Gym and Unity3D Environments.

Hari Vidharth s4031180
Ashwin Vaidya s3911888
Krishnakumar Santhakumar s4035992
Dhawal Salvi s4107624

Abstract

Deep Reinforcement learning is a sub-domain of deep learning which focuses on training agents without human intervention. Ongoing research has proposed many algorithms which show a range of results. However, one does not know which algorithm to pick for their task. In this paper we compare *Double Deep Q-Learning (DDQN)* and *Proximal Policy Optimization (PPO)* algorithms on four environments and conclude that DDQN performs better than PPO on these environments. We list the environments and their specifications so that the reader can use our results on a similar task.

1 Introduction

Reinforcement learning is a technique by which a system learns to accomplish a task or a goal by performing actions on the environment and receiving feedback from it. This is a convenient way to train neural networks for tasks which do not have sufficient training data, or for the tasks for which giving explicit information is not feasible. Many problems of robotics such as biped motion come under the purview of reinforcement learning. Recently, reinforcement learning has shown results surpassing those by humans [2, 7] and it finds uses in many real-world applications such as Traffic Light Control, Advertisement, Chemistry, Games, etc. However, choosing the algorithm from various reinforcement learning algorithms remains the central problem. These algorithms are diverse and often based on different approaches such as *Q-Learning* and *Policy Optimization*. In this paper, we analyze *DDQN* and *PPO* based on one of each approach. We conduct experiments using different environments and present the results in Section 5.

2 Background

2.1 Double Deep Q Networks (DDQN)

Q-Learning based approaches estimate the quality of the next action, based on the current state. This quality is used to select the next action. It uses Equation 1 to estimate the target value for Q.

$$Q(s, a) = r(s, a) + \gamma max_a Q(s^{'}, a) \tag{1}$$

Here Q represents the Q-value, r reward, s current state, s' next state, a actions and γ represents the discount factor. Further, the max function selects the highest Q-value from the state-action pair of the next state. However, it overestimates the Q values as well as the action values. DQN [5] also suffers from this problem of overestimation.

DDQN [11] eliminates this drawback of overestimation by splitting the Q-Network in two. It uses an online network (Q network) to greedily select the action and a target network to estimate the Q-value. The Equation 2 is used for the estimation of the target value Q in DDQN.

$$Q(s,a) = r(s,a) + \gamma Q(s^{'}, argmax_a Q(s^{'}, a)) \tag{2}$$

Here, the Q network chooses the action corresponding to the highest Q values and the Target network estimates the Q values of actions given the next state. Moreover, DDQN uses different policies for Q-value evaluation and action selection. Two separate value functions are trained using separate experiences. Also, the estimated value of the future actions is evaluated using a different policy. These additions solve the problem of overestimation and helps in faster training and more stable learning.

2.2 Proximal Policy Optimization (PPO)

Policy gradient methods have shown success in continuous environments. However, they suffer from the problem of credit assignment. It cannot discern which actions were responsible for success. This leads to gradient updates for the entire trial by assuming all the actions equally contributed to the success or failure of the trial. This assumption is naive as well incorrect and results in longer training time and instability. A better way to proceed is to predict the contribution of each action during a trial and weigh the gradients accordingly. This leads to the family of algorithms called Actor-Critic algorithms. In this algorithm, the single network is separated into an actor-network and a critic-network. The Actor-network is responsible for taking actions and the critic-network which assigns a value to each action.

The Figure 1 shows the general architecture of Actor-Critic network.

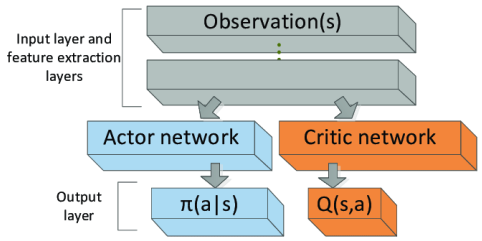

Figure 1: Asynchronous Actor Critic. Source [4]

PPO [6] is an extension to Actor-Critic method. The main contribution of this method is the clipping function as shown in Equation 3. A problem with policy gradients is that there is no limit on the step size and a very large step can diverge the policy. Further, an observation collected using a sub-optimal policy will diverge the network during the next back-propagation step. PPO solves this by the use of a clipping function. Equation 3 shows that the loss is calculated using the expectation of advantage times the gradient. This gradient is clipped using the *epsilon* parameter and ensuring that the value of the gradient does not explode. Here, $\hat{A}_t$ denotes the Advantage which is the expected discount future reward.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{3}$$

3 Implementation Details

3.1 Training Environment

Table 1 lists the 4 environments we used to train our models on in order of complexity.

Environment	Developer	Observation Space	Action Space	Actions	Goal	Reward	Time Steps
Basic	Unity	(20,)	Discrete-3	No Action, Move Left, Move Right	Move to Optimal State	-0.01 at Each Step +0.01 for Sub Optimal State +1.0 for Optimal State	100
Lunar Lander	OpenAI	(8,)	Discrete-4	Do Nothing, Fire Right Engine, Fire Main Engine, Fire Left Engine	Land in Target Area	-100 for Crash +100 for Land +10 for Ground Contact -0.3 for Fire Main Engine +200 for Solving	1000
Push Block	Unity	(210,)	Discrete-7	No Action, Turn Clockwise and Counter Clockwise, Move 4 Directions Front Back Left Right	Push Block to Goal	-0.0025 for Every Step +1.0 for Block Touching Goal	1000
Hallway	Unity	(108,)	Discrete-5	No Action, Turn Clockwise and Counter Clockwise, Move Front and Back	Choose and Move to Correct Room	+1 for Moving to Correct Room -0.1 for Moving to Incorrect Room -0.0003 Existential Penalty	500

Table 1: Environment Description.

3.2 Double Deep Q Networks (DDQN)

DDQN was implemented using the *Keras* framework, ml-agents, unity gym wrapper, OpenAI gym, and python 3.7.7. The online and the target network architecture are same which consists of an input layer the size of the observation space and the output layer the size of the action space with a linear activation. Further, two hidden layers in between them with the size of 512 each with *ReLu* activation. DDQN agents learn and improve themselves through a method called *experience replay*. It stores the transitions that the agent observes allowing this data to be reused for training later on. Random samples called mini-batch are taken from the transitions that serve as the training data for the network. It has been shown that this greatly stabilises and improves the DQN training procedure. It essentially maps (s, a) pairs to their (s', r). Now, the agent remembers each step of the episode, the memory consists of the actions taken, the state and place when the action was taken, the reward for the action and, the new state resulting from the action taken. These memories are also known as experiences. By running through experience replay, every time the agent takes an action, the online network will begin to associate certain state/action pairs with appropriate Q values, which means higher and positive Q values if the actions taken results in survival and negative Q values if the actions are taken ends the episode. The agent will begin to predict higher Q values and will start to survive for long as the agent keeps playing the game.

Also, the ϵ-greedy strategy has been implemented to influence the choice of actions. ϵ starts at 1 and decays to 0.1 or 0.01 throughout the episodes. An ϵ decay rate is set to control the amount of epsilon decay from max to min. A higher value of ϵ makes the agent choose more random actions encouraging exploration. While, lower value of ϵ acts as the threshold value for which the agent starts to take greedy actions and thus, exploitation begins. Besides that, a deterministic policy may be stuck in a local optimal during the end of the training phase. Hence, there is a minimum amount of ϵ maintained to prevent this and to encourage some amount of exploration by taking random actions avoiding exploitation/greedy actions by the agent.

3.3 Proximal Policy Optimization(PPO)

The PPO was implemented using the *Keras* library in python $3.7.4$ and the actor and critic network consist of three fully connected layers. The first layer has 64 neurons, followed by 32 neurons in the second and third layer. Further, the output layer changes depending on the action space of the respective environment. Table 1 lists the action space of each environment. The critic network has only one unit for the output layer. Moreover, the rectified linear unit (*ReLU*) [1] is used as an activation for all the layers while the final layer of the actor-network has *softmax*. We used *Adam* [3] as an optimizer for training all the environments. The loss function of the actor network was computed using Equation 3 and the critic loss is calculated using the *Mean Squared error*. The environments were interfaced using Unity's gym wrapper and OpenAI gym. Each episode runs for the default time limit by the environment designed (the default time steps are shown in the table 1 and the reason to choose this setting is to let the agent explore more about the environment) or until the goal is reached.

4 Experiment Details

4.1 Double Deep Q Networks (DDQN)

For DDQN all the experiments were conducted on the four environments mentioned in the above Table 1. The network architecture remained the same as mentioned in the implementation details 3.2 throughout the experiments. As ϵ-greedy strategy was used, so ϵ decay was the only hyperparameter that was

varied during the experiments. The maximum and minimum ϵ remained the same at 1.0 and 0.01 respectively. ϵ decay was changed as per the environment to favour exploration and exploitation accordingly and with the value 0.99 for Basic, 0.999 for Lunar Lander, 0.99999 for Push Block and Hallway. All the environments were run for 1000 episodes.

4.2 Proximal Policy Optimisation (PPO)

Similar to DDQN experiment settings 4.1, PPO also tested on the four different environments as shown in the Table 1. All the environments except push-block run for 1000 episodes whereas, push-block environment training runs for 500 episodes. The learning rate of the actor and critic model was set to 0.001 for all the experiments. Similarly ϵ (0.2), γ (0.99) and entropy values (0.001) were common for all the experiments. The threshold value (named *update_thresh*, in the program) was set to regulate the training for each environment. For lunar lander environment, the threshold value was set to 5 and for unity-basic and push-block environment, the threshold value was set to 10 and 200 respectively. However, for hallway environment, we implemented different strategy for training. Firstly, we updated the network when the agent reaches the goal faster by comparing the current steps the agent has taken to reach the goal with the previous attempts. In all the experiments the agent chooses the action based on the probability of policy actions by the actor network in the environment.

5 Results

We now discuss the performance of both the models in terms of the average scores over the total episodes. First, we take the performance of the two simple environments, Basic and Lunar Lander. From Figure 2 and 3, we can see that DDQN model shows a good performance on these environments with the average score increasing over the episodes and tapering towards the end. PPO model shows a similar performance but DDQN seems to be performing better, in the two simple environments as seen in Figure 6 and 7.

Next, we compare the performance of the two complex environments, the Push Block and Hallway, which require a bit more exploration. The DDQN model compared to the simple environments shows a decrease in performance, but also shows steady and consistent updates. Further, on the push block environment, it reaches the ideal average score of 2. While in the hallway, the average score falls a little short than the ideal average score of 1. However, we believe that due to the nature of the complexity of the environment more training and exploration will help in performance. Contrasting this with PPO, Figure 8 and 9 shows that for these two complex environments, the average score did not improve across each episode.

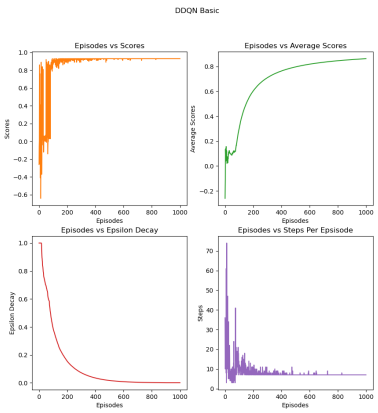

Figure 2: DDQN-Unity Basic Environment

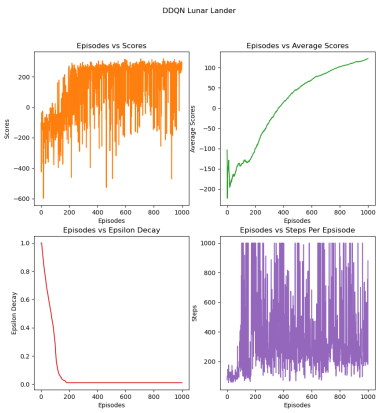

Figure 3: DDQN-OpenAI Lunar Lander Environment

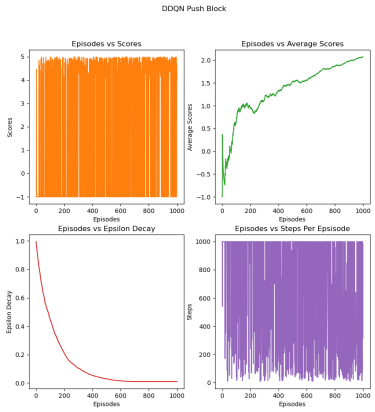

Figure 4: DDQN-Unity Push Block Environment

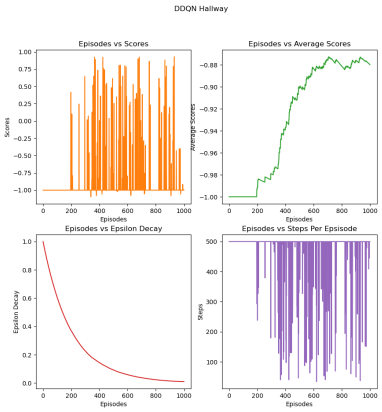

Figure 5: DDQN-Unity Hallway Environment

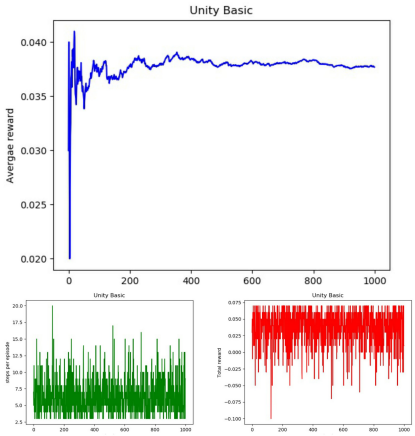

Figure 6: PPO result : Unity Basic Environment, episodes (x-axis) vs average result, steps per episodes and total rewards (y-axis)

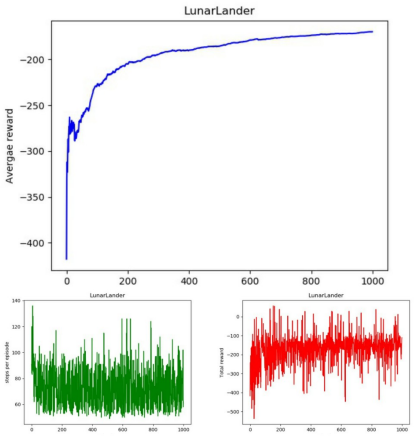

Figure 7: PPO result : Lunar Lander Environment, episodes (x-axis) vs average result, steps per episodes and total rewards (y-axis)

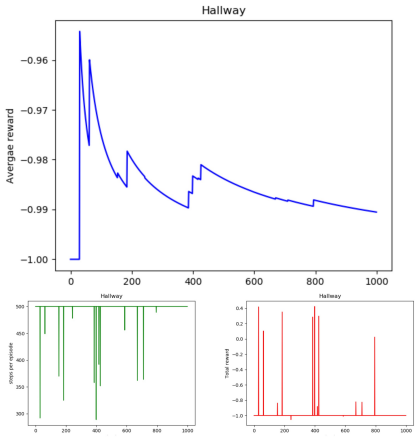

Figure 8: PPO result : Unity Hallway Environment, episodes (x-axis) vs average result, steps per episodes and total rewards (y-axis)

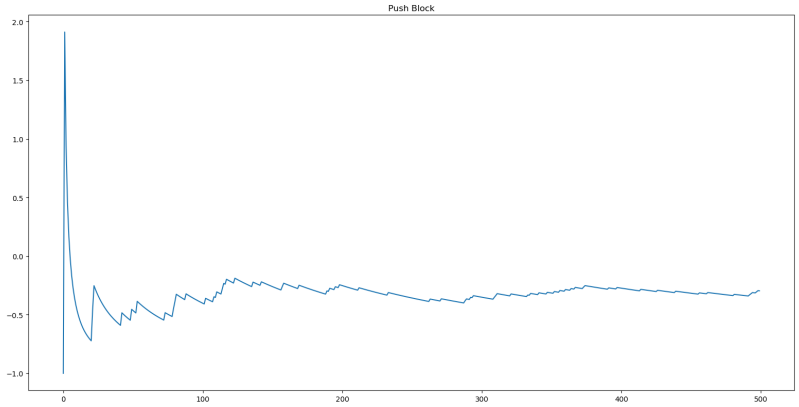

Figure 9: PPO result : Unity Push-Block Environment, episodes (x-axis) vs average score (y-axis)

6 Discussions and Conclusion

From our results, we can conclude that DDQN outperforms PPO model in these selected environments. As we are using random action selection strategies for both the models during the early training stages as mentioned in the implementation details (section 3), DDQN seems to be much more robust to domain noise than PPO. We believe that the online learning adopted by DDQN leads to its better performance over PPO which uses offline training scheme. Further, we observe that DDQN has higher data efficiency than Policy Gradients. Our results indicate that DDQN performs better with a lower number of episodes compared to PPO. As can be observed, the PPO model saturates at a lower average score compared to DDQN which rises slowly and reaches a stable and consistent performance. In the case of PPO, we observed poor result in the complex environment even after trying two different methods of training as explained in the experiment section 4.2. However, when training on the simple environment the average reward increases, with more training. We believe that in case of PPO longer training might lead to a better average score compared to DDQN. Moreover, we believe that improving the action space selection will improve the performance of the PPO agent.

Due to the time constraints, we could not do an extensive survey but we would like to increase training time for both the models. Additionally, we would like to introduce a curiosity factor as it has shown to be more successful with the complex exploration tasks of the environment. Further, we would like to implement and compare *Soft Actor Critic (SAC)* [10] and *Deep Deterministic Policy Gradient (DDPG)* [8] in which the agent learns from the trade-offs Between Policy Optimization and Q-Learning [9].

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark, 2020.

[3] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[4] Wu Menghao, Yanbin Gao, Alexander Jung, Qiang Zhang, and Shitong Du. The actor-dueling-critic method for reinforcement learning. *Sensors*, 19:1547, 03 2019.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[8] Josh Achiam OpenAI Spinning Up. Deep deterministic policy gradient. `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`.

[9] Josh Achiam OpenAI Spinning Up. Kinds of rl algorithms. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`.

[10] Josh Achiam OpenAI Spinning Up. Soft actor-critic. `https://spinningup.openai.com/en/latest/algorithms/sac.html`.

[11] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.