ELEG 6360: Modern Artificial Intelligence
Reinforcement Learning with OpenAI Gymnasium

Department of Electrical and Computer Engineering
Prairie View A&M University
Fall 2025

November 30th 2025

Jabin K Wade II
Submitted to
Dr. Lijun Qian

Table of Contents

Introduction

Open AI Gymnasium is a python toolkit used to develop and test Reinforcement Learning (RL) algorithms. It provides a standardized set of environments which we will be using to create agents with different RL algorithms and evaluate their performance. In this project, we have four tasks and five environments. Meaning, we will create and evaluate five agents against multiple tasks including different Atari games and robotic physics simulations. There are two main RL algorithms that we will be using to solve the tasks given to us in this project. Deep Q-Learning and Proximal Policy Optimization.

Our agent interacts with its environment $\boldsymbol{\varepsilon}$ in a sequence of actions, observations, and rewards. At each time step α_t the agent chooses from a set of defined game actions $A = \{0,\ldots,K\}$. Our agent can only observe the current image of the current screen at $\mathbf{x}_t$ and task, which makes it impossible for the agent to fully understand the current situation. Therefore, our agent must consider the past trajectory $\boldsymbol{\mathcal{T}}$ of actions and observations where $\boldsymbol{\mathcal{T}} = \mathbf{x}_1, \alpha_1, \mathbf{x}_2, \alpha_2, \ldots, \mathbf{x}_t, \alpha_{t-1}$ and then learn based on these sequences. This then allows us to turn our problem into a Markov Decision Process (MDP) which we can then apply RL algorithms to solve effectively.

Deep Q-Learning

The goal of our agent is to interact with the environment by selecting actions that maximizes the agent's total reward. Always assuming that future rewards are discounted by a value of $\boldsymbol{\gamma}$ over time t. Making our reward function $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$. Our optimal action function Q*(s, α) is the maximum expected return from the taken action. Q*(s, α) is a bellman equation to maximize the expected value of r + $\boldsymbol{\gamma}$Q*(s', α'). Making the equation $Q^*(s, \alpha) = \mathbf{E}_{s' \sim \boldsymbol{\varepsilon}} [r + \boldsymbol{\gamma} \max Q^*(s', \alpha') | s, \alpha]$. Using this equation, we can iteratively update our Q value faction and eventually converge to the optimal action selection. Unfortunately, in our case using only these equations will not work and we must use a non-linear function approximator for cases of neural networks or multi-layer perceptron. These can be trained by minimizing loss function $\mathbf{L}_i(\Theta_i)$ that change at each iteration i. After Differentiating the loss fuction with respect to the weights Θ. We arrive at the following equation…

$$\nabla_{\theta_i} \mathbf{L}_i(\theta_i) = \mathbf{E}_{s' \sim \boldsymbol{\mathcal{P}}(\cdot); s' \sim \boldsymbol{\varepsilon}} [(r + \boldsymbol{\gamma} \max Q^*(s', \alpha'; \Theta_{i-1}) - Q(s, \alpha; \Theta_i)) \nabla_{\theta_i} Q(s, \alpha; \Theta_i)]$$

If the weights are updated after each time step and expectations are replaced by samples from the behavior distribution $\boldsymbol{\mathcal{P}}$ and the environment $\boldsymbol{\varepsilon}$ we get the Q Learning algorithm. To streamline the implementation, I will be using Stable Baselines3 (SB3) Deep Q-Network then modifying the hyperparameters to fit my use case per task.

Proximal Policy Optimization

Similarly to Deep Q-Learning, The goal of our agent is to interact with the environment by selecting actions that maximizes the agent's total reward. Always assuming that future rewards are discounted by a value of $\boldsymbol{\gamma}$ over time t. The agent selects α_t based on its observations at $\mathbf{x}_t$. Creating a policy $\boldsymbol{\pi}(\alpha | \mathbf{x}_t; \Theta)$. Where Θ represents the neural network weights. The agents trajectory $\boldsymbol{\mathcal{T}}$ is stored to update this policy and maximize its reward by the end of the episode. This is formalized by the policy objective function, which is based on the advantage function $\mathbf{A}_t$. We then compute the probability ratio of the new policy to the old with the following equation $\mathbf{r}_t\,(\Theta)=\frac{\pi_\Theta(\alpha_t|s_t)}{\pi_{\Theta\text{old}}(\alpha_t|s_t)}$. Then, we can maximize the reward by updating the policy in a safer way by limiting how much the updated policy can diverge by ε from the prior policy in each step. We do this by computing the loss function…

$$\mathcal{L}^{CLIP}(\Theta) = \mathrm{E}_t[\min(r_t(\Theta)A_t, clip(r_t(\Theta), 1-\varepsilon, 1+\varepsilon)A_t$$

After computing this equation, the network weights Θ are updated using gradient ascent. Resulting in a steadily improving policy without sudden destabilizing shifts. Again, I will be using SB3 Proximal policy Optimization and modifying the hyperparameters to fit the use case per task.

Stable Baselines3 (SB3)

SB3 is a python library that provides implementations of RL algorithms. It includes high quality code and documentation for how to implement its RL algorithms. In this project I will be using it and modifying the hyperparameters of functions to try and get the best results from my agents. The following are explanations of all hyperparameters I used during this project.

- Learning Rate – The rate at which the network weights are updated during training. (lower = slow and steady, Higher = faster but instability)
- Gamma – Discount factor.
- Buffer Size – Number of experiences the agent has in its memory. Larger buffer means the agent has more experiences to learn from.
- Batch Size – number of experiences sampled from the buffer per training step.
- Train Frequency - number of steps between each training update.
- Target Update interval – Number of training steps between updates to the target network
- Learning Starts – Agent does not start training until this number of steps have been recached. Goal here is to fill the buffer with experiences for the agent to learn from
- Exploration Fraction - % of training time that the agent will choose epsilon greedy actions. Starting from 1.0
- Final Epsilon – The final value of the epsilon greedy at the end of training. Set to .01. By the end of training, model will always choose greedy action.
- N Steps – Number of steps to run for each environment per update.
- Entropy Coefficient – Encourages exploration by penalizing certainty in outputs.
- Clip Range – Clipping parameter, restricts policy updates to prevent large destabilizing updates. (smaller number = more conservative updates)
- GAE Lambda – Lambda parameter for advantage function. Bias-variance trade off
- Max gradient norm – Gradient clipping norm to avoid exploding gradients and stabilize training.
- Value Function coefficient – Balancing value function loss relative to policy loss in total loss.

Task 1: Discrete and Continuous Environments

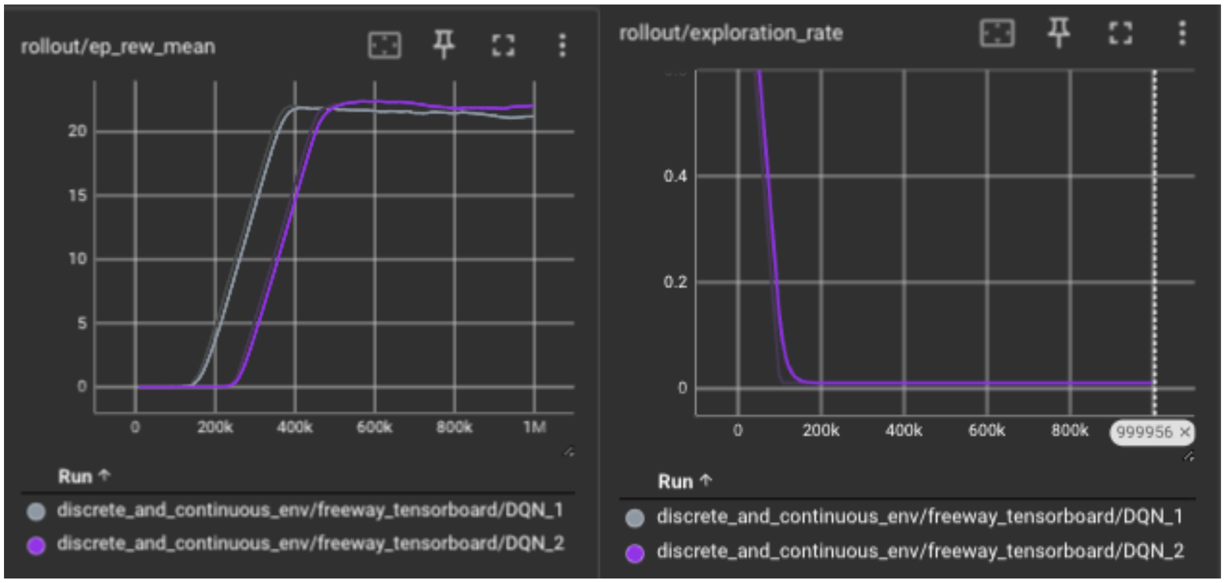
In this task we are given two google colab notebooks for the environments Atari Freeway and MuJoCo Ant. These environments are discrete and continuous respectively. In a discrete environment, there is a fixed number of possible states and actions, and time moves in distinct steps. In a continuous environment, there is an infinite number of actions and states the agent could be in and time flows continuously without any breaks.

Atari Freeway

For Atari Freeway, I implemented a DQN as it is a discrete environment with only four actions (Up, Down, Left, Right). It uses a convolutional neural network because its input is the Atari game screen. This screen is read a frame at a time and pixel by pixel with three color channels, with this large input it is too much for a MLP's basic feed forward network. For my training, I used SB3's baselined hyperparameters.

Parameter	Value
Learning Rate	0.00025
Gamma	.99
Buffer Size	100,000
Batch Size	32
Training Frequency	4
Target Update Interval	10,000
Learning Starts	50,000
Exploration Fraction	0.1
Final Epsilon	0.01

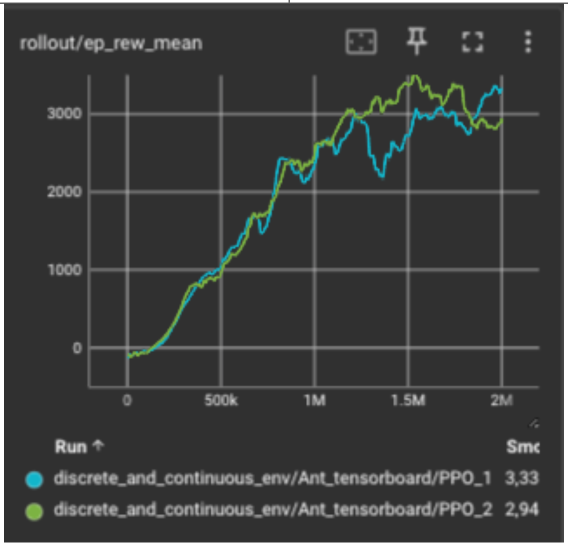
With these setting my agent was able to achieve a mean score of 22. Where it then seems, my model converges and gets optimal results.



MuJoCo Ant

MuJoCo Ant is a continuous environment where the agents' goal is to move forward as fast as possible in the positive x direction. This environment is well suited for PPO, and it is the algorithm I chose to complete this task. It uses a Multilayer Perceptron because it is better for modeling continuous actions as torques, joint angles, and velocities which is needed for the ant robot. For my training, I used a mix and combination of baselines found from OpenAI's Original PPO paper (OpenAI 2017) and SB3's published baselines

Parameter	Value
Learning Rate	0.0001
Gamma	.99
N_steps	512
Batch Size	32
Ent_coef	0.001
Clip Range	0.1
N_epochs	10
Gae_lambda	0.8
Max_grad-norm	0.6
Vf_coef	.677239

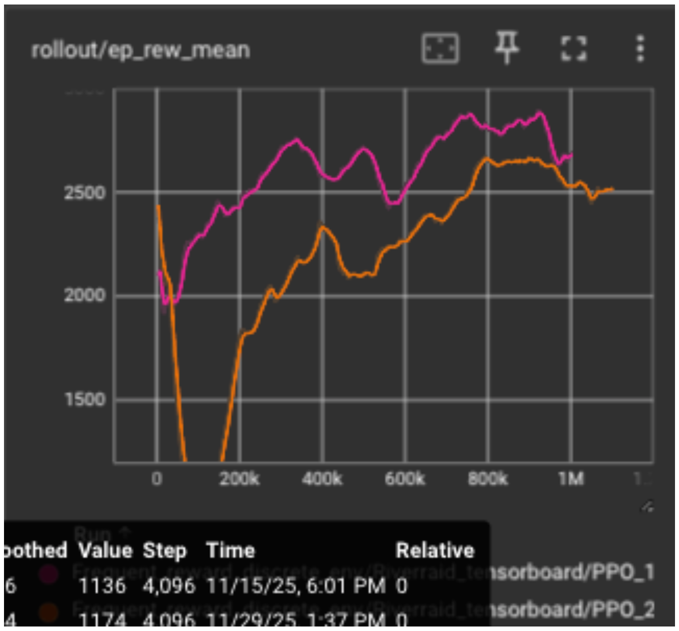


Task 2: Frequent Reward Discrete Environment

Atari Riverraid

Atari Riverraid is a discrete environment where the agent's goal is to maximize its game score. It increases its score by getting fuel, or destroying enemy objects like Helicopters, tankers, jets, and bridges. Since the input for our observation is the 2d game space, we again use a convolutional neural network. When initially attacking this task, I found that using a DQN did not produce good results. The agent would often fly straight into walls killing itself and ending the episode. I believe this is because how noisy the environment is with the ammount of rewards and actions the agent must take to get points. I then switched to a PPO network where I still did not see much of an increase in performance. The model would still actively fly into walls or fly only straight while shooting, getting some points but eventually flying into a wall or enemy killing itself. To fix this I decided to modify the rewards of the environment by adding a survival bonus reward. I implemented a function that after every 5 steps, the agent would get a small reward of .5 for surviving, then every 900 steps get a milestone reward of 10. This showed a great increase in my agents' performance by giving it extra incentive to stay alive along while also still prioritizing the game rewards.

Parameter	Value
Learning Rate	0.00025
Gamma	.99
Buffer Size	100,000
Batch Size	32
Training Frequency	4
Target Update Interval	10,000
Learning Starts	50,000
Exploration Fraction	0.1
Final Epsilon	0.01



```
othed  Value  Step   Time                 Relative
6      1136   4,096  11/15/25, 6:01 PM    0        Frequent_reward_discrete_env/Riverraid_tensorboard/PPO_1
4      1174   4,096  11/29/25, 1:37 PM    0        Frequent_reward_discrete_env/Riverraid_tensorboard/PPO_2
```

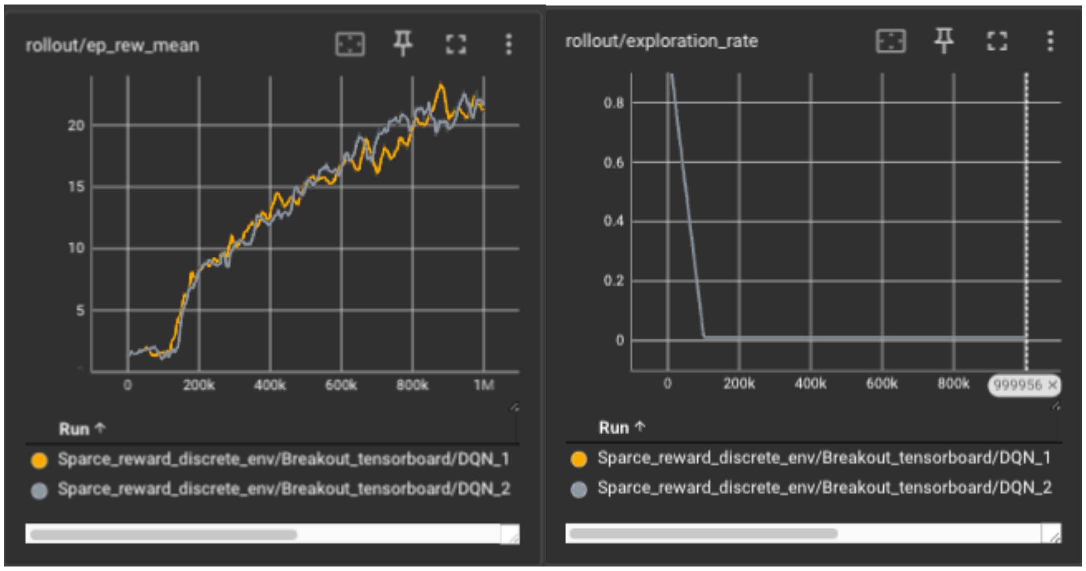
My results show that my model did not learn that well. My reward mean does not plateau and has a lot of hills and valleys. Increasing training time only resulted in catastrophic forgetting and the rewards would go back down towards 0.

Task 3: Sparce Reward Discrete Environment

Atari Breakout

Atari Breakout is another discrete environment with the goal of the agent to maximizing the game score. The game space observation is inputted to the model again as a image making this again a CNN policy. For my hyperparameters, I used the SB3 baseline I had used in the same Atari Freeway game and results which passed the task and was able to get a average reward of 21 points. I ran into a problem where when inferencing the model, the model had problems shooting the ball or would immediately go to the right side of the screen and not fire or stay there and let the ball drop with no attempt to save. I fixed this by adding small tweaks to the inferencing code. I modified the code so that when the game detects a loss of a life, it would automatically fire the next ball, then call a NOOP command for 4 steps (about a half second or so) so the model can correctly observe the game space with no actions. After these additions, the model performed perfectly. Our reward curve does not plateau after 1 million steps of training. However, It is still seems to be in early stages of rising and would perform better with more training steps. Our model could be close to convergence but due to time, we will settle with our high score of 19.

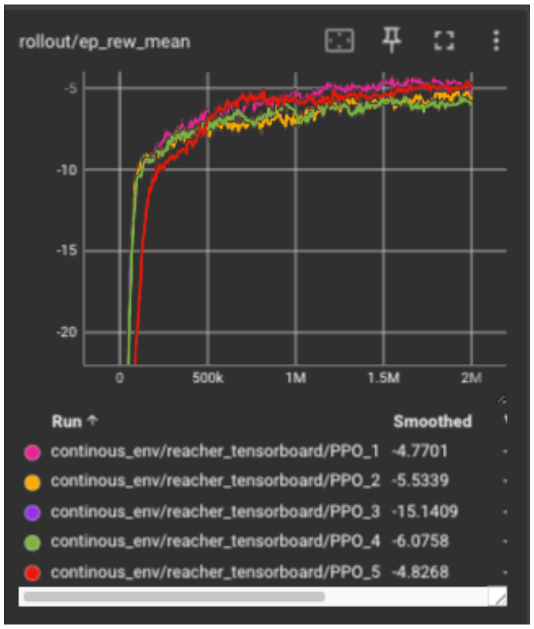
Parameter	Value
Learning Rate	0.00025
Gamma	.99
Buffer Size	100,000
Batch Size	32
Training Frequency	4
Target Update Interval	10,000
Learning Starts	50,000
Exploration Fraction	0.1
Final Epsilon	0.01



Task 4: Continuous Environment

MujoCo Reacher

MuJoCo Reacher is a continuous environment where the goal is to have a two jointed robot arm as close to the target that is spawned at a random position in the game space. The agent has 50 timesteps to complete this goal. If it doesn't it fails. To solve this, I used a PPO algorithm similar to the one I used for MuJoCo Ant. My reward curve looks as expected with a plateau at the top showing that we've converged and met optimal or close to optimal performance.



Run ↑	Smoothed
● continous_env/reacher_tensorboard/PPO_1	-4.7701
● continous_env/reacher_tensorboard/PPO_2	-5.5339
● continous_env/reacher_tensorboard/PPO_3	-15.1409
● continous_env/reacher_tensorboard/PPO_4	-6.0758
● continous_env/reacher_tensorboard/PPO_5	-4.8268

Conclusion

In this project we learned how to implement different reinforcement learning algorithms to play various Atari and MujoCo games. I learned the differences in continuous verses discrete environments and how to use Deep Q Learning or Proximal Policy Optimization to solve them and the differences between the two. On every task, I found myself asking, "Do I need my agent to explore more, or exploit more?" Once I knew the answer I could modify my hyperparameters or figure out a way to eventually solve the problem. This project was an amazing learning experience, and I am very happy to be able to capstone the semester by completing it. I learned a lot about the methods of DQN and PPO in a real world practical manor and understood how to read the outputs of the tensor board to figure out what I needed to modify with my model. In the end, I found myself applying more and more of what I learned in class to the code I was implementing.

Reference:

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. arXiv preprint arXiv:1312.5602

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.