CS4246 Project Report: Understanding the Impact of an Adversarial Agent on Training a Reinforcement Learning Agent

Github Link: https://github.com/vincentaurellio/CS4246

Matthew Antonio Wijaya
A0244490E
e0866231@u.nus.edu

Vincent Aurellio Budianto
A0244496U
vincent.aurellio@u.nus.edu

November 24, 2024

Contents

1 Introduction

1.1 Background and Problem Statement

Reinforcement learning (RL) has demonstrated its effectiveness in solving various decision-making and control problems. However, real-world applications often involve substantial randomness and uncertainty, which can adversely impact the performance of RL agents. These challenges may arise from environmental stochasticity, adversarial interference, or unforeseen dynamic changes.

This project aims to investigate whether standard RL training techniques are sufficient to equip agents to handle such challenges and to explore methods for enhancing their robustness. Specifically, we introduce an adversarial agent into a highly stochastic GridWorld environment to analyze the impact of adversarial dynamics on training and to evaluate the adaptability of RL agents in these scenarios.

The central question we aim to address is: *What happens when an RL agent operates in a highly stochastic environment? Will the agent be able to handle it effectively?* To answer this, we designed an experiment comparing the performance of a standard RL algorithm in a highly stochastic environment against its performance when trained alongside an adversarial agent.

1.2 Objectives

The objectives of this project are as follows:

1. To apply RL algorithms introduced in class to a practical problem.
2. To modify and extend these algorithms to support multi-agent scenarios, incorporating adversarial dynamics.
3. To analyze and understand the impact of adversarial agents on the training process and performance of RL models.

2 Related Works

Our project aims to investigate the impact of adversarial environments on enhancing the robustness of RL agents. This topic has been extensively explored in prior research, which has inspired the direction of our work.

For instance, Pinto et al. [1] explored robust adversarial reinforcement learning in complex 3D Gym environments, demonstrating how adversarial training can improve the resilience of RL agents. Similarly, Gleave et al. [2] studied adversarial policies in simulated environments, highlighting the vulnerabilities of RL agents to adversarial perturbations. Pan et al. [3] focused on robust adversarial reinforcement learning, providing strategies for agents to better navigate hostile environments. In contrast, Huang et al. [4] examined adversarial dynamics in simpler domains like Atari games, showcasing how adversarial policies can exploit weaknesses in agent behavior.

Despite these significant contributions, research on adversarial RL in highly stochastic environments remains relatively unexplored. Therefore, our project seeks to address this gap by analyzing the effects of adversarial interventions in a simpler, yet highly stochastic, environment.

3 Methodology

3.1 Environment Design

We designed our environment using a GridWorld setting, creating a highly customized environment tailored specifically for the objectives of this project. The maze consists of one player, multiple walls, and a single goal. The primary objective for the player is to reach the goal while minimizing the number of steps taken. However, a unique challenge is introduced: the positions of the walls are scrambled after each action taken by the player. This dynamic behavior necessitates careful planning and adaptability from the RL agent.

3.1.1 Maze and Movement Mechanics

The player can move one square at a time in any of the four cardinal directions: up, down, left, or right. After every player move, the walls are randomly repositioned within the maze. To implement this, we developed separate methods for handling player movements and wall scrambling, as illustrated in the diagram below.

3.1.2 Ensuring Maze Solvability

One major challenge in this setup was ensuring the maze remained solvable despite the randomized wall scrambling. To address this, we implemented a self-generating path algorithm. This algorithm dynamically generates a guaranteed solvable path between the player's starting position and the goal.

The self-generating path algorithm works as follows:

1. From the player's starting position, the algorithm makes a series of random moves with variable step sizes.
2. After a few steps, the algorithm creates a direct path to the goal by prioritizing either the x-direction or y-direction.
3. Any walls obstructing this generated path are removed to ensure the maze is solvable.

This approach guarantees a solvable maze while maintaining the dynamic and challenging nature of the environment. Additionally, it ensures that adversarial agents do not create unsolvable mazes during training.

3.1.3 Gym Environment Wrapper

To streamline integration and standardization, we encapsulated our environment within a custom OpenAI Gym-compatible wrapper. This wrapper allows for seamless interaction with RL libraries and facilitates retesting and experimentation. By adhering to the Gym interface, we ensure that the environment is reusable and provides a robust foundation for further experiments, enabling enhancements to the RL algorithms.

3.2 Reinforcement Learning Agent

As one of our objectives is to apply RL algorithms from class, we selected the Actor-Critic algorithm coupled with Proximal Policy Optimization (PPO). These algorithms represent a robust and widely used approach in modern RL settings. Their flexibility allows for customization, such as tuning the neural network, balancing exploration and exploitation, and adapting to the requirements of a dynamic and adversarial environment like ours. Below, we discuss the components in detail and highlight the adjustments made to suit our project.

3.2.1 Actor-Critic Algorithm

The Actor-Critic algorithm is a foundational method in RL that combines two key components:

- **Actor**: Responsible for selecting actions based on a policy. It takes the current state as input and outputs the probability distribution over possible actions. This component is optimized to maximize the expected reward.
- **Critic**: Evaluates the actions taken by estimating the value of the current state or state-action pair. The value function helps guide the actor by providing feedback on whether the chosen actions are beneficial.

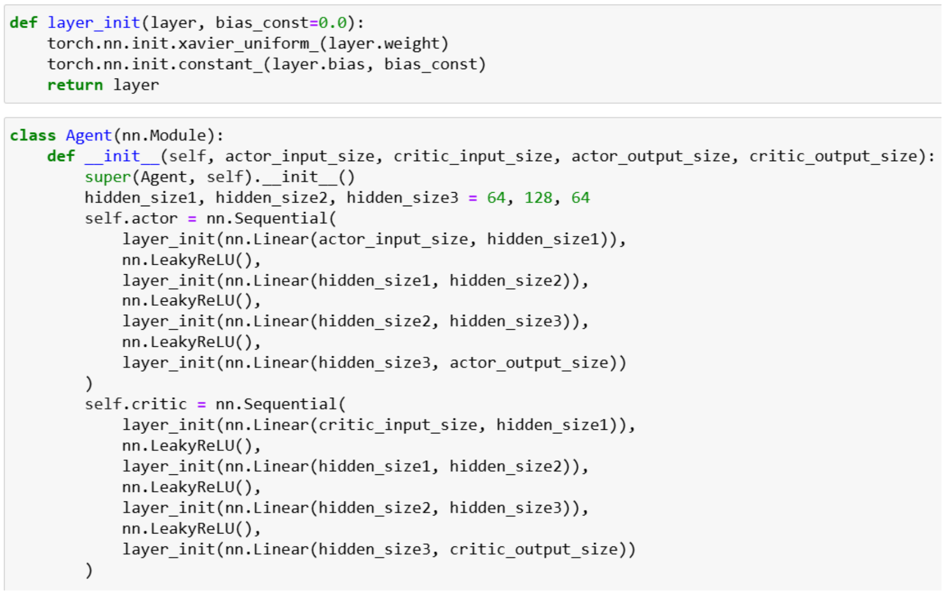
The integration of these components creates a feedback loop where the actor learns to improve its policy based on the critic's evaluation. This setup allows the agent to balance short-term and long-term rewards, making it particularly suited to environments with stochastic and adversarial elements, such as our GridWorld.

Our actor and critic networks are implemented using deep learning models tailored to our project requirements. Both models consist of three linear layers followed by an output layer. Below, we detail the tuning choices for these layers and the rationale behind each decision:

1. **Activation Function**: We use the Leaky ReLU activation function to prevent the vanishing gradient problem and to handle negative values more effectively. Unlike standard ReLU, Leaky ReLU introduces a small, non-zero gradient for negative inputs, ensuring better flow of gradients during training.

2. **Regularizer**: To mitigate overfitting in our highly stochastic environment, we incorporate regularization techniques. By doing so, we prevent the model from overfitting to noise and improve its generalization capabilities. In our implementation we used weight decay to ensure our agent is not overfitting.
3. **Initialization**: We use Xavier initialization (Glorot initialization) to set the initial weights of the network. This method helps ensure that the weights are neither too large nor too small, promoting stable training by maintaining a balance between input and output variances across layers.

Below is the code snippet for the actor and critic networks:



```
def layer_init(layer, bias_const=0.0):
    torch.nn.init.xavier_uniform_(layer.weight)
    torch.nn.init.constant_(layer.bias, bias_const)
    return layer
```

```
class Agent(nn.Module):
    def __init__(self, actor_input_size, critic_input_size, actor_output_size, critic_output_size):
        super(Agent, self).__init__()
        hidden_size1, hidden_size2, hidden_size3 = 64, 128, 64
        self.actor = nn.Sequential(
            layer_init(nn.Linear(actor_input_size, hidden_size1)),
            nn.LeakyReLU(),
            layer_init(nn.Linear(hidden_size1, hidden_size2)),
            nn.LeakyReLU(),
            layer_init(nn.Linear(hidden_size2, hidden_size3)),
            nn.LeakyReLU(),
            layer_init(nn.Linear(hidden_size3, actor_output_size))
        )
        self.critic = nn.Sequential(
            layer_init(nn.Linear(critic_input_size, hidden_size1)),
            nn.LeakyReLU(),
            layer_init(nn.Linear(hidden_size1, hidden_size2)),
            nn.LeakyReLU(),
            layer_init(nn.Linear(hidden_size2, hidden_size3)),
            nn.LeakyReLU(),
            layer_init(nn.Linear(hidden_size3, critic_output_size))
        )
```

Figure 1: Agent Actor Critic

3.2.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) builds on the Actor-Critic framework and introduces a novel approach to stabilize policy updates. In traditional policy gradient methods, large updates to the policy can lead to instability or suboptimal performance. PPO addresses this issue by:

- **Clipping Policy Updates**: It limits the change in the policy by applying a clipped objective function, ensuring that updates remain within a defined range. This prevents overly aggressive changes that might destabilize learning.
- **Surrogate Objective Function**: PPO optimizes a surrogate objective that balances exploration and exploitation while maintaining policy stability. It ensures that the new policy is not significantly different from the old policy.

These features make PPO computationally efficient and robust, allowing it to handle complex environments like ours effectively.

To compute the advantage function $\hat{A}_t$, we use Generalized Advantage Estimation (GAE):

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

Where:

- $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: The temporal difference error.
- γ: The discount factor.
- λ: The GAE parameter, controlling the trade-off between bias and variance.

The clipped surrogate objective (actor loss) is defined as:

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

Where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}$: The probability ratio of the new policy π_θ to the old policy $\pi_{\theta_{\mathrm{old}}}$.
- $\hat{A}_t$: The estimated advantage function at timestep t.
- ϵ: The clipping parameter, typically a small value (e.g., 0.1 or 0.2), which restricts how much $r_t(\theta)$ can deviate from 1.

The clipping operation ensures that the policy update stays within a trust region by restricting $r_t(\theta)$ to the range $[1-\epsilon, 1+\epsilon]$, promoting stable training and penalizing overly large updates.

The clipped value function loss (critic loss) is defined as:

$$L^{\mathrm{VF}}(\theta) = \mathbb{E}_t\left[\max\left(\left(V_\theta(s_t) - V_t^{\mathrm{target}}\right)^2,\, \left(\mathrm{clip}\left(V_\theta(s_t), V_{\mathrm{old}}(s_t)-\epsilon, V_{\mathrm{old}}(s_t)+\epsilon\right) - V_t^{\mathrm{target}}\right)^2\right)\right]$$

Where:

- $V_\theta(s_t)$: The predicted value of state s_t under the current policy.
- V_t^{target}: The target value for state s_t, typically computed as $r_t + \gamma V_\theta(s_{t+1})$.
- $V_{\mathrm{old}}(s_t)$: The value predicted by the previous policy iteration.

The total loss for PPO includes additional terms for value function loss and entropy bonus:

$$L(\theta) = \mathbb{E}_t\left[L^{\mathrm{CLIP}}(\theta) - c_1 L^{\mathrm{VF}}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

Where:

- $S[\pi_\theta](s_t) = -\sum_a \pi_\theta(a|s_t)\log\pi_\theta(a|s_t)$: The entropy of the policy, which promotes exploration.
- c_1, c_2: Coefficients to balance the contributions of the value loss and the entropy bonus.

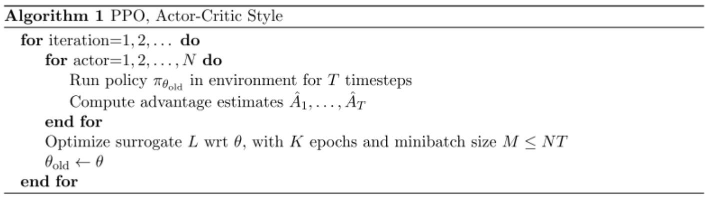
The policy gradient update is computed as:

$$\theta \leftarrow \theta + \alpha\nabla_\theta L(\theta)$$

Where:

- α: The learning rate.
- $\nabla_\theta L(\theta)$: The gradient of the total PPO loss.

This is the algorithm for PPO:



```
Algorithm 1 PPO, Actor-Critic Style
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\mathrm{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \le NT$
    $\theta_{\mathrm{old}} \leftarrow \theta$
end for
```

Figure 2: PPO Algorithm

Source: Proximal Policy Optimization Algorithms, Schulman et al. [5]

3.2.3 Agent Reward

The reward function plays a critical role in guiding the agent's learning process. In our setup:

- The agent receives a reward of 50 × (maze length + maze width) when it reaches the goal. This high reward incentivize the agent to focus on successfully completing the maze.
- A penalty equal to the negative Manhattan distance from the agent's new position to the goal is applied after each move. This encourages the agent to minimize its distance to the goal and discourages unnecessary exploration.

This reward structure primarily incentivize the agent to minimize the distance to the goal and rewards it significantly upon reaching the objective.

3.3 Adversarial Agent

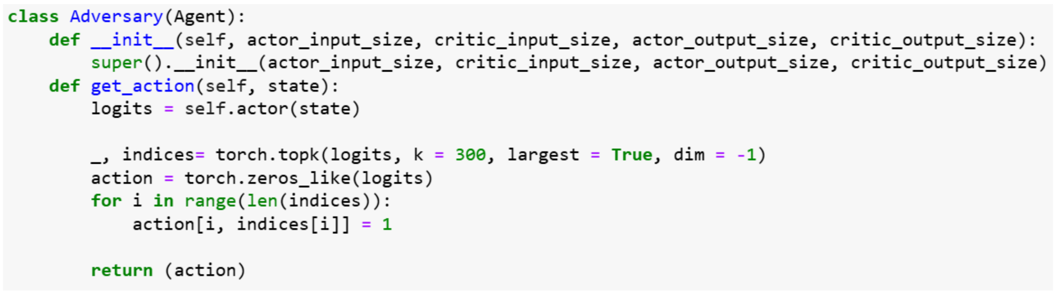
To see the effects of adversarial training in accordance to our objectives, we employed an adversarial agent during training for one of our agents. The adversarial agent design is similar to the RL Agent with some modifications to better suit its purpose. The adversarial agent. Below we will discuss the components of the adversarial agents in detail and highlight the adjustments made to suit our project.

3.3.1 Modified Actor-Critic Algorithm

The adversarial agent uses the same Actor-Critic algorithm as the RL Agent, with modifications on the Actor.

1. **Actor**: Responsible for selecting actions based on a deterministic policy. It takes the path that will be generated by the path generating algorithm as input and outputs the set of walls that will be generated on the maze. This is done by selecting the top k values based on the output values of the Actor Neural Network, where k is the number of walls to be placed in the maze. This component is optimized to minimize the RL Agent's expected rewards.
2. **Critic**: Evaluates the actions taken by estimating the value of the current state or state-action pair. The value function helps guide the actor by providing feedback on whether the chosen actions are beneficial.

We modified the actor component to be deterministic because of difficulty in fitting a known probability distribution to our desired input and output. By modifying the actor component, we ensure that the actor and critic component are able to work properly in creating a feedback loop to improve our adversarial agent in a similar way to the RL Agent, creating an adversary that learns alongside the agent. Below is the architecture of the adversarial agent's actor, which has been adjusted to accommodate this task:

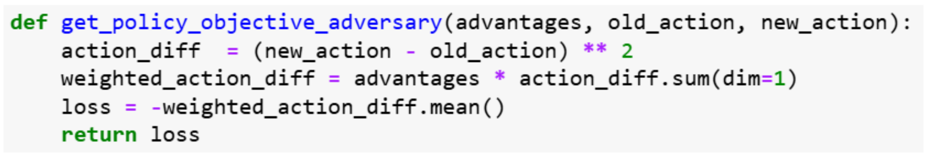


```python
class Adversary(Agent):
    def __init__(self, actor_input_size, critic_input_size, actor_output_size, critic_output_size):
        super().__init__(actor_input_size, critic_input_size, actor_output_size, critic_output_size)
    def get_action(self, state):
        logits = self.actor(state)

        _, indices= torch.topk(logits, k = 300, largest = True, dim = -1)
        action = torch.zeros_like(logits)
        for i in range(len(indices)):
            action[i, indices[i]] = 1

        return (action)
```

Figure 3: Adversary Agent

3.3.2 Proximal Policy Optimization (PPO)

The adversarial agent uses the same PPO algorithm as the RL Agent with some modifications on the Actor Loss Function, since there is no log probability for a deterministic policy. We used the squared difference of wall placement multiplied by minus advantage for the loss.



```
def get_policy_objective_adversary(advantages, old_action, new_action):
    action_diff  = (new_action - old_action) ** 2
    weighted_action_diff = advantages * action_diff.sum(dim=1)
    loss = -weighted_action_diff.mean()
    return loss
```

Figure 4: Adversary Loss Function

3.3.3 Reward Function

To make this a zero sum game we made the adversarial agent's reward to equal to minus the RL Agent's reward. This reward structure is designed to make sure the game is fair for the agents and to incentivize the adversarial agent to minimize the RL Agent's reward.

3.4 Experiment Design

3.4.1 Training

Training on the agents are performed on a 20 x 20 GridWorld maze as described in training environment section with start being placed at (3, 3) and goal at (18, 18) for every episode. After the agent and the maze is initialized, the agent takes in the state of the maze and chooses an action and will get rewarded based on the reward function for the action. After an action is taken, all the states, actions and rewards will be recorded in a buffer. An episode will end when the agent reaches the goal or when the truncation limit is reached. At the end of each episode, the agent will perform updates based on the PPO algorithm using the values stored in the buffer.

For the agent trained with the adversarial agent, there will be slight modifications to the training process. The agent and the adversarial agent will take turns in taking their actions, starting with the adversarial agent placing the walls and receiving reward for the wall placement, followed by the agent taking the wall placement as input and choosing where to move. The agent will then get rewarded based on the move and the process is repeated. At the end of each episode, both agents are updated based on their own update methods as explained in previous sections.

The training is parallelized by initializing 16 environments and running them concurrently. When an environment has reached the end of their episode, it will wait for the others to finish before using information obtained from each environment to perform the episodic update. The training will end when the allocated number of steps is reached.

3.4.2 Agent Tuning

To further expand on the effects of parameters of our agents in the performance of the agent, we will also be tuning our agents. The parameters tuned are:

1. **Adversary Addition**
   Employing an adversary agent to control the wall maze instead of a stochastic maze. In accordance to our goal in this project, we want to see the effects of adversary in training, so we made our baseline agent trained in the stochastic maze and another agent trained with the adversarial agent controlling the maze for comparison.

2. **Entropy Coefficient**
   Responsible for the agent's tendency to explore compared to exploit. We want to see the effects of this parameter on our agent, so we employed our baseline agent with a higher entropy coefficient of 0.1 and another agent with a lower entropy coefficient of 0.01 for comparison.

3. **Truncation**
   Terminates an episode when a certain limit is reached, even if the agent have not reached the goal. We are also interested in the effects of implementing truncation during our training, so we employed our baseline agent without truncation and we employed another agent with truncation for comparison. The truncation limit is set at five times the size of the maze.

4. **Training Steps**
   Total number of steps that the agent will be trained across all the parallel environments. We also want to observe if the number of training steps can affect our agent's performance, so for each agent mentioned previously we made two variations, one with lesser training steps and one with

more training steps. The baseline training steps will be set at 3200000 and the lesser training steps will be set at 800000.

3.4.3 Evaluation Metrics

For evaluating our agents, we will be using the following metrics.

1. **Mean Rewards**: Average amount of reward achieved across all episodes of testing. This metric evaluates the performance of the agent by measuring how close it is able to get to the goal on average.
2. **Goals Reached**: Number of times the agent is able to reach the goal across all episodes of testing. This metric evaluates the agent by observing how many times it is actually able to reach the goal.

4 Experiments and Results

4.1 Experimental Setup

Testing will be performed on 4 agents (Baseline Agent, Agent with Adversary, Agent with Less Entropy, Agent with Truncation) and an additional 4 agents using the same configuration but with a reduced number of training steps.

Testing will also be performed on various environments.

1. **Training Environment**: Exact same start and goal setup as training.
2. **Directional Start and Goal**: The goal will be slightly random in one of four general directions relative to the start (Up Right, Up Left, Down Right, Down Left).
3. **Swapped Start and Goal**: Start and goal positions will be swapped from the training positions, so the start is at (18, 18) and the goal is at (3, 3).
4. **Random Start and Goal**: Start and goal positions is set completely at random.

From the agents and environments mentioned above, all the agents will be tested on all the environments. Testing will be performed by initializing the selected agent inside the selected environment and running the episode until either the agent reaches the goal or a step limit of 5000 is reached. If the episode is terminated by the step limit, all the rewards achieved during the episode will be averaged for the mean rewards metric. This will be repeated for 10 times for each agent and environment choice. The performance of each agent in each environment will then be collected and measured based on previously mentioned evaluation metrics.

4.2 Results

Agent	Training Environment	Swapped Goal and Start	Random Right Up	Random Down Left	Random Down Right	Random Up Left	Random Environment
Base	-94354.7	-32440.4	-78536.8	-51251.0	-145533.9	-13841.7	-82185.1
Adversarial	-75600.0	-9206.1	-27256.4	-54823.4	-52801.8	-61576.2	-30136.7
Truncated	-59692.4	-26816.8	-67310.1	-41434.0	-30474.4	-145610.0	-62310.3
Less Exploration	-95139.0	-95151.1	-81137.2	-100502.9	-157471.8	-22170.5	-93336.8

Table 1: Mean Rewards Normal Training Steps Agents

Agent	Training Environment	Swapped Goal and Start	Random Right Up	Random Down Left	Random Down Right	Random Up Left	Random Environment
Base	0	0	0	0	0	3	0
Adversarial	0	7	2	0	0	0	3
Truncated	0	0	0	0	0	0	0
Less Exploration	0	0	0	0	0	3	0

Table 2: Goals Reached Normal Training Steps Agents

Agent	Training Environment	Swapped Goal and Start	Random Right Up	Random Down Left	Random Down Right	Random Up Left	Random Environment
Base	-44399.3	-26840.1	-37823.2	-18916.8	-10726.7	-129721.0	-42540.2
Adversarial	-116974.4	1482.3	-101224.8	-1366.2	-70800.3	-70053.4	-71122.8
Truncated	-85074.0	-24300.3	-79829.0	-21977.5	-11909.9	-123216.1	-47409.6
Less Exploration	-95157.1	-95034.4	-90594.3	-94028.8	-24825.3	-156453.0	-80757.0

Table 3: Mean Rewards Less Training Steps Agents

Agent	Training Environment	Swapped Goal and Start	Random Right Up	Random Down Left	Random Down Right	Random Up Left	Random Environment
Base	0	0	1	4	5	0	3
Adversarial	0	10	0	9	0	0	1
Truncated	0	0	0	1	3	0	1
Less Exploration	0	0	0	0	2	0	1

Table 4: Goals Reached Less Training Steps Agents

4.3 Robustness Analysis

As evident in the results, the agent trained in adversarial environment consistently performs better compared to other agents and this is especially evident in cases that differ largely from the training environment, such as the Random Start and Goal environment. This proves that adversarial environment helps in improving the robustness of an agent, allowing it to generalize better to unseen cases compared to agents that are not trained in adversarial environment. Adversarial Environment is designed to emulate the worst outcome possible for a random environment such as our GridWorld maze. This might be the reason why it is able to handle unseen cases better, since it is already trained on the worst case.

5 Discussion

5.1 Insights on Agent Tuning

5.1.1 Adversarial Training

As evident from the results, agent trained in adversarial environment perform better on unseen test cases compared to other agents, such as in the Random Start and Goal environment. Ultimately, this test case is our main test for generalization, since we aim for this environment to simulate the real world where anything can happen and the environment might differ largely from the training environment which the agent has seen. We think that by performing well in this test, the agent has proved itself to be capable of generalization, even in extreme cases.

We also think that the adversary enables a more focused learning, since it is putting the walls to hinder the agent on purpose, thus the agent learns better on how to overcome this. In contrast, our baseline agent without the presence of adversary, might just fit to the noise of the stochastic wall movements instead, which results in the agent not reaching the goal.

5.1.2 Entropy Coefficient

We can see from the results that the agent with less entropy coefficient consistently performs the worst in all test cases. We think it is caused by the highly stochastic nature of our environment which favors exploration over exploitation, since the environment changes at a rapid pace which makes it difficult to learn how to exploit the state.

5.1.3 Truncation

From the results we can observe that the agent with truncation consistently performs better than our baseline agent in terms of mean rewards. However, our truncated agent reaches the goal significantly less compared to the baseline agent. We think that this is caused by the truncation hindering the agent's learning. The agent only learns how to get closer to the goal, as evident by the good mean rewards performance, but fails to actually reach the goal, as evident in goals reached performance. This is possibly caused by the episode getting truncated before the agent reaches the goal, which causes the agent to not learn how to actually get to the goal.

5.1.4 Training Steps

As we can observe from the results, in the case of our baseline agent, when trained with less training steps, the agent performs better on similar environments to the training environment. However, the performance significantly drops when the agent with less training steps is tested on an unseen environment. We think this might be caused by a simpler policy that is learned compared to a more complex one that is learned over more training steps. The simpler policy performs better on the seen environments because our environment is highly stochastic more complex policy will just overfit on the noise. However, on unseen environments, the simple policy is unable to account for the drastic change and thus performs significantly worse compared to more complex policies.

5.2 Challenges and Limitations

5.2.1 Challenges

Even though we managed to achieve the goals we set for this project, it is not without its challenges.

1. **Environment Design**
   In designing the environment, we want to make a stochastic maze while still ensuring it is solvable. We finally settled with our path generating algorithm to solve this challenge, however it is not perfect. It only generates a single path, and it is rather streamlined so there is room for improvement.
2. **Adversarial Agent Algorithm**
   We settled for a deterministic adversary because we were not able to fit any known distribution that are compatible with the Gym wrapper for our desired input and output.
3. **Defining Adversarial Behavior**
   Designing the adversarial agent's reward function to ensure it effectively hindered the main agent without making tasks unsolvable was challenging. Ensuring the adversary introduced meaningful obstacles rather than arbitrary disruptions required iterative adjustments and testing.

5.2.2 Limitations

From the challenges explained above, we attempted to solve it to the best of our ability. However, our agents still fell short of our expectations where most agents are not able to reach the goal consistently.

1. **Knowledge**
   In this project we tried to implement the materials taught in class to the best of our ability, complemented with our own research and experiments. However, we are not experts in this subject and some aspects of this project are lacking due to our limited knowledge.
2. **Time and Resources**
   Due to limited time and resources that we have access to, our agents might not have performed to

our expectations.

5.3 Implications for Real-World Applications

As demonstrated by our project and prior research, adversarial environments can significantly enhance the robustness of RL agents across various settings. This suggests that such approaches could be adopted in real-world RL applications to improve an agent's ability to handle unseen and unpredictable scenarios. By fostering resilience and adaptability, this method can contribute to the development of agents that generalize effectively and perform consistently across diverse and dynamic situations—an essential requirement for real-world applications.

6 Conclusion

Based on the results of our project, we can conclude that the presence of an adversarial agent during training can improve the robustness of a RL agent. RL agent also requires high entropy coefficient to solve problems in highly stochastic environments, such as ours. We also found that truncation hinders the RL agent's learning because it truncates an episode before the agent reaches the goal. Finally, we also found that to generalize better, RL agent need to be trained over more training steps to be able to learn more complex policies that can generalize better to unseen environments.

7 Future Works

To address the limitations of our project and to expand further on the topic of our project, we propose some future improvements.

1. **More Training**
   From our project we can see that more training improves an agent's ability to adapt to unseen environments. However, due to time and resource constraints we were unable to do this. This can be a good starting point to improve our project.
2. **Less Stochastic Environment**
   From the results of our project we can see that our agents are still unable to fully solve this stochastic maze problem. Our highly stochastic maze might be to difficult to find patterns in, therefore it is worth attempting to see if a less stochastic environment is better for agent learning.
3. **Different Adversarial Algorithm**
   Due to our limited knowledge, we implemented a deterministic adversary for our project. This provides room for improvement by trying different adversary designs and algorithms that might perform better.

References

1. Pinto L, Davidson J, Sukthankar R, and Gupta A. Robust Adversarial Reinforcement Learning. 2017 Mar. Available from: `http://arxiv.org/abs/1703.02702`
2. Gleave A, Dennis M, Wild C, Kant N, Levine S, and Russell S. Adversarial Policies: Attacking Deep Reinforcement Learning. 2019 May. Available from: `http://arxiv.org/abs/1905.10615`
3. Pan X, Seita D, Gao Y, and Canny J. Risk averse robust adversarial reinforcement learning. Proceedings - IEEE International Conference on Robotics and Automation 2019 Mar; 2019-May:8522–8. DOI: `10.1109/ICRA.2019.8794293`. Available from: `http://arxiv.org/abs/1703.02702`
4. Huang S, Papernot N, Goodfellow I, Duan Y, and Abbeel P. Adversarial Attacks on Neural Network Policies. 2017 Feb. Available from: `http://arxiv.org/abs/1702.02284`
5. Schulman J, Wolski F, Dhariwal P, Radford A, and Klimov O. Proximal Policy Optimization Algorithms. 2017 :1–12. Available from: `http://arxiv.org/abs/1707.06347`

8 Appendix

8.1 Training Plots

8.1.1 Base Agent

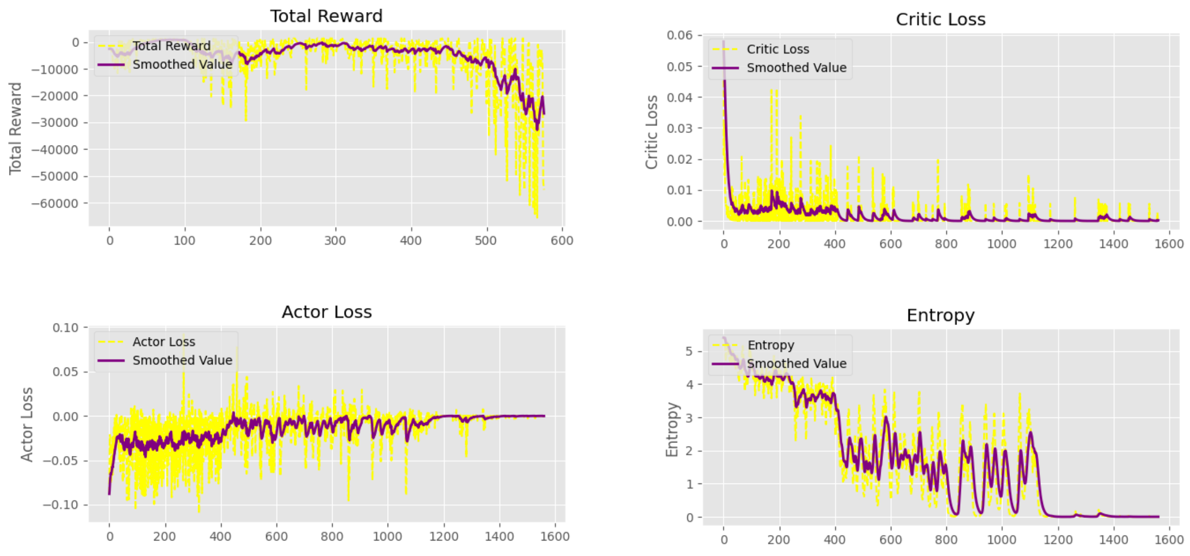


Figure 5: Base Model Training Plot

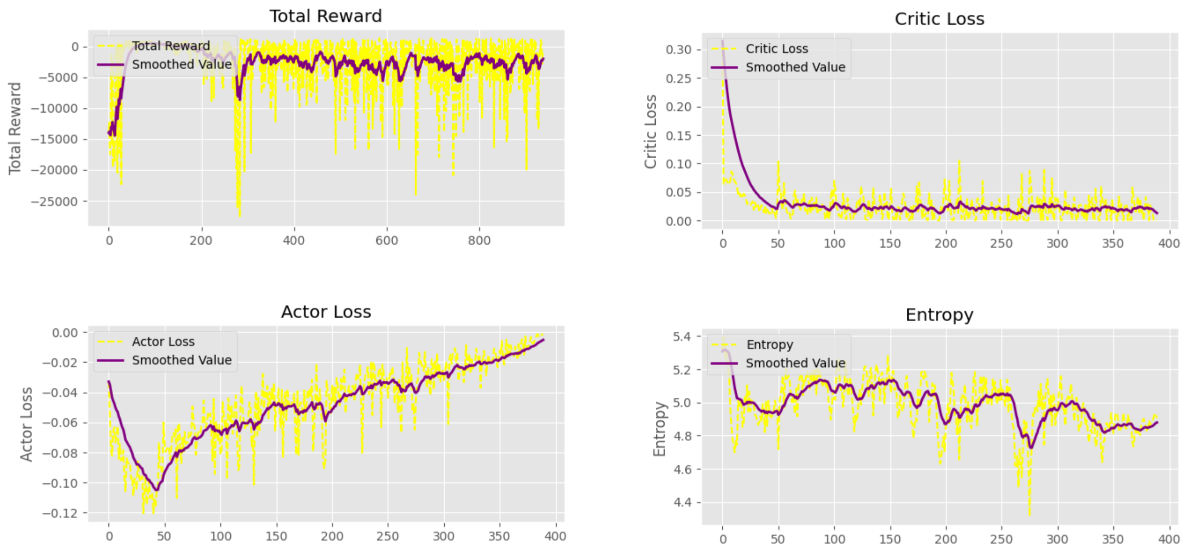


Figure 6: Base Model Training Plot (less step)

8.1.2 Agent with Adversary

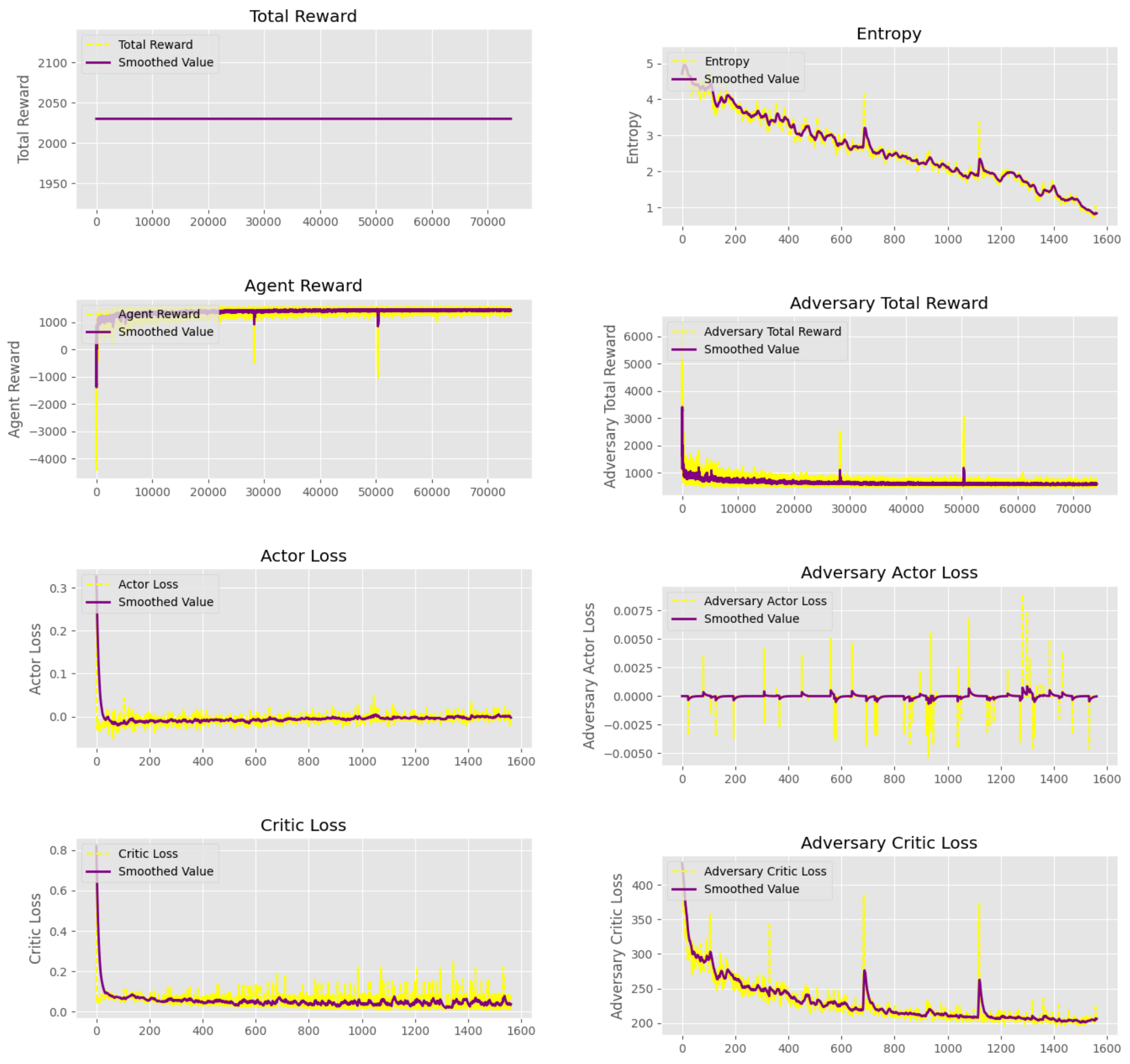


Figure 7: Adversary Model Training Plot

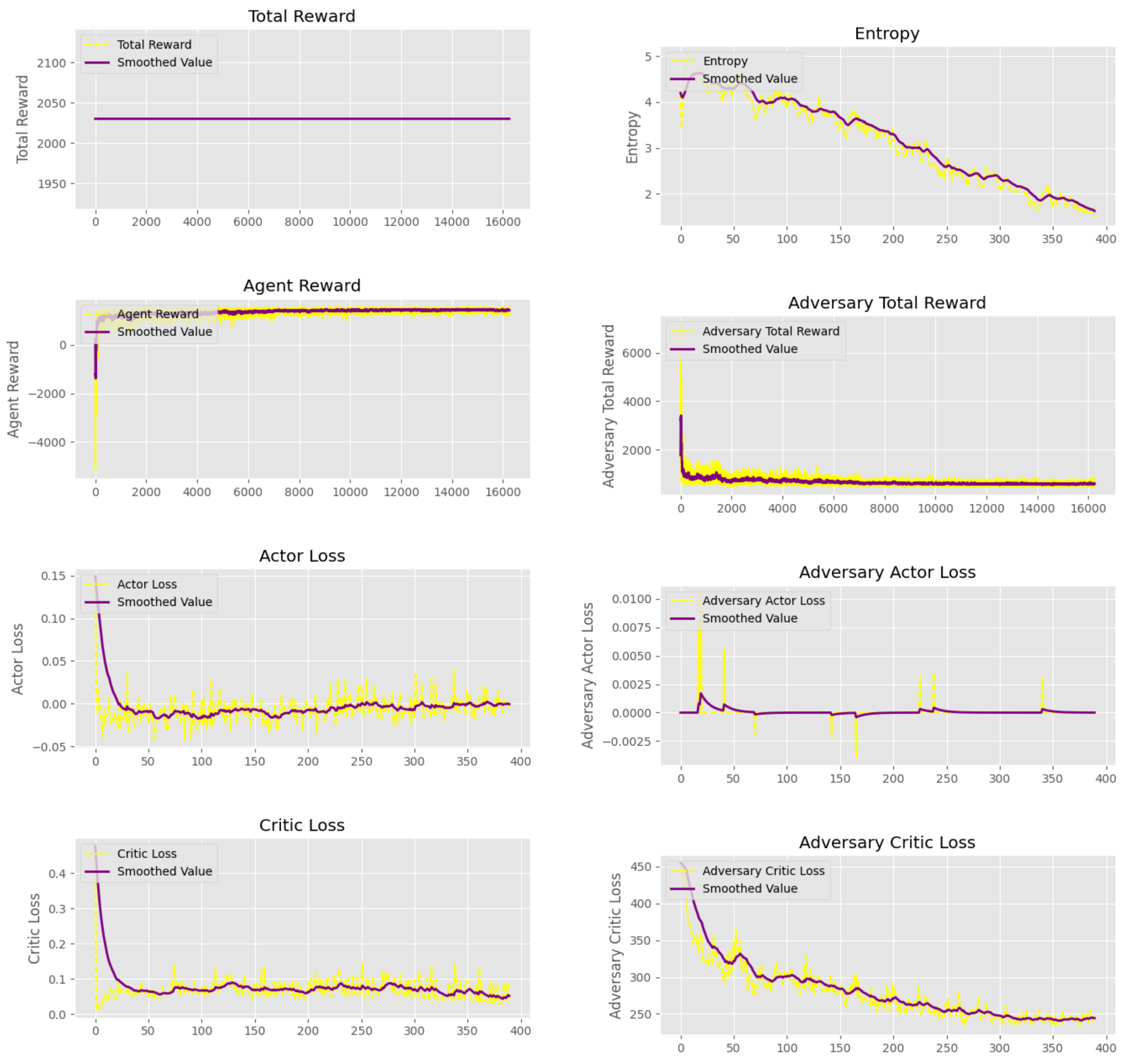


Figure 8: Adversary Model Training Plot (less step)

8.1.3 Truncated Environment

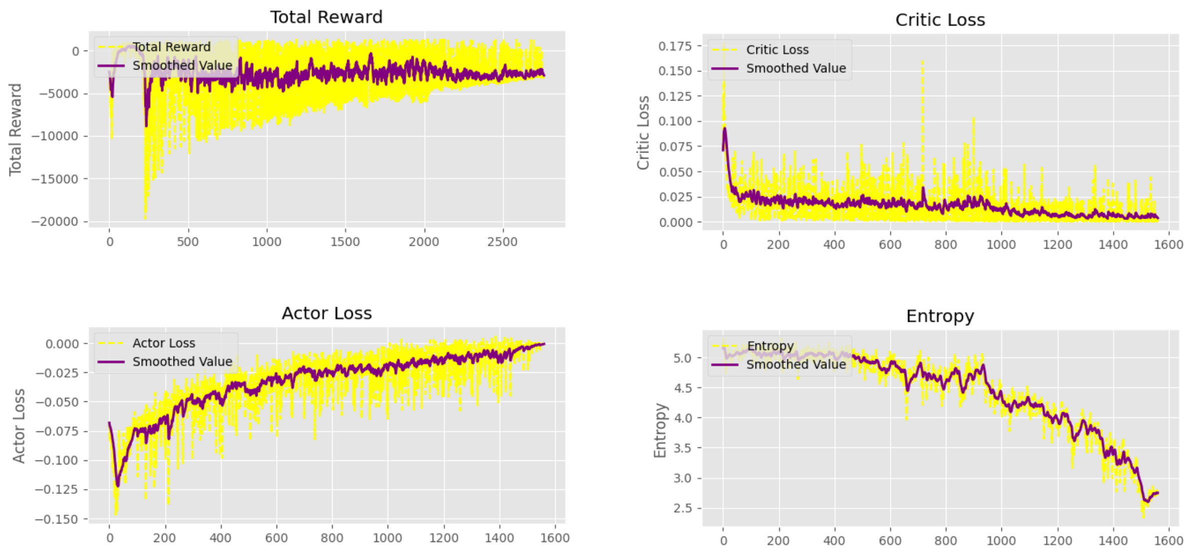


Figure 9: Truncated Model Training Plot

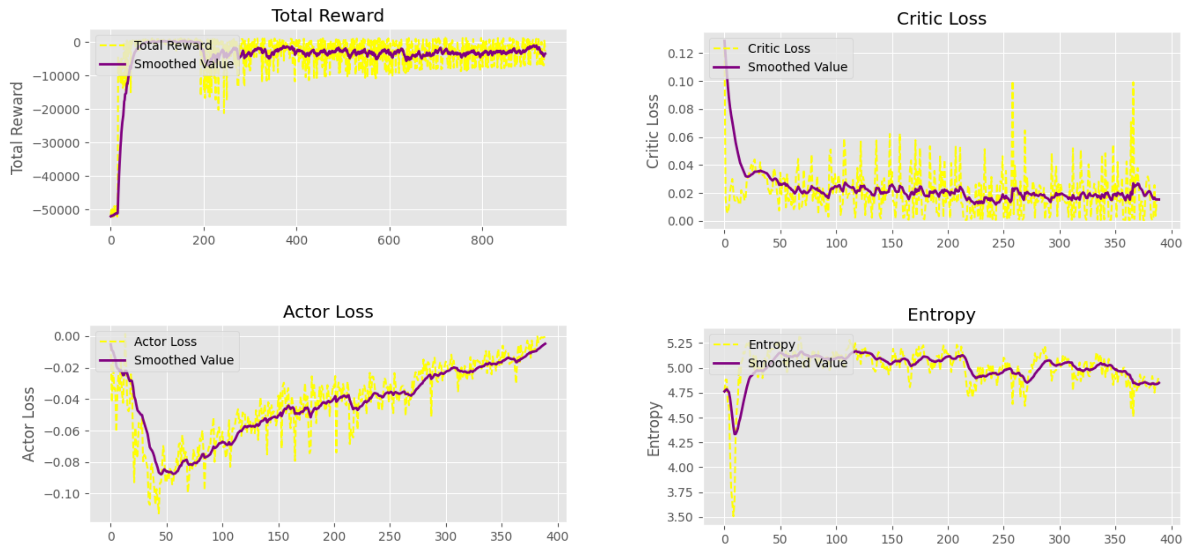


Figure 10: Truncated Model Training Plot (less step)

8.1.4 Less Exploration Agent

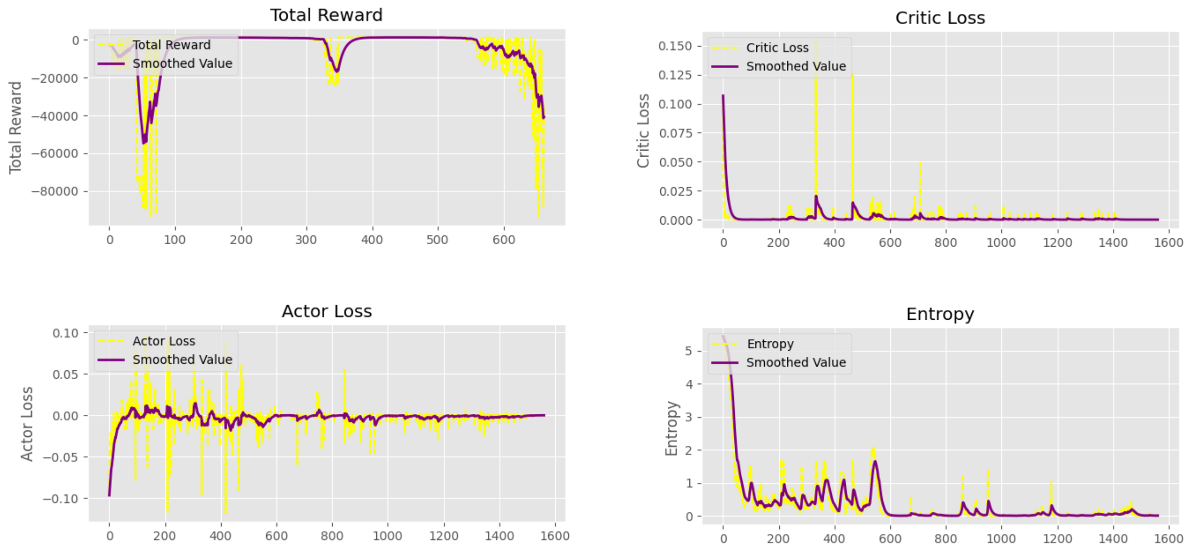


Figure 11: Less Exploration Model Training Plot

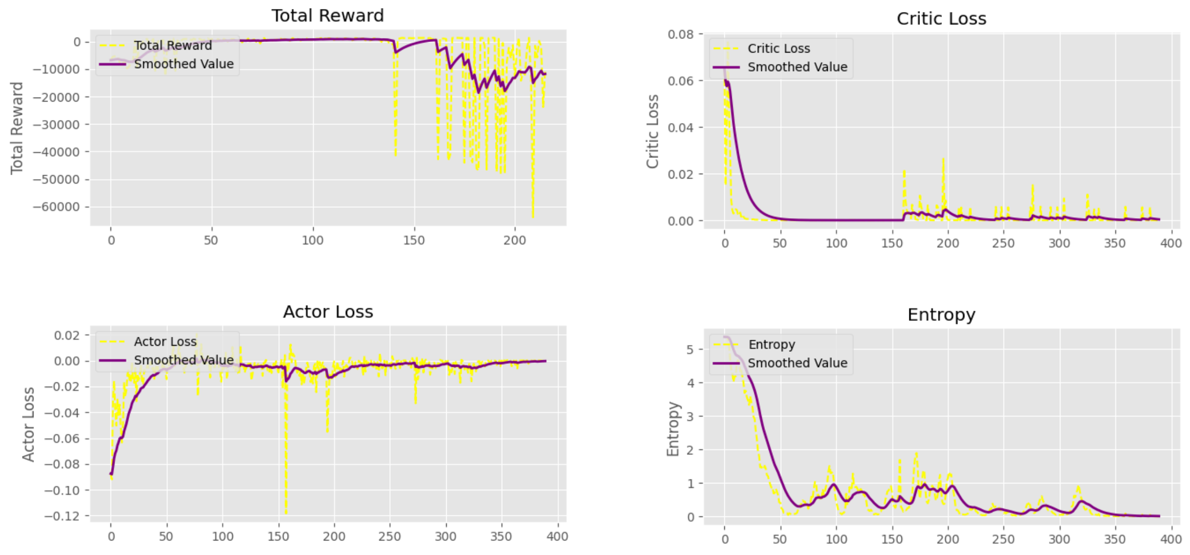


Figure 12: Less Exploration Model Training Plot (less step)

8.2 Division of works

Vincent: Environment Creation, Agent Creation, Training
Matthew: Adversary Addition, Experimentation