Optimization and Reinforcement Learning
Final Project Report

Hong-Son Nguyen, N16137037

June 25, 2025

1 Problem Overview

The objective of this project is to develop a reinforcement learning (RL) agent capable of playing a custom space ship game, with the primary goal of maximizing the in-game score (up to 10,000 points). The game, implemented in Python using Pygame, features a player-controlled spaceship that must avoid and destroy falling rocks, collect power-ups, and survive as long as possible. The RL agent must learn to control the spaceship efficiently, balancing survival and aggressive scoring strategies.

The main requirements for this project are as follows:

- *Custom RL Environment:* The RL environment must be constructed around the provided game code, exposing appropriate state, action, and reward interfaces for agent training.
- *State Representation:* The state can be designed flexibly, such as using raw game images (single frames or stacked), or structured data like positions and velocities of game objects.
- *Reward Design:* The reward function is up to the designer, but it must encourage behaviors that maximize the final game score at the end of each episode.
- *Game Integrity:* The core game mechanics—including health points (HP), damage, and scoring rules—must remain unaltered to ensure fairness and comparability.
- *Training Efficiency:* During training, rendering can be disabled or FPS increased to accelerate learning. However, for final evaluation, the trained agent must play the game visibly at 60 FPS.
- *Evaluation Metric:* The agent's performance is measured by the highest score it can achieve in a single episode under the standard game settings.

This project challenges students to integrate RL concepts with practical environment engineering, state and reward design, and model evaluation, all within the constraints of a real

2 Environment Setup

2.1 Game Environment

The core of the environment is a custom space ship game implemented in Python using the Pygame library (see `game.py`). The game features a player-controlled spaceship that must avoid and destroy falling rocks, collect power-ups, and survive as long as possible. The main components of the game include:

- *Player:* The spaceship, which can move left or right and shoot bullets.
- *Rocks:* Obstacles that fall from the top of the screen. Colliding with rocks reduces the player's health.
- *Power-ups:* Items such as shields and guns that provide temporary advantages when collected.
- *Score:* Points are awarded for destroying rocks and collecting power-ups.
- *Game State:* The game maintains internal state variables such as score, player health, lives, and the positions of all objects.

The game logic is encapsulated in the `Game` class, which provides methods for updating the game state, handling collisions, and rendering the game screen. The game runs at a fixed resolution (`WIDTH` $\times$ `HEIGHT`) and a target frame rate of 60 FPS.

2.2 RL Environment Wrapper

To interface the game with reinforcement learning algorithms, a custom environment wrapper (`env_wrapper.py`) was implemented. This wrapper exposes the game as a standard RL environment with the following features:

- *Action Space:* The agent can choose from four discrete actions: do nothing, move left, move right, or shoot.
- *Observation Space:* The environment returns the current game screen as an RGB image array, which can be used as input for convolutional neural networks.
- *Reward Function:* The reward is designed to encourage survival and score maximization. The agent receives a small positive reward for surviving each step, additional rewards for collecting power-ups, and a penalty if the player's health decreases.
- *Episode Termination:* An episode ends when the player loses all lives or achieves the maximum score (10,000 points).
- *Rendering:* The environment supports both headless (no rendering) and visible modes, allowing for faster training and visual evaluation at 60 FPS.

The wrapper manages the game loop, handles action execution, collects observations and rewards, and provides methods for resetting and rendering the environment. This setup enables seamless integration with RL libraries and facilitates efficient agent training and evaluation.

3 State Representation

In this project, the state representation is based on raw game images, which are processed and stacked to provide temporal context for the reinforcement learning agent. The environment (`SpaceShipEnv` in `env_wrapper.py`) returns the current game screen as an RGB image array with shape $(\texttt{WIDTH}, \texttt{HEIGHT}, 3)$. To make the state suitable for deep reinforcement learning, especially for convolutional neural networks, several preprocessing steps are applied (see `utils.py`):

- *Grayscale Conversion:* Each RGB frame is converted to grayscale to reduce computational complexity and focus on essential visual information.
- *Resizing:* The grayscale image is resized to 84×84 pixels, a standard input size for many RL algorithms.
- *Normalization:* Pixel values are normalized to the range $[0, 1]$ by dividing by 255.
- *Frame Stacking:* To capture temporal dynamics and motion, four consecutive preprocessed frames are stacked along the first dimension, resulting in a state tensor of shape $(4, 84, 84)$. This allows the agent to infer object velocities and directions from changes across frames.

The preprocessing and stacking are implemented in the `preprocess_frame` and `stack_frames` functions in `utils.py`. At the start of each episode, the stack is initialized with four identical frames. During gameplay, the newest frame is appended to the stack, and the oldest is removed, maintaining a fixed-length history.

This state representation provides the agent with both spatial and temporal information necessary for effective decision-making in the space ship game environment.

4 Reward Design

The reward function is a critical component in reinforcement learning, as it directly influences the agent's learning behavior and ultimate performance. In this project, the reward design aims to encourage the agent to maximize the game score by surviving longer, collecting power-ups, and avoiding damage.

The reward at each time step is computed as follows (see `calculate_reward` in `env_wrapper.py`):

$$r_t = r_{\text{survival}} + r_{\text{powerup}} + r_{\text{damage}}$$

Where:

- r_{survival} is a small positive reward for surviving each step, set to 0.25.
- r_{powerup} is a bonus for collecting power-ups:

$$r_{\text{powerup}} = \begin{cases} 50, & \text{if collected shield} \\ 25, & \text{if collected gun} \\ 0, & \text{otherwise} \end{cases}$$

- r_{damage} is a penalty applied if the agent's health decreases in the current step, set to -100.

$$r_{\text{damage}} = \begin{cases} -100, & \text{if health decreasing} \\ 0, & \text{otherwise} \end{cases}$$

This reward structure encourages the agent to:

- Survive as long as possible (by providing a constant positive reward).
- Seek out and collect power-ups for additional rewards.
- Avoid taking damage, as health loss incurs a significant penalty.

The episode terminates when the agent loses all lives or achieves the maximum score (10,000 points). The cumulative reward over an episode reflects the agent's overall performance in maximizing the game score while minimizing damage.

5 Model Architecture

In this project, three reinforcement learning (RL) model architectures were explored: Double Dueling Deep Q-Network (D3QN), Soft Actor-Critic (SAC), and Proximal Policy Optimization (PPO). Each architecture leverages deep neural networks to approximate value functions or policies, and is described mathematically as follows:

5.1 Double Dueling Deep Q-Network (D3QN)

The Double Dueling Deep Q-Network (D3QN) combines the advantages of double Q-learning [3] and the dueling network architecture [4] to improve stability and performance in value-based reinforcement learning.

The D3QN model processes the input state (a stack of 4 grayscale frames of size 84×84) through a series of convolutional layers, followed by separate fully connected streams for value and advantage estimation:

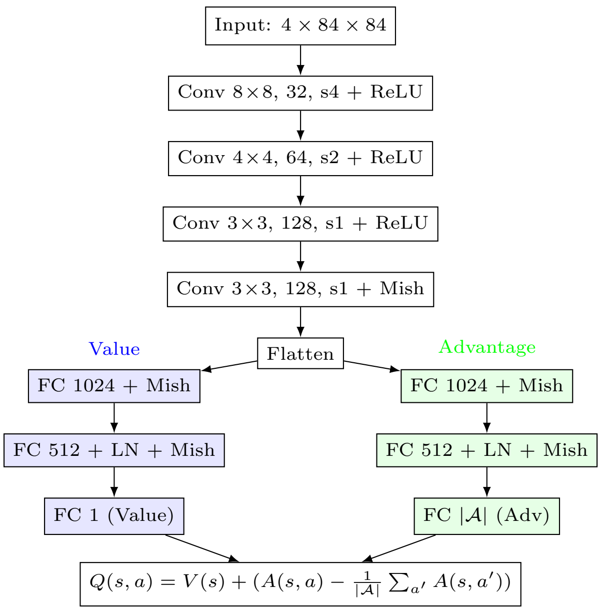

Figure 1: Compact Dueling DQN (D3QN) Network Architecture

The final Q-value for each action is computed as:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

where $V(s)$ is the scalar value output and $A(s,a)$ is the advantage vector.

Update Policy:

The Double Q-learning update is used to reduce overestimation bias [3]. The target for the Q-network is:

$$y = r + \gamma Q_{\text{target}}(s', \arg\max_{a'} Q_{\text{online}}(s', a'))$$

where Q_{online} is the current network and Q_{target} is the target network (periodically updated).

The loss function minimized during training is the mean squared error:

$$L(\theta) = \mathbb{E}_{(s,a,r,s',d)} \left[(y - Q(s,a;\theta))^2 \right]$$

where d is the done flag indicating episode termination.

Exploration: An ϵ-greedy policy is used for exploration, where the agent selects a random action with probability ϵ and the action with the highest Q-value otherwise. ϵ decays over time from 1.0 to 0.1.

This architecture enables efficient and stable learning of optimal policies in high-

5.2 Soft Actor-Critic for Discrete Action Spaces (SAC-Discrete)

Soft Actor-Critic (SAC) is an off-policy RL algorithm originally designed for continuous action spaces [1]. In this project, we adapt SAC for discrete action spaces, following recent advances that extend entropy-regularized RL to discrete domains. The key idea is to use a categorical policy (actor) and Q-value critics, both parameterized by deep convolutional neural networks. It consists of three networks: an actor (policy) and two critics (value function).

Why Discrete SAC?

The SAC model processes the input state (a stack of 4 grayscale frames of size 84×84) through a series of convolutional layers. The original SAC uses a Gaussian policy for continuous actions. For discrete actions, we replace the policy with a categorical distribution over actions, allowing the agent to sample actions and compute entropy in a discrete setting. This enables the benefits of entropy regularization (better exploration, robustness) in environments like the space ship game, where actions are discrete (e.g., stay, left, right, shoot).

Update Policy: At each training step, the SAC agent updates its networks as follows:

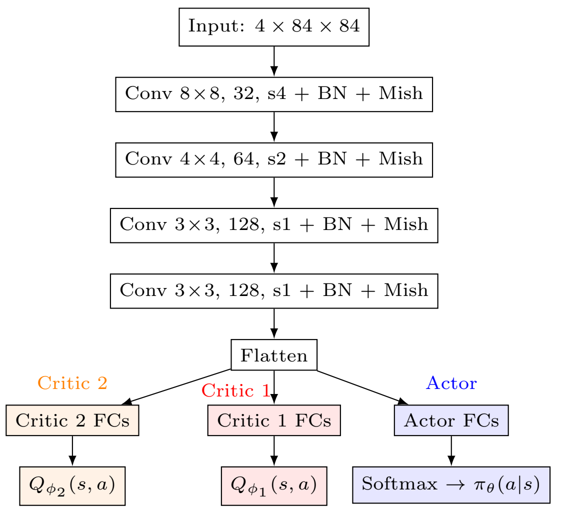
Input: 4 × 84 × 84
Conv 8×8, 32, s4 + BN + Mish
Conv 4×4, 64, s2 + BN + Mish
Conv 3×3, 128, s1 + BN + Mish
Conv 3×3, 128, s1 + BN + Mish
Flatten
Critic 2
Critic 1
Actor
Critic 2 FCs
Critic 1 FCs
Actor FCs
$Q_{\phi_2}(s,a)$
$Q_{\phi_1}(s,a)$
Softmax → $\pi_\theta(a|s)$

Figure 2: SAC-discrete network: shared convolutional feature extractor, one actor (policy) head, and two critic (Q-value) heads.

Critic targets:

$$Q_{\text{target}}(s',a') = \min_{i=1,2} Q_{\phi_i}(s',a')$$

$$V(s') = \sum_{a'} \pi_\theta(a'|s') \left[Q_{\text{target}}(s',a') - \alpha \log \pi_\theta(a'|s')\right]$$

$$y = r + \gamma(1-d)V(s')$$

where r is the reward, d is the done flag, γ is the discount factor, and α is the entropy temperature.
Critic loss:

$$L_{\text{critic}_i} = \mathbb{E}\left[(Q_{\phi_i}(s,a) - y)^2\right], \quad i = 1,2$$

Actor loss:

$$L_{\text{actor}} = \mathbb{E}_{s\sim D}\left[\sum_a \pi_\theta(a|s)\left(\alpha \log \pi_\theta(a|s) - Q(s,a)\right)\right]$$

where $Q(s,a) = \min(Q_{\phi_1}(s,a), Q_{\phi_2}(s,a))$.

Target network soft update:

$$\phi'_i \leftarrow \tau\phi_i + (1-\tau)\phi'_i, \quad i = 1,2$$

These updates ensure stable learning by using double Q-learning, entropy regularization, and target networks, and are implemented efficiently for discrete action spaces in this project.

5.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient method that improves training stability by constraining the policy update at each step [2].

In this project, PPO is implemented for discrete action spaces using separate convolutional neural networks for the policy (actor) and value (critic) functions. This model processes the input state (a stack of 4 grayscale frames of size 84×84) through a series of convolutional layers.

Network Structure:

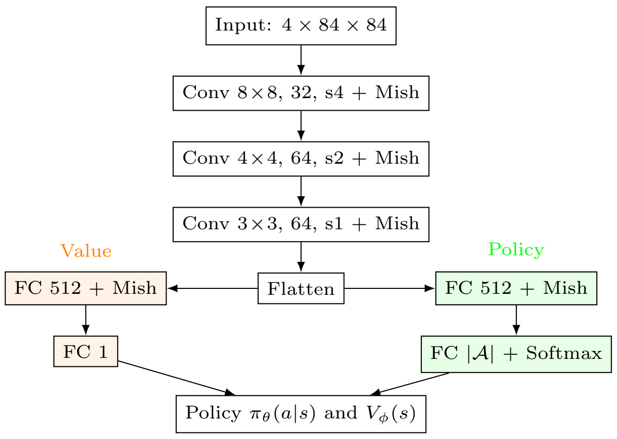

Figure 3: Compact PPO Network Architecture: separate convolutional actor and critic.

Update Policy: PPO uses a clipped surrogate objective to limit the deviation from the previous policy, improving stability:

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

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies.
- $\hat{A}_t$ is the advantage estimate, computed using Generalized Advantage Estimation (GAE).
- ϵ is a hyperparameter controlling the clip range.

The value function is updated by minimizing the mean squared error:

$$L_{\text{value}}(\phi) = \mathbb{E}_t\left[(V_\phi(s_t) - R_t)^2\right]$$

This PPO agent uses separate convolutional networks for the policy and value functions, enabling stable and efficient learning in high-dimensional, discrete-action environments such as the space ship game.

All three architectures use convolutional neural networks to process stacked image frames as input, enabling the agent to learn effective representations of the game environment for decision-making.

6 Training Process

The training process for all three RL agents (D3QN, SAC-Discrete, and PPO) follows a similar high-level structure, with method-specific details in agent updates and buffer management. Each agent interacts with the custom SpaceShipEnv environment, collects experience, and updates its neural networks to maximize the game score.

6.1 General Training Loop

The overall training loop for each agent can be described as follows:

Algorithm 1 General RL Training Loop

```
1:  for episode = 1 to $N_{\text{episodes}}$ do
2:    Reset environment and preprocess initial state $s_0$
3:    Initialize episode reward $R \leftarrow 0$
4:    for timestep = 1 to max steps per episode do
5:      Select action $a_t$ according to current policy (with exploration)
6:      Execute $a_t$ in environment, observe $r_t$, $s_{t+1}$, $done$
7:      Store transition $(s_t, a_t, r_t, s_{t+1}, done)$ in buffer
8:      $R \leftarrow R + r_t$
9:      if update condition met then
10:       Sample batch from buffer
11:       Update agent's networks (policy/value)
12:     end if
13:     $s_t \leftarrow s_{t+1}$
14:     if $done$ then
15:       break
16:     end if
17:   end for
18:   Log episode statistics (reward, score, health)
19:   Save model if performance improved
20: end for
```

6.2 D3QN Training

The D3QN agent uses a prioritized replay buffer to sample important transitions more frequently. The agent's ϵ-greedy policy balances exploration and exploitation, with ϵ decaying over time. The target network is updated every 1000 steps to stabilize learning. The agent's networks are saved periodically and whenever a new best score is achieved.

Key hyperparameters:

- Replay buffer size: 50,000
- Batch size: 64
- Learning rate: 1×10^{-4}
- Target network update frequency: 1000 steps
- Maximum episodes: 4000

6.3 SAC-Discrete Training

The SAC-Discrete agent uses a large replay buffer and updates its networks every fixed number of timesteps (2048). The agent samples actions from its learned categorical policy and stores transitions in the buffer. Critic and actor networks are updated using the SAC loss functions, and target networks are softly updated. Model checkpoints are saved every 100 episodes.

Key hyperparameters:

- Replay buffer size: 100,000
- Batch size: 128
- Learning rate: 5×10^{-4}
- Update frequency: every 2048 steps
- Maximum episodes: 5000

6.4 PPO Training

The PPO agent collects trajectories in a buffer and updates its networks every 2048 timesteps using mini-batch optimization. The agent stores log-probabilities for each action to compute the PPO surrogate loss. The policy and value networks are checkpointed every 50 episodes and whenever a new best score is achieved.

Key hyperparameters:

- Replay buffer size: 100,000
- Batch size: 128
- Learning rate: 5×10^{-4}
- Update frequency: every 2048 steps
- Maximum episodes: 4000

6.5 Logging and Evaluation

During training, episode rewards, scores, and health are logged and saved as CSV files for later analysis. The best-performing models are checkpointed for final evaluation. After training, the highest score achieved by each agent is reported as the main performance metric.

7 Results

In this section, we present and compare the training outcomes of four reinforcement learning approaches applied to the space ship game environment: D3QN with a normal replay buffer, D3QN with a prioritized replay buffer, Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC) for discrete actions. The evaluation focuses on the agent's ability to maximize game performance as measured by both immediate and long-term metrics.

For each method, we report the following key results:

- **Reward per Episode:** The total reward accumulated by the agent in each episode during training.
- **Score per Episode:** The final game score achieved by the agent in each episode.
- **Average Reward per Episode:** The moving average of episode rewards, providing insight into learning stability and overall performance trends.
- **Average Score per Episode:** The moving average of episode scores, reflecting the agent's ability to consistently achieve high scores over time.

These results are visualized in training curves for each method, enabling a direct comparison of learning speed, stability, and final performance. The comparative analysis highlights the strengths and weaknesses of each approach in the context of the space ship game, providing insights into the impact of replay buffer strategies and algorithmic choices on RL agent effectiveness.

7.1 D3QN with Normal Replay Buffer

The Double Dueling Deep Q-Network (D3QN) with a standard replay buffer serves as a baseline for value-based reinforcement learning in this environment. The agent samples experience uniformly from the buffer, which helps break correlations between consecutive transitions and stabilizes training. Figure 4 shows the training curves for reward and score per episode, as well as their moving averages for this method.

7.2 D3QN with Prioritized Replay Buffer

In this variant, the D3QN agent utilizes a prioritized replay buffer, where transitions with higher temporal-difference (TD) error are sampled more

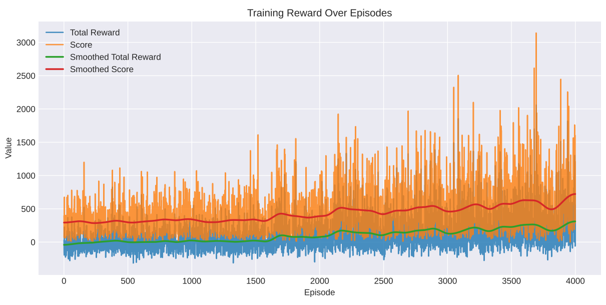

Figure 4: D3QN with Normal Replay Buffer training plots

frequently. This approach accelerates learning by focusing updates on more informative experiences. The impact of prioritized sampling on convergence speed and final performance is illustrated in Figure 5.

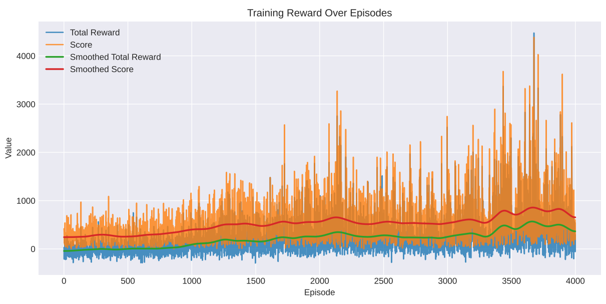

Figure 5: D3QN with Prioritized Replay Buffer training plots

7.3 Proximal Policy Optimization (PPO)

PPO is a policy gradient method known for its stability and robustness. The PPO agent collects trajectories and updates its policy using a clipped surrogate objective, which prevents large, destabilizing policy updates. The learning dynamics and performance of PPO are presented in Figure 6, including reward and score trends across episodes.

7.4 Soft Actor-Critic (SAC) Discrete

The SAC agent is adapted for discrete action spaces, leveraging entropy regularization to encourage exploration and prevent premature convergence. By optimizing both value and entropy objectives, SAC aims for high and stable performance. Figure 7 displays the training results for SAC discrete, highlighting the effect of entropy-driven exploration on reward and score outcomes.

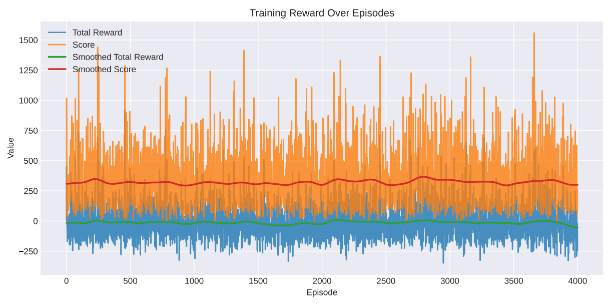

Figure 6: PPO training plots

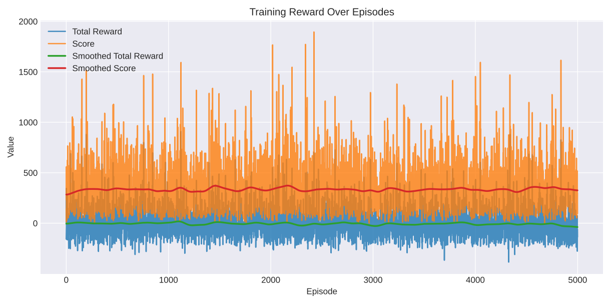

Figure 7: SAC training plots

Table 1 summarizes the quantitative results of each reinforcement learning method after the training process. For each approach, we report the highest score achieved, the average score across all episodes, and the highest reward obtained during training. This comparison provides a clear overview of the relative performance and effectiveness of each algorithm in maximizing both immediate and long-term objectives in the space ship game environment.

Table 1: Training Performance between Proposed RL Algorithms

Method	**Best Score**	**Average Score**
D3QN	3141.0	425.21
D3QN-Prior	4384.0	511.67
PPO	1558.0	320.51
SAC	1895.0	333.66

From the visual results in the training plots and the comparison table of training performance across methods, it is evident that the off-policy and value-based algorithms significantly outperform the other two in training the space ship game model. Thanks to the simple yet effective architecture of DQN—enhanced by architectural variants such as Dueling networks and the Double Q-learning update mechanism—the D3QN model demonstrates superior performance and efficiency compared to the two other models: PPO (on-policy) and SAC, which rely on more complex network architectures and require longer training periods.

In particular, the D3QN agent with a prioritized replay buffer achieves the highest scores and rewards, as well as faster and more stable convergence during training. The prioritized replay mechanism enables the agent to focus on more informative experiences, accelerating the learning process and improving final performance. In contrast, PPO and SAC, while robust and theoretically appealing, exhibit slower learning and lower peak performance in this discrete, image-based environment.

Overall, these results highlight the effectiveness of value-based, off-policy methods—especially when combined with experience prioritization—for challenging arcade-style games with high-dimensional visual input. Check out the game-play (see git-repo).

8 Conclusion

This project explored and compared several state-of-the-art reinforcement learning algorithms for maximizing performance in a custom space ship game environment. Through extensive experimentation, it was found that the D3QN model, particularly when equipped with a prioritized replay buffer, consistently outperformed both PPO and SAC in terms of learning speed, stability, and final game score.

The findings suggest that, for discrete-action, image-based environments like the space ship game, value-based off-policy methods with architectural enhancements and experience prioritization offer a practical and efficient solution. The prioritized D3QN agent was therefore selected for the final validation phase, demonstrating its suitability for real-world deployment in this context.

Future work may explore further architectural improvements, hybrid approaches, or transfer learning to enhance generalization and performance across a broader range of game scenarios.

References

[1] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018. URL `https://arxiv.org/abs/1801.01290`.

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

arXiv:1707.06347, 2017. URL https://arxiv.org/abs/1707.06347.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016. URL https://arxiv.org/abs/1509.06461.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016. URL https://arxiv.org/abs/1511.06581.