UAB
Universitat Autònoma
de Barcelona

Bachelor's Degree in Artificial Intelligence

PROJECT

Reinforcement Learning

Authors:
Martina Carretta (1673930)
Mariona Pla (1616474)

December 11, 2024

Contents

1 Introduction

This project investigates reinforcement learning (RL) through a structured progression of tasks that focus on the development and evaluation of RL agents in varying levels of complexity. The initial phase involves solving a straightforward environment using custom implementations of RL models to analyze and compare performance. The second phase advances to a more complex scenario, applying established RL frameworks to optimize agent behavior and evaluate outcomes. The final phase centers on multi-agent RL, requiring the training and optimization of agents in a competitive environment.

2 Simple ALE environment: SKIING

The Skiing game consists of an agent (skier) whose main objective is to go in between 20 flags in the least amount of time.

Figure 1: Skiing environment.

2.1 Action space

Pacman's action space is Discrete(3). The different actions correspond to:

- 0: NOOP
- 1: RIGHT
- 2: LEFT

2.2 Observation space

Observation space is Box(0, 255, (250, 160, 3), uint8). An array with shape (210, 160, 3) where each element is an integer between 0 and 255

2.3 Episode characteristics

In each episode, the skier is supposed to ski down the slope, trying not to miss gates and do it in the least amount of time. There is a total of 20 gates, so the episode finishes where the last gate is found.

2.4 Reward function

Skiing reward function is complicated since all the rewards are negative. The total reward corresponds to the time and the corresponding penalisations. In every timestep, the agent receives a negative reward of -6 or -7 corresponding to the time elapsed. Hitting a tree or a gate does not penalise, however, time is needed to recover and continuing skiing down the slope.

- Miss a gate -500
- Grey spots (ice) -2
- Time -6/-7

2.5 Untrained agent performance

As seen in Figure 2, the untrained agent gets scores in the range of -20.000 up to -13.000. With a mean reward -16466.

For comparison, manually played episodes that successfully passed through all the gates achieved rewards around -3,500, which is close to the best possible scores.

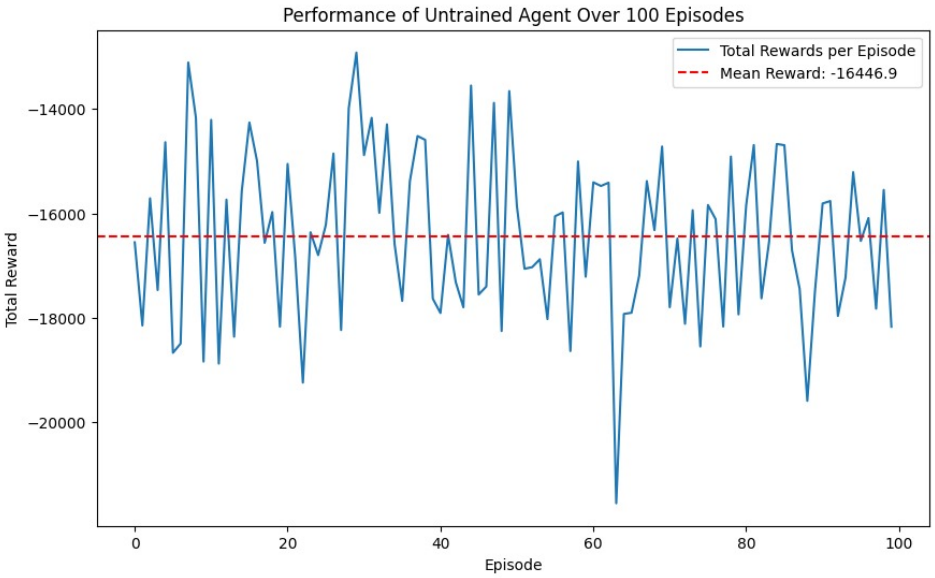

Figure 2: Untrained agent Skiing rewards.

2.6 DQN

2.6.1 Training

It has 4 main sections:

- Preprocessing: Frames from the environment are converted to grayscale, resized to 84x84, and normalized to the range [-1, 1].
  - Pass the frame to grayscale
  - Resize to (84, 84)
  - Normalize
- DQN architecture: The network uses convolutional layers to extract features, followed by fully connected layers to compute Q-values for each action.
- Experience Replay Buffer: A replay buffer stores transitions, enabling the agent to learn from past experiences, breaking temporal correlations.

- Target network: A separate target network stabilizes training by providing fixed Q-value targets that are updated periodically.

Training was initially conducted with a purely automated approach, but the agent's performance declined rapidly. Adjustments were made to hyperparameters, such as the epsilon decay, to slow down the exploration-to-exploitation transition. However, in three different configurations:

- Epsilon Decay = 0.9999 with 5000 episodes
- Epsilon Decay = 0.995 with 2000 episodes
- Epsilon Decay = 0.995 with 3600 episodes

the agent's rewards consistently decreased as training progressed, see Figure 3.

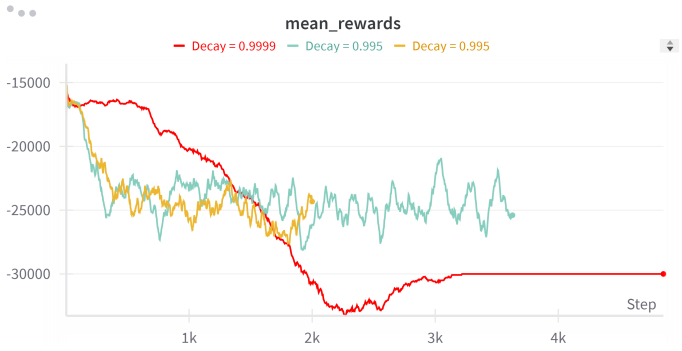

Figure 3: Skiing mean rewards - Epsilon value.

To address this issue, manually played episodes were added to the replay buffer. Initially, the buffer size was set to 10, matching the number of manual episodes, but performance remained suboptimal. It was suspected that the First-In-First-Out (FIFO) behavior of the replay buffer led to transitions being overwritten too quickly, hindering the agent's ability to learn effectively.

The buffer size was then increased to 30,000, and the following tests were conducted, see Figure 4:

- 1 Manual Episode
- 4 Manual Episodes
- 15 Manual Episodes

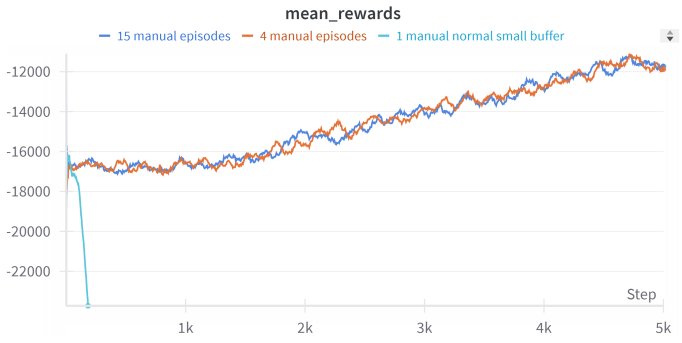

Figure 4: Skiing mean rewards - Manual episodes.

These results showed a significant improvement compared to the previous experiments with epsilon decay. Surprisingly, the results from the 4 and 15 manual episode setups were very similar, with no notable improvement in performance.

A hypothesis was formed that longer training might yield better results, so the maximum number of episodes was increased, see Figure 5:

- 10,000 Episodes: Performance dropped almost immediately.
- 6,000 Episodes: Results were similar to those obtained with fewer episodes.
- 4,000 Episodes: The performance drop was slower, but it still occurred.

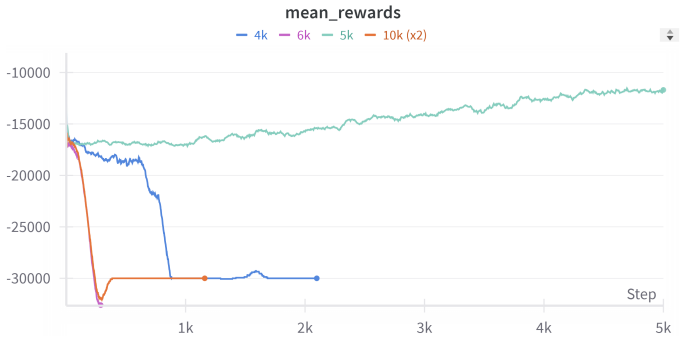

Figure 5: Skiing mean rewards - Nº episodes.

At this point, the reasons behind the agent's struggles remained unclear, despite adjustments to both the training duration and buffer content.

As a last resort, the decision was made to train the same DQN architecture using the weights and biases obtained at the end of the 5k episode run. However, the results were disappointing. The rewards were comparable to those obtained during the 10k episode training, showing no significant improvement, see Figure 6.

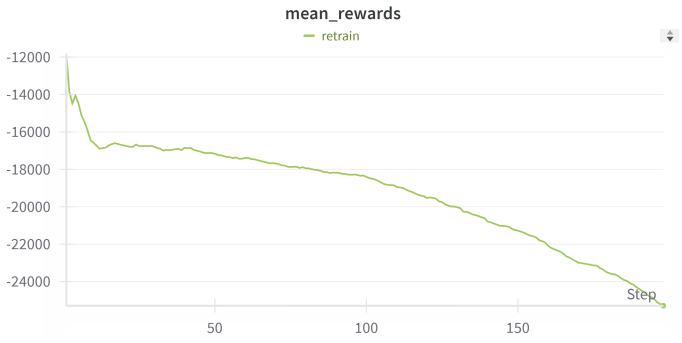

Figure 6: Skiing mean rewards - Retraining.

After training the DQN, an extension, **Double DQN (DDQN)**, was tested by maintaining the same hyperparameters but changing the network architecture. The training behavior of DDQN was very similar to that of the basic DQN, following the same general learning curve, Figure 7.

However, a noticeable improvement was observed in the mean rewards at the end of the training, with an increase of approximately 200 points. This suggests that, while the overall behavior remained similar, the introduction of DDQN helped the agent make more stable and accurate decisions, leading to a slight improvement in performance over the standard DQN.

2.6.2 DQN Parameters

The final model has the following parameters:

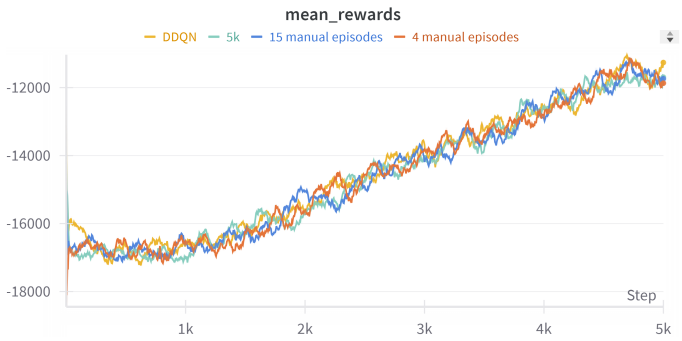

Figure 7: Skiing mean rewards - DDQN.

- Learning Rate (LR) = 0.001
- Memory Size = 8452*2
- Maximum Episodes = 4000
- Epsilon = 0.99
- Epsilon Decay = 0.999
- Gamma = 1
- Batch Size = 64
- Burn-In = 1000
- DNN Update Frequency = 1
- DNN Sync Frequency = 2500
- Channels = 1
- Window Size = 80
- Number of Actions = 3 (possible actions the agent can take)
- Input Shape = (1, 80, 80) (PyTorch format for channels, height, and width)
- Hidden Size = 256

Given the long execution times, it's computationally expensive to fine-tune these parameters or test other DQN architectures. Therefore, a decision was made to work with parameters and architectures that are known to be effective in many projects, even if they may not be optimal for the current environment.

2.6.3 DQN Architecture

Convolutional Layers:

- Conv2d(1, 32, kernel_size=8, stride=4) followed by ReLU.
- Conv2d(32, 64, kernel_size=4, stride=2) followed by ReLU.
- Conv2d(64, 64, kernel_size=3, stride=1) followed by ReLU.

These layers extract spatial features from the game's frames.

Fully Connected Layers:

- **Fully Connected (Linear) Layer** with 256 hidden units followed by **ReLU**.
- **Output Layer** with units equal to the number of possible actions (3).

2.6.4 Optimizer:

Adam optimizer with a learning rate of 0.001.

2.6.5 Loss Function:

Mean Squared Error (MSE) between predicted Q-values and target Q-values. This setup represents a standard approach for a DQN architecture, designed to process image-based inputs and output action predictions based on the learned Q-values. The next steps involve training the model, but given the computational cost, it seems that further architectural or hyperparameter adjustments are not feasible at this point.

2.6.6 Testing

The test phase reveals that the agent still struggles to play Skiing effectively. However, a comparison of the mean rewards provides insight into the agent's progress. The untrained agent received a mean reward of -16500, while the trained agent achieved a mean reward of -13800. This indicates an improvement of 2700 points, suggesting that the agent has made progress in learning the task. While the performance is still far from optimal, the results demonstrate that the agent has improved.

Figure 8: DQN test rewards.

2.7 REINFORCE

2.7.1 REINFORCE Parameters

- Learning rate: 0.0001
- Discount factor (gamma): 0.99

- Hidden size: 256
- Horizon (max steps per episode): 15000
- Maximum trajectories: 5000

The optimizer used is Adam, with a learning rate of 0.0001. Loss Function: The loss is calculated as the negative log probability of the selected actions weighted by the discounted rewards.

2.7.2 Training

The frames have been preprocessed in the following way:

- Pass the frame to grayscale
- Resize to (84, 84)
- Normalize

The Reinforce architecture used is the following:

- Convolutional Layers: Extract features from image inputs through three convolutional layers with increasing filters (64, 128, and 64) and ReLU activations.
- Fully Connected Layers: After flattening the convolutional output, two fully connected layers are used. The first maps features to a hidden layer of size 256 (HIDDEN_SIZE), and the second maps to the action space with a softmax activation to produce probabilities for each action.

Training process:

- Collect trajectories by interacting with the environment for up to HORIZON steps or until the episode ends.
- Preprocess frames into grayscale, resized, normalized images.
- Calculate the discounted rewards for the episode using the discount factor.
- Compute the loss as the negative log probability of actions taken, weighted by the discounted rewards.
- Backpropagate the loss and update the policy network using the Adam optimizer.
- Log episode rewards, mean rewards, and loss to Weights & Biases (wandb).
- Save the best model weights based on the highest episode score and mean rewards.
- Optionally save GIFs of the best-performing episodes.

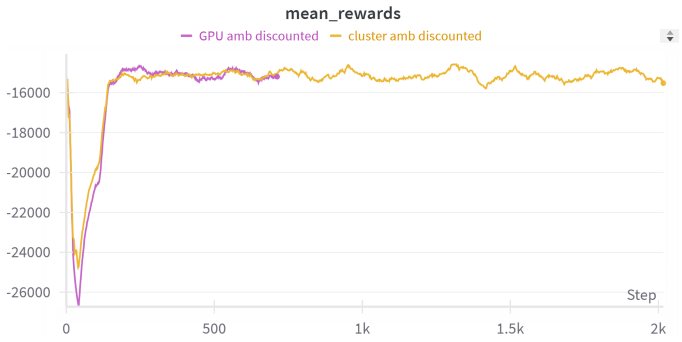

Figure 9: REINFORCE training mean rewards.

2.7.3 Testing

The agent consistently got a reward of -9013. After analyzing the episodes, it can be seen that the agent has found a trick: it points the skies straight down the slope so it goes at maximum speed. Even if using this approach is misses flags, the time makes up for it.

2.8 A2C

A2C (Advantage Actor-Critic) was implemented to address the complexity of the environment. The architecture combines a policy network and a value network into a unified framework, enabling the agent to learn more efficiently by reducing the variance in policy updates through the advantage function.

Despite these advantages, the agent converged to the same suboptimal strategy as observed with REINFORCE, consistently achieving a reward of -9013.

2.9 Other trained models

Skiing, being one of the more challenging environments, has seen improvements over time through the use of different algorithms. Notably, models like Agent57, and Go-Explore (a family of algorithms that solve some of the Skiing problems by explicitly remembering promising states and first returning to such states before intentionally exploring) have shown significant performance advancements. These models are equipped with enhanced exploration strategies or more sophisticated planning capabilities, which give them a clear advantage in the Skiing environment compared to more basic algorithms like DQN and REINFORCE (the ones trained in this project).

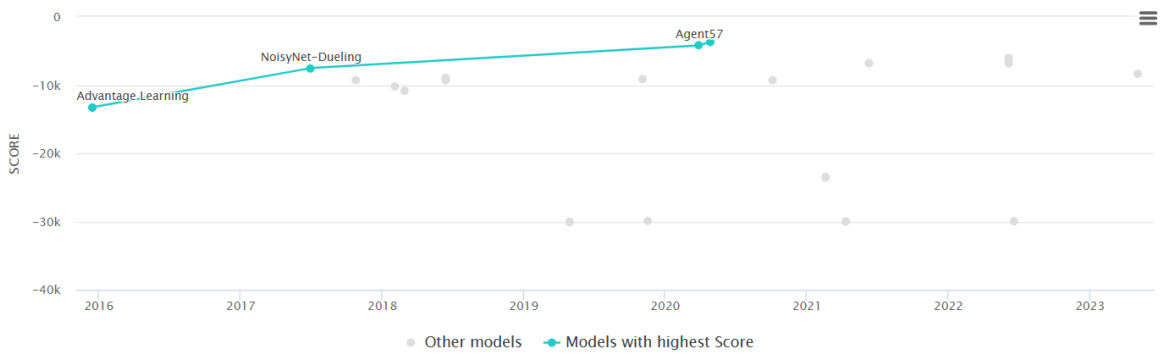

Figure 10: Trained models mean rewards.

For further details on these results and additional visualizations, please refer to the official page of the graphic: link.

2.10 Conclusions

The environment demands agents to balance speed and accuracy, as rewards are heavily influenced by both completing the task quickly and properly interacting with the environment (passing through flags).

The observed convergence to suboptimal policies across all algorithms indicates that the environment's reward structure played a critical role in shaping agent behavior. Specifically, the heavy emphasis on penalties and its delay led agents to exploit shortcuts, prioritizing speed over accuracy.

3 Complex ALE environment: PACMAN

Pac-Man is a classic video game where players control a yellow, circular character navigating a maze to eat pellets while avoiding ghosts. The game challenges players to strategize, collect bonus items, and clear levels by consuming all pellets while using power pellets to temporarily defeat ghosts.

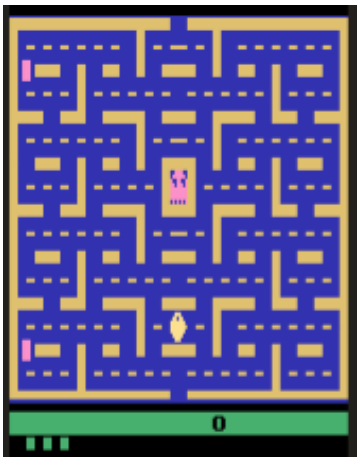

Figure 11: Pacman environment.

3.1 Action space

Pacman's action space is Discrete(5). The different actions correspond to:

- 0: NOOP
- 1: UP
- 2: RIGHT
- 3: LEFT
- 4: DOWN

3.2 Observation space

Observation space is Box(0, 255, (250, 160, 3), uint8). An array with shape (210, 160, 3) where each element is an integer between 0 and 255

3.3 Episode characteristics

In each episode, the player has 3 lives, represented by green squares at the bottom-left corner of the environment. The episode ends when the player loses all three lives by being caught by the ghosts or completes the level by consuming all the pellets in the maze.

3.4 Reward function

- Small pellets (yellow): +1
- Power pellets (pink): +5
- After consuming a power pellet, Pac-Man can eat ghosts (while they are blue). The first ghost provides a reward of +20, and the reward doubles for each additional ghost eaten.

3.5 Untrained agent performance

As seen in Figure 12, the untrained agent typically gets a score within the range of 10 to 30. Occasionally, it exceeds this range, achieving rewards between 50 and 100.

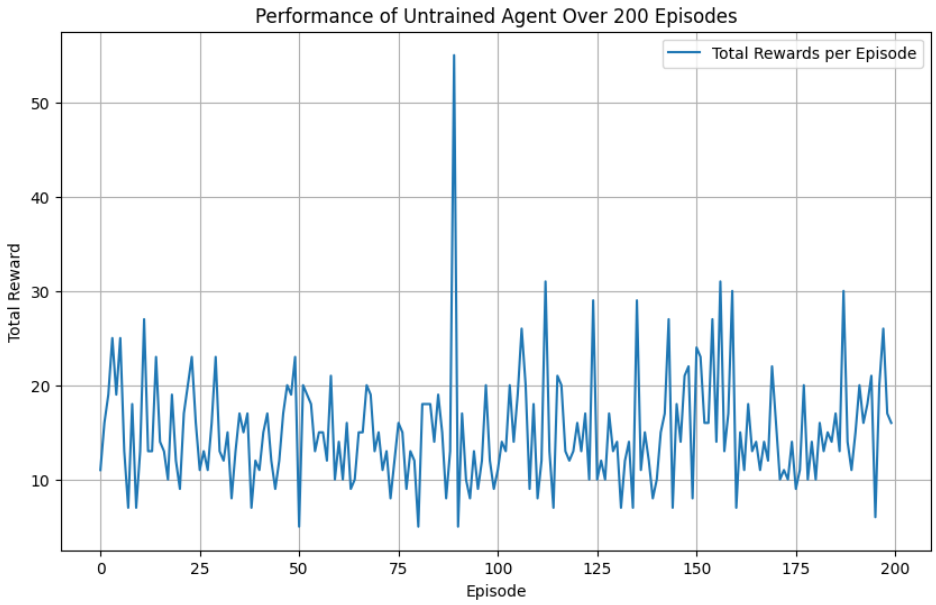

Figure 12: Untrained agent Pacman rewards.

3.6 Proximal Policy Optimization

Proximal Policy Optimization (PPO) offers stable and reliable training by constraining the policy updates. It uses a clipped objective to ensure that the policy changes incrementally, avoiding instability caused by aggressive updates. This characteristic is particularly beneficial in Pacman, where small improvements in strategy can lead to significant performance gains. PPO's robustness and sample efficiency allow it to predict ghost movements and optimise navigation paths, making it a powerful choice for mastering Pacman's strategic challenges.

3.6.1 PPO parameters

Due to the high training complexity and time requirements, many of the parameters used are set to their default values from PPO. Additionally, the inherent stochasticity of the environment further complicates the comparison of different runs, as identical parameter settings can lead to different outcomes in each execution.

- policy = "CnnPolicy": This specifies a convolutional neural network policy, which is suitable for environments with image-based inputs.
- learning rate = 0.0005: A small learning rate that ensures gradual updates to the model, preventing unstable training.
- gamma = 0.99: A high discount factor that emphasizes long-term rewards in decision-making.
- gae lambda = 0.95
- n steps = 128
- ent coef = 0.01
- vf coef = 0.5

- clip range = 0.2
- clip range vf = 1
- n epochs = 6
- batch size = 128
- max grad norm = 0.4
- total timesteps = 2000000: Set to a high number to allow the agent to train for as long as necessary. The training is manually stopped if needed.

3.6.2 Training

PPO implementation from Stable Baselines was used.

The model is saved every 2000 timesteps in order to have backups and it is evaluated every 1000 timesteps and only saved if the evaluation results are better than the best result.

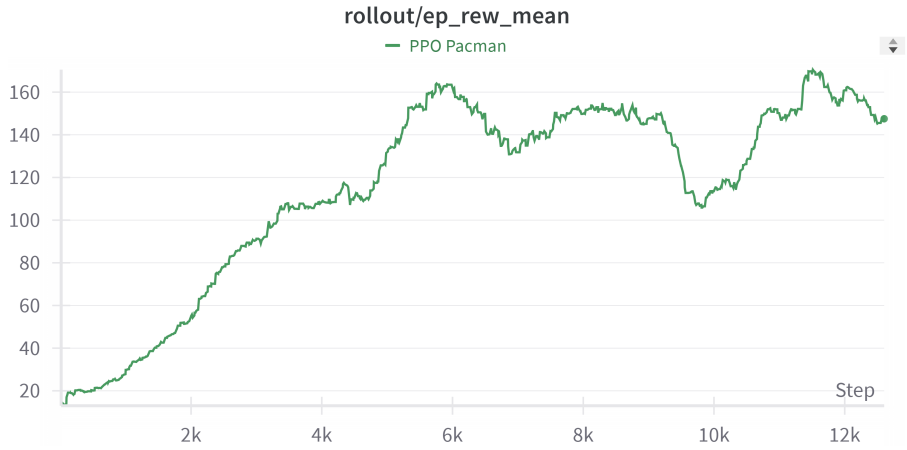

Figure 13: PPO Training mean rewards.

3.6.3 Testing

The model was tested over 100 episodes to gain a comprehensive understanding of the agent's performance. As a point of reference, untrained agents typically score between 10 and 30.

As shown in Figure 14, the PPO-trained model achieves scores ranging from 100 to 400, consistently getting rewards of 350; indicating that the agent has successfully learned to play Pacman. The mean test reward is approximately 330.

3.7 Advantage Actor-Critic

Advantage Actor-Critic (A2C) is a straightforward and efficient policy gradient algorithm that works properly in environments with moderately complex dynamics, like Pacman. It combines value-based and policy-based learning to reduce variance in updates, leveraging an "advantage function" to guide decisions. A2C runs multiple environments in parallel, which improves sample efficiency and exploration by exposing the agent to diverse scenarios in each training batch. This parallelization helps the agent learn to navigate Pacman's maze effectively, balancing the objectives of collecting dots and avoiding ghosts.

Figure 14: PPO Test rewards.

3.7.1 A2C parameters

Due to the high training complexity and time requirements, many of the parameters used are set to their default values from PPO. Additionally, the inherent stochasticity of the environment further complicates the comparison of different runs, as identical parameter settings can lead to different outcomes in each execution.

- policy = 'CnnPolicy'
- learning rate = 0.0005
- gamma = 0.99
- n steps = 64
- vf coef = 0.5
- ent coef = 0.01
- max grad norm = 0.4
- total timesteps = 3000000

3.7.2 Training

A2C implementation from Stable Baselines was used. 8 parallel environments are used for training. The model is saved every 2000 timesteps in order to have backups and it is evaluated every 1000 timesteps and only saved if the evaluation results are better than the best result.

3.7.3 Testing

The model was tested over 100 episodes to gain a comprehensive understanding of the agent's performance. As a point of reference, untrained agents typically score between 10 and 30.

As shown in Figure 16, the A2C-trained model achieves scores ranging from 100 to 700, indicating that the agent has successfully learned to play Pacman. The mean test reward is approximately 320.

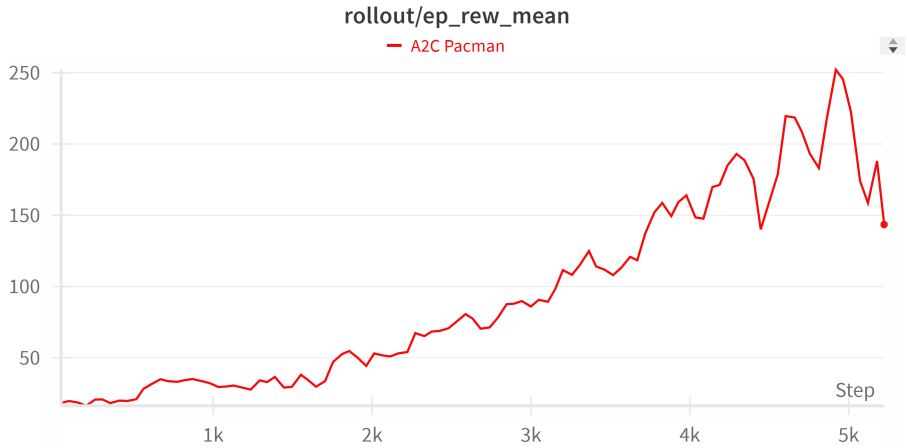

Figure 15: A2C Training mean rewards.

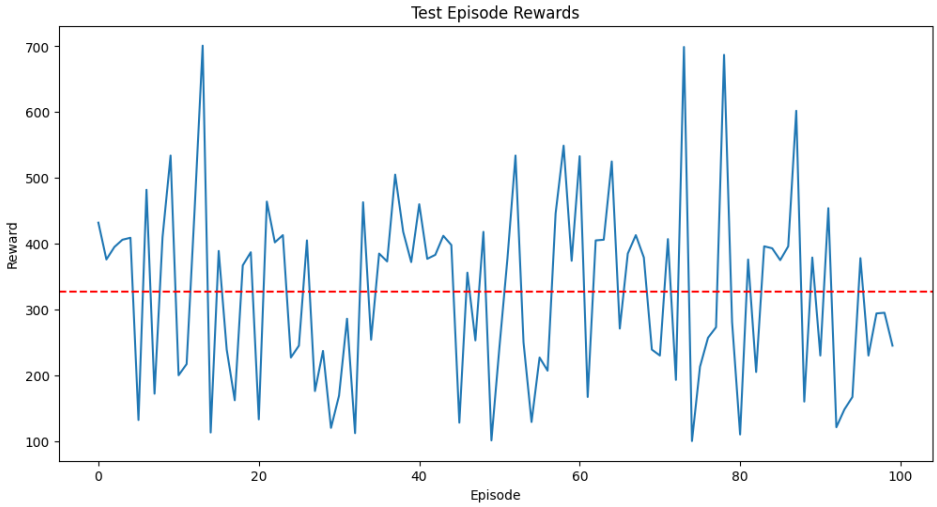

Figure 16: A2C Test rewards.

3.8 Conclusions

PPO demonstrates more consistent rewards, stabilizing around 350 points, but struggles to achieve higher scores. In contrast, A2C exhibits a broader range of rewards, with some runs reaching up to 700 points. Despite these differences in performance patterns, the average rewards for both algorithms are similar, indicating that their overall performance is comparable.

An analysis of the executions detailed in this report reveals that PPO takes significantly longer to converge compared to A2C. However, due to the inherent stochasticity of the environment, direct comparisons are challenging. Identical executions with the same parameters can yield highly variable performances, complicating a definitive evaluation of the algorithms.

4 PONG World Tournament

Pong is a two-player arcade game where the objective is to score points by hitting a ball past the opponent's paddle. By default, the player controls the right paddle.

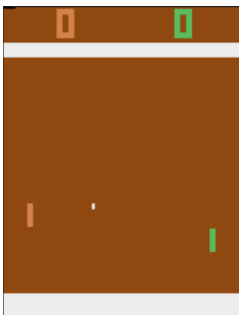

Figure 17: Pong environment.

4.1 Action space

Pong's action space is Discrete(6). They correspond to:

- 0: NOOP
- 1: FIRE
- 2: RIGHT
- 3: LEFT
- 4: RIGHTFIRE
- 5: LEFTFIRE

4.2 Observation space

The default observation space is Box(0, 255, (210, 160, 3), uint8).

4.3 Episode characteristics

In each episode, there are 21 balls, each one corresponding to a possible point score. After the 21 points are played, the episode ends.

4.4 Reward function

- Score a point +1
- Opponent scores a point -1

4.5 Untrained agent performance

The untrained agent consistently gets scores between -21 and -19. Taking into account that the worse reward that can be obtained is -21, the untrained agent is scoring no points at all in the majority of the episodes.

4.6 Proximal Policy Optimization

Since Pong is a multi-agent environment, the agent needs to be capable of playing on both sides of the game. Therefore, two different agents will be trained: one for the left side and one for the right side. Due to the timing, the training began before the tournament preprocessing was clarified. The following sections explain the followed steps.

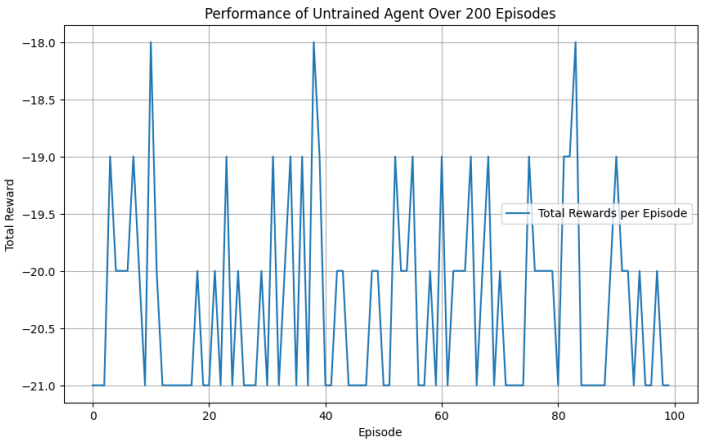

Figure 18: Untrained agent Pong rewards.

4.6.1 Preprocessing

The environment was preprocessed in the following way (this does not follow the tournament specifications):

- Convert the environment to grayscale.
- Frame skipping of 4
- Resize frames to 84x84 pixels.
- Stack the last 4 frames for context.
- Convert data to integers.
- Transposes the observation axes for compatibility.

4.6.2 Training

The pong environment was initialized using 'gymnasium.make(PongNoFrameskip-v4)'. In the default configuration of the Gymnasium environment, the right player is controlled, and this setting cannot be changed. The right agent was trained using a PPO, Figure 19 shows the mean rewards during the training process.

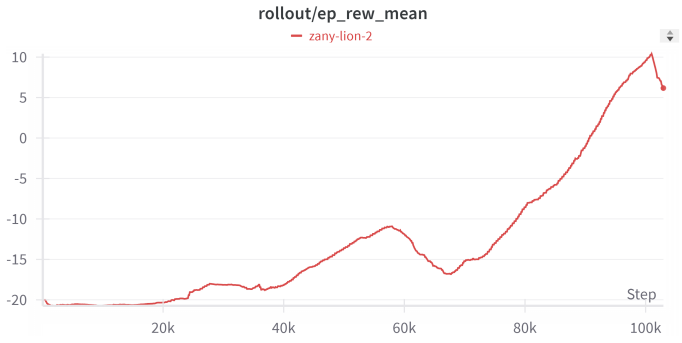

Figure 19: PPO training mean rewards.

The left agent was trained by flipping the observations along the horizontal axis, however, the agent did not learn how to effectively play Pong. After that another approach was used, it included flipping the observations and changing the symbol of the obtained reward. Despite these changes, the agent completely failed to learn how to play pong.

4.6.3 Testing

As seen in Figure 20, the right agent consistently gets scores above 19, with only some episodes underachieving this threshold.

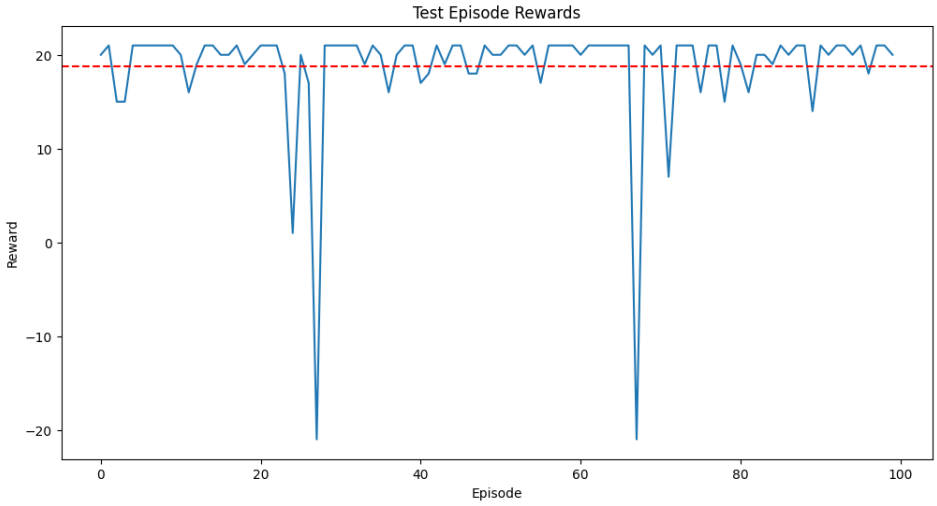

Figure 20: PPO test mean rewards.

4.6.4 Preprocessing tournament

Since the agent needs to be able to play in the Pong tournament, the preprocessing needs to be very specific.

- Convert the environment to grayscale.
- Resize frames to 84x84 pixels.
- Stack the last 4 frames for context.
- Convert data to 32-bit floating-point format.
- Normalizes observations to the range [0, 1].
- Transposes the observation axes for compatibility.

4.6.5 Training

The environment was initialised using 'gymnasium.make(PongNoFrameskip-v4)' and with the tournament preprocessing applied. Despite following this criteria, neither the right agent nor the left made any improvements in the performance.

After realising that the initialised environment does not support simultaneous gameplay with two trained agents, some modifications were implemented. The environment was now loaded using PettingZoo and preprocessed according to the tournament specifications. However, due to limited documentation and technical guidance on using PettingZoo for this specific task, it was not possible to train the agents successfully.

4.7 Conclusions

Training separate agents for the left and right sides of the Pong environment was essential due to its multi-agent nature. Initially, preprocessing steps did not adhere to tournament specifications. While tournament-compliant preprocessing was implemented later, it did not lead to good outcomes.

The right agent, trained with PPO in the default Gymnasium configuration, consistently achieved high scores, surpassing 19 in most episodes. However, the left agent, trained using horizontally flipped observations, failed to learn effective gameplay.

The Gymnasium environment presented limitations, as it did not support simultaneous gameplay with two trained agents. This necessitated a transition to the PettingZoo framework, which was pre-processed according to tournament specifications. Unfortunately, training the agents with PettingZoo was unsuccessful due to limited documentation and technical challenges.