RL Project - Snake Game

Eddy Frighetto - ID: 2119279

Abstract

Reinforcement Learning (RL) algorithms have attracted significant attention from the scientific community due to their ability to learn optimal behaviors through direct interaction with the environment. This project focuses on the design and implementation of RL algorithms capable of training agents to play the game Snake. Algorithms such as Advantage Actor-Critic (A2C), Deep Q-Networks (DQN), and Double Deep Q-Networks (DDQN) were employed. The report outlines the strengths and limitations of each RL approach in the context of the Snake game, along with the training outcomes.

1. Introduction

The Snake Game is a classic arcade-style game where the player controls a growing line (the snake) that must navigate a grid to collect food while avoiding collisions with walls or itself. As the snake eats food, it grows longer, increasing the difficulty of maneuvering without crashing. The Snake environment is really simple, is a quadratic matrix where each value represents:

- **Wall** $\rightarrow 0$
- **Empty space** $\rightarrow 1$
- **Fruit** $\rightarrow 2$
- **Body** $\rightarrow 3$
- **Head** $\rightarrow 4$

Either moves are represented by a numerical value, that are used in the function *move* that influence the environment and returns the relative reward.

- **Up** $\rightarrow 0$
- **Right** $\rightarrow 1$
- **Down** $\rightarrow 2$
- **Left** $\rightarrow 3$

The rewards returned after a move are:

- **Win Reward** $\rightarrow 1$
- **Fruit Reward** $\rightarrow 0.5$
- **Step Reward** $\rightarrow 0$
- **Ate Himself Reward** $\rightarrow -0.2$
- **Hit Wall Reward** $\rightarrow -0.1$

2. Heuristic/Baseline

To establish a performance baseline, has been implemented a simple rule-based (heuristic) policy for the Snake game. This heuristic does not rely on any learning mechanism but instead follows a predefined strategy to select actions based on the current game state. The logic of the heuristic is as follows:

1. **Greedy movement toward the fruit**: The primary objective is to minimize the Manhattan distance between the snake's head and the fruit. This is achieved by evaluating all possible directions and selecting the one that brings the head closest to the fruit.

2. **Obstacle avoidance**: Before moving in the chosen direction, the heuristic checks whether the next cell contains a wall or the snake's body. Unsafe directions are filtered out to avoid collisions.

3. **Fallback strategy**: If the greedy direction is not safe, the heuristic considers the next closest directions (in terms of Manhattan distance) and chooses the best valid one. If no valid moves are available, it selects a random action to continue.

Algorithm 1 Heuristic Policy for Snake Environment

```
Input: Environment env
Output: Actions for each board
actions ← empty list
for each board in env.boards do
    Get head position (x_h, y_h)
    Get fruit position (x_f, y_f)
    candidates ← empty list
    for each action in {UP, RIGHT, DOWN, LEFT} do
        Compute (x', y') ← next position after action
        if (x', y') is valid and not a WALL or BODY then
            distance ← |x' − x_f| + |y' − y_f|
            Append (action, distance) to candidates
        end if
    end for
    if candidates is empty then
        next_action ← random action
    else
        next_action ← action minimum distance to the fruit
    end if
    Append next_action to actions
end for
return actions
```

3. Advantage Actor-Critic

The Advantage Actor-Critic (A2C) algorithm integrates the benefits of both policy-based and value-based reinforcement learning techniques. It consists of two core components: an **actor**, responsible for selecting actions based on the current state via a learned policy, and a **critic**, which evaluates the chosen actions by estimating the expected return.

3.1. Architecture and Learning Approach

The actor and critic share a common feature extractor and diverge into separate output heads: one producing a probability distribution over actions (policy logits), and the other estimating the state-value function. This shared architecture helps reduce redundancy and accelerates convergence by providing both components with a consistent state representation.

3.2. Training the Actor

The actor is trained using the policy gradient method, which adjusts the policy to increase the likelihood of actions that lead to higher returns. To guide this process, the algorithm computes the *advantage*, a measure of how much better an action is compared to the average. These advantages are normalized to reduce variance and stabilize learning. Additionally, an **entropy term** is incorporated into the loss function to encourage exploration by preventing the policy from becoming overly deterministic too early in training (Shrivastav, 2025).

3.3. Training the Critic

The critic is updated by minimizing the mean squared error (MSE) between its predicted state values and the estimated returns. The returns are calculated as a discounted sum of future rewards along a trajectory, with the option of bootstrapping from the critic's own estimate at the final state. This allows the model to benefit from both empirical returns and value-based bootstrapping.

4. Deep Q-Network

The Deep Q-Network (DQN) algorithm extends traditional Q-learning by using a deep neural network to approximate the Q-function, thereby enabling reinforcement learning in high-dimensional state spaces where a tabular approach is infeasible. In this implementation, the neural network maps raw game states to Q-values for each possible action (Mnih et al., 2013).

To improve training stability, DQN employs two key techniques:

- **Experience Replay**: A replay buffer stores tuples of past experience $(s, a, r, s', done)$. During training, mini-batches of these experiences are sampled uniformly at random to break the temporal correlations between consecutive updates.
- **Target Network**: A second neural network, referred to as the target network, is used to compute the target Q-values. It is periodically updated by copying the weights from the main Q-network. This approach helps stabilize learning by preventing the target values from shifting too quickly, as they are computed using a separate, slowly updated network.

4.1. Training Procedure

At each training step, the agent performs the following:

- Uses an **epsilon-greedy** strategy to select actions: with probability ϵ, it explores random actions; otherwise, it exploits the learned policy by selecting the action with the highest predicted Q-value.
- Stores the transition in the replay buffer.
- Samples a batch of experiences from the buffer to perform a gradient descent step, minimizing the mean squared error (MSE) between the predicted Q-values and the target Q-values computed via the Bellman equation: $Q_{target} = r + \gamma \cdot max_a Q_{target}(s', a)$

- Periodically updates the target network with the weights from the Q-network.

The training loop also applies an exponential decay to the ϵ parameter, gradually reducing exploration over time.

5. Double Deep Q-Network

One of the main issues with standard Q-learning and DQN is the tendency to overestimate action values. This occurs because the same Q-network is used both to select the action with the highest estimated value and to evaluate that action. This can lead to overoptimistic value estimates, which degrade learning performance (van Hasselt et al., 2015).

Double Deep Q-Network (Double DQN) addresses this problem by decoupling action selection from action evaluation. Instead of using the same network to perform both roles, Double DQN uses:

- The **online Q-network** (also called the main network) to select the best action in the next state
- The **target Q-network** to evaluate the value of that action

This result in the following target for the Q-learning update:

$$Q_{target} = r + \gamma \cdot Q_{target}(s', argmax_a Q_{main}(s', a))$$

This formulation reduces the overestimation bias by eliminating the max operator's double use on the same Q-function.

5.1. Implementation Details

The implementation of Double DQN extends the standard DQN setup in the following ways:

- The Q-network architecture remains the same, consisting of two convolutional layers followed by a fully connected layer and an output layer predicting Q-values for each action.
- Experience replay is used to sample mini-batches of past transitions for training.
- The target network is synchronized with the main Q-network every fixed number of training steps.
- During training, the main network selects the action with the highest Q-value for the next state, and the target network evaluates this action to compute the target Q-value.

The rest of the training process, including epsilon-greedy action selection, reward bootstrapping, and loss minimization via mean squared error (MSE), mirrors the standard DQN pipeline but incorporates this key correction in target estimation.

6. Justification for Chosen Algorithms

These algorithms were chosen for several key reasons.

Firstly, all selected methods are well-suited for solving Markov Decision Processes (MDPs), which accurately model the dynamics of the Snake environment. Moreover, the game's sparse and delayed reward structure necessitates algorithms capable of promoting stable learning and efficient exploration, capabilities that each of these approaches provides. Lastly, their integration of deep neural networks makes them particularly effective in handling the high-dimensional and complex state spaces inherent to the game.

7. Training

All agents were trained for 5000 iterations in a parallelized Snake environment simulating 1000 boards concurrently. Metrics such as average reward, number of fruits eaten, wall collisions, and training loss were tracked and are reported in the subsequent figures.

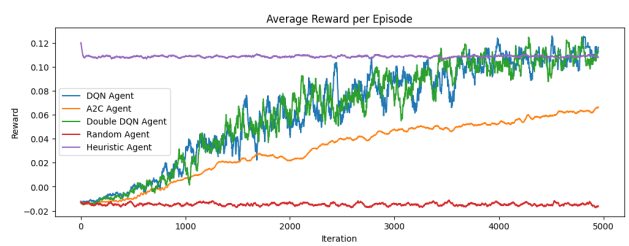

Figure 1. Average reward per episode - Fully Observable

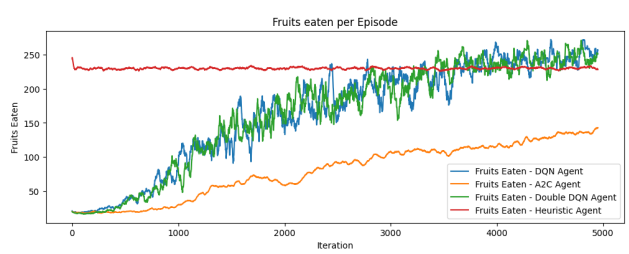

Figure 2. Average fruits eaten per episode - Fully Observable

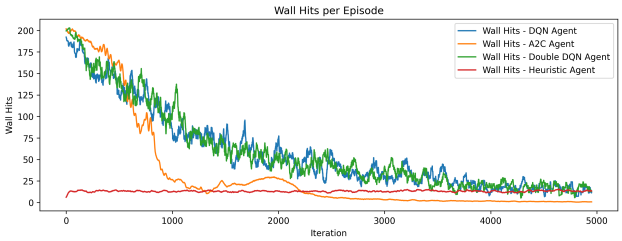

Figure 3. Average walls hitted per episode - Fully Observable

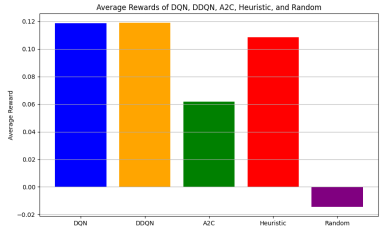

Figure 4. Average rewards between the agents - Fully Observable

8. Results

Figure 1 illustrates that the Heuristic policy performs strongly, providing a solid baseline for evaluating the performance of the trained agents. In contrast, the Random policy, as expected, shows poor performance with consistently low rewards. The Advantage Actor-Critic (A2C) algorithm demonstrates noticeable improvement over the Random policy but still falls short of the heuristic baseline in terms of average reward. On the other hand, the DQN and Double DQN agents yield closely aligned results, effectively learning to navigate the environment by avoiding obstacles and actively collecting fruits, thereby achieving the highest overall performance among the evaluated approaches.

Further insights from Figure 2 show that both DQN and Double DQN have effectively learned to play the game, demonstrating intelligent behavior that goes beyond mere survival. Their performance slightly surpasses that of the heuristic baseline, indicating a more reward-driven strategy. In contrast, the A2C agent, as illustrated also in Figure 3, appears to have focused mainly on avoiding wall collisions, without developing a reliable strategy for fruit collection.

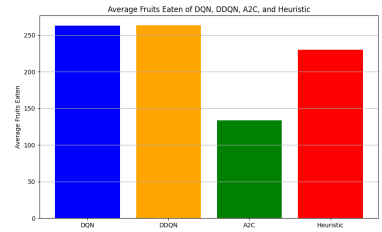

Figure 5. Average fruits eaten between the agents - Fully Observable

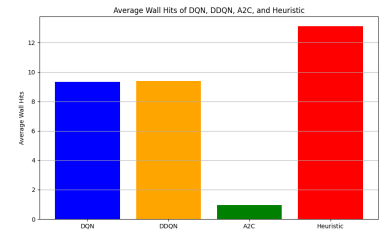

Figure 6. Average wall hitted between the agents - Fully Observable

Table 1. Average results - Fully Observable

METHOD	REWARD	FRUITS EATEN	WALL HITS
RANDOM	-0.01	-	-
HEURISTIC	0.11	230.00	13.12
A2C	0.06	133.70	0.96
DQN	0.12	262.81	9.32
DOUBLE DQN	0.12	263.55	9.38

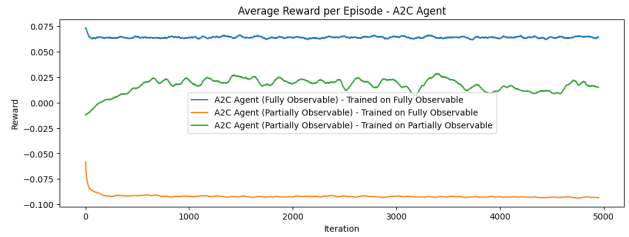

Figure 7. Average reward A2C - Fully vs Partially Observable

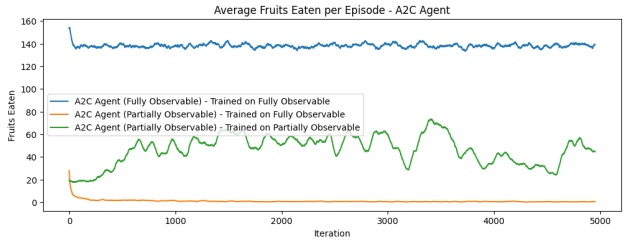

Figure 8. Average fruits eaten A2C - Fully vs Partially Observable

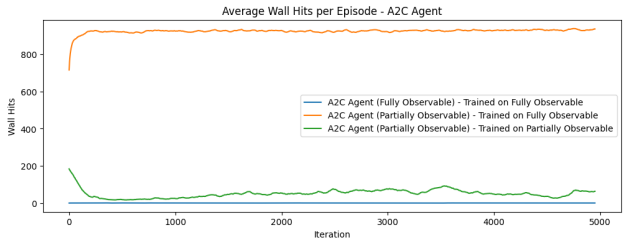

Figure 9. Average wall hits A2C - Fully vs Partially Observable

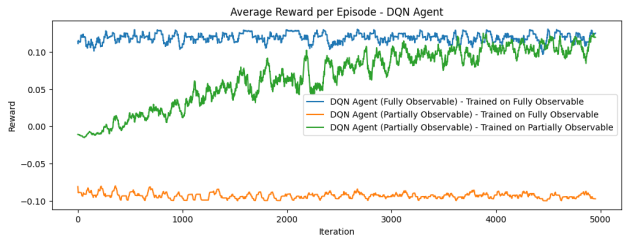

Figure 10. Average reward DQN - Fully vs Partially Observable

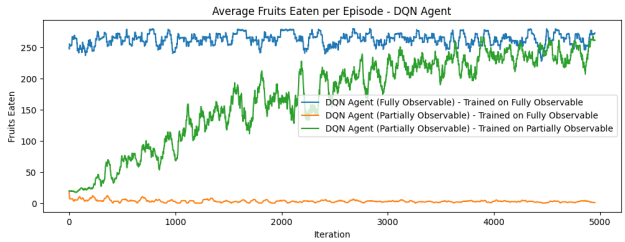

Figure 11. Average fruits eaten DQN - Fully vs Partially Observable

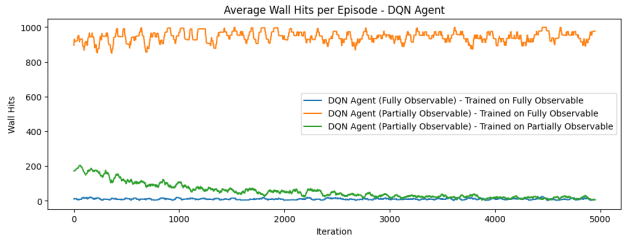

Figure 12. Average wall hits DQN - Fully vs Partially Observable

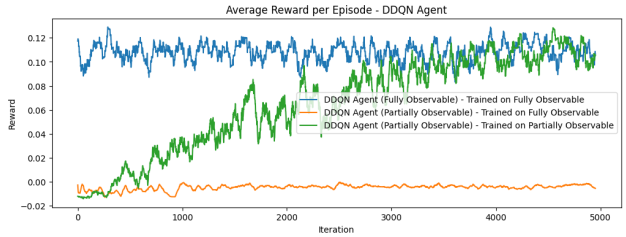

Figure 13. Average reward DDQN - Fully vs Partially Observable

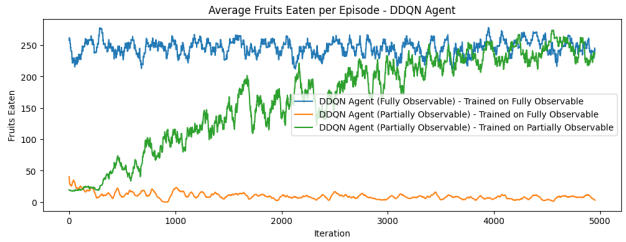

Figure 14. Average fruits eaten DDQN - Fully vs Partially Observable

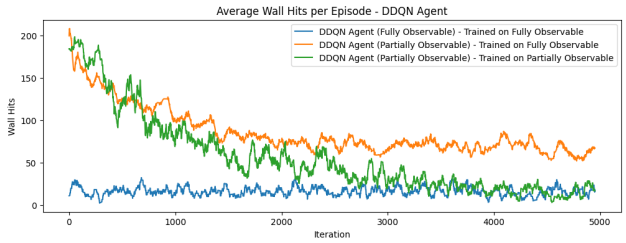

Figure 15. Average wall hits DDQN - Fully vs Partially Observable

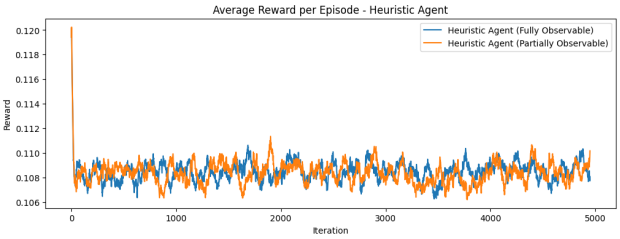

Figure 16. Average reward Heuristic - Fully vs Partially Observable

During the evaluation phase, we also carried out experiments in a partially observable environment. As shown in Figure 7, the A2C agent performed worse when using weights trained in the fully observable environment, an expected outcome given the reduced availability of information. We then retrained the A2C agent directly within the partially observable setting. While this led to a slight improvement in performance compared to the transferred model, it still did not reach the levels achieved in the fully observable environment. These results suggest that partial observability has a significant negative effect on the learning effectiveness of the A2C agent.

For the DQN and Double DQN agents, the results were notably different. When using the weights trained in the fully observable environment, both agents initially performed poorly in the partially observable settings. However, after retraining them directly within the partially observable environment, their performance improved significantly and matched that achieved in the fully observable scenario. These findings suggest that partial observability does not have a lasting negative impact on the learning capabilities of DQN and Double DQN agents, as they are able to adapt effectively to the new environment.

9. Conclusions

Based on the results, it is evident that the different agents adopted distinct strategies for playing Snake. The A2C agent, as previously discussed, appears to have learned that avoiding fruit prevents the snake from growing, allowing it to safely navigate the environment by consistently avoiding both walls and fruits. In contrast, the DQN and Double DQN agents emerged as the top performers, exhibiting nearly identical results. These agents effectively learned to balance fruit collection with obstacle avoidance, adopting reward-maximizing strategies that not only matched but surpassed the performance of the heuristic baseline.

References

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Shrivastav, S. Advantage actor-critic (a2c): Understanding the magic of reinforcement learning, January 2025. URL https://shivang-ahd.medium.com/advantage-actor-critic-a2c-understanding-the-magic-of-reinforcement-learning-3515fce988a4.

van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.