Reinforcement Learning Project
Playing Snake game with deep RL

Matteo Dal Nevo (2087919)

May 14, 2024

Abstract

Reinforcement Learning (RL) algorithms have attracted a lot of attention in the field of artificial intelligence because of their ability to learn optimal behavior through interaction with an environment. The implementation of RL algorithms applied to the classic Snake game is explored in this report, which is a suitable testing ground due to its challenges of strategic decision-making, reward optimization, and sequential action selection.
The discussion will revolve around the design and implementation of RL algorithms that can train agents to play Snake, specifically algorithms like Double Deep Q-Networks (DDQN), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO). The analysis will look at their ability to learn effective strategies for maximizing game scores while avoiding collisions with obstacles or the snake's body. For comparison purposes, a baseline policy will be also included and discussed.
Furthermore, various tests will be conducted to investigate the impact of different network architectures, and training strategies on the learning efficiency and final performance of the RL agents.
Through experimentation and evaluation, this report aims to provide insights into the strengths and limitations of different RL algorithms when applied to the Snake game domain. These findings demonstrate the potential of RL algorithms in mastering complex tasks such as playing Snake, showcasing their adaptability and capability to learn efficient policies through trial and error.

1 Critical Issues and Challenges

While the Snake game provides a conducive platform for RL experimentation, several critical issues and challenges merit consideration:

1. **Sparse Rewards**: One of the primary challenges in the Snake game is the sparsity of rewards, meaning that it is only provided infrequently or in certain states under certain actions. This can make it difficult for the agent to learn effectively the optimal policy.
2. **Exploration-Exploitation Tradeoff**: Balancing exploration and exploitation is crucial in RL, particularly in environments with sparse rewards. RL algorithms must balance the need to explore new actions in different states in a given environment with the need to exploit the current knowledge of the environment to maximize the reward function over time. Furthermore, the game dynamically change over time, increasing its complexity.
3. **Delayed Rewards**: In the Snake game, rewards are often delayed, requiring to the agent to make decisions based on information that may not be available for a long period of time. For RL agents can be challenging to attribute actions to corresponding outcomes accurately. Delayed rewards can hinder learning progress, requiring sophisticated algorithms and training strategies to overcome temporal credit assignment issues.
4. **Generalization and Transfer Learning**: Achieving generalisation and transfer learning capabilities is essential for RL agents to apply learned policies to new, unseen state of the game effectively. Ensuring that agents can adapt and generalise their learned behaviours beyond the training environment is critical for their applicability.
5. **Approximation and State Representation Dependencies**: RL algorithms often rely on function approximation techniques, such as neural networks, to approximate value functions or policies. The choice of state representation and the approximation capabilities of the neural network can significantly impact the learning process and the quality of learned policies. Dependencies between the state representation and the neural network's ability to generalize may introduce biases or limitations, requiring careful consideration and potentially specialized architectures or training approaches.

2 Baseline Strategy

The implemented baseline strategy revolves around making intelligent decisions for the snake's movement to optimize its chances of reaching the food without colliding with its own body. This section provides an extensive explanation of the baseline strategy, and why it is considered reasonable for the task.

Objective

The primary objective of the baseline strategy is to guide the snake towards the food while avoiding collisions with its own body. This involves analyzing the current game state, including the positions of the snake's head, the food, and its body segments, to determine the optimal direction of movement. The action space dimension is four, corresponding to **UP, RIGHT, DOWN** and **LEFT**.

Components of the Baseline

The baseline strategy consists of several components that work together to achieve the objective:

Position Analysis: The baseline begins by analyzing the positions of the snake's head and the food obtained from the actual state of the environment. The horizontal and vertical distances between the head and the food are computed to determine the relative positions.

Direction Selection: Based on the relative positions of the head and the food, the baseline selects the optimal direction for the snake to move. It prioritizes movements that minimize the distance to the food while ensuring no collision with the snake's body occurs. This approach essentially creates a decision tree where the algorithm checks each direction, choosing the best option based on these criteria.

Collision Avoidance: To prevent the snake from colliding with its own body, the algorithm checks if the proposed movement direction would result in a collision with any part of the snake's body. This is achieved through the `is_colliding_with_body` function, which efficiently identifies collisions based on the current body positions.

Fallback Mechanism: In case there are no feasible movements that satisfy the conditions (e.g., all directions lead to collisions), a fallback mechanism is employed in the baseline to randomly select a direction. This ensures that the snake continues to move even in challenging scenarios.

This algorithm assumes that the chosen environment is the fully observable. At each iteration an action is chosen based on the logic explained above and is fed to the environment to collect the reward and move to the next state. This iterative process ends after a fixed number of steps (e.g. 20000).
The flowchart is shown in Fig. 1.

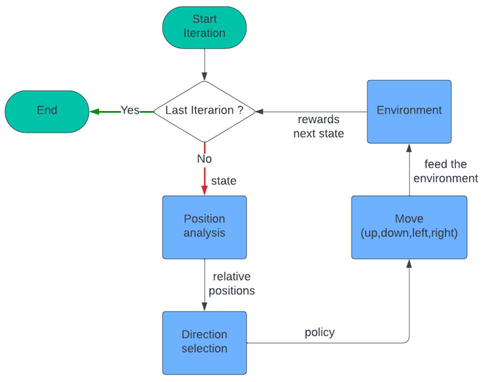


Figure 1: Flowchart of the heuristic algorithm

In conclusion, by using the baseline strategy to solve the game, the snake can be guided towards the food without colliding with its own body. Its heuristic-based decision-making process, focus on safety and efficiency, adaptability to game dynamics, and simplicity make it a viable starting point for developing more advanced strategies.

3 Deep Reinforcement Learning

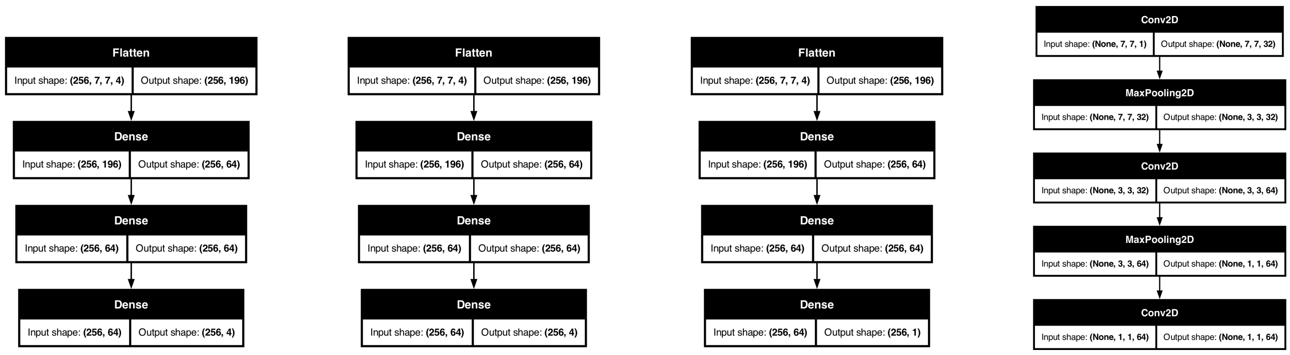
This section discusses about the reinforcement learning (RL) algorithms implemented in the project, namely Double Deep Q-Networks (**DDQN**) [3], Advantage Actor-Critic (**A2C**) [1], and Proximal Policy Optimization (**PPO**) [2]. These algorithms are suitable candidates for training RL agents in the Snake game due to their unique advantages and trade-offs. The parameters of the neural networks and their architectures are reported in Table 1 and in Fig. 2, respectively. The only difference compared to the paper [3] refers to the experience replay buffer, which has not been implemented, but has been partialy mitigated by parallelizing the environments.

Double Deep Q-Networks (DDQN): Double-DQN is an extension of the traditional Deep Q-Network (DQN) algorithm, designed to address overestimation bias in Q-learning. By decoupling action selection and action evaluation, Double-DQN improves the stability and convergence of Q-learning algorithms. This specific approach was chosen for its simplicity, effectiveness, and proven performance in various RL domains.
Additionally, Double-DQN is well-suited for environments with discrete action spaces, making it a natural choice for the Snake game. The action is selected according to an ϵ-greedy approach to balance the exploration and exploitation, this parameter decay with a linear behaviour that depends on the number of training steps.
The used architecture consists of a feedforward neural network with three dense layers. The input layer is flattened to accommodate the state representation (that is the provided categorical representation of the board game), followed by two (dense) hidden layers (64 neurons for each hidden layers) with ReLU activation functions chosen to deal with the vanishing gradients problem. The final output layer predicts Q-values for each action in the given state.

Advantage Actor-Critic (A2C): A2C is a policy gradient-based RL algorithm that combines the advantages of both actor-critic and advantage estimation techniques. By using a separate actor and critic network, A2C facilitates more stable and efficient learning. The actor network outputs the policy distribution over actions, while the critic network estimates the state-value function. This algorithm was selected for its suitability for environments with sparse rewards, such as the Snake game. A2C's asynchronous training paradigm also allows for parallelization, leading to faster convergence and improved sample efficiency.
For both networks the input layer is the same as that of the DDQN network, then it consists of two (dense) hidden layers with hyperbolic tangent (tanh) activation functions and a softmax activation function for the actor's output layer and a linear activation function for the critic's output layer (both have 64 neurons for each hidden layers).

Proximal Policy Optimization (PPO): PPO is a state-of-the-art policy optimization algorithm that addresses issues of sample efficiency and stability in policy gradient methods. By constraining the policy update step to ensure limited divergence from the previous policy, it achieves more reliable and robust learning. This choice was motivated by its inherent simplicity, ease of implementation, and demonstrably strong performance across diverse reinforcement learning tasks. PPO's adaptability to different environments and its ability to handle both discrete and continuous action spaces make it a versatile choice for training RL agents in the Snake game. The Neural Network architecture used is the same as the ones used for the A2C algorithm.

CNN:
For the purpose of improving the performance and test different state representation an alternative architecture was developed. This variant utilizes convolutional layers to process image-based state representations (normalized game board). The neural networks start with convolutional layers, which are composed first of 32 and then 64 neurons, a 3 by 3 kernel, and a ReLU activation function. Then followed by max-pooling layers to extract features from the input images. The extracted features are then flattened and fed into the FFNN reported previously.


Flatten
Input shape: (256, 7, 7, 4) Output shape: (256, 196)
Dense
Input shape: (256, 196) Output shape: (256, 64)
Dense
Input shape: (256, 64) Output shape: (256, 64)
Dense
Input shape: (256, 64) Output shape: (256, 4)
Flatten
Input shape: (256, 7, 7, 4) Output shape: (256, 196)
Dense
Input shape: (256, 196) Output shape: (256, 64)
Dense
Input shape: (256, 64) Output shape: (256, 64)
Dense
Input shape: (256, 64) Output shape: (256, 4)
Flatten
Input shape: (256, 7, 7, 4) Output shape: (256, 196)
Dense
Input shape: (256, 196) Output shape: (256, 64)
Dense
Input shape: (256, 64) Output shape: (256, 64)
Dense
Input shape: (256, 64) Output shape: (256, 1)
Conv2D
Input shape: (None, 7, 7, 1) Output shape: (None, 7, 7, 32)
MaxPooling2D
Input shape: (None, 7, 7, 32) Output shape: (None, 3, 3, 32)
Conv2D
Input shape: (None, 3, 3, 32) Output shape: (None, 3, 3, 64)
MaxPooling2D
Input shape: (None, 3, 3, 64) Output shape: (None, 1, 1, 64)
Conv2D
Input shape: (None, 1, 1, 64) Output shape: (None, 1, 1, 64)

(a) DDQN network (b) A2C actor network (c) A2C critic network (d) CNN

Figure 2: Neural networks architectures

Overall, the selection of DDQN, A2C, and PPO algorithms provides a comprehensive framework for training RL agents in the Snake game environment. By leveraging the unique strengths of each algorithm and conducting rigorous experimentation, the final goal is to gain deeper insights into the learning dynamics and performance characteristics of RL agents in complex, dynamic environments.

Hyperparameters	DDQN	A2C	PPO
Input Layers neurons	256	256	256
Hidden Layers	2	2	2
Hidden Layers neurons	64	64	64
Hidden Activation function	ReLu	tanh	tanh
Output Layers	4	4 / 1	4 / 1
Output Activation function	Linear	Soft-Max / Linear	Soft-Max / Linear
Learning Rate	1e-3	3e-4	3e-4
Optimizer	Adam	Adam	Adam

Table 1: Models architectures

4 Experimental Results

This section presents the obtained experimental results, comparing the learning progress of the RL agent trained using DDQN, A2C, or PPO algorithms with the baseline strategy in the fully observable scenario. Its is also discussed the evaluation metrics used to measure the performance of the RL agents and other different trial for improving the final result.

4.1 Evaluation Metrics

To measure the performance several evaluation metrics are employed, including:

- **Average Reward**: The average reward obtained by the RL agent over a fixed number of episodes during training. Higher average rewards indicate better performance.
- **Average eaten fruit**: The average fruit eaten by the RL agent during training. Higher value of eaten fruit suggest improved efficiency and effectiveness in achieving the task goals.
- **Average Collision Rate**: The rate of collisions with obstacles or self encountered by the RL agent during training. Lower collision rates indicate better navigation and avoidance strategies.

These evaluation metrics are computed for both the RL agents and the baseline strategy to assess the relative performance and effectiveness of each approaches.

4.2 Learning Progress Comparison

Base case:
To compare the learning progress of the RL agent with the baseline strategy, the metrics are tracked for each agent over time. The results of the training are shown in Fig. 4, it illustrates the learning progress comparison between the RL agents and the baseline over the course of training epochs. Each epoch is 200 steps, so a total of 20000 steps have been taken in the environment. The only difference is on PPO, the algorithm was trained for a total of 150000 steps, so one epoch corresponds to 1500 steps.
The hyperparameters used during training are $\gamma = 0.99$ for all the algorithms and for the DDQN the exploration-exploitation parameter decays linearly starting from 1.0 to a minimum of 0.1 in 2000 steps, as shown in Fig.3.

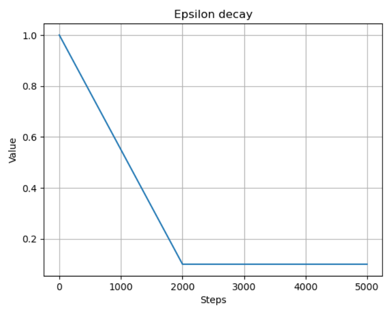


Figure 3: ϵ - linear schedule

The agents trained with DDQN, A2C or PPO algorithms exhibit distinct patterns of learning progress, with each algorithm showcasing its own strengths and weaknesses. Among these algorithms, PPO emerges as the algorithm with the slowest learning pace and the poorest performance. This can be attributed to the constraints imposed on policy update, which must not differ too much from the previous one. A possible tuning of the parameters (e.g. learning rate, clipping ratio, entropy bonus) could lead to better results, but due to the extensive training time of 3 hours, no further tests were conducted.
In contrast, both Double-DQN and A2C algorithms demonstrate similar trends in terms of average reward and fruit consumption as they interact more with the environment. However, they diverge in their learning strategies. DDQN prioritize to avoid collisions with walls and consuming the fruits simultaneously, while A2C first focuses on not to collide with walls and then to consumes the fruits.
This results in distinct learned policies between the two algorithms. Indeed, the number of collisions with the body differs significantly between the them. It seems that A2C places less emphasis on the risk of self-collision and prioritizes getting the fruit as soon as possible. In contrast, DDQN has learned to adopt a more cautious approach by avoiding direct paths to the fruit, resulting in fewer collisions with its own body. This distinction highlights the delicate differences in the learned policies of the two algorithms.
Initially, DDQN tends to converge faster, but A2C exhibits more stable and consistent performance over long training periods. This suggests that the choice between these algorithms may depend on the specific requirements of the task and the desired trade-off between learning speed and stability.
Comparing the heuristic policy with the learning algorithms, it is not possible to notice a significant difference in performance. In fact, in some cases, they are even worse. In terms of average rewards, eaten fruits and wall hits the steady state performance are pretty close to each other, while the average body hits are very different.

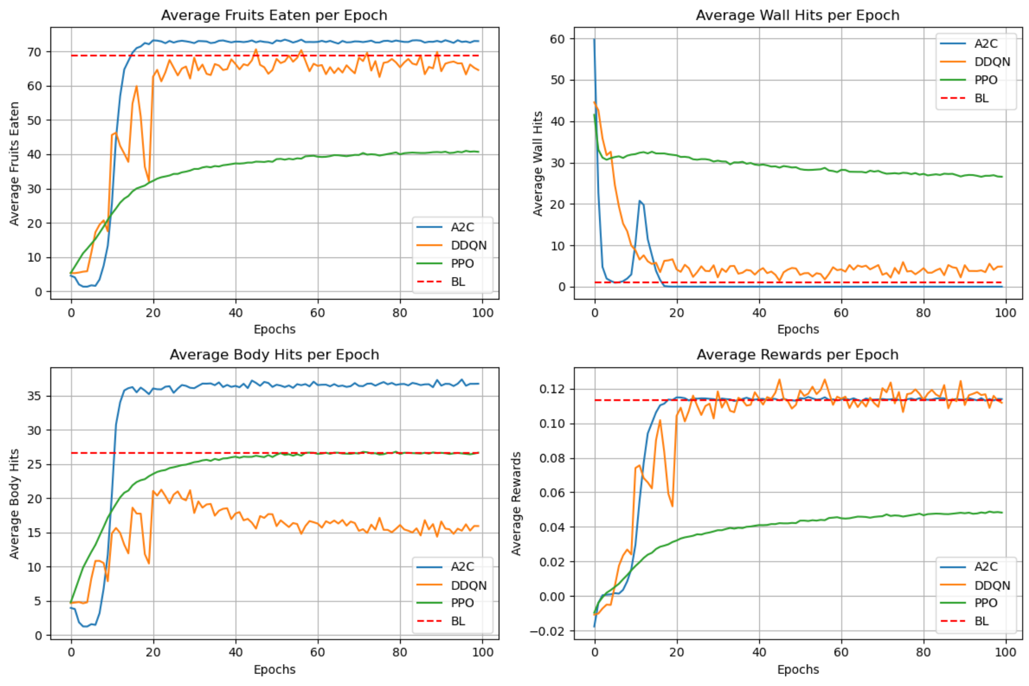


Figure 4: Learning Progress Comparison between RL Agents and Baseline

As shown in Fig. 5, the evaluation of the agents confirms the previous comments. The DDQN agent is able to beat the heuristic policy w.r.t all study metrics. The evaluation of the agents was carried out by exploiting the optimal policy, i.e. by selecting the action that maximizes the Q-value function or by choosing the action with the highest probability.
In this scenario there are almost no clashes with the walls and it can be seen that the average reward is higher than the heuristic policy, meaning that on average the DDQN agent are able to choose the optimal action for a given state more effectively than the heuristic policy or the other RL algorithms.

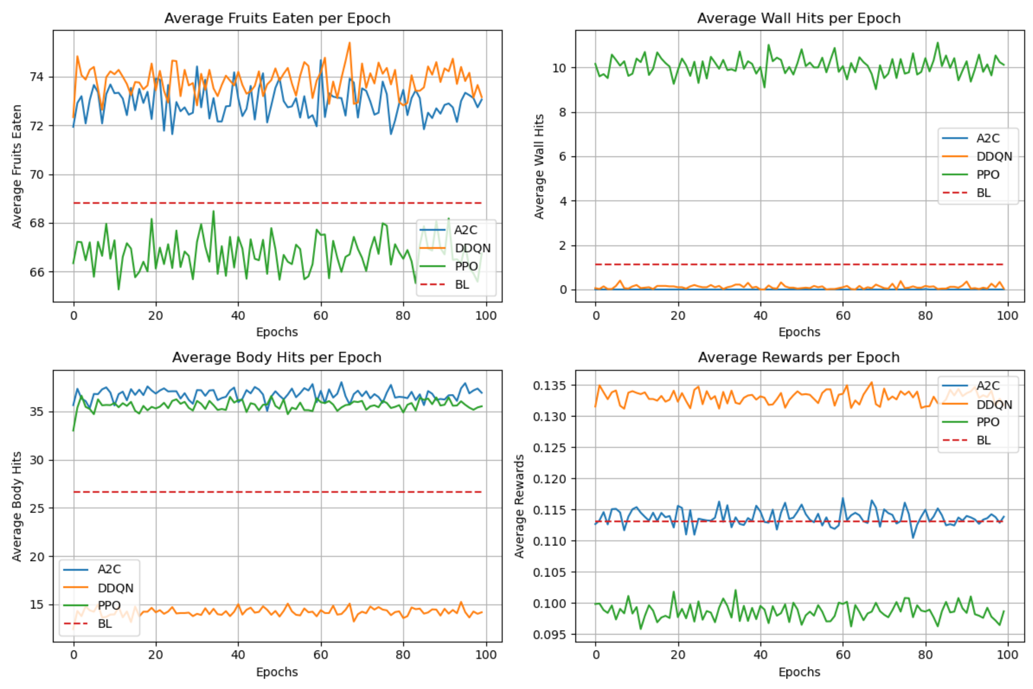


Figure 5: Evaluation Comparison between RL Agents and Baseline

Different state representation:
Contrary to expectations, as depicted in Fig. 6 and 7, the performance of both DDQN and A2C agents trained with CNN-based feature extraction and the modified state representation was lower compared to previous experiments. Despite the agents' ability to grasp the basic concepts of the game state, they struggled to learn effective policies as the game complexity increased, resulting in suboptimal performance, worse than heuristic policy during the training phase and a little bit better during the evaluation phase.
These negative results suggest that the alternative state representation may not be the best option.
The experiment indicates that the use of CNNs for feature extraction does not always improve the performance of DDQN and A2C agents when using a different state representation. This underscores the importance of carefully designing and evaluating state representations in conjunction with neural network architectures for reinforcement learning tasks.

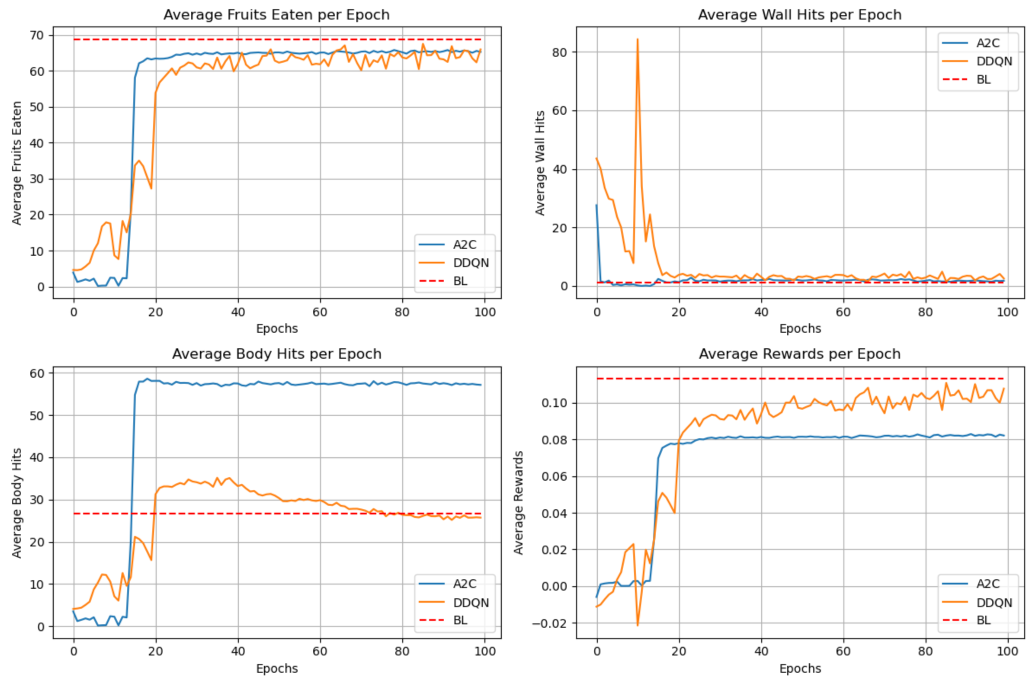


Figure 6: Learning Progress Comparison between RL Agents and Baseline using different state representation

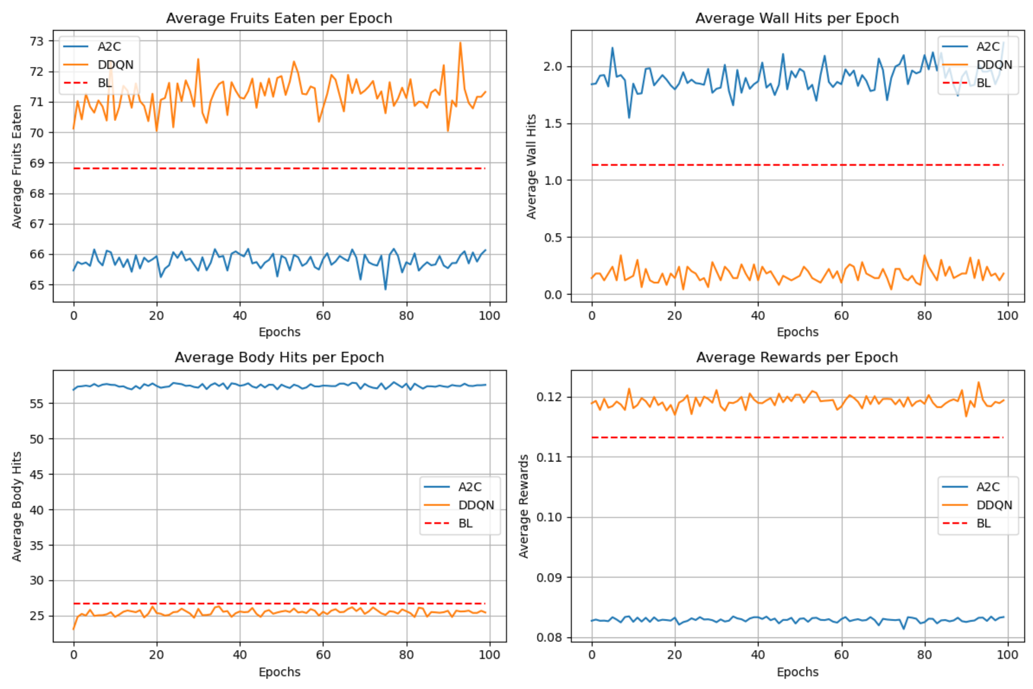


Figure 7: Evaluation Comparison between RL Agents and Baseline using different state representation

Other tests:
In an effort to further improve the performance of the agents, additional avenues were explored. However, the outcomes of these experiments did not surpass the performance achieved in previous tests:

- **Masking Actions:** Initial tests involved masking actions to prevent the snake from making movements that could potentially lead to collisions with walls. However, as the agents demonstrated adeptness in avoiding walls after a small number of training steps, this approach did not yield significant improvements. Future work could focus on refining the action masking strategy to also prevent self-collisions, potentially leading to better outcomes.

- **Reward Function Modification:** Another approach was to modify the reward function to penalize more the snake for body collisions. Despite efforts in this direction, the adjusted reward structure did not result in significant performance gains. Instead, it led to a reduction in the average reward. To effectively motivate desirable behaviors, it may be necessary to explore other reward structures.

- **Hyperparameters Modification:** The last attempt was made by reducing the minimum ϵ parameter, bringing it from 0.1 to 0.01. This led to an improvement during the final phase of the training, but no significant difference was observed in the evaluation phase.

These additional tests underscore the iterative process of research in reinforcement learning, emphasizing the importance of experimentation and refinement in discovering effective strategies and enhancing agent performance. While these specific methods didn't achieve the desired results, they offer valuable insights for future exploration and potential improvement paths.

5 Conclusions

Overall, the DDQN agents performs better than the heuristic policy and the findings of this study demonstrate the potential of RL algorithms in mastering complex tasks such as playing the Snake game, showcasing their adaptability and capability to learn efficient policies through trial and error. However, there are still avenues for improvement and further research, particularly in fine-tuning algorithm parameters, exploring alternative architectures, and refining state representations to enhance performance.

References

[1] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.