Introduction to Reinforcement Learning with Snake Game

Seoho Yun

Abstract

In this project, we utilize the snake game from Git Hub as the basis for our analysis and training using Reinforcement Learning techniques. The primary objective of this paper is to elucidate interactions and effects that various parameters have on the overall winning rate of the game. Also, the secondary objective is to explain how these techniques can be applied to real-life scenarios.

1 Introduction

So far, we focus mostly on implementation of linear-regression and neutral networks. In this project, we aim to broaden our understanding of various sides of machine learning, particularly focusing on the field of Reinforcement Learning (RL). Overall, this project serves as a practical exploration of RL through the lens of a simple game. To achieve this, we have chosen a simple game developed by Vincent during SPR 2023 as an educational tool. It is essential to emphasize that the game's source code should only be shared and utilized for educational purposes. Distribution beyond these educational boundaries is strictly prohibited.

2 Contextual and Theoretical Background

The primary objective of this snake game is to guide the snake's movements in such a way that it can consume the apples. However, it is crucial to be mindful of certain conditions that can result in the game over. Firstly, the snake must remain within the specified boundaries of the game area. If it ventures beyond these boundaries, it will meet an unfortunate end, and the game will conclude. The given code can change the dimension of the boundaries, but we will focus on only 4 by 4 case to simplify the calculation. Furthermore, the snake's survival is contingent on avoiding a self-collision. If the snake's head collides with any part of its own body during its movement, it will result in an immediate game over. By considering these two critical conditions, the player must control the snake's movement so that the snake can fulfill the entire grid.

For the training phase, this model uses Monte-Carlo methods instead of dynamic programming. The rationale behind this choice stems from the inherent complexity of real-life environment dynamics which has to deal with noise of the environment, agents, and sensors. Rather than focusing on the entire dynamics, the model relies on simulating every possible movement of the snake. This approach, commonly referred to as Monte-Carlo simulation, provides a practical solution for training the snake game.

A question one may have is how this model is trained. The training process for this snake game involves four important hyperparameters: the number of episodes, ϵ , γ, and a reward vector.

First of all, the number of episodes determines how many times the model is trained. Generally, a larger number of episodes leads to better learning. However, due to limitations of hardware(the absence of GPUs on local machines), training beyond 50,000 episodes is computationally heavy and may damage the local machines. Therefore, it is critical to make a balance between the number of episodes and computability.

Also, the exploitation-exploration constant, ϵ, plays a huge role in training process. Typically, in RL, this constant ranges between 0 and 1, determining the balance between exploitation and exploration. When ϵ is 0, the model always chooses the same path as it believes that way is the most optimal way. This strategy is known as exploitation, as it focuses on maximizing performance based on existing information. On the other hand, when ϵ is equal to 1, the model randomly guesses without learning anything. This strategy is known as exploration, as it aims to gather more information about the available choices and explore the environment to potentially discover better decisions.

Next, γ adjusts the importance given to future rewards compared to immediate rewards. In the game, after the snake consumes an apple, a new apple appears in the non-occupied squares in the grid uniformly at random. Since the model cannot predict the exact location of future apples, it needs to make decisions to focus on both current and future apples based on probability. γ handles this trade-off, where a value of 0 indicates no consideration for future rewards, and a value of 1 represents complete emphasis on future rewards.

Finally, the reward vector is crucial for training the model. This vector allows for the assignment of reward values to four different cases: losing move, inefficient move, efficient move, and winning move. The weights or rewards associated with these cases determine how the model is trained, as it aims to optimize its decision-making based on the assigned reward values. By appropriately tuning these hyperparameters and designing a well-calibrated reward vector, the model can be trained to make efficient and effective decisions within the game, ultimately maximizing its winning rate.

3 Implementation Details

To run the code successfully, it is necessary to obtain the entire file from the GitHub repository, rather than just the launch.py file. Moreover, it's important to note that this code may not execute in local IDEs such as Jupyter Notebook since it accesses multiple files from different directories. In order to overcome this, it is recommended to create an Anaconda environment that includes the gymnasium package for reinforcement learning. Once the environment is set up, the code should work as intended and you should be able to play the snake. If you encounter any bugs or issues, please report them on the corresponding GitHub page.
The complete launch.py file should look like this:

```python
from game.Snake import Snake
from reinf.SnakeEnv import SnakeEnv
from reinf.utils import perform_mc, show_games

# Winning everytime hyperparameters
grid_length = 4
n_episodes = 30000
epsilon = 1.0
gamma = 0.8
rewards = [-500,-50, 300, 100000]
# [Losing move, inefficient move, efficient move, winning move]

# Playing part
# game = Snake((800, 800), grid_length)
#game.start_interactive_game()

# Training part
env = SnakeEnv(grid_length=grid_length, with_rendering=False)
q_table = perform_mc(env, n_episodes, epsilon, gamma, rewards)


# Viz part
env = SnakeEnv(grid_length=grid_length, with_rendering=True)
show_games(env, 1000, q_table)
```

Please note that in order to deactivate the game play and focus solely on the training and testing phases, you are required to comment out the playing part.

4 Computational Results

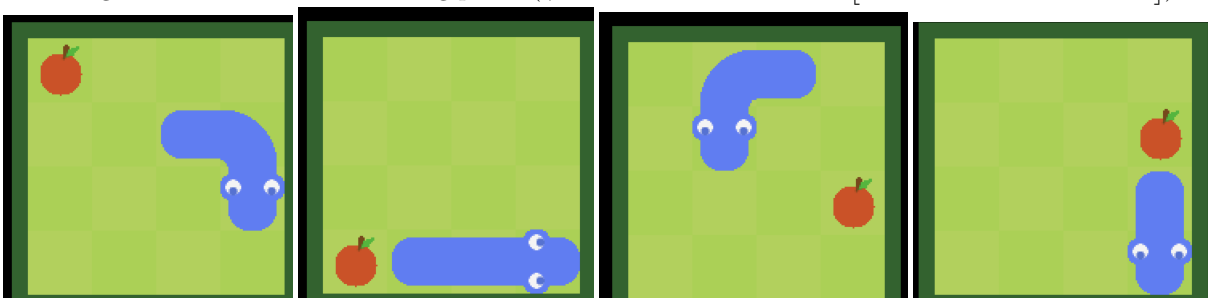
As stated in the mathematical background, our focus will not be on the number of episodes due to limitations imposed by the local machine.

4.1 Exploration-Exploitation

First of all, when $\epsilon = 0$, it encounters a failure. It attempts to select an action based on the current knowledge or policy. However, since it lacks any information about the game, it is unable to make any

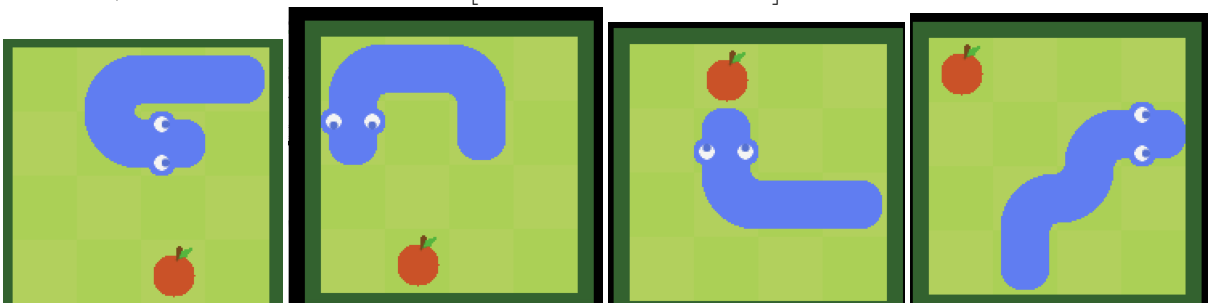
moves. In other words, it is constrained to follow the previous movement, but in this case, there is no previous movement available. This code throws an error.

If $\epsilon = 1$, the model continuously explores without saving its results or comparing its past and current paths. Consequently, the snake keeps moving without any meaningful trainings of the model. Here are four images of snakes after the training phase ($\gamma = 0.2$ and reward vector $\begin{bmatrix} -300 & -50 & 300 & 10000 \end{bmatrix}$):



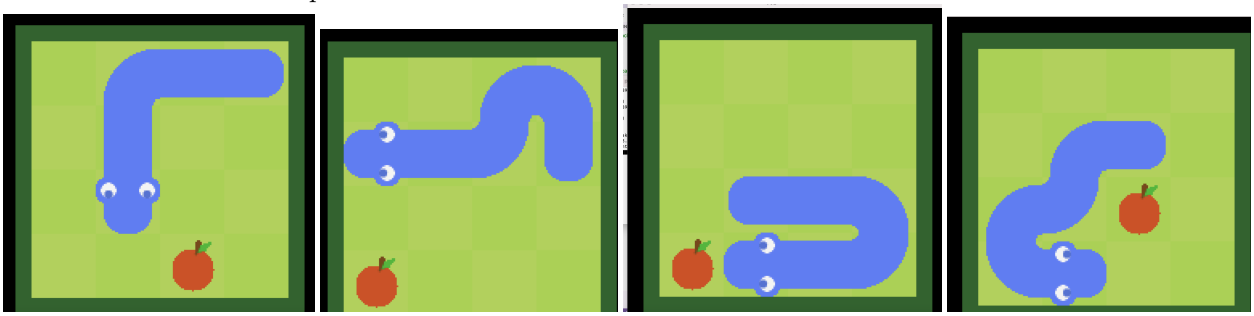
Each screenshot represents a different testing round. In the first screenshot, the snake moves to a random location without recognizing the apple because it wants to try a new strategy, which, unfortunately, proves to be a disastrous choice. In the second screenshot, when a new apple is present, the snake does not prioritize other parameters such as γ to believe ignoring it yields better solutions. The third screenshot shows that sometimes the snake finds a reasonable movement to locate an apple. However, since it does not save successful results and build upon them, every movement it makes is purely probabilistic, preventing the creation of any statistical models. Finally, due to the snake's almost chaotic movement, it occasionally exhibits strange behavior, such as ignoring apples or repeatedly moving back and forth (which causes a bug in the code).

When $\epsilon = 0.5$, the model simultaneously exploits and explores with equal probabilities. The parameters used are $\gamma = 0.2$ and the reward vector $\begin{bmatrix} -300 & -50 & 300 & 10000 \end{bmatrix}$. Here are four screenshots:



Each screenshot represents a different testing round. In the first screenshot, the snake follows the strategy that was trained and is on the verge of exploring a new strategy. In the second screenshot, when a new apple appears, the snake efficiently navigates towards it by exploiting the previously trained moves. The third screenshot demonstrates that this combined exploration and exploitation approach works well, as the snake successfully focuses on the apple while still exploring potential better moves. However, the fourth screenshot reveals that sometimes this exploration strategy fails, resulting in a game-over scenario.

When $\epsilon = 0.25$, the model primarily exploits the learned knowledge and sometimes explores new possibilities($\gamma = 0.2$, and the reward vector is $\begin{bmatrix} -300 & -50 & 300 & 10000 \end{bmatrix}$). Presented below are four screenshots of the snake's performance:

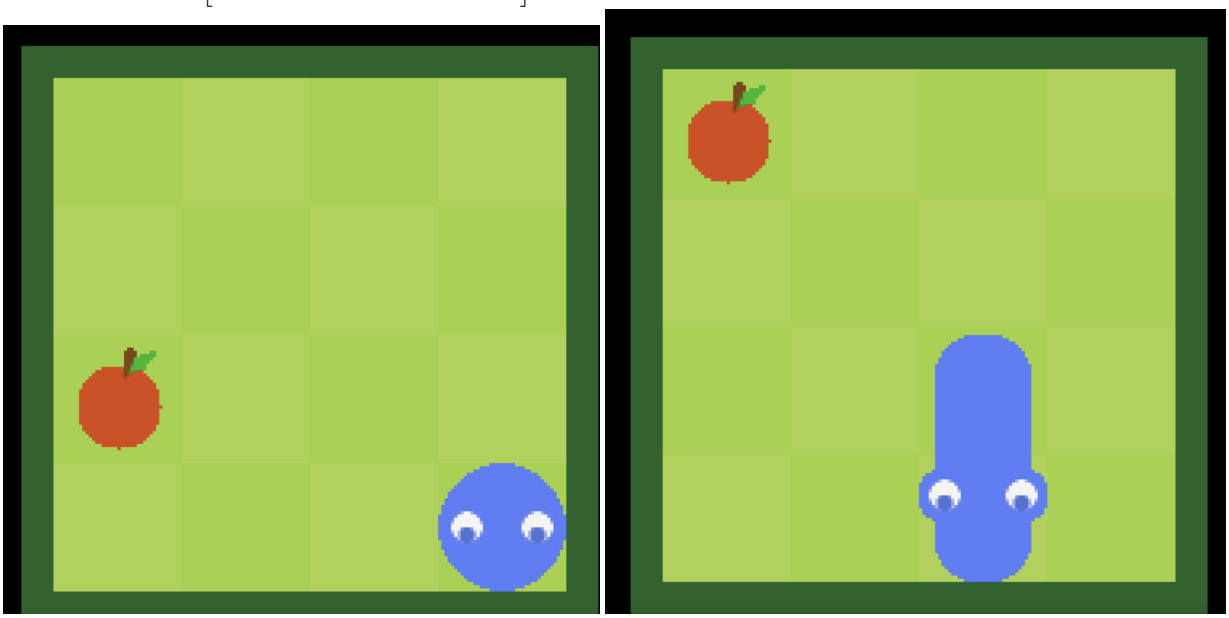


One notable observation is that the length of the snake is actually longer compared to the previous

scenarios. This is because the snake has acquired an understanding of how to optimize while also experimenting with new approaches to further enhance its performance. The first three images demonstrate that the model has learned valuable insights about controlling the snake. Furthermore, in the fourth image, although the snake is attempting a new strategy, it proves to be highly effective as it builds upon the previously optimized approaches. As the model continues to learn, it becomes increasingly challenging for the user to provoke game over, as the snake learns how to survive more frequently and effectively.

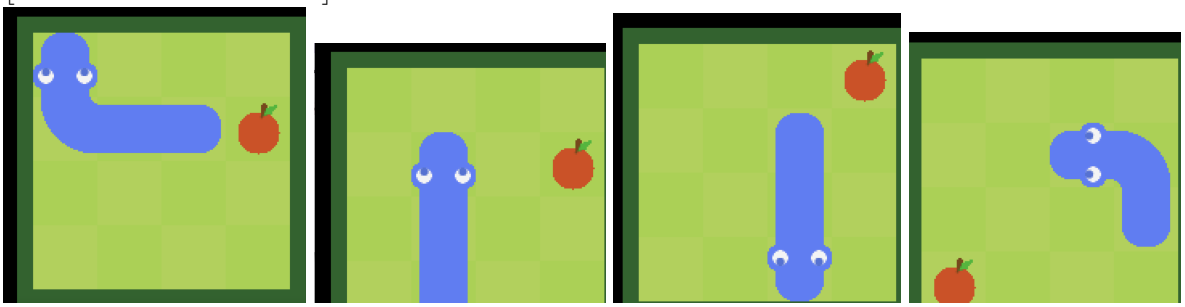
4.2 Future Emphasis

When $\gamma = 0$, this model prioritizes the current reward without considering future rewards. Presented below are two images showcasing the snakes' behavior after the training phase, utilizing $\epsilon = 0.25$ and a reward vector of $\begin{bmatrix} -500 & -50 & 300 & 100000 \end{bmatrix}$:



As the model exclusively focuses on the current reward, the snake becomes fixated on reaching the apple. Additionally, with γ set to the extreme value of 0, the snake continuously oscillates back and forth, aware of the difficulty in approaching the apple. This behavior typically occurs after the snake dies three or four times. Consequently, while the current reward may appear favorable, the snake finds itself in a state of stagnation, unable to effectively reach the apple.

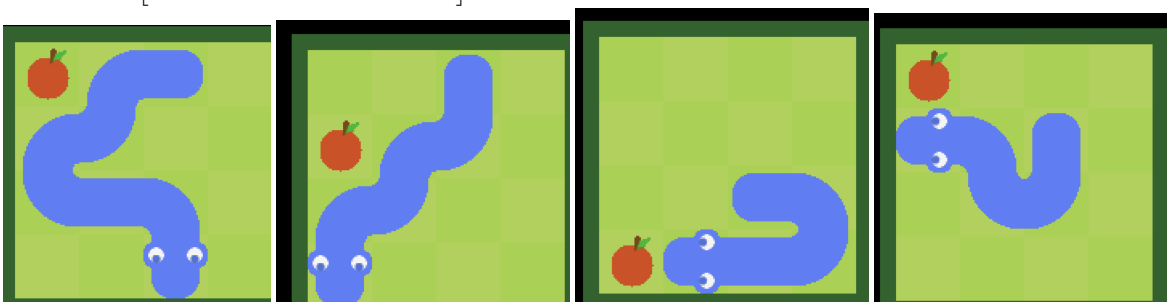
When $\gamma = 1$, this model prioritizes future rewards over the current apple. Here are four screenshots showcasing the behavior of the snake after the training phase, using $\epsilon = 0.25$ and a reward vector of $\begin{bmatrix} -500 & -50 & 300 & 100000 \end{bmatrix}$:



As the model solely focuses on future rewards, the snake tends to disregard the current apple. The first two screenshots clearly demonstrate that the snake does not prioritize the current apple. Particularly, the third screenshot reveals that the snake even moves away from the current apple to optimize its position for future rewards. Fortunately, as shown in the fourth screenshot, there are instances where the snake may approach the current apple if it predicts that a future apple will appear in close proximity.

When $\gamma = 0.5$, this model aims to balance the importance of current and future rewards. Here are four screenshots illustrating the behavior of the snake after the training phase, using $\epsilon = 0.25$ and a

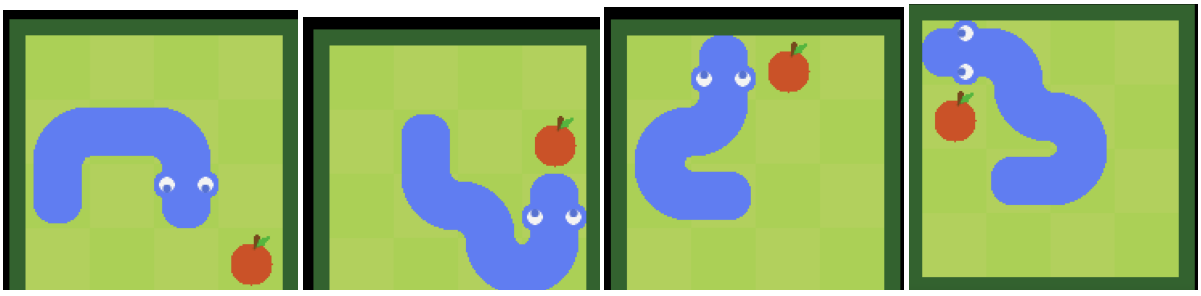
reward vector of $\begin{bmatrix}-500 & -50 & 300 & 100000\end{bmatrix}$:



As the model considers both current and future rewards, the snake tends to survive more often. This is evident in the last two pictures. However, there are instances, as depicted in the first two pictures, where the snake may overlook the current apple, believing that obtaining a future reward is easier and more valuable. Overall, the snake seeks a balance between immediate gains and long-term strategic planning.

When $\gamma = 0.25$, this model aims to strike a balance between prioritizing current rewards while still considering future rewards to some extent, ensuring that the snake does not ignore the current apple.

Presented below are four screenshots illustrating the behavior of the snake after the training phase, using $\epsilon = 0.25$ and a reward vector of $\begin{bmatrix}-500 & -50 & 300 & 100000\end{bmatrix}$:



As the model considers both current and future rewards, the snake demonstrates improved survival strategies compared to when $\gamma = 0.5$. This is evident in all four pictures, where the snake exhibits the ability to capture the current apple while strategically positioning itself to minimize potential locations for future apples. By optimizing its head position, the snake increases its chances of successfully obtaining future apples, while still attending to the immediate reward at hand. The balanced approach allows the snake to adapt its movement patterns to maximize overall rewards.

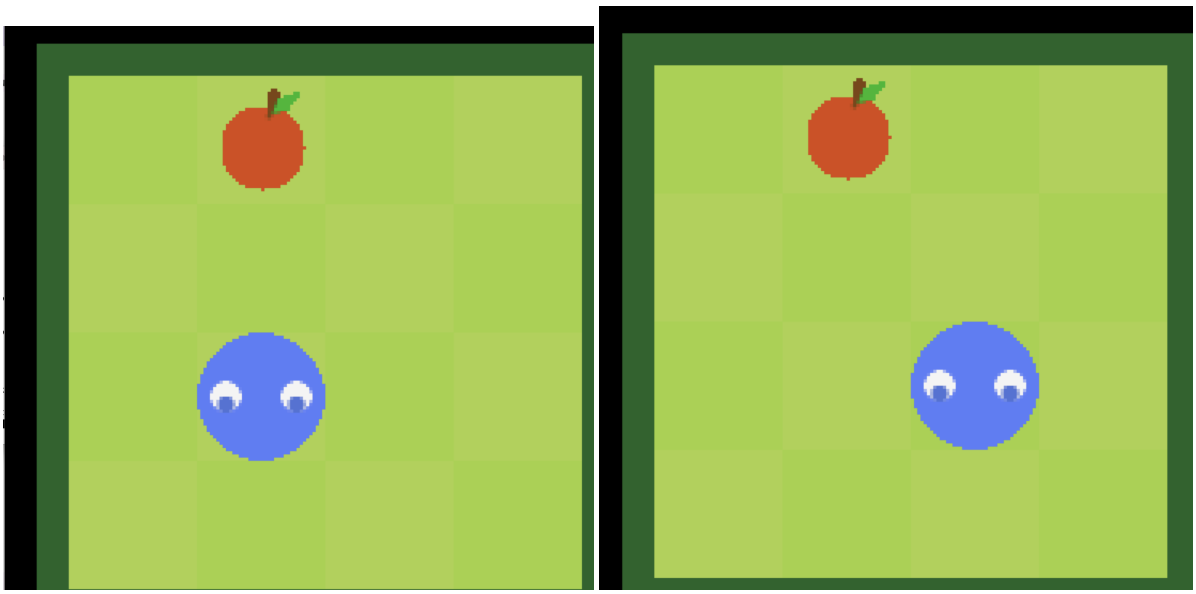
4.3 Reward vector

Since it is impossible to display every possible scenario, the presented examples focus on meaningful extreme cases rather than attempting to optimize every element of the reward vector. By highlighting these diverse instances, the aim is to provide a comprehensive understanding of how this model trains.

Case 1(when losing move is too negative):

The reward vector is equal to $\begin{bmatrix}-500000 & -50 & 300 & 100000\end{bmatrix}$ with $\gamma = \epsilon = 0.25$. It is important to note that the training time for this case is significantly longer compared to other scenarios, as the model computes more extensively to minimize the rate of dying.

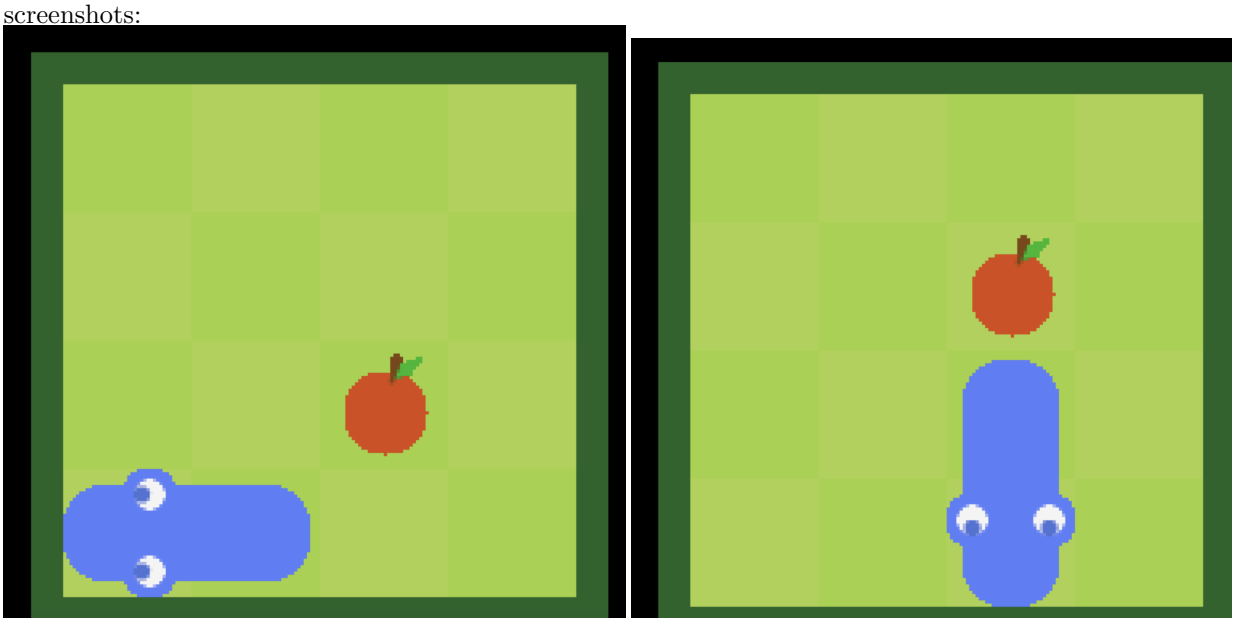
Here are the corresponding screenshots:



Due to the substantial penalty associated with dying, the snake's strategy in this case is to avoid taking risks and prioritize self-preservation. As a result, the snake tends to remain in place or move back and forth instead of actively pursuing apples. This risk-averse behavior is driven by the desire to prolong its lifespan and accumulate rewards over time, resulting in this boring case.

Case 2(when the weight inefficient move is too negative):

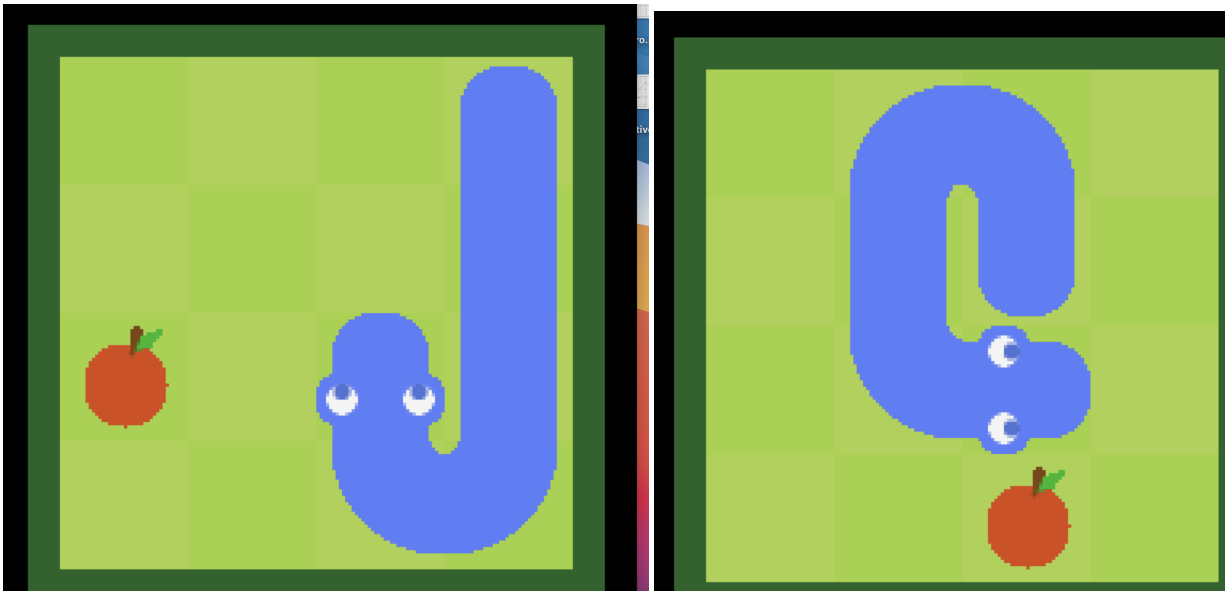
The reward vector is $\begin{bmatrix}-500 & -500 & 300 & 100000\end{bmatrix}$, with $\gamma = \epsilon = 0.25$. Here are the corresponding screenshots:



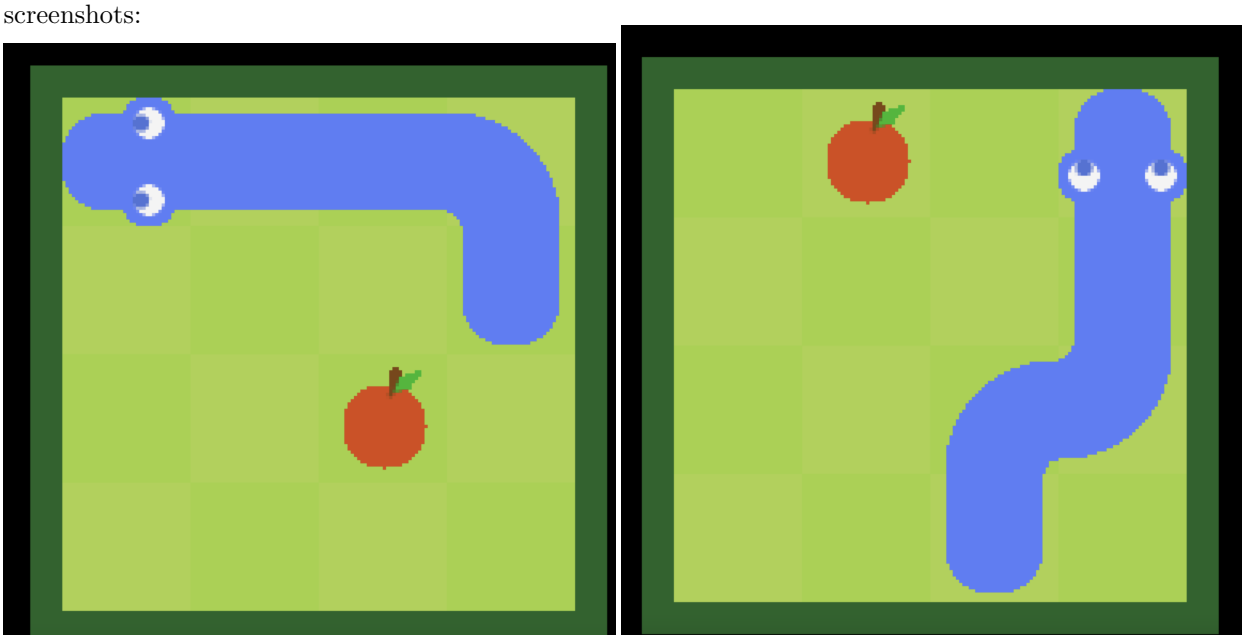
As the weight assigned to inefficient moves is significantly negative, the snake prioritizes avoiding these inefficient moves at all costs. Surprisingly, the snake chooses to sacrifice its own survival and intentionally avoids consuming the apples altogether. This strategy arises from the understanding that eating the apples could potentially result in subsequent inefficient moves, which the snake seeks to avoid.

Case 3(when the weight inefficient move is close to 0):

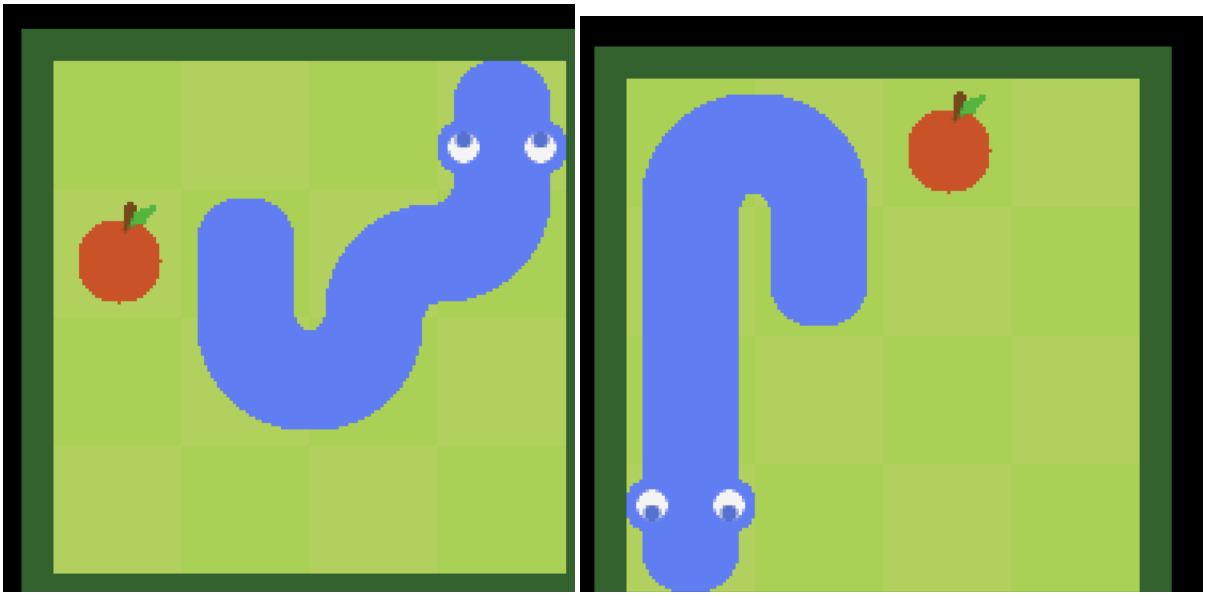
The reward vector is $\begin{bmatrix}-500 & -1 & 300 & 100000\end{bmatrix}$, with $\gamma = \epsilon = 0.25$. Here are the corresponding screenshots:



With the negligible deduction for inefficient moves, the snake tends to prioritize survival over efficiency, leading to the execution of every possible inefficient move while consuming apples. While this strategy allows the snake to maintain a high survival rate, it comes at the cost of increased time and the adoption of peculiar and non-optimal movement patterns. Case 4(when the weight efficient move is too big):
The reward vector is $\begin{bmatrix}-500 & -50 & 30000 & 100000\end{bmatrix}$, with $\gamma = \epsilon = 0.25$. Here are the corresponding screenshots:



Similar to the previous edge cases, the snake in this scenario consumes apples and grows in size. However, its primary focus is on efficiently maneuvering through the game board rather than actively pursuing apples. As a result, the snake often tends to ignore the apples altogether and exhibits counter intuitive movements. Case 4(when the weight winning move is too big):
The reward vector is $\begin{bmatrix}-500 & -50 & 300 & 1000000000\end{bmatrix}$, with $\gamma = \epsilon = 0.25$. Here are the corresponding screenshots:



Similar to the previous edge case for the inefficient move, the deduction for inefficient moves is negligible compared to the winning reward. Therefore, the snake comes up with most inefficient and counter intuitive movements to win the game at all cost.

5 Conclusion

Throughout this game, we have gained valuable insights into the implementation and real-life applications of Reinforcement Learning. Furthermore, we have learned the significance of hyperparameters in training RL models. The hyperparameters play a crucial role in determining the outcome of the trained model. In the context of real-world systems, careful consideration must be given to optimizing and effectively describing the overall set of hyperparameters. It becomes crucial to find a balance and cross-validate these hyperparameters to prevent the occurrence of extreme cases that could lead to meaningless outputs.

The understanding gained from this game underscores the importance of thoughtful hyperparameter selection and validation, as it directly influences the performance and effectiveness of the implemented RL architecture in real-life scenarios.