Snake Report

Povegliano Giorgio – 2089066

In this document I will describe and discuss the work that I've done related to the "Snake Project" for the second part of the Reinforcement Learning exam.

Inside *"main.ipynb"* can be found the extensive code used to train the agent and define the baseline policy. In addition, I found useful to define an instance of the environment and plot it while testing both the learnt and heuristic policy, in order to have a graphical tool to confront them and better understand their issues.

Running *"evaluate.py"* both the baseline policy and the agent will be loaded and tested on an instance of the environment containing 1000 boards, for 1000 iterations. The results will be discussed later inside this report.

Baseline Policy

Follow The Fruit

I will start by describing my approach to defining a suitable Heuristic Policy. I began with an observation: if the snake's head follows the direction of the fruit, it will never collide with the wall. This is trivially true because the fruit is always placed within the boundaries of the board, so the shortest path to the fruit will never include a wall cell. However, additional considerations need to be made to address the eventuality of the snake hitting its own body.

The idea is to evaluate every cell surrounding the snake's head: if the cell contains a fruit, the snake should move to eat it. Otherwise, the snake should choose the direction of the free cell that has the shortest distance to the fruit (a free cell meaning it's neither a part of the snake's body nor a wall cell). This approach also accounts for the possibility of the snake returning to the cell it occupied in the previous iteration (one of the issues I struggled to adjust).

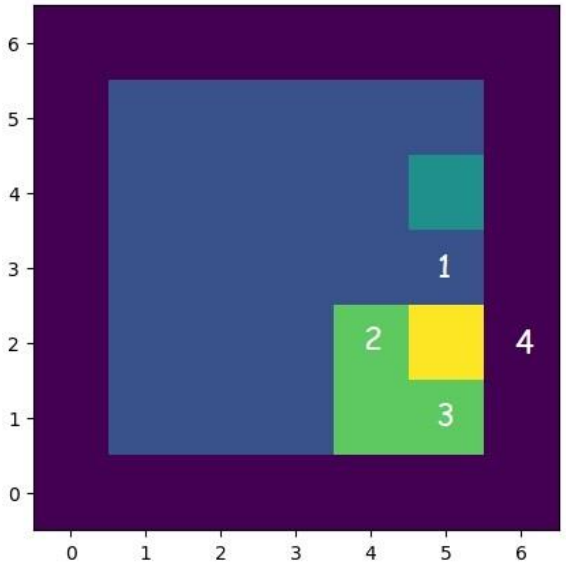

Figure 1: Example of operation of the Baseline Policy

In *Figure 1*, a brief example of the algorithm is shown: in this case, the cells surrounding the snake's head are labeled as 1, 2, 3, and 4. Both 2 and 3 are occupied by the body of the snake, while 4 is a wall. In this case, the only solution is to go into cell 1, which is also on the shortest path that leads to the fruit.

This approach works well and can complete the game, leading to the highest reward. However, the most notable issue that can't be overlooked is that it requires a fully observable environment. The choices at each iteration are made solely based on the structure of the board; if the cells cannot be distinguished, this policy (unlike a learned one) obviously cannot work.

One of the challenges I faced while defining the baseline policy was managing the event of the snake being surrounded by non-empty cells (either its own body or the wall). There are three strategies to handle this situation:

1. **Staying Still**: In this approach, the snake's head remains stationary. As the snake's body reduces at each iteration, a negative reward of -0.1 is assigned. This approach results in the snake slowly freeing itself over time.
2. **Always Heading Towards the Fruit**: Here, the snake moves in the direction of the fruit, even if it means eating its own body. This results in a negative reward of -0.2 but has the advantage of instantly resetting the game by eliminating the body.
3. **Hybrid Approach (The Chosen Solution)**: This approach is a compromise between the first two. If the snake is surrounded only by its own body, it should eat itself. If it is adjacent to a wall, it should bounce into it, which effectively means doing nothing and results in a negative reward of -0.1. This approach was chosen after observing the results of all three strategies.

My concern was that the snake is typically surrounded by non-empty cells when its body is very long, leading to a long wait for the head to be freed. If the reward is -0.1 per iteration, it may be more advantageous, reward-wise, for the snake to eat itself and restart the game.

RL Agent

Q-Learning

To tackle the Reinforcement Learning task, I used Q-Learning as it's model-free algorithm: the agent learns directly from its interactions with the environment. This approach offers significant advantages over the baseline policy, particularly in handling partially observable environments. In scenarios like the Snake game, where states and actions are discrete and limited, Q-Learning is very effective as it leverages Q-values to determine optimal actions among the four possible directions (up, right, down, and left). To address the exploration-exploitation dilemma, I implemented a basic epsilon-greedy strategy.

The backbone of my algorithm is a Deep Neural Network composed of a total of 5 trainable layers:

- *Convolutional Layer 1* with 32 filters of size 3x3
- *Convolutional Layer 2* with 64 filters of size 3x3
- *Flatten Layer* that connects the 2D output of the Convolutional Layers to the 1D output of the Fully Connected ones.
- *Fully Connected Layer 1* with 256 neurons
- *Fully Connected Layer 2* with 128 neurons
- *Fully Connected Output Layer* with 4 units (representing the 4 actions that can be taken)

The optimizer chosen is ADAM with a learning rate of 1e-4.

For what concerns the Q-Learning algorithm itself I decided to use these parameters:

- *Early Stopping* so that if after the 10th iteration results don't improve stops the learning.
- *Update Frequency (C)* to update the target network every 10 iterations.
- *Discount Factor (GAMMA)* set to 0.9 to balance the importance of future rewards (in this case the completion of the game that gives the highest reward possible).
- *Epsilon Greedy* exploration strategy with epsilon set to 0.1.

I trained the agent for 10000 iterations, simulating 1000 7x7 boards at each iteration.

Results

Following, in *Figure 2* is shown a graph that compares the agent's mean rewards at each iteration during the process of training (mean of the 1000 rewards given by each board) with the baseline policy and the random policy (both tested on 10000 iterations).

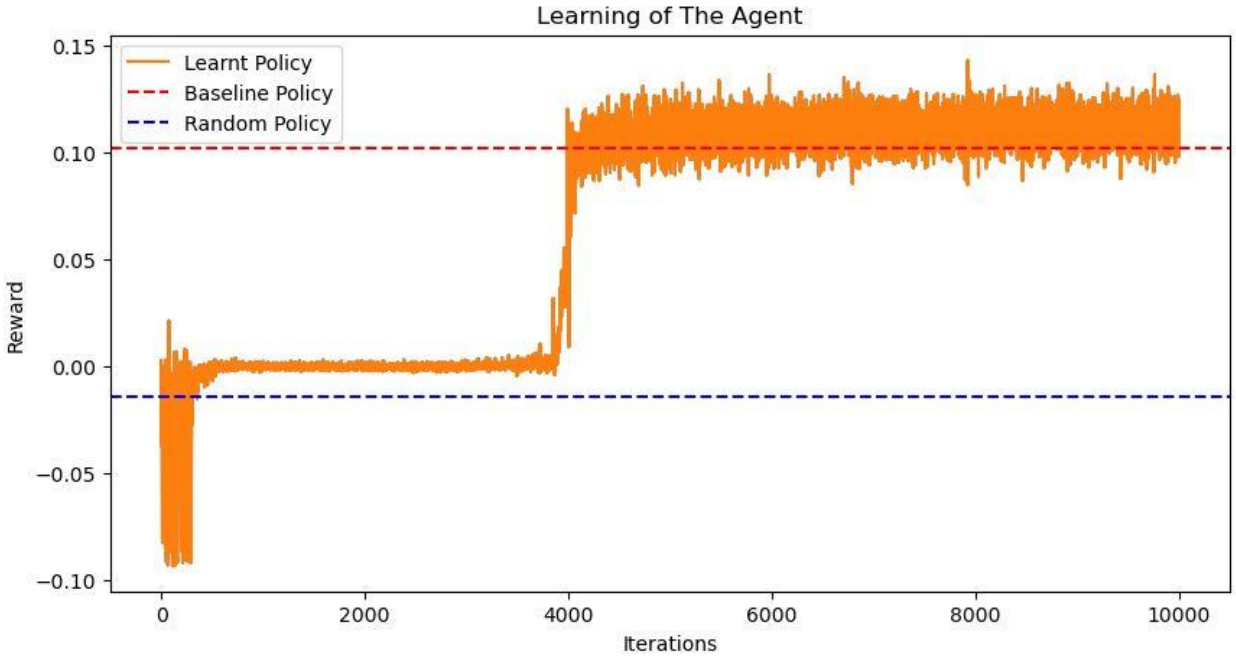

Figure 2: Plot of the rewards of the agent during the training process.

As shown, after just 500 training iterations, the agent outperforms the Random Policy. It then takes another 4000 iterations to match the Baseline Policy's performance. By the end of the training, the agent consistently outperforms the Baseline Policy.

In Figure 3, the performance of the Heuristic and Learnt Policies is plotted over 1000 iterations using an environment with 1000 7x7 boards. After an initial phase where both policies perform similarly, the Learnt Policy averages a reward of 0.132, while the Baseline Policy averages just over 0.1.

These results are replicable running the file *"evaluate.py",* that loads the pre-trained model and the informative policy and tests them on 1000 iterations.

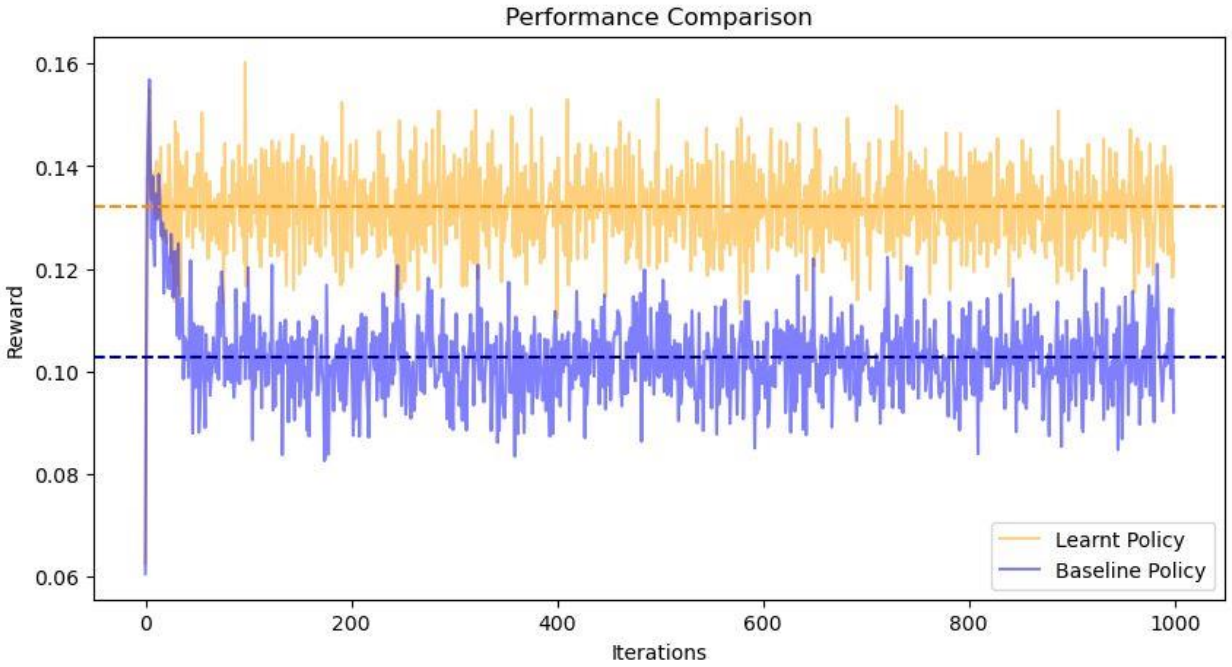

Figure 3: Agent versus Baseline Policy performance comparison.

In conclusion, despite the effort to develop a comprehensive and reliable Baseline Policy, the agent, through direct interaction and learning within the environment, achieves better results. This confirms the correctness of the assumptions made to build the model, from the choice of the Reinforcement Learning algorithm (in this case Q-Learning) and its parameters to the structure of the DNN.

Aside from the issues mentioned earlier with developing an optimal baseline policy, I also struggled with balancing various parameters, particularly the learning rate of the neural network, the exploration parameter epsilon, and the number of iterations. Also, deciding on the number of layers in the network was challenging. I felt that having too many layers might be overkill and would require significantly more computational power and training time. Balancing all these factors was not easy, but seeing promising results, I ultimately decided to use a 5-layer deep neural network and train for 10,000 iterations.