Snake game with Deep Q-Learning

Michele Scapinello, 2087617

Contents

1 Project critical issues

The snake game present a series of critical issues that may be tackled during the development of an heuristic or reinforcement policy. Some of these issues may be common to both approaches while others might be critical only for one of the two. During the analysis of the problem some of the main critical issues that have arisen and might be encountered during its development are:

- **Sparse Rewards**: In the snake game provided rewards are sparse, meaning that when dealing with a reinforcement learning agent the feedback received is positive when it eats a fruit or negative when it collides with a wall or itself. Training a DRL agent with this definition of rewards can make learning slow and challenging. One possible solution could imply having a different reward function, for example the distance of each cell from the fruit.
- **Dynamic Environment**: The snake game's environment is dynamic, with the snake's length changing over time as it eats fruit or himself. This dynamic nature adds complexity to the problem, requiring agents to adapt their strategies in real-time to reduce the probability of eating himself.
- **Generalization**: Agents should generalize well to unseen game configurations or variations. They should not overfit to specific patterns or configurations present in the training data but should instead learn general strategies that can be applied in various scenarios and that would allow the RL agent to play the snake game unconditionally from the environment.
- **Exploration vs. Exploitation**: In the reinforcement learning scenario balancing exploration (trying new actions to discover potentially better strategies) and exploitation (leveraging known strategies to maximize immediate rewards) is very important as the

agent must explore the state space effectively without getting stuck in sub-optimal behaviors. To cope with this problem ϵ-greedy approaches are used and as the learning process goes on the ϵ parameter is decreased (ϵ-decay trace).

- **State Representation**: Designing an appropriate state representation is crucial for effective learning. The state should encode all relevant information about the game state, such as the positions of the snake's body segments, the location of the fruit, and the boundaries of the game board, in a way that makes the learning process effective.
- **Function Approximation**: DRL algorithms often use function approximation (e.g., neural networks) to approximate the value function or policy. Training deep neural networks can be challenging due to issues like vanishing gradients, overfitting, and instability. To cope with this problem, proper techniques and structures are used in neural networks, for example ReLU activation function, to avoid gradient related problems.
- **Delayed consequence of taking an action**: The snake game requires long-term planning and foresight to avoid running into dead-ends and to maximize the length of the snake. However, this can be challenging for agents, particularly those relying on short-term reward signals.
- **Hyperparameter Tuning**: DRL algorithms have various hyperparameters (e.g., learning rate, discount factor, exploration rate) that need to be carefully tuned to achieve optimal performance. Finding the right set of hyperparameters can significantly impact the training process and the final performance of the agent. The selection of the hyperparameters is not trivial and since they are not learnable in the training process it is necessary to perform various experiment with different settings to find the optimal one.

2 The baseline policy

The functions used in the baseline policy are the following:

- **retrieveBoardPositions(board)**: This function retrieves the positions of different elements (head, walls, fruit, body) on the game board in the form of tensors containing the respective indexes.
- **getDirections(step, heads)**: This function calculates the direction of the movement based on the difference between the step indexes and the snake's head indexes and return the direction in the form of string (UP, RIGHT, DOWN, LEFT).
- **getProbabilities(directions)**: This function converts the direction of the movement into probabilities for each possible action in a similar way as a one-hot encoding. The probability of the direction is set to 1, while the others to 0. The Table 1 shows the structure of the encoding for every possible action.

Direction	Probabilities	Action index (argmax)
UP	[1, 0, 0, 0]	[0]
RIGHT	[0, 1, 0, 0]	[1]
DOWN	[0, 0, 1, 0]	[2]
LEFT	[0, 0, 0, 1]	[3]

Table 1: Encoding of probabilities and action indexed according to the direction.

- **nextMove(head, body, walls, prev_head)**: This function is executed if a path to a fruit is not found with the `BFS()` function, for example when the body is blocking the head of the snake in a corner, Fig.1 (a). It determines the next move based on certain conditions such as checking for valid moves that avoid collision with walls or the snake's body. If there are no valid moves, it selects a random move among the possible one and avoids to eat 'behind' himself to not lose the whole body of the snake, Fig. 1 (b).

Figure 1: The snake is stuck as shown in (a) and the `nextMove()` function randomly selects to go DOWN. The result is shown in (b).

- **BFS(head, walls, fruit, body)**: This function performs Breadth-First Search (BFS) to find the shortest path from the snake's head to the fruit on the board, avoiding walls and the snake's body, Fig. 2. It returns an array containing the indexes of the board cells the snake has to visit to reach the fruit.

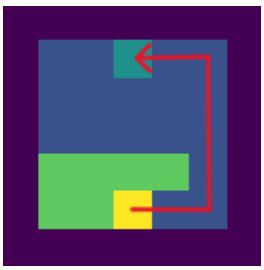

Figure 2: Path computed for the given scenario by the `BFS()` function.

- **computePath(board, boards_size, prev_head)**: This function is used to call the functions `BFS()` and `nextMove()`, to find a path from the current position of the snake's head to the fruit. It first retrieves the positions of game elements using `retrieveBoardPositions()` and then attempts to find a path using the `BFS()` function. If no path is found, it falls back to a different strategy determined by the `nextMove()` function.

Figure 3: Scenario in which the `BFS()` function does not find a path and thus the computation is done by the `nextMove()` function.

The heuristic approach used to solve the snake game is an easy way of dealing with such problem. It has been programmed such that the snake will never hit the wall in any circumstances as requested by the description of the problem and at the same time it tries to maximize the amount of fruits eaten during the game. The flow of the algorithm is shown in Fig. 4.

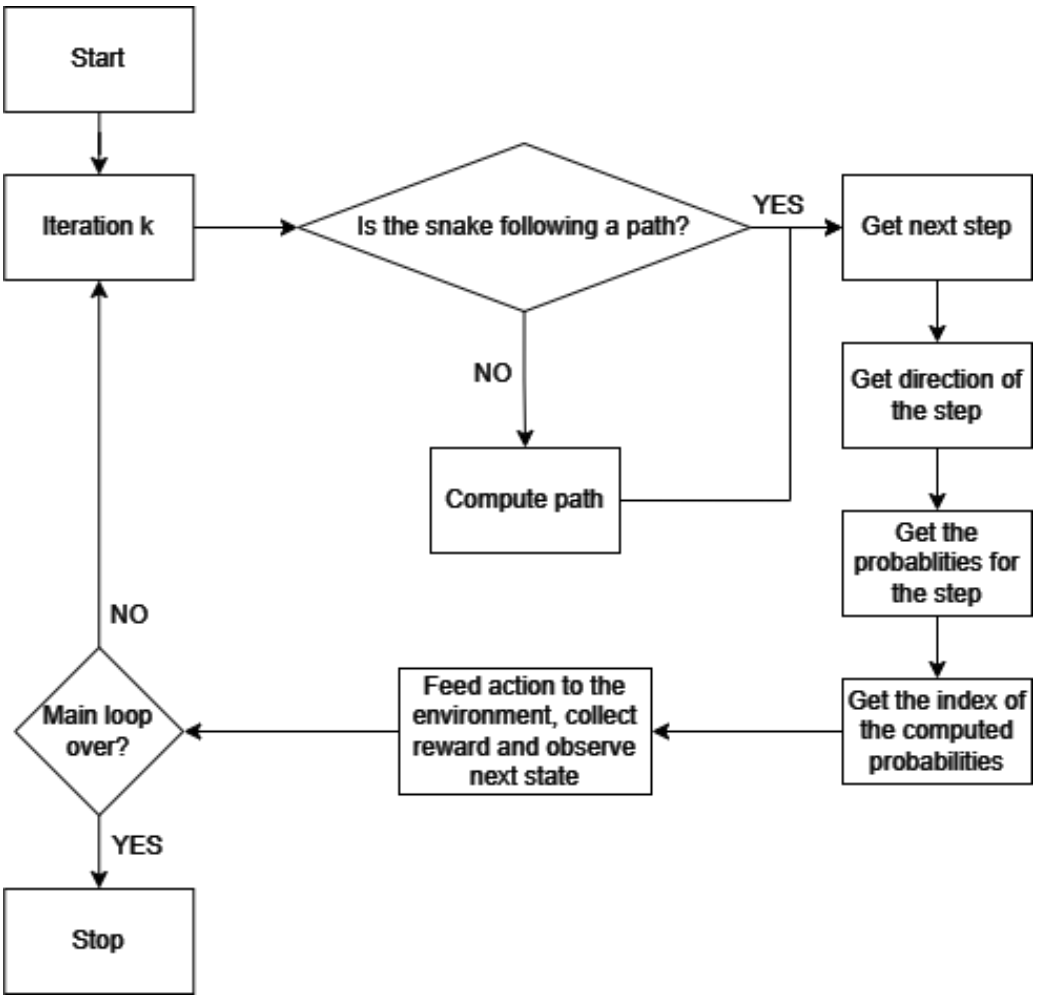

Figure 4: Flow diagram of the algorithm implemented for the heuristic approach.

Once the environment is chosen, that in our case is set to be the **fully observable environment**, the game can start. The proposed algorithm is an iterative implementation that follows at each iteration the logic explained afterwards, up to a defined number usually set to 50000 or more iterations. At the beginning of each iteration it is checked if the snake is already following a path to the fruit or not and if not, one is found with the `computePath()` function. In order to follow the computed path, at the beginning of each iteration, the first occurrence in the array returned from the latter function is extracted and is fed in input to the `getDirections()` function. The result of the `getDirections()` function is used as input for the `getProbabilities()` function, to get the corresponding probability enconding, as show in Table 1. These probabilities are then fed to the `argmax()` function of the Tensorflow library to extract the index of the action to be taken to cope with the environment structure and functionalities and the way the actions are indexed. After all these steps the action is fed to the environment to collect the reward and move to the next state. Overall, this algorithm combines heuristic-based pathfinding (BFS) with a simple random policy to cope with the possibility of BFS not working. It represents a straightforward approach to play the snake game without using deep reinforcement learning techniques or involving any learning functionality.

3 Deep Reinforcement Learning

The algorithm chosen to solve the problem of the snake game is the Double Deep Q-Networks (Double DQN) algorithm, an extension of the Deep Q-Network (DQN) algorithm that addresses the issue of overestimation bias present in the traditional Q-learning methods. The Double DQN algorithm uses two separate neural networks, one called **Online Network** (ON) used to select actions during training which takes in input the current state and outputs the Q-values for all possible actions. The action is selected according to an ϵ-greedy approach to balance the exploration and exploitation during training:

$$A_t = \begin{cases} \text{argmax}_a \, Q_t(a) & \text{with probability } 1 - \epsilon \\ \mathcal{U} \sim a_1, \ldots, a_n & \text{with probability } \epsilon \end{cases}$$

Also, the ϵ parameter is decreased as the learning process goes on according to a decay trace approach. The other network, called **Target Network** (TN), is used to evaluate the selected actions and takes in input the next state to output the Q-values. The choice of evaluating the selected action with the TN helps mitigate the overestimation of the bias present in traditional Q-learning approaches. The weights of the TN are periodically updated with the weights of the ON to stabilize training. Moreover, during the learning process the weights of the ON are updated at each step of the learning process with Adam optimizer, a stochastic gradient descent method, where the loss function adopted is the Mean Square Error (MSE) computed between the predicted Q-values from the ON and the target Q-values. The target Q-values are computed using the Bellman equation as the immediate reward plus the discounted maximum Q-value of the next state, calculated using the target network:

$$Y_i = R_{t+1} + \gamma \text{max}_a(Q(S_{t+1}, a, \hat{\theta}))$$

Thus, the loss is computed as:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (Y_i - Q(s_i, a_i, \theta))^2$$

The choice of implementing Double DQN algorithm has been done because of the State-of-the-Art Performances of such algorithm in various reinforcement learning environments. Applying this algorithm to the Snake game led to good results with reference to the heuristic approach, as will be explained in section 4. Moreover, the algorithm does not have a tedious implementation and its logic is easy to understand with reference to other algorithms such as Proximal Policy Optimization which has been implemented at the first moment but unsuccessfully and therefore did not yield to proper results for comparison. The reduced overestimation bias allows to have a more accurate Q-value estimate with reference to the DQN algorithm and also allows to better generalize across similar states in the game of Snake which very important if we consider that a game board can have multiple different configurations but with common features. Reducing overestimation bias and having a better generalization thus implies to make more informed decisions in various games scenarios and optimize the reward function.

3.1 Neural Network architecture

The architecture of the networks employed in the algorithm is the one of Feed Forward Neural Networks (FFNN).

The input layer flattens the input, that is a multi-dimensional array representing the state of the environment (the game board). Two fully connected (dense) layers of 128 neurons each are employed as hidden layers to learn the relationships between the input features. After

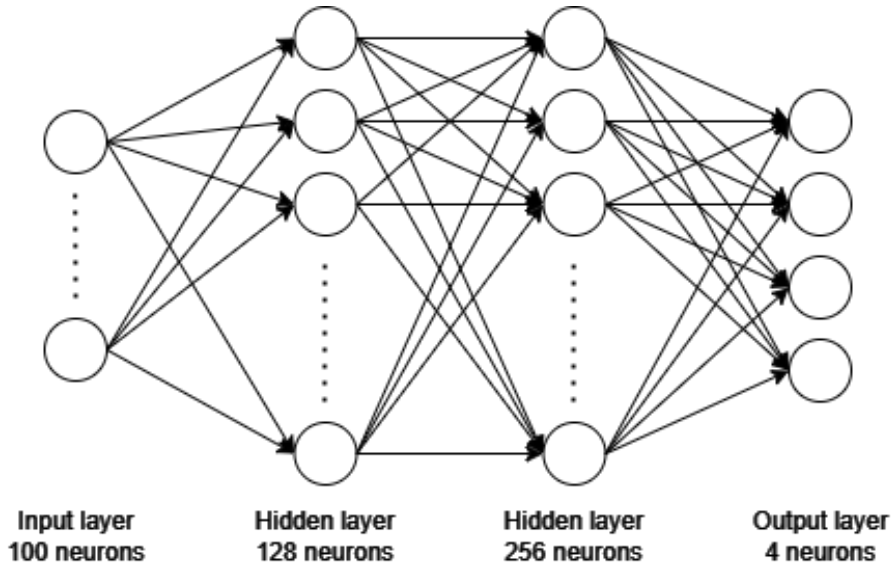

Figure 5: Feed Forward Neural Network architecture.

each dense layer, batch normalization layers are added to help stabilize and accelerate the training process. The activation function used in the provided FFNN is the Rectified Linear Unit (ReLU) chosen to cope with the vanishing gradients problem. The output layer consists of four units corresponding to the Q-value of the possible actions that the agent can take in the game of Snake. The activation used in the last layer is linear, meaning that the network directly outputs the Q-values for each actions without applying any further transformation. The kernel initializer used for the dense layers is the He normal initialization to facilitate learning. The parameters of the neural network are summarized in Table 2.

Hyperparameter	**Value**
Learning rate	0.001
Discount factor	0.99
Optimizer	Adam
Hidden Layers	2
Hidden Layer Neurons	128
Output Layer Neurons	4
Dense activation function	ReLU
Kernel initializator	HeNormal
Output activation function	Linear

Table 2: Model Configuration.

4 Results

The results of the training of the agent are shown in Fig. 6 and the whole process of training and evaluation has been carried out in the fully observable environment. The plots show the progress of each characteristic over 50000 iterations where an epoch is equal to 1000 iterations. Reward and fruits are correlated between each other thus the progress of the functions is the same but from the plots it is possible to see that they differ in therms of time instance in which they surpass the heuristic policy, which may be caused by the agent performing worse than the heuristic policy in terms of wall hits and body hits. As the number of iterations increases the

agent learns how to play the game in order to maximize the reward. This means that it tries to both maximize the number of fruits eaten and minimize the wall and body hits. At a certain point in the training the agent is capable of surpassing the heuristic policy in terms of average fruits eaten over 1000 iterations (roughly around 25000 iterations) and similarly it is capable of surpassing the heuristic policy in terms of average reward (roughly around 30000 iterations).

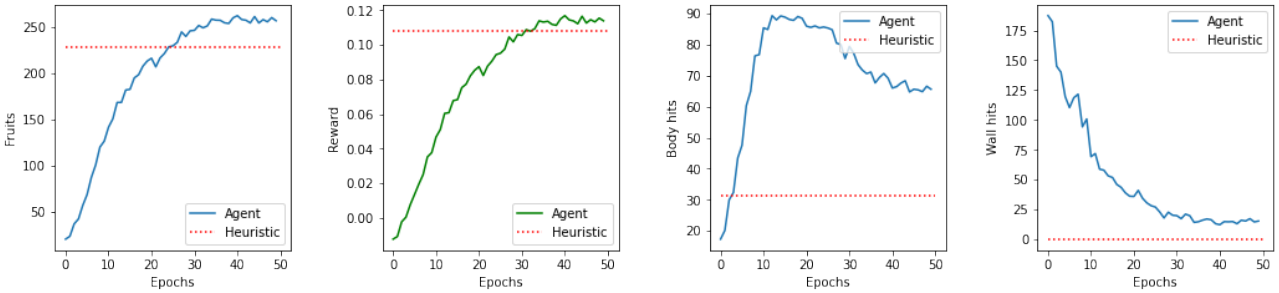

Figure 6: Training step statistics.

The evaluation results are shown in Fig. 7. The results displayed show how the agent is capable of beating the heuristic policy in terms of fruits eaten and average reward over 50000 iterations (1 epoch equals 1000 iterations). The selection of the action in the evaluation has been done with a greedy approach, thus picking the action maximizing the Q-value function of the the pair state, action. The body hits in the evaluation are decreased and the wall hits are almost near to 0 on average showing that the agent is capable of selecting an action by also considering the delayed consequences of such decision and that it avoids to hit the wall in almost all the possible scenario in which it is near one.

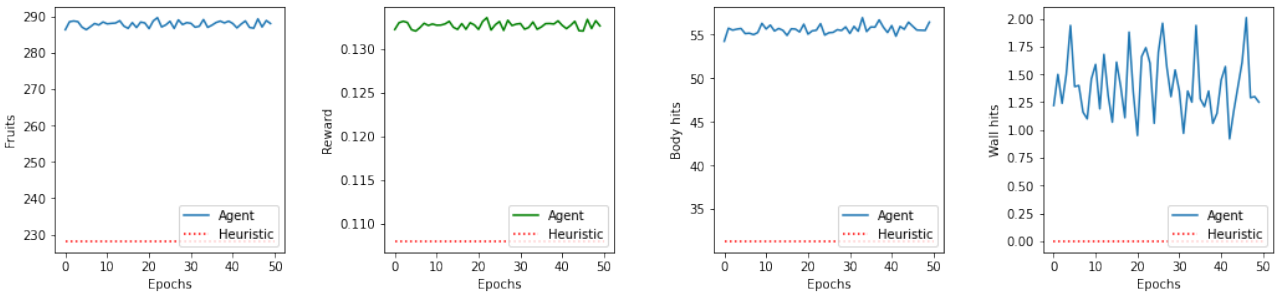

Figure 7: Evaluation step statistics.

5 Conclusions

As explained in chapter 4 the agents performs better than the heuristic policy in average reward and fruits eaten characteristics but does not surpass it in all fields. Some improvements might allow the agent to overcome the heuristic policy, for example by modifying the values of the reward function and heavily penalize wall hits and body hits. Another possible improvement would be changing the representation of the state and feed in input to the FFNN the distance in various directions of the snake head from its body, the fruit and the walls together with the positions of the latter. These two improvements together might allow to decrease the number of wall and body hits in spite of training and evaluation time. Also, a change in the structure of the NN might allow to improve the Agent performances as FFNN are known to not be able to preserve the spatial correlation of the inputs. Using a Convolutional Neural Network might overcome such problem and achieve better performances than the heuristic policy.