Winning At Snake Using Three Brains

Piai Luca

Abstract

This project utilizes the Deep Q-Network (DQN) algorithm to train an agent capable of winning the game of Snake. We evaluate three distinct state-space representations, identifying the vision-based AtariAgent as the top performer. To improve the performances, we introduce the "CerberusAgent," which dynamically switches between three pre-trained heads based on the current score to balance aggressive scoring with survival. Finally, we compare this solution against a Hamiltonian Cycle baseline, demonstrating that while the baseline ensures victory, our agent achieves significantly higher movement efficiency.

1. Goal and structure of the report

The goal of the project is to train an agent capable of winning at the Snake game. In this experiment, we focus on a single algorithm, namely the **DQN algorithm**. The motivation for this choice is that the algorithm has proven successful in the context of Atari games (Mnih et al., 2013).

We will consider three different agents that differ from one another in their internal state-space representations. Then we will select the agent with the best performances, we further refine it and finally compare the resulting solution with a *baseline algorithm* that does not involve any learning.

Any reader can access the code published on GitHub[1].

2. The environment

Snake is an old and very popular game that doesn't need any introduction. In our experiment, the game looks like Figure 1.

When the game starts, the starting position of the snake is always the same, instead the apple is placed randomly within the environment. The snake board is a 10×10 cells, the snake starting length is 3, therefore the maximum score of the game is 97.

[1]https://github.com/luca037/Snake-RL

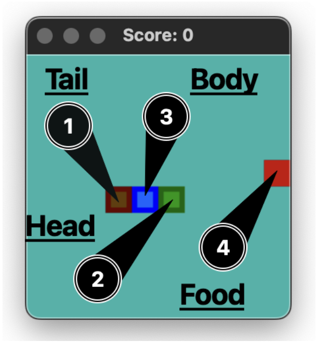

Figure 1. The starting frame of the game.

2.1. The Set of Actions

The game keeps track of the snake's current *direction*, which can be one of the following: *up, down, left,* or *right*. At the start of the game, the snake is always pointing to the right. The possible **actions** are just three: *going straight, turning right* or *turning left*. By knowing the current direction and the action selected by the agent, the game is able to correctly update the position of the snake.

2.2. The Reward Function

The reward function used is reported in Table 1. More complex reward functions could be designed; however, we chose to keep it as simple as possible for two main reasons. First, a simple reward function does not bias the agent toward learning a specific strategy: it must learn on its own how to solve the game. Second, this choice allows for a fair comparison among the three agents and their respective state-space representations.

Table 1. Reward function.

Description	**Reward**
Snake hits the wall or eats itself	-1
Snake eats apple	+1
Snake survives 1 step and length ≥ 10	+0.01

The agent receives a positive reward when the snake's length is greater than 10 and the executed action does not result in the snake's death. The motivation behind this choice is to discourage the agent from moving directly toward the apple; instead, we encourage more varied movement patterns, reducing the likelihood that the snake traps itself.

3. The algorithm

The pseudo-code of the algorithm used to train the agents is Algorithm 1.

Algorithm 1 Deep Q-learning with Experience Replay

```
1: Initialize: replay memory $\mathcal{D}$ to capacity $N$
2: Initialize: action-value function $Q$ with random weights $\theta$
3: Initialize: target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
4: for each episode do
5:   Initialize starting state $s_1$
6:   for each time step $t$ do
7:     With probability $\varepsilon$ select a random action $a_t$
8:     otherwise select $a_t \leftarrow \arg\max_a Q(s_t, a; \theta)$
9:     Execute action $a_t$, observe reward $r_t$ and state $s_{t+1}$
10:    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
11:    Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1}) \sim \mathcal{D}$
12:    if episode terminates at step $j+1$ then
13:      $y_j \leftarrow r_j$
14:    else
15:      $y_j \leftarrow r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
16:    end if
17:    $\theta \leftarrow \theta - \alpha \nabla_\theta \sum_j [y_j - Q(s_j, a_j; \theta)]^2$
18:    Every $C$ steps reset $\hat{Q} = Q$
19:  end for
20: end for
```

Note that, for the sake of simplicity, two aspects are omitted.

Exploration Decaying The update of the parameter ε is not included in the pseudocode: each agent must define a decay function that controls the probability of selecting a random action. In our experiments, we considered both an exponential decay and a linear decay. The decay rate is treated as a hyperparameter.

Initial Random Steps The pseudo-code does not account for the initial phase in which the agent selects actions completely at random for a fixed number of steps. This was introduced in order to ensure a proper exploration around the initial state.

Replay buffer implementation The replay buffer has a fixed size. However when is full the older samples are deleted and replaced by the new ones.

4. The Three Agents

In this section, we present the three agents considered in our study: the **BlindAgent**, the **LidarAgent**, and the **AtariAgent**. In the following, we describe each of them in detail, focusing on their state-space representations and on the architecture of the action-value function Q (i.e., the Q-network).

4.1. BlindAgent

The BlindAgent is the simplest one. Each state is defined as a binary vector of size 11. The meaning of each element is reported in Table 2. At each time step the agent build the state vector, using the current information of the game. We call this agent blind because it has a partial view of the environment, indeed it can detect obstacles only when they're one distance block with respect to the head of the snake.

Table 2. BlindAgent State-Space Representation.

Index	Feature	Value
1	Obstacle in front of the head (straight)	0 or 1
2	Obstacle directly on the right of the head	0 or 1
3	Obstacle directly on the left of the head	0 or 1
4	Snake direction == left	0 or 1
5	Snake direction == right	0 or 1
6	Snake direction == up	0 or 1
7	Snake direction == down	0 or 1
8	Apple is on the left of the head	0 or 1
9	Apple is on the right of the head	0 or 1
10	Apple is above the head	0 or 1
11	Apple is below the head	0 or 1

The Q-network of this agent is a simple feed-forward neural network as illustrated in Figure 2.

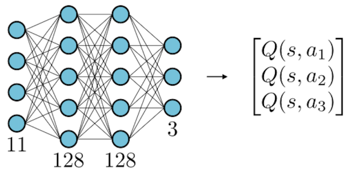

Figure 2. Schematic representation of the Q-network of BlindAgent. The activation function used is the `ReLU`; the output function of the network is the identity.

4.2. LidarAgent

The LidarAgent perceives the environment through a LIDAR that is placed upon the head of the snake. The LIDAR scans in eight different directions $d_1, \ldots, d_8$ (see Figure 3) and, for each of them, provides three pieces of information:

1. The inverse of distance to the wall[2].

[2]We consider the inverse of the distance so that each value lies

2. Whether the apple lies in that direction.
3. Whether a part of the snake's body is present in that direction.

Finally the state is build by concatenating the information coming from all the directions and by adding, at the very end, the current direction of the snake. Therefore, the state is represented by a $8 \times 3 + 4 = 28$-dimensional vector (see Table 3).

Table 3. LidarAgent State-Space Representation.

Index	Feature	Value
1	Inverse distance to the wall along direction d_1	(0, 1]
2	Apple is along direction d_1	0 or 1
3	A piece of body is along direction d_1	0 or 1
$\vdots$	repeat for all directions d_i, $i = 2, \dots, 8$	$\vdots$
25	Snake direction == left	0 or 1
26	Snake direction == right	0 or 1
27	Snake direction == up	0 or 1
28	Snake direction == down	0 or 1

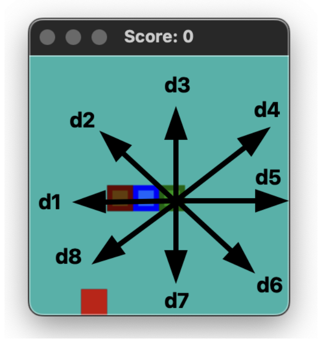

Figure 3. The 8 directions of the LidarAgent.

The Q-network used for the LidarAgent has the same structure of the one used for the BlindAgent (see Figure 2). The only difference is in the dimension of the input state (11 vs 28).

4.3. AtariAgent

The AtariAgent is the most complex among the three. The current game frame can be represented as a two-dimensional matrix: each body segment is encoded with the value 0.5, the head with 1, the food with 2, and empty cells with 0. The state is constructed by stacking the last four frames of the game, resulting in a $4 \times 12 \times 12$ tensor.

A single frame is represented as a 12×12 matrix, even though the actual game grid is 10×10. An external border filled with zeros is added so that, when the snake's head reaches this border, it indicates a collision with the wall.

Finally the Q-network is a CNN; a schematic representation is reported in Figure 4. The name of the agent derives from the fact that the same input representation is used in (Mnih et al., 2013).

in the range $(0, 1]$, which is beneficial for the stability of neural network training.

5. Training and results

In this section we compare the results obtained by the three agents. The hyperparameters used are reported in Table 4.

Table 4. Hyperparameters used to train each agent.

Parameter	BlindAgent	LidarAgent	AtariAgent
γ	0.99	0.99	0.99
starting ε	1	1	1
last ε	10^{-4}	10^{-4}	10^{-4}
ε decaying	exponential	linear	linear
random steps	0	$25 \cdot 10^3$	$50 \cdot 10^3$
batch size	32	32	32
replay buffer capacity	$100 \cdot 10^3$	$150 \cdot 10^3$	$250 \cdot 10^3$
target sync (or C)	10	$2 \cdot 10^3$	$5 \cdot 10^3$

The metrics used to monitor the training of the agents are the *mean reward over the last 100 games* and the *average value of the action-value function* Q, computed as:

$$\text{avg\_q} := \sum_{i=1}^{250} \max_a Q(s_i, a; \theta) \qquad (1)$$

where states s_i belongs to a fixed set of size 250 created at the very beginning of the training and then never modified.

5.1. Observations on the results

Slow convergence From Figure 5 we can see that each agent has reached convergence. However, it appears that the **LidarAgent** and the **AtariAgent** converge very slowly. This behavior is due to the linear decay function, which decreases the exploration rate at a slow pace. By design, the minimum value of ε is reached only when the replay buffer is full. A solution for this problem is to increase the decaying factor for both.

Average Predicted Action-Value Figure 6 shows the evolution of the average predicted action-value Q during training. From a theoretical perspective, we expect this quantity to gradually increase as the agent improves its policy and eventually converge to a stable value.

Interestingly, in our experiments both the **BlindAgent** and the **LidarAgent** exhibit a noticeable spike when ε reaches its minimum value. This behavior indicates an overestimation of the expected return. A possible cause may be

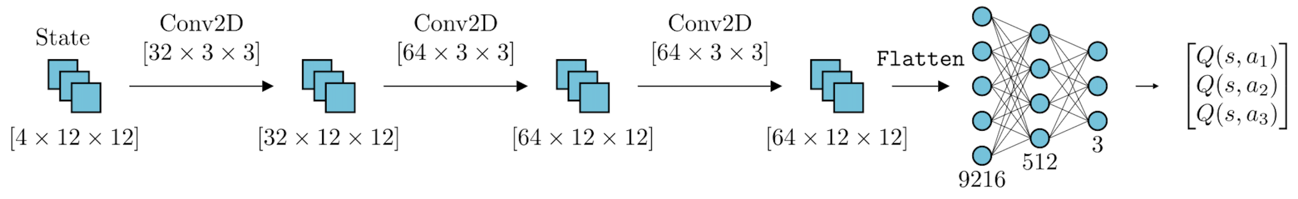

Figure 4. Schematic representation of the Q-network of the AtariAgent. Each convolutional layer uses a kernel with size 3 × 3, stride 1 and padding 1. The activation function used is the `ReLU`; the output function of network is the identity.

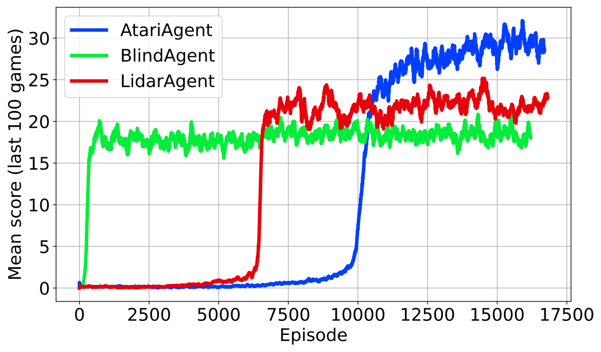

Figure 5. Mean score of last 100 games. BlindAgent reaches minimum ε after ≈ 200 games; LidarAgent after ≈ 6500 instead Atari agent after ≈ 10k.

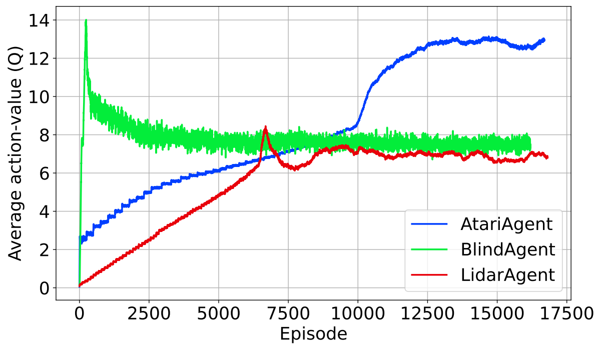

Figure 6. Average predicted action-value. The average was computed considering 250 states.

the *target sync* parameter; increasing its value could help mitigate this issue.

AtariAgent is the winner The record achieved during training are reported in Table 5. As we can see the AtariAgent is able to reach a score of 61, meaning that the snake occupies the 64% of the total area of the board.

Table 5. Record Score achieved by the three agents during training.

Agent	Record/Max Score
BlindAgent	47/97
LidarAgent	41/97
AtariAgent	**61**/97

The main limitation of the **BlindAgent** is that it cannot avoid trapping itself due to its very simple state-space representation. As a result, reaching the end-game is not feasible for this agent. The **AtariAgent**, on the other hand, has full visibility of the board and can therefore learn strategies to avoid self-trapping. In principle, the **LidarAgent** should also be able to avoid trapping itself, however, in our experiments, it exhibited poor performance with respect to the other two agents. The reasons for this behavior are not yet clear, this require further investigation.

6. Improving The Agent

In this section we present the main problems related to AtariAgent and we present a solution to improve its performances.

6.1. Why AtariAgent Fails?

The state-space representation fully describes the environment, so why the trained agent is not able to win the game? We can identify two main reasons:

1. **State space not fully explored**: The replay buffer is mostly filled with early-game frames in which the snake is short. As a result, the agent is not able to learn an effective end-game strategy.

2. **Indecision in strategy selection**: By looking at the learned policy, it appears that the agent rushes towards the food (similarly to the BlindAgent). However, once the snake becomes long enough, it *sometimes* switches to a different strategy that is effective at avoiding self-trapping.

Regarding the second point: rushing towards the food is a greedy policy and it is great at the beginning. However, as the game progresses, the agent must switch to a more conservative strategy. Although the AtariAgent is capable of performing this switch (as indicated by its maximum score of 61), it appears to lack a reliable criterion for deciding when to make the transition.

6.2. Solution: a biased replay buffer

In this subsection, we briefly describe an **unsuccessful approach**. This was an attempt to solve the first issue discussed in previous subsection.

Rare and common samples The idea was to split the replay buffer in two smaller ones: the first one containing *common samples*, the second one containing *rare samples*. By rare samples we mean frames in which the length of the snake is above the current average, as well as frames in which the snake dies. The training batch is then constructed by sampling a small fraction (e.g., 5%) from the rare-sample buffer, with the remaining samples drawn from the common-sample buffer.

Failing The solution proposed didn't provide any significant improvement. It was difficult to tune the dimension of the two replay buffers and the percentage of rare samples in a batch.

6.3. Solution: CerberusAgent

A simple solution for both issues is the CerberusAgent. Like the *mythological Cerberus*, the agent is composed by three heads and in our context the heads are 3 pre-trained AtariAgents.

- **Head 1**: a standard AtariAgent trained from the default initial state.
- **Head 2**: an AtariAgent trained starting from a state in which the snake has length 20.
- **Head 3**: an AtariAgent trained starting from a state in which the snake has length 47.

After training the three heads, we observe that **Head 1** has learned a policy that aggressively moves toward the food, whereas **Head 2** and **Head 3** have learned a more conservative *survival* policy that is very effective at reaching the end-game.

How the Cerberus Plays Snake After all heads are trained, the **CerberusAgent** can be assembled. During gameplay, the agent selects which head to use based on the current score. Figure 7 illustrates the simple selection logic and Figure 8 shows the distribution of the score of the agent.

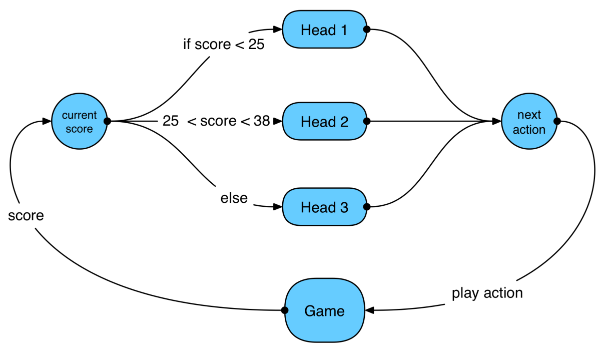

Figure 7. Selection logic used by CerberusAgent. At each step, only one head decide the next action to perform. The heads are selected based on the current score.

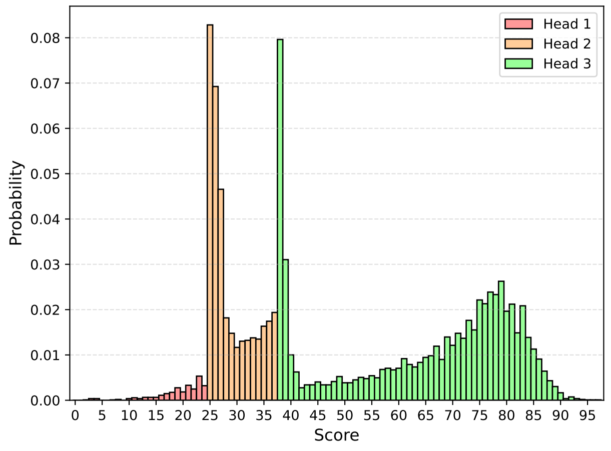

Figure 8. Distribution of the score of the CerberusAgent. The mean is ≈ 50, the record is 97 (max score). In the plot, the bars are colored according to the head that was responsible for the snake's death.

7. Comparison with a Baseline

In this section we compare the CerberusAgent with a *baseline* algorithm that doesn't involve any learning.

7.1. Baseline algorithm

In our experiment the baseline algorithm is the **Hamiltonian Cycle**. In graph theory, an Hamiltonian cycle is a path in a graph that visits every node exactly once and returns to the starting node, forming a closed loop. In Snake world the nodes of the graph correspond to the cells of the underlying grid. In environment, the Hamiltonian Cycle represents a solution that is very simple to implement and always wins the game. Figure 9 shows the graphical representation of the algorithm.

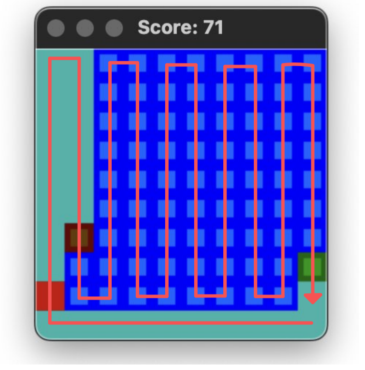

Figure 9. Hamiltonian Cycle in Snake.

7.2. CerberusAgent vs Baseline algorithm

There are several metrics that can be used to compare the two strategies. In this experiment, we consider the *average score* and the *efficiency*. The efficiency depends on the number of steps used to reach a certain score. A strategy is more efficient than another if uses less steps to reach every possible score.

Regarding the average score (and the winning rate in general) there is no discussion: the baseline algorithm represents a perfect solution that never dies and always wins the game. However its weakest point is the **efficiency**.

The baseline algorithm follows a survival policy that is optimal for the end-game but inefficient during the early stages of the game. On the other hand, the overall policy learned by our agent is very fast at the beginning, when the snake is short, and then gradually switches to a survival strategy. As shown in Figure 10, the baseline algorithm requires significantly more steps to complete the game compared to our agent.

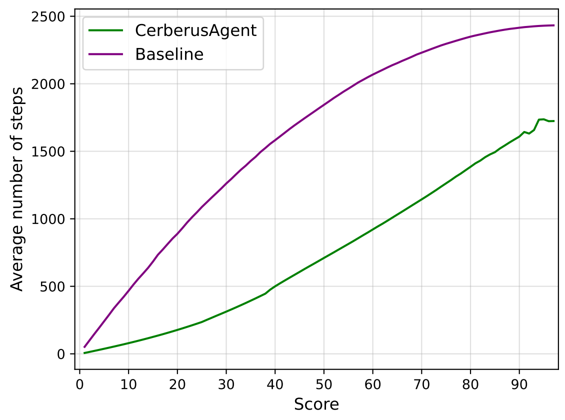

Figure 10. Average number of steps to reach each possible score of the game.

8. Conclusions

In this report we've presented a way to build an agent that can win snake, using a model free approach. However the proposed solution is far from being perfect. In the following we consider the main limitations.

8.1. Low Winning Rate

The agent reaches the end-game very frequently; however, it rarely manages to win. One possible improvement is to design a more sophisticated reward function to enhance performance.

One issue with the policy learned by the third head is that it tends to create holes in the snake's body configuration. To address this, a positive reward could be assigned when no holes are present, encouraging the snake to remain compact.

The goal is to guide the agent toward learning a strategy closer to that of the baseline algorithm.

8.2. Approach Not Scalable

In our experiments, the game grid was relatively small (10×10). The proposed solution does not scale well to larger grids. In such cases all the heads of the agent must be trained from scratch, or a preprocessing step would be required to downscale the game frames.

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.