Efficient Deep Reinforcement Learning and model generalization for the Snake Game

Valentin Badea, Swann Bessa, Vivien Brandt, Ethan Weitzmann

Abstract :
This research project delves into the training of a Deep Q-Learning vision-based agent within the dynamic environment of Snake, employing extensive parametrization to attain optimal performance. Our study focuses on a small square grid of width and length 8, where risks of collision are higher and the training of the agent is thus particularly challenging. We derived a specific preprocessing for the images to strike a balance between complexity and sufficiency of information. To allow for memory-efficient training, we redesigned an environment so that it aligns with the data structures we used as input to our model. By tuning hyperparameters, testing different rewards, and optimizing the action-selection policy, we derived an agent which attains known performances with limited information and computational power. Finally, an original contribution of our paper is to generalize our agent to larger grids by reusing its convolution layers. Full code available here

Introduction

A. Rules of the game. The game of Snake takes place on a rectangular grid with a defined number of rows and columns. The main objective of Snake is to navigate through the grid to eat items (usually represented as dots or apples) that appear at random positions on the screen.

The snake moves continuously in one of four directions: up, down, left, or right. The player controls the direction, but the snake never stops moving. Each time the snake eats an item, its length increases by a predetermined amount, usually one tile.

The game is over either when the snake collides with a border of the grid, or when it collides with its own body. The difficulty increases as the snake grows longer, making navigation and avoidance of self-collision more challenging.

The player uses directional inputs to change the direction the snake is moving. However, the snake cannot move in the opposite direction of its current movement to prevent it from immediately colliding with its own body.

B. Previous work. Deep Reinforcement Learning has previously been used for various video games, most notably Atari (1) with vision-based approaches. The Snake game provides an interesting case of application, since its grid structure is particularly suitable for vision applications (2) (3) (4). (2) provides a modified DQN model with a specific replay buffer to optimize memory management. (4) shows how reward crafting can benefit the training of the agent. While (3) uses categorical indices to encode the different tiles of the image, (2) and (4) directly use a preprocessed version of the original image. Finally, (5) provides a lightweight environment which allows for fast training.

C. Aim of the article. This article builds on previous work to attain an optimal DRL agent for the Snake game, in a context where limited computation resources are available. While the fully observed environment (respecting the Markov property) allows for various other architectures, our aim was to attain an efficient agent through parametrization, comparable with previous approaches. This is why we focused on Deep Reinforcement Learning which has already been used for this game. We derived a specific image preprocessing to attain a tradeoff between complexity and sufficiency of information, and redesigned an existing environment taken from (5) so that it fits optimally to the data representation. After optimizing the agent through extensive parametrization, we explored how agents trained on smaller grids could be used for the training on larger grids, which is much heavier computationally.

Background

A. Double Deep Q-Network. The standard Reinforcement Learning setting and the issues raised by Vanilla Deep-Q-Learning are detailed in the Supplementary Note 1. To prevent those issues we follow the work done in (6) and apply a DDQN approach, where action selection and evaluation are decoupled. The corresponding algorithm 1 inspired from (7) is detailed in the appendix. We will therefore use a model Q_ϕ (for action evaluation) and a target model $Q_{\phi'}$ (for action selection). We compute the mean squared error between y_t^{DDQN} and $Q_\phi(s,a)$, where y_t^{DDQN} is:

$$y_t^{DDQN} = r_t + \gamma max_{a'} Q_{\phi'}(s_{t+1}, a)$$

where ϕ' is updated following the parameters of the prediction network (less often). To reduce the correlation between samples, we also use a replay buffer to simulate iid sampling. At every time step, to select a new action, we implement an ϵ-greedy strategy, where we either take an action that maximizes our action-value estimate (with probability $1-\epsilon$) or a random action (with probability ϵ). This enables to obtain certain a trade-off between exploration and exploitation.

B. Snake in Reinforcement Learning. The Snake game is a fully observed environment, since the player has access to the whole grid, as well as the dynamic of the snake. It is also deterministic, since the next direction chosen by the player leads deterministically to a single next state.

Considering that at each state we know the exact dynamic of the snake, the Snake game also verifies the Markov Property. In conclusion, the Snake game is a Markov Decision Process in a fully observed deterministic environment.

Methodology & Approach

A. Environment. The snake environment is a 2D grid of dimension (n_{rows}, n_{cols}). For a given set of coordinates (x, y), we would access the corresponding point in the grid with grid[x][y], so that Numpy's array convention aligns with the problem coordinates.

Food: One unit of food randomly spawns on the grid (anywhere outside of the snake's body) at the beginning of the game and every time a piece of fruit has been eaten by the snake.

Snake: The snake is a Python list of contiguous 2D-coordinates $(x_i, y_i)_{i=1,\dots,length}$ that evolve through time. At the beginning of the game, the snake spawns in the middle of the grid, with initial length 3. At every time step, the snake moves according to the policy learned by the agent. Let (x_h, y_h) be the new position of the head. To update the list, we append (x_h, y_h). If a fruit was present at those coordinates, the snake's size increases by 1, so the list we have well corresponds to the new state of the snake. If not, we pop the snake's tail, resulting in the visual impression that the snake is moving one unit further. The snake's head can move along 3 relative directions (or 4 absolute directions). From its own perspective, the snake can move straight, left or right. From the observer's perspective, it can go right, left, up or down (one of these directions however is not accessible as the snake head can't go back).

Game termination: The game ends when the snake hits a wall or its own body parts (i.e. when the next position of the snake's head is already in the snake's body or is outside of the grid).

B. Reward system. The most straight-forward way of designing a reward system for this game would consist in applying a positive reward (r_{food}) whenever the snake eats a piece of food and a negative reward ($-r_{termination}$) whenever the game ends. However, most of the difficulty behind this approach comes from the fact that rewards are often delayed from the moment the agent took a specific action. A possibility of improving this comes from a reward regularization mechanism (shown in (4)), which is based on the log-distance between the snake's head and the piece of food. At any time step $t > 0$, the snake gets rewarded with:

$$\delta r_t = log_{L_t}(\frac{L_t + D_t}{L_t + D_{t-1}})$$

where L_t is the snake's length and D_t is the euclidean distance between the snake's head and the piece of food.

To complete this reward scheme, we also add a timeout strategy (also taken from (4)) to make sure that the agent does not become complaisant (and pointlessly hangs around the piece of food). If the snake stays too long without eating (namely more than $0.7L_t + 10$ iterations), we apply the following penalty:

$$\delta r_t = -\frac{1}{2L_t}$$

We will be comparing these two reward mechanisms later in this paper.

C. Network architectures. Throughout this project, we used 2 main architectures depending on the state types we considered. Supplementary note 2 details and discusses the pros and cons of different state types.

Vector state: This approach consists in predicting the action output (with size 3) from a boolean vector of size 11 which encodes the dangers around the snake (see last paragraph of supplementary note 2). The most natural way to proceed is thus to use a Multi-layer perceptron. To perform this Q-learning task, we defined a simple neural network with one fully-connected hidden layer ($n_{hidden} = 256$) using Relu activation function.

Board state: We adopted an architecture with convolutional layers to exploit the grid structure of the game, in a vision-based fashion as previously done with Atari Games ((1)). Here, the visual patterns to identify are quite simple : the shape of the snake, the position of the head and the position of the food. Therefore, we opted for an architecture inspired from LeNet5 (8), a model used to identify hand-written digits. The CNN trains on the game boards with two channels that we designed : one to track the position of the snake body (1 for the snake's body, -1 for its head, 0 elsewhere), one to track the food position on the grid (1 for the food position, 0 elsewhere). As mentioned in the supplementary note 1, this environment is not quite Markovian, but might be sufficient since we're keeping track of the position of the snake's head differently from the rest of its body. The CNN we use has 2 convolutional layers connected to 2 fully connected dense layers with resp. size $32 \times n_{rows} \times n_{cols}$ and 512. You may see a scheme of the network architecture on Figure 1.

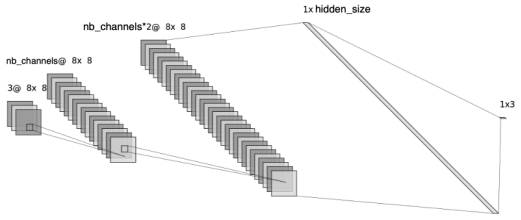

Fig. 1. CNN architecture chosen

Results and discussion

We carried out our experiments on a grid of size 8 (64 tiles), as represented in figures 10 and 11 of the appendix. While previous works focus on larger grids, we chose this size so as to put our RL agent in challenging situations as quickly as possible. For example, a snake of length 16 already occupies

a quarter of the board, thus making the avoidance of collision tricky.
As previously detailed, we used a Deep-Q-Learning agent, with a target network and a replay buffer.

A. Optimizing the neural network architecture. We optimized our neural architecture by looking at two specific parameters : the number of channels of the convolution layers, and the hidden size of the classifier (parameters $nb\_channels$ and $hidden\_size$ on Figure 1). We trained 3 models for 4000 episodes on each set of parameters and averaged them. Figure 2 shows the results obtained.

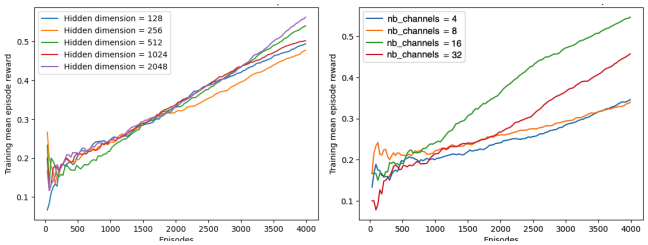

Fig. 2. Left : Mean scores depending on the MLP's hidden dimension, Right : Mean scores depending on the $nb\_channels$ parameter of the convolution layers (see architecture)

Due to the parallelization ability of the GPU, the computing time was quite similar for all parameters. Interestingly, the parametrization of the convolution layers have much more impact on the mean scores than that of the classifier, thus showing the importance and interest of the convolutional step. According to the right plot, setting the $nb\_channels$ parameter to 16 is optimal. Lower values fail to capture the complexity of the task, while higher values seem to overparametrize the model. The $hidden\_size$ parameter of the MLP had less impact on the averaged mean scores. The scores don't benefit much from a $hidden\_size$ above 512, which we concluded was the optimal tradeoff between complexity and performance.

B. Finding the right tradeoff between exploitation and exploration. As described previously, we used ϵ-greedy as our action-selection policy, thus we tuned the initial epsilon in order to find the best tradeoff between exploitation and exploration. The snake needs enough exploitation in order to increase in length, and learn how to progress in the game. This is especially true since the longer it gets, the harder the game is. Exploration is needed to find new routes and strategies. We trained 3 models on 2000 episodes with different epsilon-values. We also recorded the times taken for the training of each epsilon values. Then, we tested the resulting agent in a full-exploitation mode on 300 games to record performance. The results are summed up in figure 3 and the table below.

ϵ	0.1	0.2	0.4	0.6	0.8
time (s)	104	108	66	51	35
test score	0.36	0.52	0.47	0.44	0.29

Interestingly, epsilon values that are above 0.6 tend to make the training mean score saturate. Even though the test score

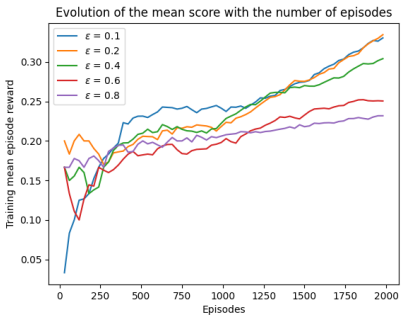

Fig. 3. Mean scores depending on value of $epsilon$

is decent for $\epsilon = 0.6$ (see table), the saturation of the training score shows that this value of epsilon lacks enough exploitation to get to higher lengths. The randomness provokes too many unintended collisions which prevent the snake from progressing. While the value of $\epsilon = 0.2$ achieves the best train and test scores, it takes twice as much time as $\epsilon = 0.4$. This is in part because the forward passes in the network are more numerous (random steps are faster to compute), and also because the games are longer since less randomness causes less collisions. To get the best tradeoff between exploitation, exploration and computation time, we thus opted for $\epsilon = 0.4$.

C. Optimizing learning. To optimize the learning process, we tuned the learning rate, the discount factor γ, as well as the $batch\_size$ parameter of our replay buffer (see algorithm 1). Again, we trained 3 models for each set of parameters for 2000 games and averaged them. Since the $batch\_size$ significantly affects the computation time of our model, we also recorded the computation times for 2000 games for each parameter. The results are presented In figure 4 for the $batch\_size$ and the learning rate. The figure 8 for γ is in the appendix.

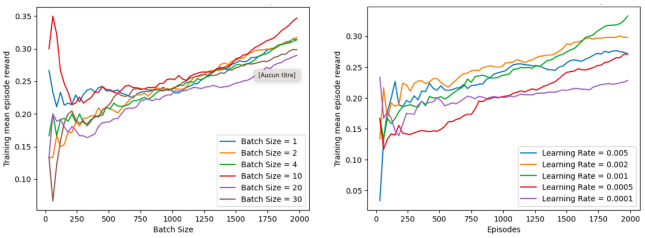

Fig. 4. Left : Evolution of mean scores depending on the $batch\_size$, Right : Evolution of mean scores depending on the learning rate

$batch\_size$	1	2	4	10	20	30
time (s)	186	125	84	67	52	50

Those figures allow us to choose the $\gamma = 0.95$ and $learning\_rate = 0.001$, which are usual in RL. Regarding the $batch\_size$, a value of 10 seems to strike the best balance between computation time and performance.

D. Reward crafting. We tested two reward schemes for our agent, as described in Methodology B. We trained agents on 20,000 games, and then compared performances on figure 5.

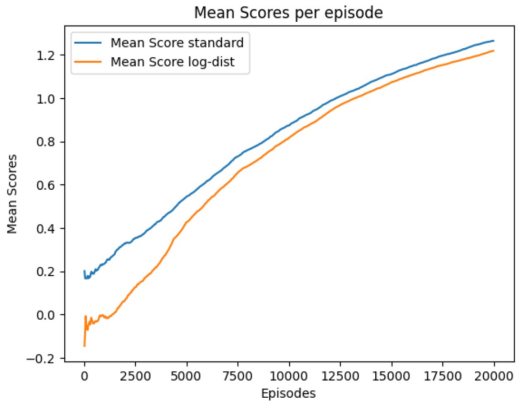

Fig. 5. Evolution of the mean training score depending on the reward scheme

Our findings suggest that both approaches have similar learning curves, and the snake doesn't benefit much from the log-distance based alternative. This might be because the learning-rate and γ were already fine-tuned, thus the snake's long term reward optimization was already successful.

E. State types. In this subsection, we trained an MLP (1 hidden layer of size 512) Q-Learning model for an agent with a precomputed 11-boolean vector as state (see Methodology C). We then compared it with the previously optimized convolutional approach on 30,000 episodes. We used the same hyper-parameters for both agents.

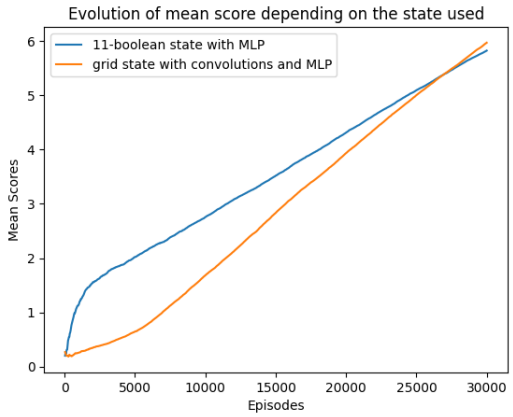

Fig. 6. Evolution of the mean training score depending on the state space used as input of the neural network

Figure 6 shows that the first agent learns very fast at the beginning, before slowing down progressively. Conversely, the vision-based agent's curve increases and even achieves better mean score after 30,000 episodes. These results can be explained by the fact that the 11-boolean vector provides explicit yet incomplete information about the snake. Conversely, the grid state almost completely describes the whole environment in a vision-based manner. While the learning takes more time to be initiated for the vision-based agent, since the model needs to extract visual features from the grid, the agent obtains a steady learning curve after 10,000 episodes allowing it to overtake the 11-boolean agent.

F. Performance of the optimized agent. Given the previous results, we trained our optimized agent on 50,000 games. Figure 9 of the appendix represents the rewards obtained for 5000 games once the agent is already trained and the action-selection policy is in full exploitation.
Our agent obtains average test scores of 14.2, with a record of 31. A score of 31 is particularly impressive since it corresponds to the snake occupying half of the board (length of 34). When looking at the actual games played by the snake, we observe that the snake adopts common strategies for human players such as moving along the edges of the grid to reduce collision risks. The average test length for 50,000 training epochs is similar to what is obtained in (3), an approach that uses 2 frames as input while we only used 1.

G. Model generalization. Finally, we tested how our agent trained with a convolutional network on 8*8 grids was able to generalize to larger 16*16 grids. The MLP part is not directly transposable to larger grids, because the number of nodes is inevitably different. However, the convolutional layers, which extract visual features, are reusable.
This is why we carried the following experience : we trained 2 agents for the Snake game on a 16*16 grid. On the first one, the weights of the neural network were randomly initialized as before. On the second one, the convolutional layers were initialized with the weights of our optimized 8*8 agent trained on 50,000 games. Figure 7 shows our results.

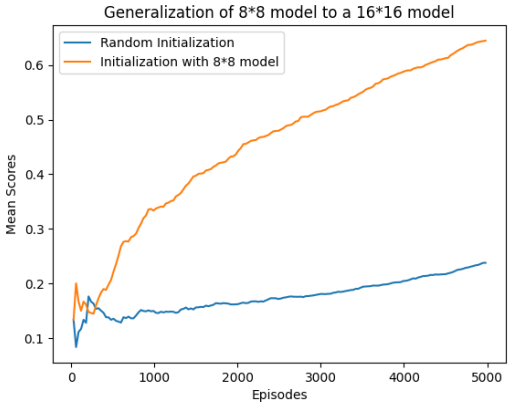

Fig. 7. Evolution of training mean scores on a 16*16 grid for a randomly initialized agent (blue) and an agent initialized with the 8*8 convolutional weights (orange)

The learning is impressively faster for the agent initialized with 8*8 grid convolutional parameters. This result is crucial for the training of larger grids, since they can be very burdensome computationally. There are 4 times more tiles in a 16*16 grid than in a 8*8 grid, thus resulting in much slower training. By pretraining an 8*8 agent, one can already extract visual features and use them to train a 16*16 agent much more rapidly, thus saving time and computational power.

Conclusion

In this paper, we used extensive parametrization to derive an optimized DRL agent for the Snake Game, which performs as well as previous approaches with limited informa-

tion. By carefully tuning learning parameters, finding a right exploration-exploitation tradeoff, and optimizing the neural architecture, we obtained mean test scores of 14 and a record of 31. An original addition of the paper consists in achieving model generalization by reusing convolutional weights. Our results show that such a practice substantially speeds up the training of agents on large grids, which would otherwise be very burdensome computationally.

Appendix

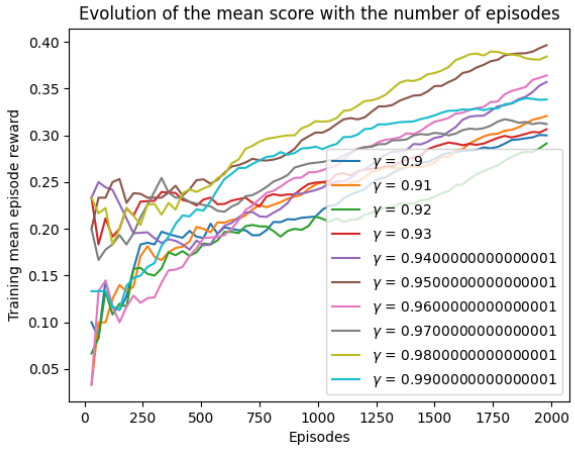

Fig. 8. Left : Evolution of mean scores depending on the $batch\_size$, Right : Evolution of mean scores depending on the learning rate

Algorithm 1 DDQN Algorithm

Initialize replay buffer $\mathcal{D}$ and deep neural networks Q_ϕ and $Q_{\phi'}$ with $(\phi' \leftarrow \phi)$
$s_1 \sim p(.)$
for $t = 1, \ldots, T$ **do**
  $a_t \sim \pi_\phi(.|s_t)$
  $r_t, s_{t+1} \sim p(.,.|s_t, a_t)$
  $\mathcal{D} \leftarrow D \cup (s_t, a_t, r_t, s_{t+1})$
  if $t\%batch\_size = 0$ **then**
    $(s_{t_i}, a_{t_i}, r_{t_i}, s_{t_i+1})_{i=1,\ldots,batch\_size} \sim D$
    $y \leftarrow r_t + \gamma max_a Q_{\phi'}(s_{t+1}, a)$
    $\hat{y} \leftarrow Q_\phi(s_t, a_t)$
    $\phi \leftarrow \phi - \alpha \nabla_\phi (y - \hat{y})$
  end if
  if $update(\phi')$ **then**
    $\phi' \leftarrow \phi$
  end if
end for

Fig. 9. Test rewards for 5000 games after hyperparametrization and training over 50,000 games

Bibliography

1. D. Silver A. Graves I. Antonoglou D. Wierstra M. Riedmiller V. Mnih, K. Kavukcuoglu. Playing Atari with Deep Reinforcement Learning. *[cs.LG]*, 2013.
2. Shahnewaz Siddique Md. Rafat Rahman Tushar. A Memory Efficient Deep Reinforcement Learning Approach For Snake Game Autonomous Agents. *CSAI*, 2023.
3. https://github.com/dragonwarrior15/snake-rl. *GitHub*, 2024.
4. Ming Zhang Ah-Hwee Tan Chunyan Miao You Zhou Zhepei Wei, Di Wang. Autonomous Agents in Snake Game via Deep Reinforcement Learning. *IEEE International Conference on Agents (ICA)*, 2018.
5. Stefan Comanita. https://www.kaggle.com/code/stefancomanita/snake-game-ai-with-reinforcement-learning. *Kaggle*, 2023.
6. D. Silver H. van Hasselt, A. Guez. Deep Reinforcement Learning with Double Q-Learning. *[cs.LG]*, 2015.
7. Jesse Read. Lecture VI - Reinforcement Learning III. *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.
8. Yoshua Bengio Patrick Haner Yann LeCun, Leon Bottou. Gradientbased learning applied to do cument recognition. *IEEE*, 1998.

Supplementary Note 1: Reinforcement Learning setting and Deep Q Learning

A. Generality. As a starting point, let us come back to some basics on Reinforcement Learning. We will follow the framework depicted in the lecture (see reference (7)): at every time step t, an agent observes the environment state $s_t \in S$ (to be determined), selects an action $a_t \in A$ and gets a reward r_t accordingly.
The agent's objective is therefore to maximize the expected discounted return, defined as:

$$R_t = \sum_{k=t}^{\infty} \gamma^k r_k$$

Here, $\gamma \in [0,1]$ is a discount factor that determines how the agent will balance immediate reward vs future rewards. This is however slightly more tricky for our Snake game, as the standard reward policy (agent gets rewarded only when a fruit is eaten) implies mostly future rewards.
Given an agent following a stochastic policy π, we define the Q-value (or action-value function) associated with state-action pair (s,a) as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t|(s_t,a_t) = (s,a)]$$

In Q-learning, we will be approximating an optimal action-value function, as it will give our optimal policy (and optimal action to take):

$$a^* = argmax_{a'} Q^*(s,a')$$

B. Deep Q-network. In Deep Q-network (DQN), we use deep learning to approximate action-value functions with a multi-layered neural network (1). For a given state s, with a set of parameters θ, the network should output $Q(s,a:\theta)$. To train this network at time t, one optimizes its weights following the mean squared error between y_t^{DQN} and $Q(s,a:\theta_t)$, where y_t^{DQN} is defined as:

$$y_t^{DQN} = r_t + \gamma max_{a'} Q(s_{t+1}, a' : \theta_t)$$

with r_t, the observed reward and θ_t the network parameters at time t. This architecture however leads to 3 major issues (7): samples are taken chronogically, thus fostering overfitting and biasing the network to learn a replay. Also, since θ_t always evolves, the network chases a non-stationary target, which is problematic in a learning context. Finally, (6) underlines that this architecture leads to a systematic over-estimation of action-value functions (specifically when taking the max in y_t^{DQN}).

Supplementary Note 2: State types : what it takes to respect the Markov property

Different state types $s \in S$ can be used for the Snake game. The most natural approach is to take a grid (as a numpy array for example), on which categorical variables describe whether each tile is empty, filled with the body of the snake, or filled with the head of the snake. An additional variable can then describe the direction of the head.
However, this approach does not fully convey the dynamic of the snake, and thus does not rigorously allow for the Markov property to be verified. Indeed, in figure 2, the dynamic of the last 4 snake tiles is ambiguous : two movements are possible.

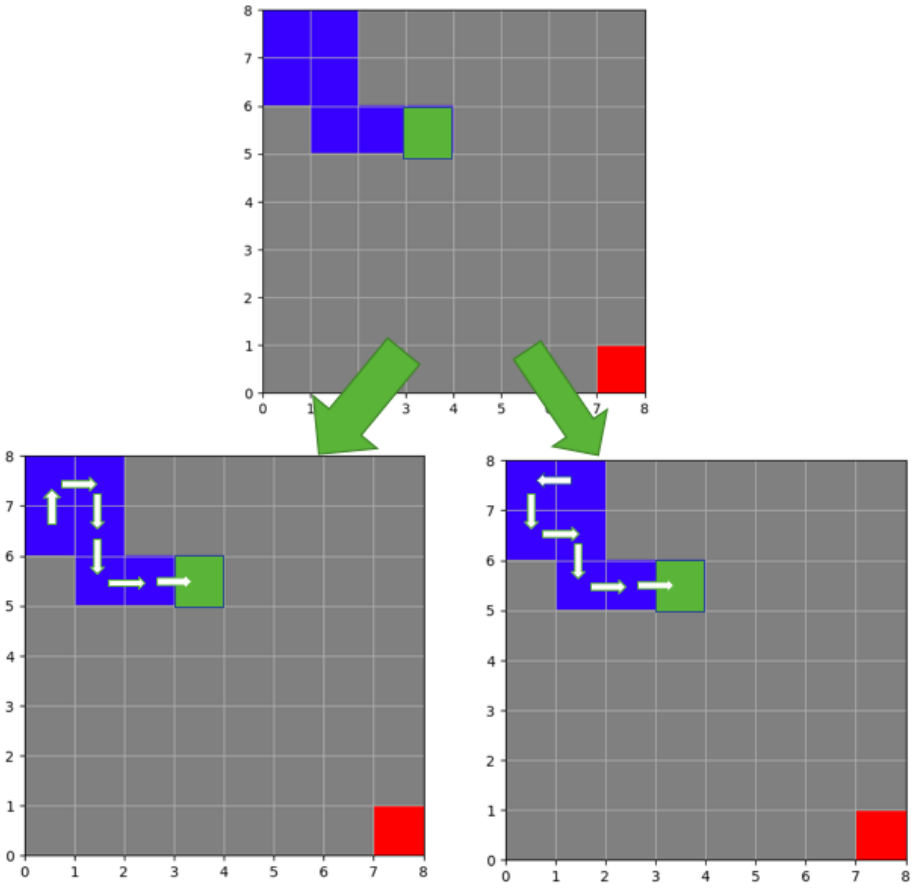

Fig. 10. Two possible dynamics for a single frame (the head of the snake is in green, the body in blue and the food in red)

This is why several frames can be used, as often done in Atari games, so that the movement can be reconstructed by our policy. Figure 3 represents the two frames which allow to infer the movement of the snake we used.

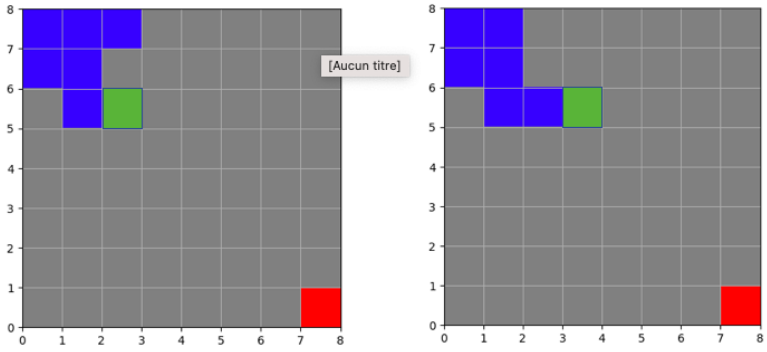

Fig. 11. The two frames allowing to infer the dynamic of the snake

Alternative approaches can directly encode the direction of each tile of the snake in the grid, by using categorical variables with multiple classes. But the closer we get to a rigorously markovian Snake game, the more complex the model gets. There is thus

a tradeoff to find, especially when Snake Games have high numbers of columns and rows.
Finally, a simplistic approach can allow for fast training while seriously limiting the information available to our agent. It uses a 11-dimensional boolean vector as state, which compares the position of the head to that of the food and potential dangers. More specifically, the state indicates with booleans if the food is right/left/up/down compared to the head. It also indicates whether there is a direct collision if the snake moves left/right/forward. All this information, coupled with the direction of the snake, result in a 11-dimensional boolean vector.