

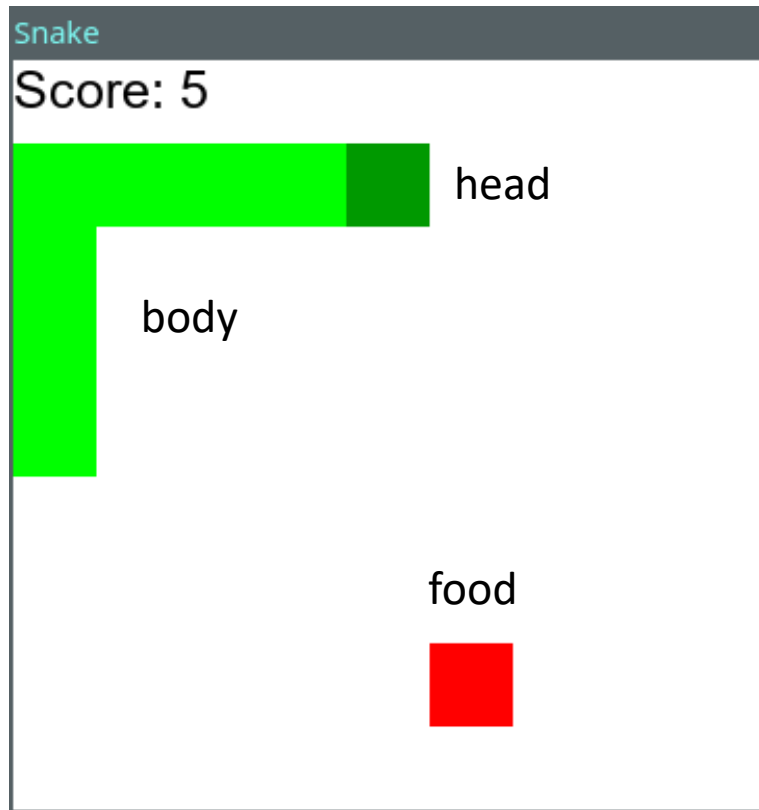
# Reinforcement Learning Snake

- Simone Brusatin
- 10 July 2023
- [github.com/Brusa99/Snake\\_RL](https://github.com/Brusa99/Snake_RL)

# What is Snake?

- Snake is a 1977 video game.
- The player maneuvers the end of a growing line, often themed as a **snake**.
- The objective is to grow as much as possible by eating **food**.
- While maneuvering the player has to avoid obstacles, such as the walls and the **body** of the snake. Hitting the obstacles results in game over.

# How I represent it



- You can play the game by running:  
`python Game.py`
- You can move in 4 directions using the arrow keys. The **head** will move and the **body** will trail on.
- If the **head** reaches the **food**, the snake will become one **block** longer and you will increase your score.
- The objective of the game is to score as high as possible.

# Model of the environment

- The playground is a matrix of size `self.w * self.h`
- Current direction, score, head and food position, and rest of the body position are class attributes.
- Walls are not modeled: head position outside of boundaries is considered as a collision.

```
# init game state
self.direction = Direction.DOWN
# snake
self.head = Point(self.w // 2, self.h // 2) # start in the center
self.snake = [self.head,
               Point(self.head.x, self.head.y - 1),
               Point(self.head.x, self.head.y - 2),
               ]
self.score = 0
self.food = None
self._place_food()
```

# Reinforcement Learning Framework

# Action Space

## 4 absolute actions

- Up
- Down
- Left
- Right



## 3 relative actions

- Turn Left
- Straight
- Turn Right

All actions are deterministic.

# Relative action

- In this way the action space has a lower cardinality, which reduces model complexity.
- Effectiveness of the model is equivalent: going backwards always results in game over.
- The agent must know current direction.

```
# determine direction based on action ~ maps action to direction
clock_wise = [Direction.RIGHT, Direction.DOWN, Direction.LEFT, Direction.UP]
idx = clock_wise.index(self.direction)
if np.array_equal(action, LEFT_TURN):
    new_dir = clock_wise[(idx - 1) % 4]
elif np.array_equal(action, STRAIGHT):
    new_dir = clock_wise[idx]
else: # np.array_equal(action, RIGHT_TURN):
    new_dir = clock_wise[(idx + 1) % 4]

self.direction = new_dir
```

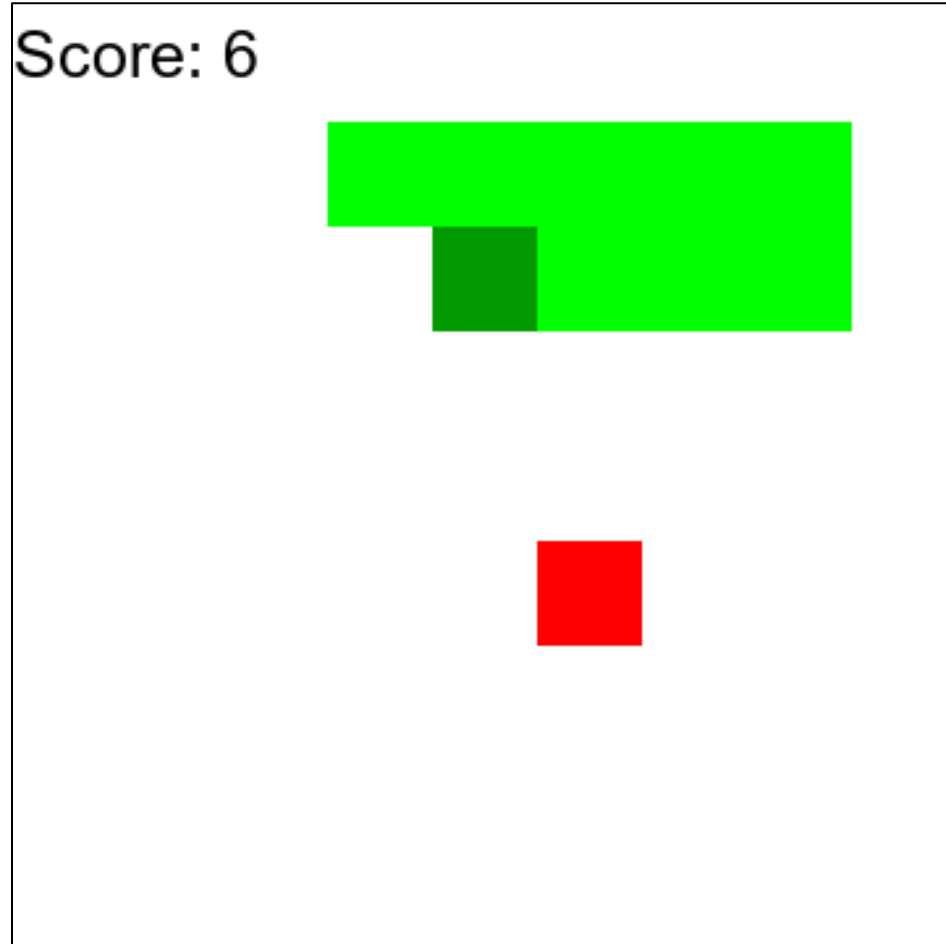
# State Space

- 3 Boolean attributes that signal an immediate collision for each relative direction
- 4 Boolean attributes for each direction
- 4 Boolean attributes for food position:
  - `food.x < head.x`
  - `food.x > head.x`
  - `food.y < head.y`
  - `food.y > head.y`

**\*Note** that this isn't the only option. Different state spaces will be discussed later.

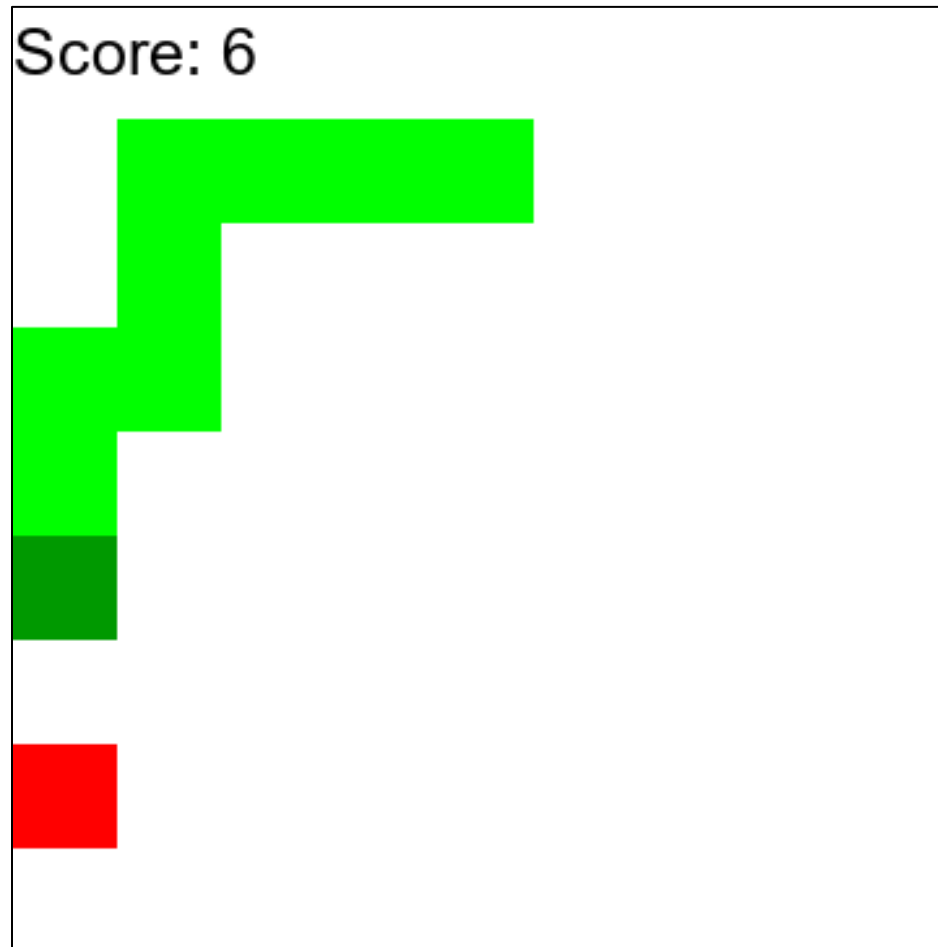


# Example



- 0 danger left
- 0 danger straight
- 1 danger right
- 1 direction left
- 0 direction right
- 0 direction up
- 0 direction down
- 0 food left
- 1 food right
- 0 food up
- 1 food down

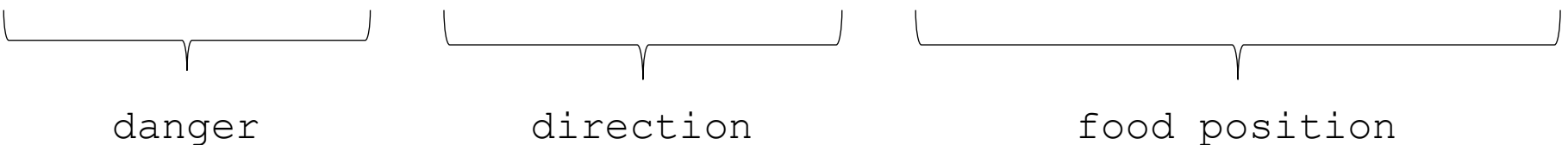
# Example



- 0 danger left
- 0 danger straight
- 1 danger right
- 0 direction left
- 0 direction right
- 0 direction up
- 1 direction down
- 0 food left
- 0 food right
- 0 food up
- 1 food down

# State Space Dimension

- We have 11 variables.
- All variables are Boolean.
- We have a total of  $2^{11} = 2048$  possible states.

$$S = D_l \times D_s \times D_r \times L \times R \times U \times D \times Fx_{<} \times Fx_{>} \times Fy_{<} \times Fy_{>}$$


danger                      direction                      food position

# Rewards

- +15 for eating the food
- -10 for colliding
- -10 if the game goes on for too long ( $\text{MAX\_ITER} * \text{len}(\text{snake})$ )
- -0.1 else\*

\*this is designed to avoid the loop bottleneck: In a vast playground the chance of randomly stumble into the food is very low. So, the agent may adopt “don’t die” as a policy, which is guaranteed in a loop.

Solving the problem

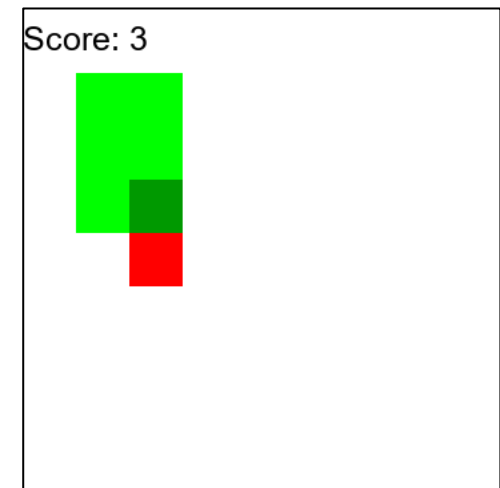
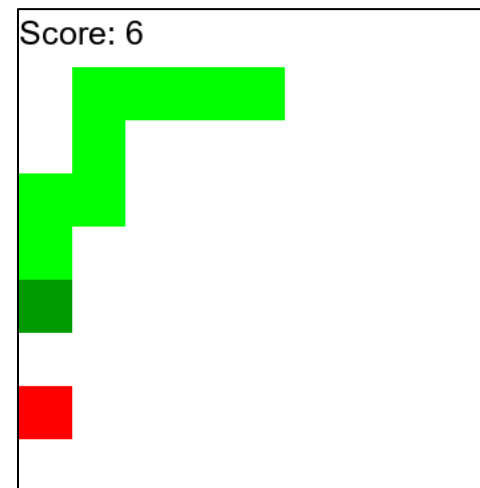
# Model based or model free?

- We have no prior knowledge on the transition probability  $p(s'|s, a)$
- We have limited knowledge on the expected immediate reward  $r(s, a)$ .

**Example:** The figures on the right are interpreted as the same state.

Taking action `Straight` will lead to a reward of 0 in the left game configuration and of +15 in the right configuration.

Taking action `Turn right` will lead to a reward of -10 (and game over) in the first case, while in the second case it will produce a reward of -0.01.



# We will opt for a model free approach

- In a model free context, we know  $N$  sequences obtained by the agent by interacting with the environment:

$$S_0^n, A_0^n, R_1^n, S_1^n, A_1^n, R_2^n, \dots \text{ with } n = 1, \dots, N$$

- We will use TD-Learning to estimate the state-action value:

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

# Value of states

- We can get the value of the states from the action-value. Since

$$\begin{aligned}Q_{\pi}(s, a) &= \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a \right] \\&= \sum_{s'} P(s' | s, a) \left[ r(s, a, s') + \gamma \sum_{a'} \pi(a' | s) Q_{\pi}(s', a') \right] \\&= \sum_{s'} P(s' | s, a) [r(s, a, s') + \gamma V_{\pi}(s')]\end{aligned}$$

- We obtain:

$$V_{\pi}(s) = \sum_{a'} \pi(a' | s) Q_{\pi}(s', a')$$



# TD-Learning

- To estimate the state action values, we will use the TD-Learning algorithm.
- Other methods, such as Monte Carlo, can be used.

---

**Algorithm 2** TD-Learning

---

**Input** Learning Rate  $\alpha \in (0; 1]$ , small  $\epsilon > 0$

- 1: Initialize  $\hat{Q}(s, a) \forall s \in S, a \in A$
- 2: **loop** for each episode:
- 3:     Initialize  $s$
- 4:     **loop** Derive  $\pi$  from  $\hat{Q}$  (with  $\epsilon - greedy$ )
- 5:         Choose  $a$  from  $A$
- 6:         Take  $A$ , observe  $R, S'$
- 7:         Compute TD-error  $\delta$
- 8:         Compute eligibility  $e$
- 9:          $Q \leftarrow Q + \alpha \delta e$
- 10:         $S \leftarrow S'$

---

- For eligibility we will use  $1(S_t = s, A_t = a)$
- To calculate the TD-error we will try different algorithms, for example SARSA:

$$\delta = R + \gamma \hat{Q}(S', A') - \hat{Q}(S, A)$$

- We will follow an  $\epsilon$ -greedy policy, that means that we will take the action which we estimate has the most value with probability  $1-\epsilon$  otherwise we will take an action at random. This permits exploration.

---

**Algorithm 2:** Epsilon-Greedy Action Selection
 

---

**Data:** Q: Q-table generated so far,  $\epsilon$ : a small number, S: current state

**Result:** Selected action

**Function** *SELECT-ACTION*(Q, S,  $\epsilon$ ) **is**

```

  n ← uniform random number between 0 and 1;
  if  $n < \epsilon$  then
    | A ← random action from the action space;
  else
    | A ← maxQ(S,.);
  end
  return selected action A;

```

**end**

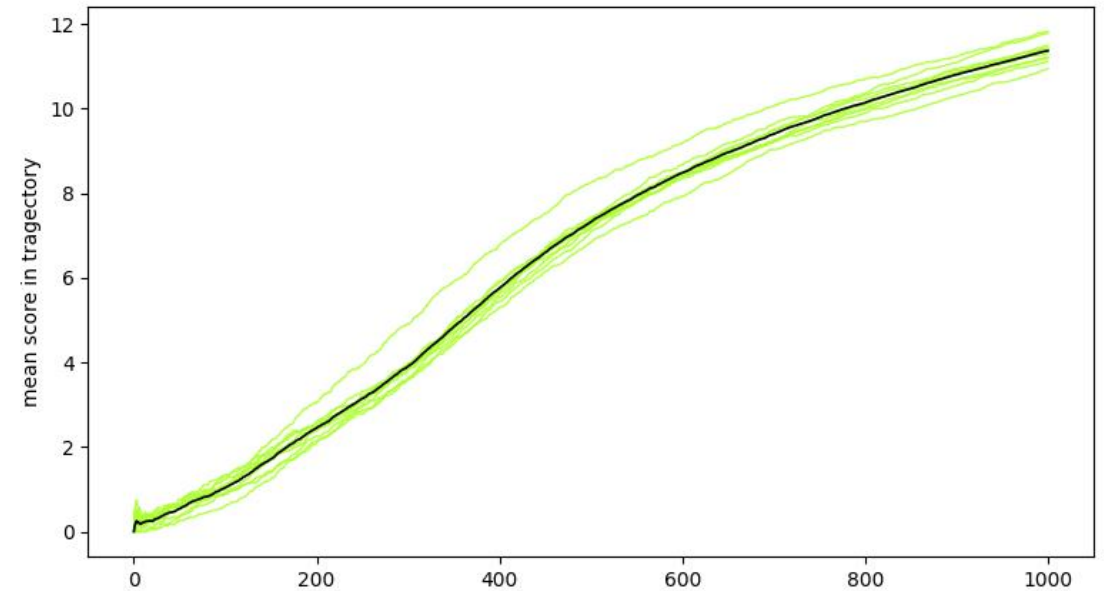
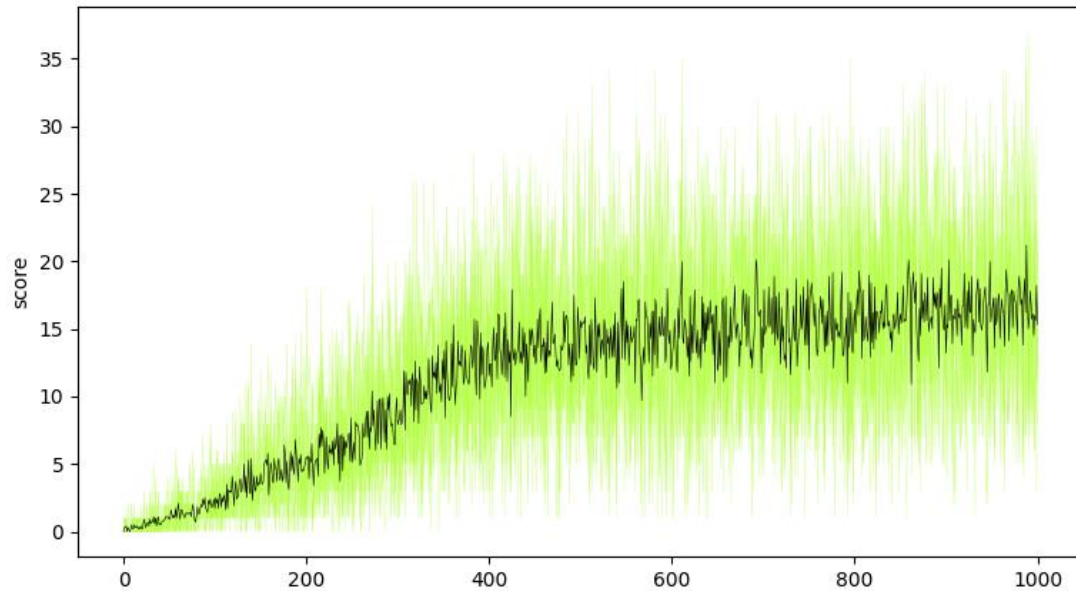
---

# Results

# Training the model

- We trained the model on a  $9 \times 9$  board.
- At around 400 games, the agent stops to learn
- We repeated the learning process 10 times.

SARSA



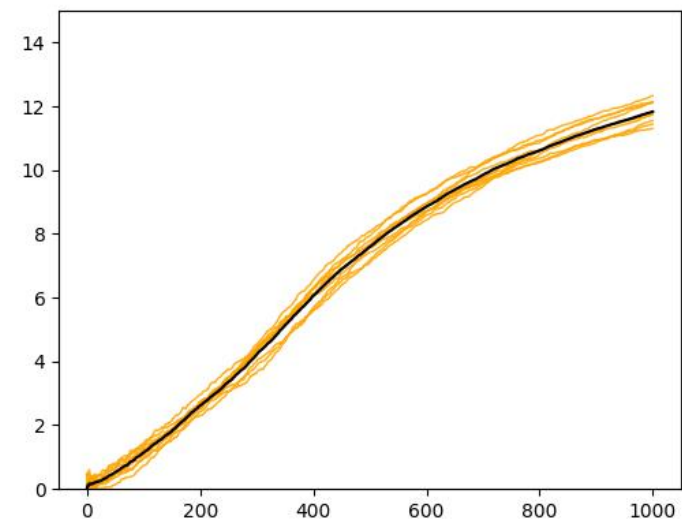
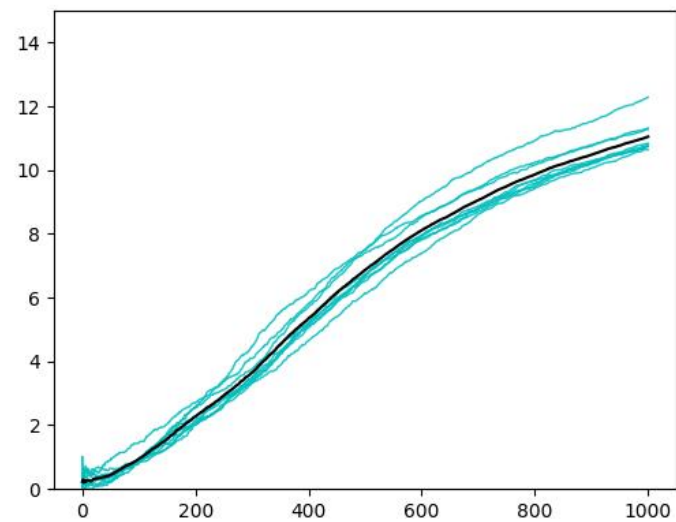
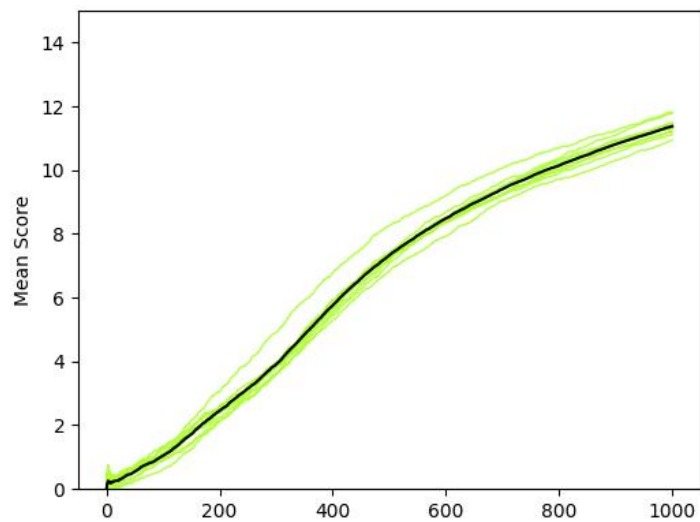
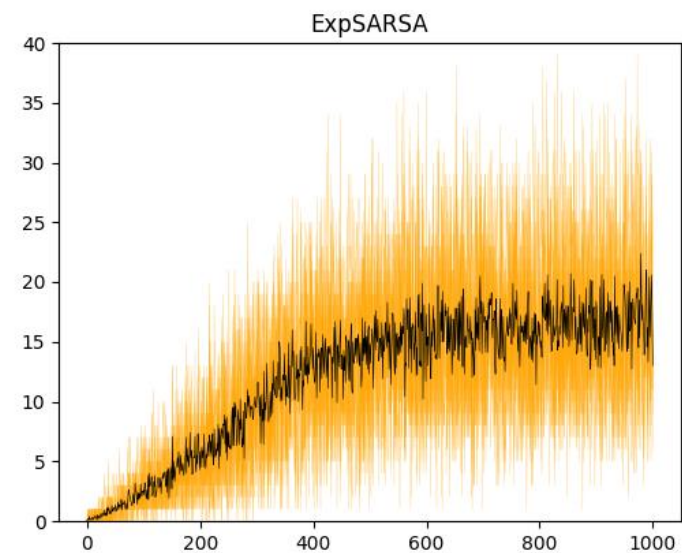
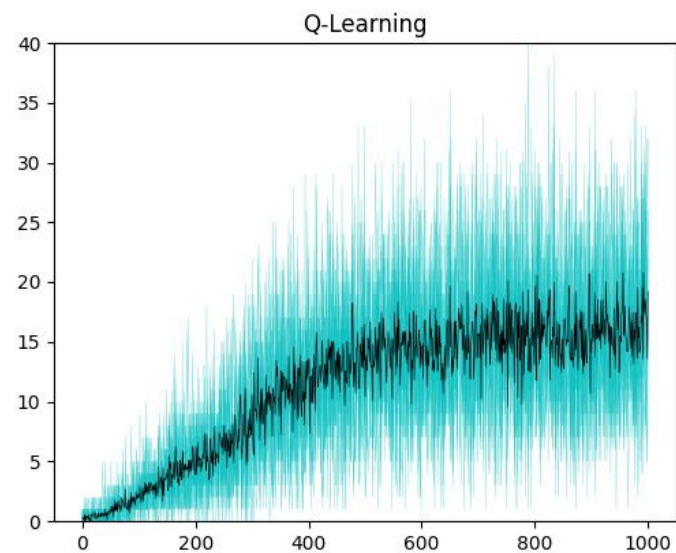
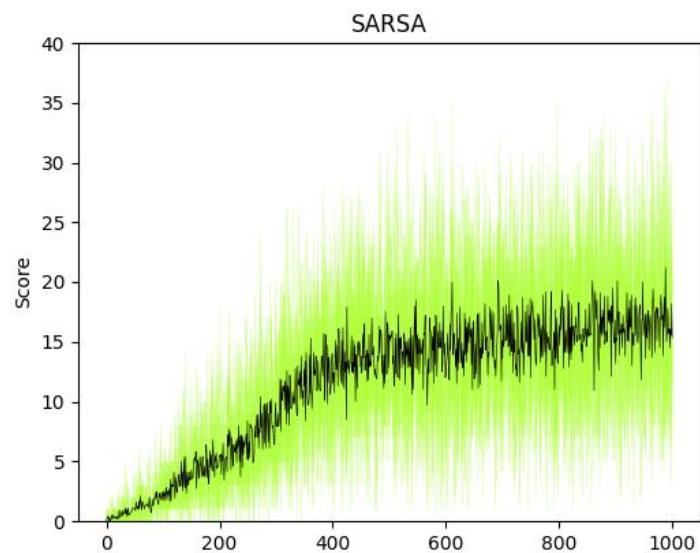
# Other algorithms

- We can try to use different TD-errors for the algorithm, for example:
- Q-Learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

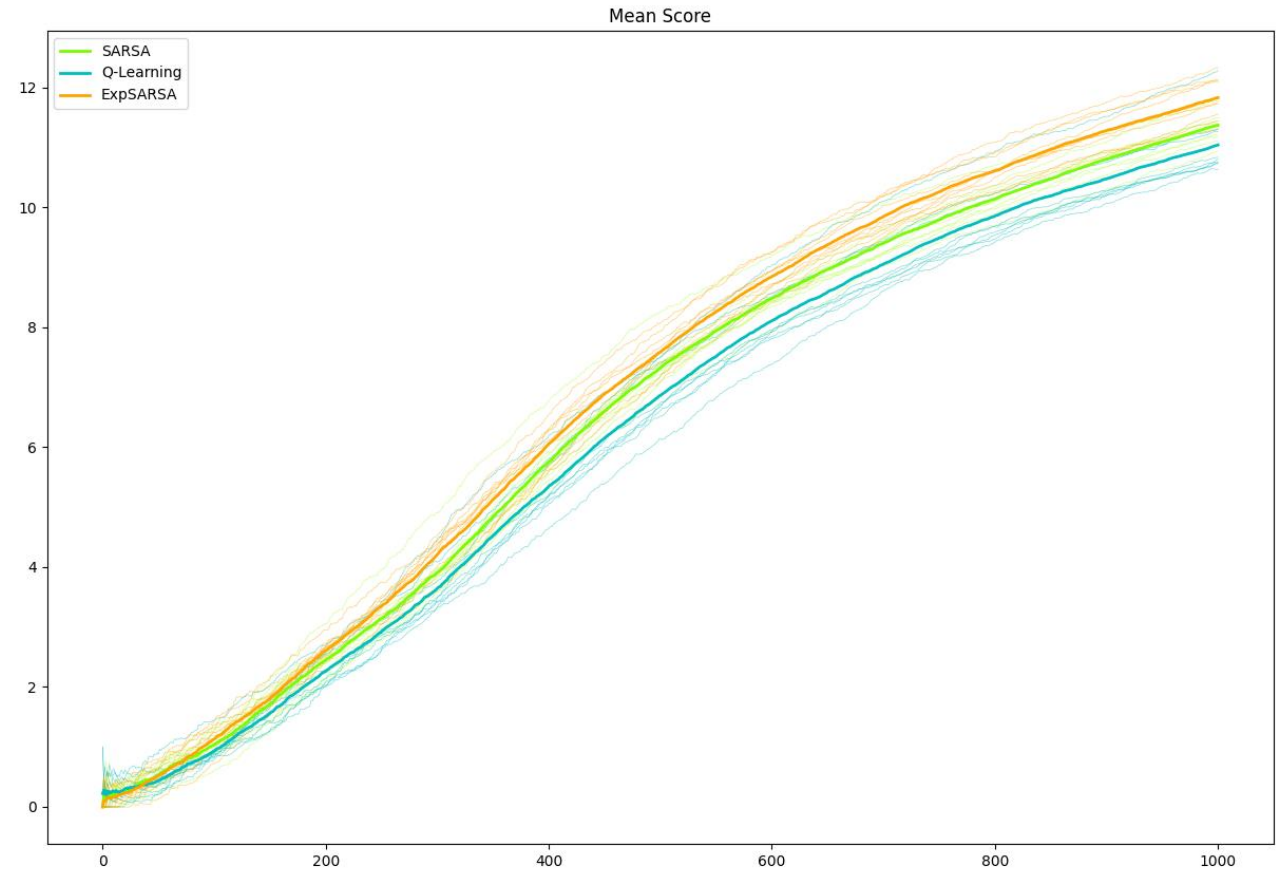
- Expected SARSA:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \\ &\leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \end{aligned}$$



# Comparing the algorithms

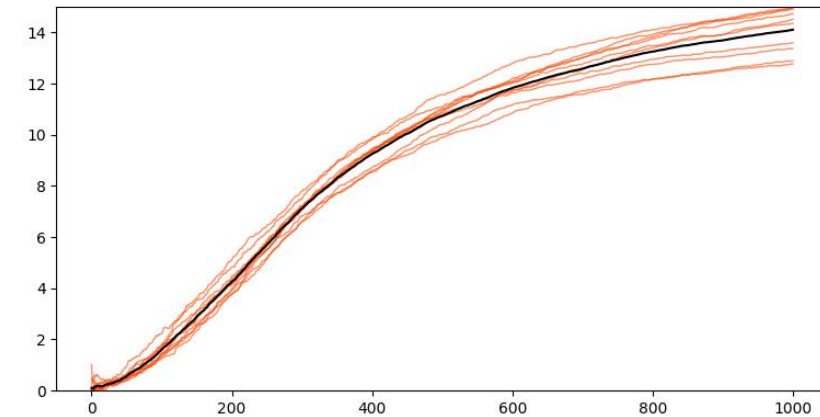
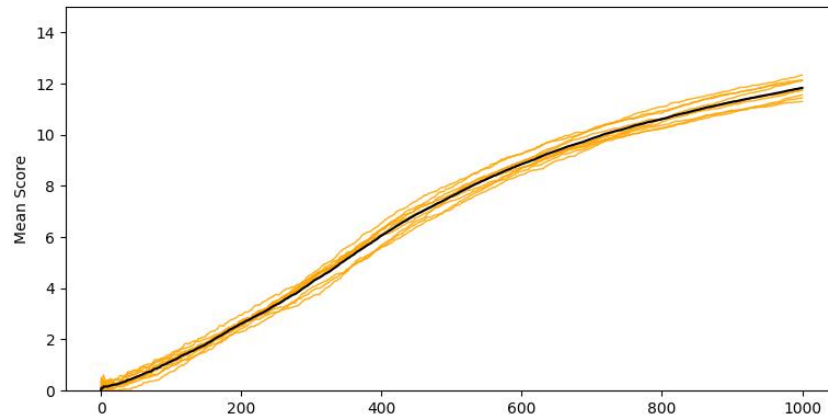
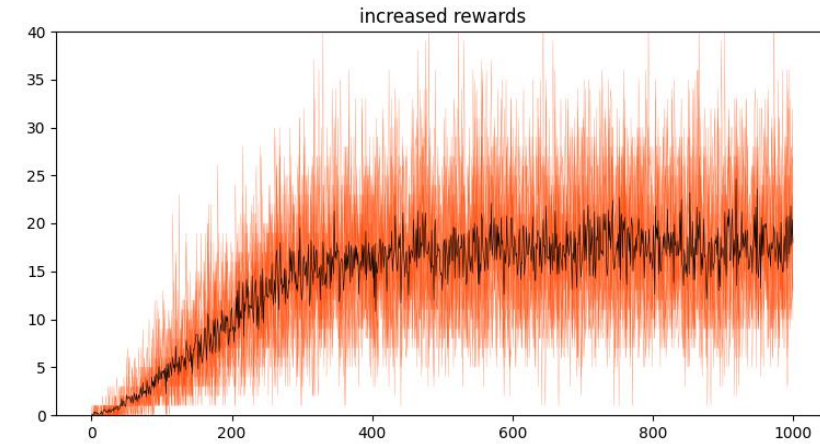
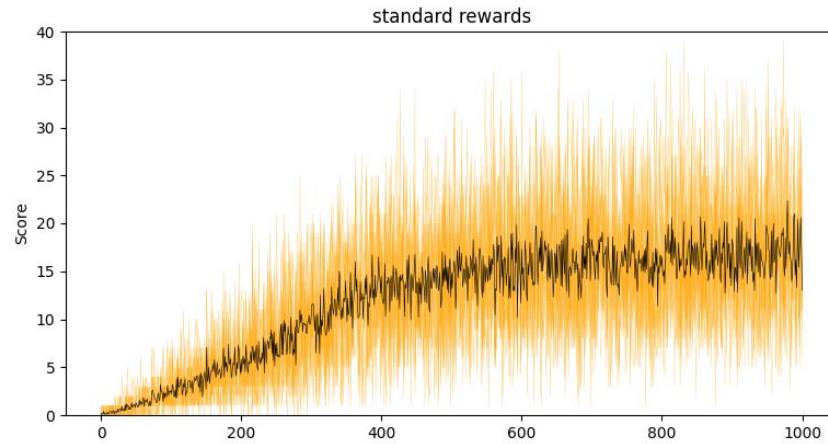
- We are looking at the average score of the models.
- The idea is that the faster it grows, the sooner the model has learned the optimal policy.
- The fastest model uses the expected SARSA algorithm.



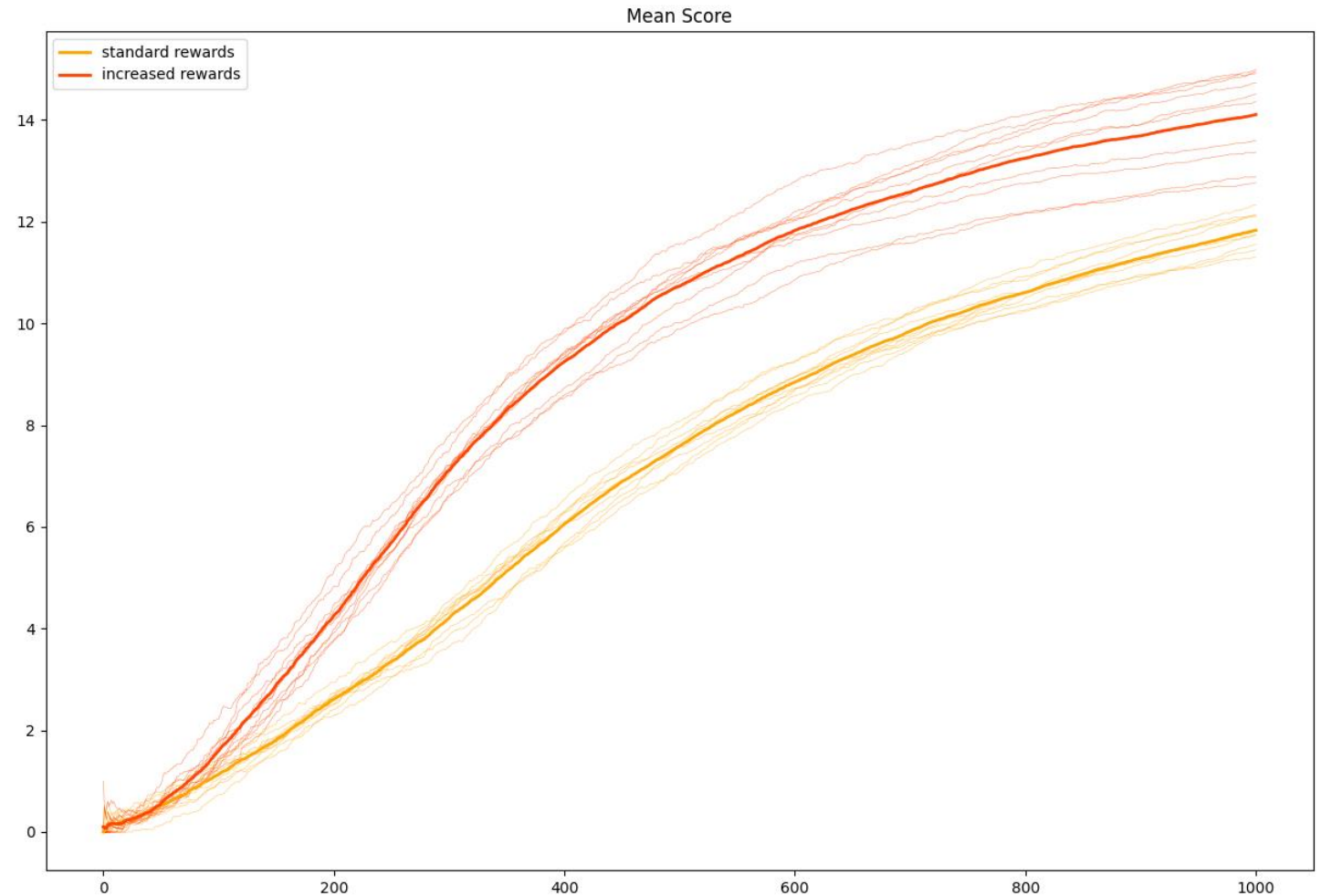
Changing parameters



# Increased rewards



- We increased the positive of eating food from 15 to 150.
- The algorithm converges faster but it has more variance.
- This is due to the rewards being sparse.



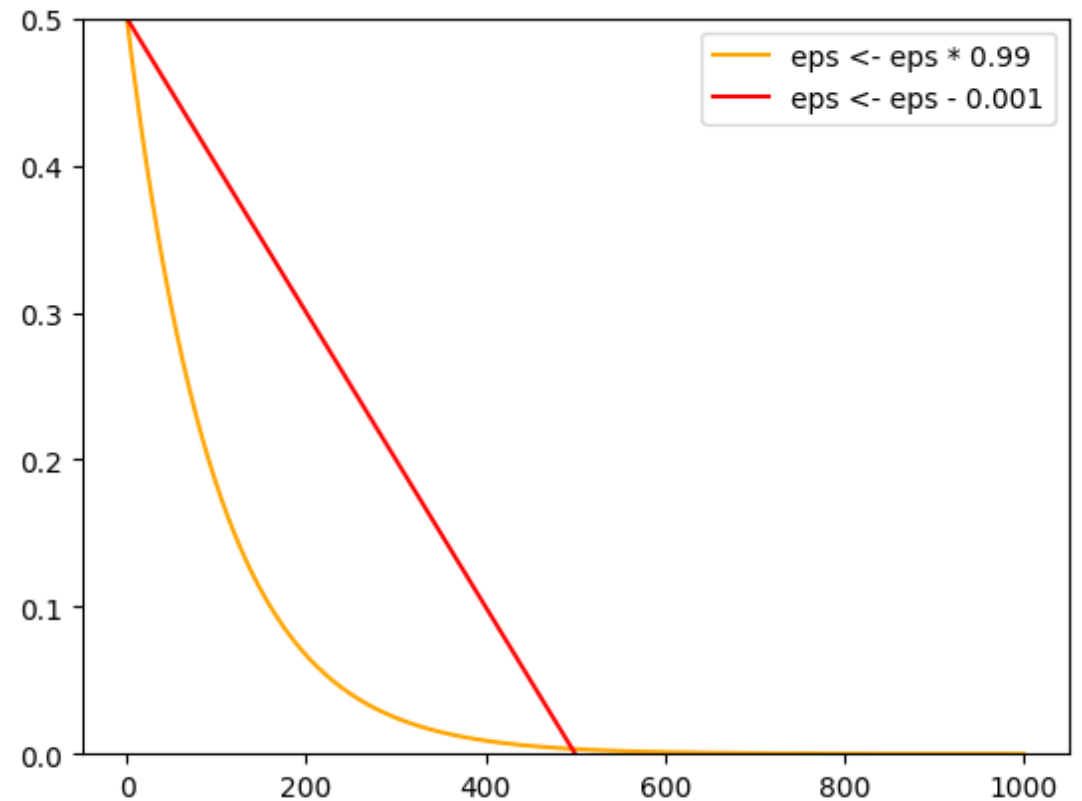
# Change how $\epsilon$ decreases.

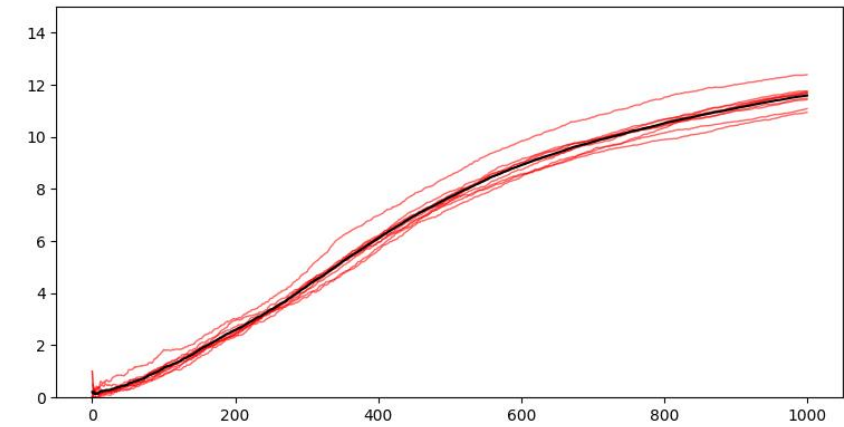
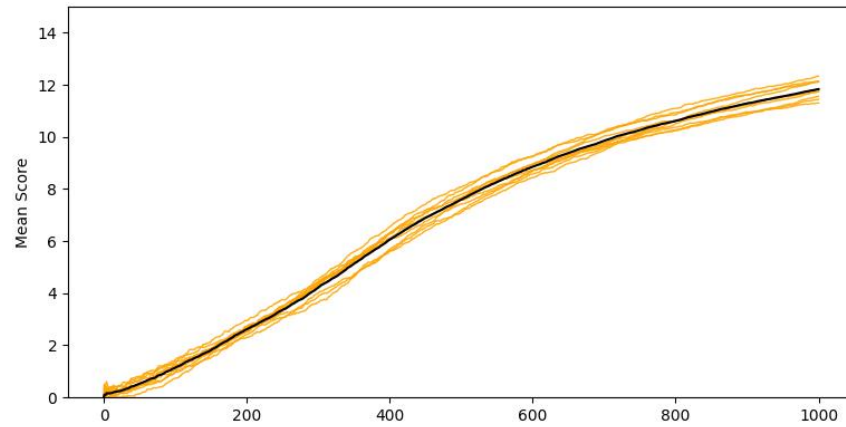
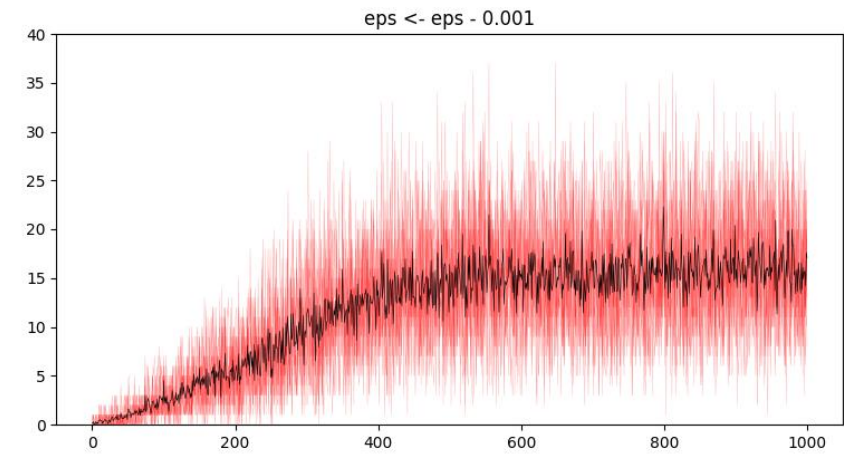
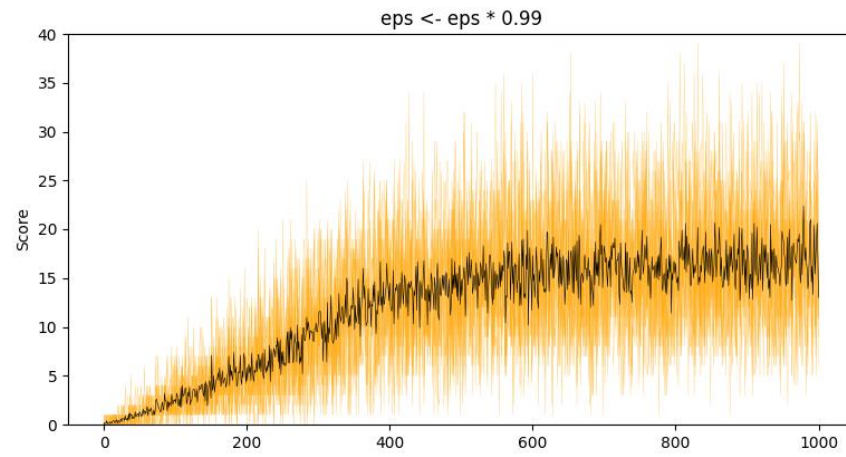
- In the previous models, to update  $\epsilon$ , we used the rule:

$$\epsilon_t = 0.99 \epsilon_{t-1}$$

- We tried instead a linear decrease:

$$\epsilon_t = \epsilon_{t-1} - 0.001$$



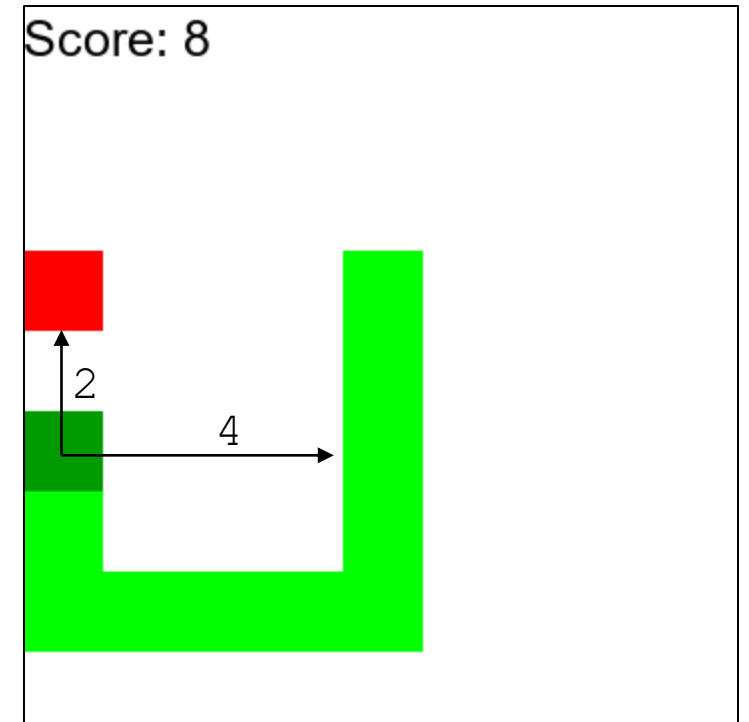


- It doesn't have any effect

How could it improve

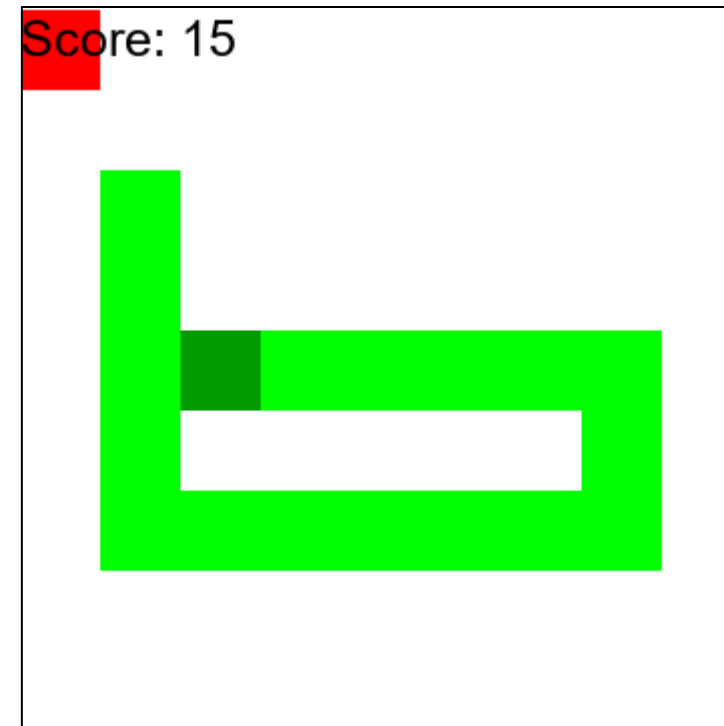
# Different state space configurations

- Instead of signaling danger only in three directions, we can also signal the corners.
- Instead of a Boolean we could use distances.
- We could use binning to limit computational complexity increase.



# The problem

- From the picture we can see that if the snake turns left he will become stuck by his own body.
- The snake isn't aware of his body movement, due to how the game is represented through states.
- Getting stuck is the only way to lose if playing with the optimal policy.



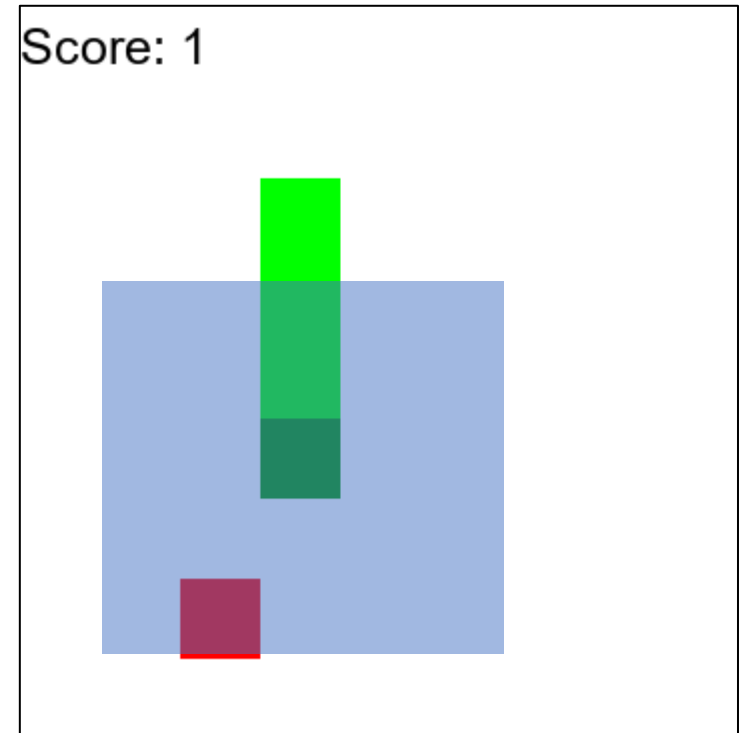
# Solution?

- We could change our state space.
- We can make the state space a vector representing each cell of the playground, with different values representing `empty`, `blocked`, `head`, `food`.
- This will lead to a total of  $4^{w*h}$  possible states. Which is computationally unfeasible.
- For example, for a  $9 * 9$  board, there will be in the order of  $10^{48}$  states.
- Moreover, the learned policy won't work on different sized playgrounds.



# Slight improvements

- Instead of giving the whole board we can only give a limited area around the snake head.
- Still computationally expensive but better.
- State no longer depend on playground size.
- Doesn't solve the problem at its core.



# Trivial Solution: Space filling curves

- Compute a curve that fills the square.
- Make the snake follow the curve in a loop.
- No actual learning.

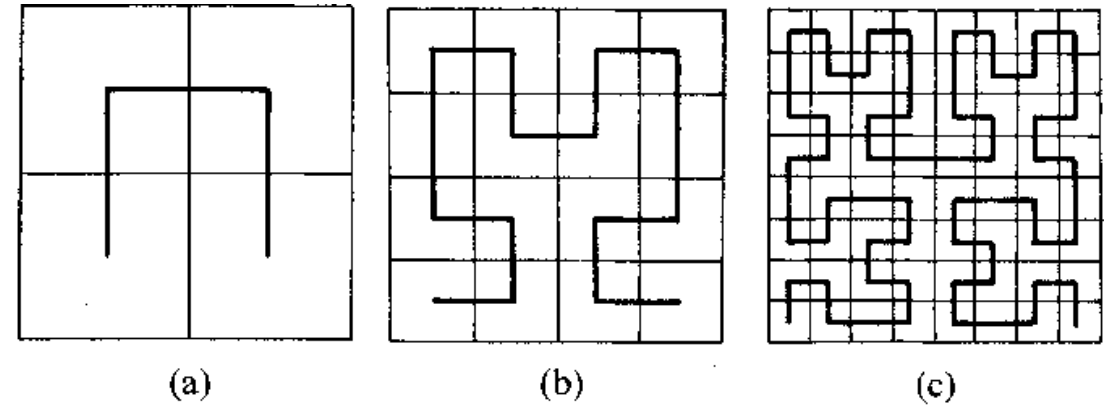


Fig. 1. Hilbert's geometric construction of a space-filling curve.

The End