# 50.039 Theory and Practice of Deep Learning

## W11S2 Introduction to Reinforcement Learning

Matthieu De Mari

# About this week

1. What is **Reinforcement Learning**?
2. What are the key ideas behind **reinforcement learning** and its **framework**?
3. What is the **exploration vs. exploitation tradeoff**?
4. How do we **train** an **RL agent** by exploring, then progressively exploiting?
5. What are some **advanced strategies** in **multi-arm bandit problems**?
6. What are the **Q** and **V functions** for a RL problem?
7. What is **Q-learning** and how can it be implemented in RL problems?

# About this week

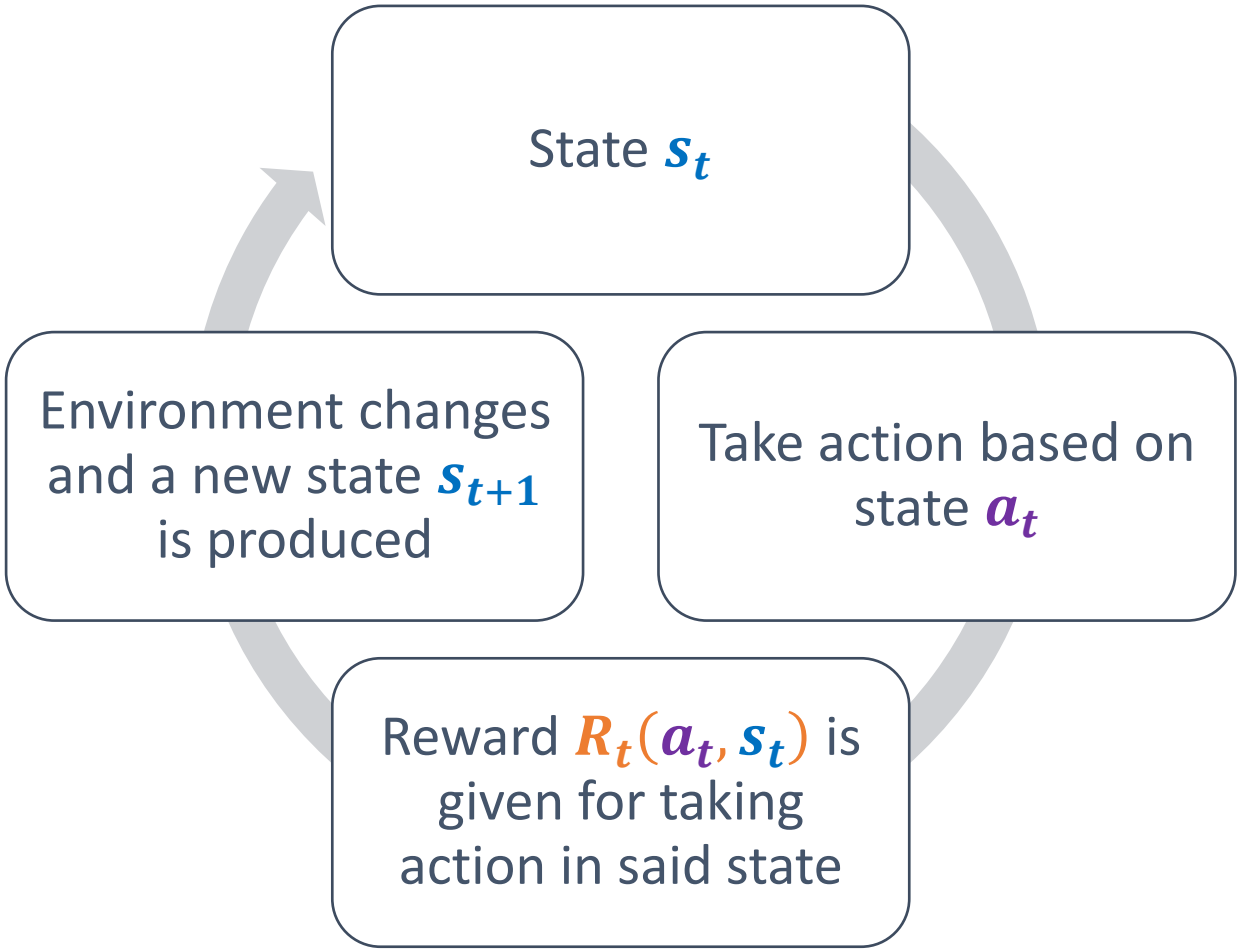
8. What is **Deep Q Learning**? And which problem does it address?
9. What is **RL with Human Feedback**?
10. How to implement RLHF via the **REINFORCE** algorithm?
11. What are **actor-critic** learning methods? And which problems do these approaches address?
12. What are more advanced problems in RL?
    - Markov states
    - Partially observable environment
    - SARSA
    - Non-stationary problems

# Reusing the RL formalism

**Definition (agent):**

In RL, we refer to the AI, as an **agent**. At each step, the **agent**:

- Looks at the current **state $s_t$**,
- Then takes an **action**, in this given state, $a_t$.
- The **action** has an effect, which is eventually measured in terms of **reward**, $R_t$.
- And a **new state $s_{t+1}$** is produced.



# Exploration and exploitation tradeoff

**Definition (exploration and exploitation):**

The phase during which, you try out things to acquire knowledge about the problem is called **exploration**.

The second phase, where you rely on your acquired knowledge, and play what you feel is the best move, is called **exploitation**.

**Definition (exploration vs. exploitation tradeoff):**

A good RL-based AI, needs to smartly combine **exploration** and **exploitation** phases.

- **Too much exploration?** You have wasted coins trying out bad machines.
- **No enough exploration?** You might end up choosing the wrong machine as the “best” one.

# A second toy problem

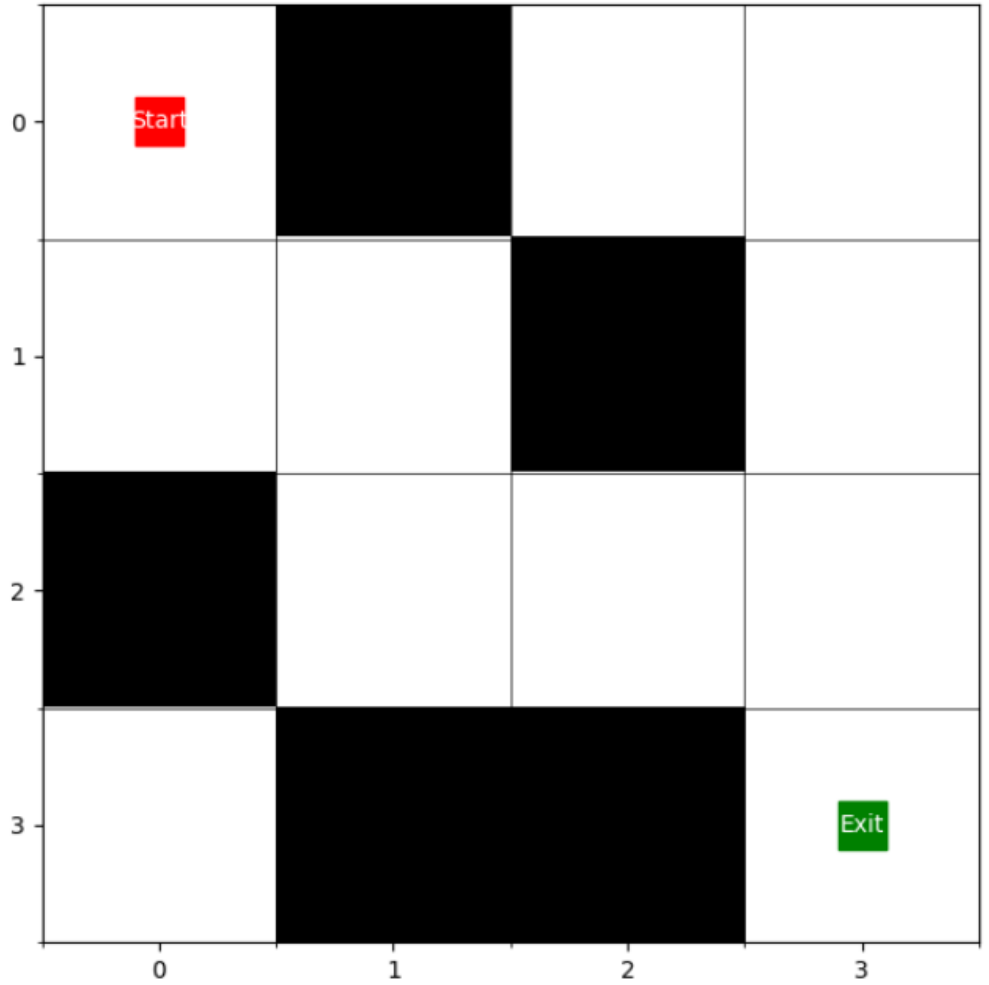
The Maze problem

# The maze problem

Let us consider the maze problem, described on the right.

The objective is

- to find the shortest path between the start and goal squares,
- without any knowledge of what the maze is beforehand.



# The maze problem

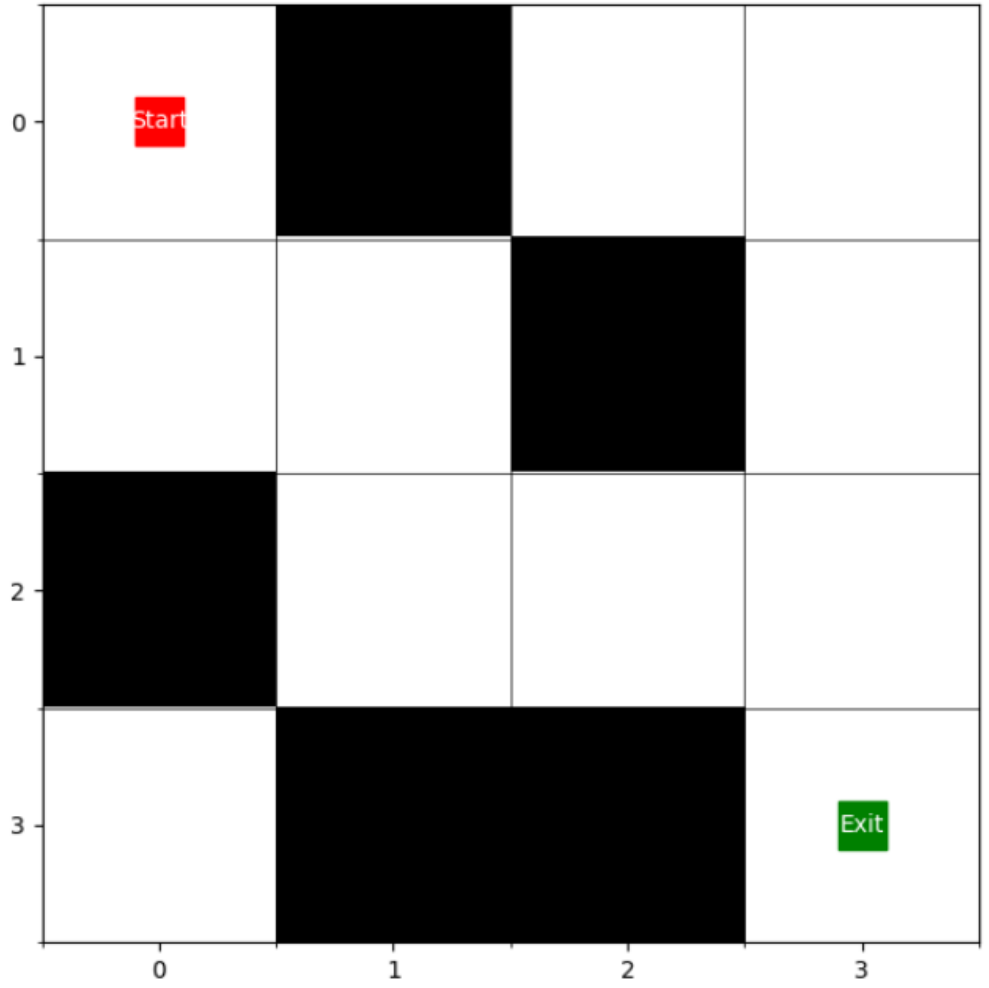
**Agent state:**

- Our agent will start at the starting position, with coordinates **(0,0)**.
  This is the initial **state $s_1$**.

**Action space:**

- In every position of the maze, four **actions** are possible

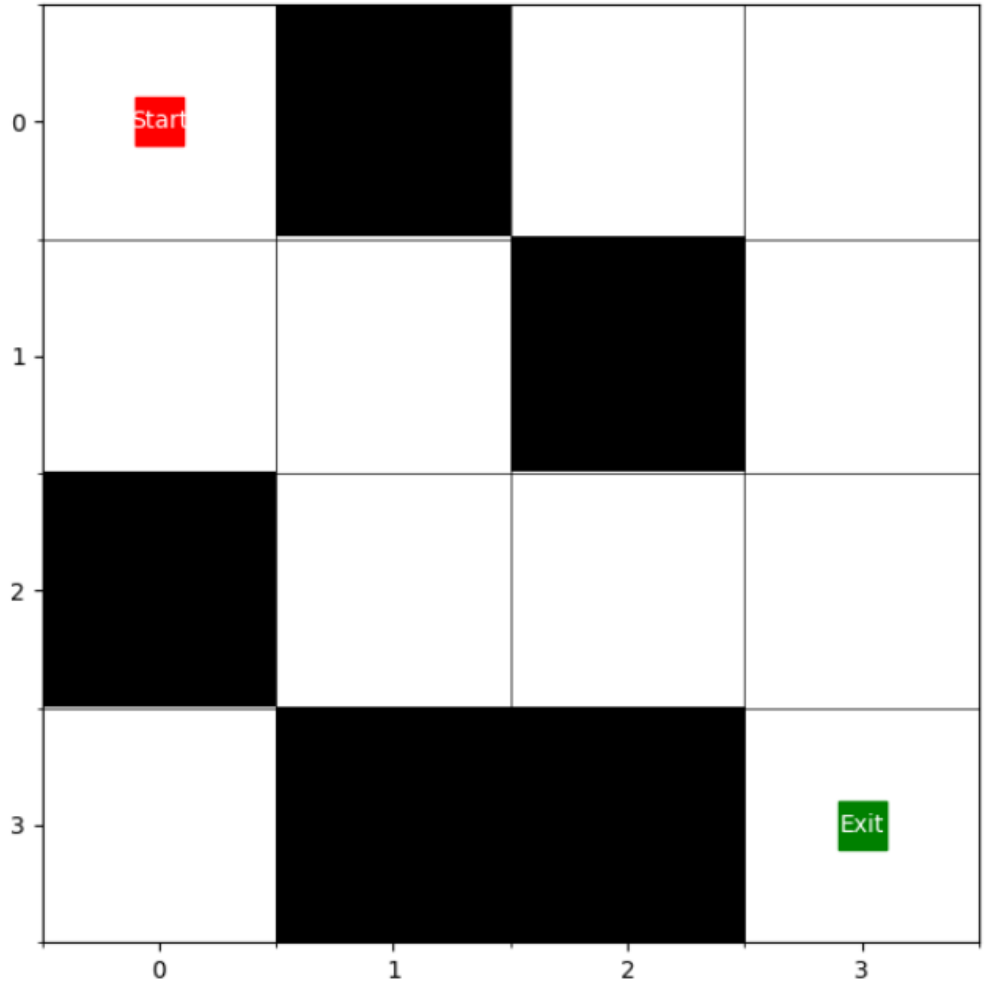
$$A = \{Go\ Up, Go\ Down, Go\ Left, Go\ Right\}$$



# The maze problem

**State update:**

- For a given **state $s_t$**, and an **action $a_t$**, the **state** will be updated following these rules.
- If the **action $a_t$** moves the user in a wall or out of the maze, the **new state $s_{t+1}$** will remain the same as the **previous state $s_t$**.

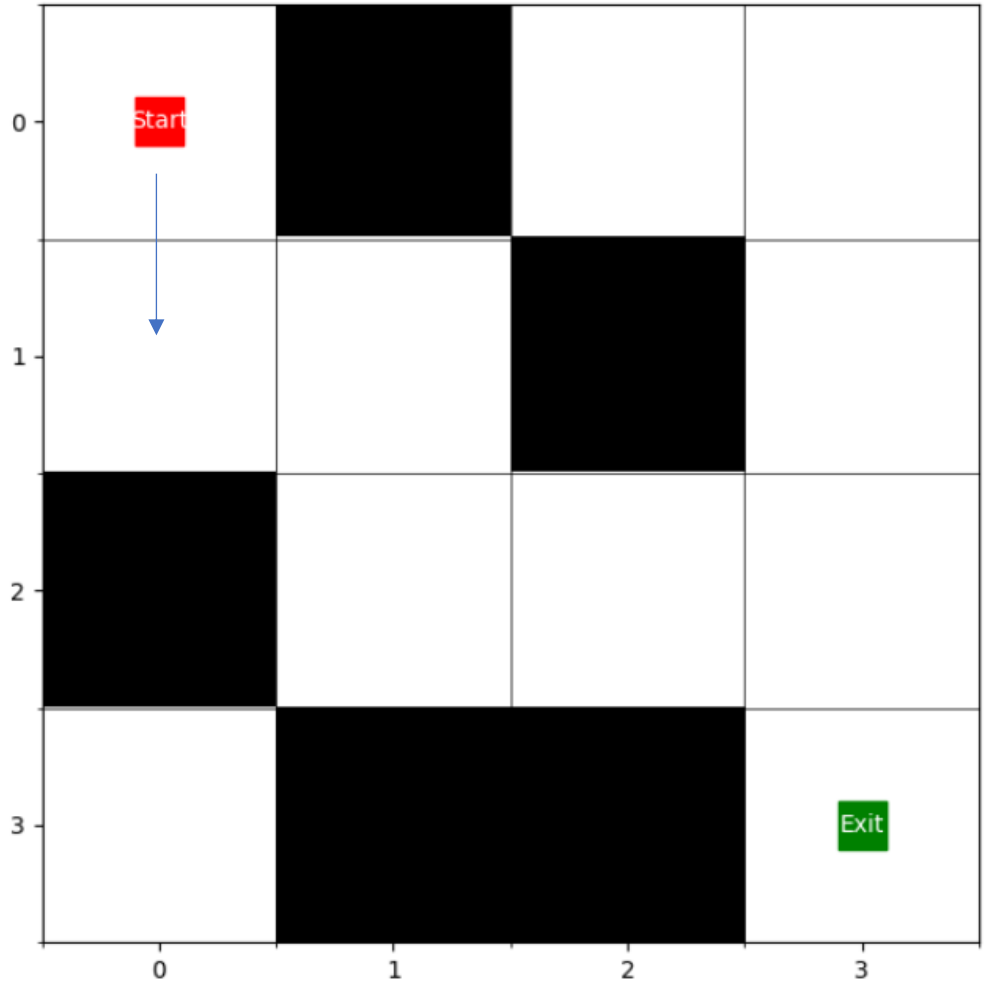


# The maze problem

**State update:**

- For a given **state $s_t$**, and an **action $a_t$**, the **state** will be updated following these rules.
- If the **action $a_t$** moves the user in a wall or out of the maze, the **new state $s_{t+1}$** will remain the same as the **previous state $s_t$**.
- Otherwise, we move the agent to the next square and this becomes $s_{t+1}$.

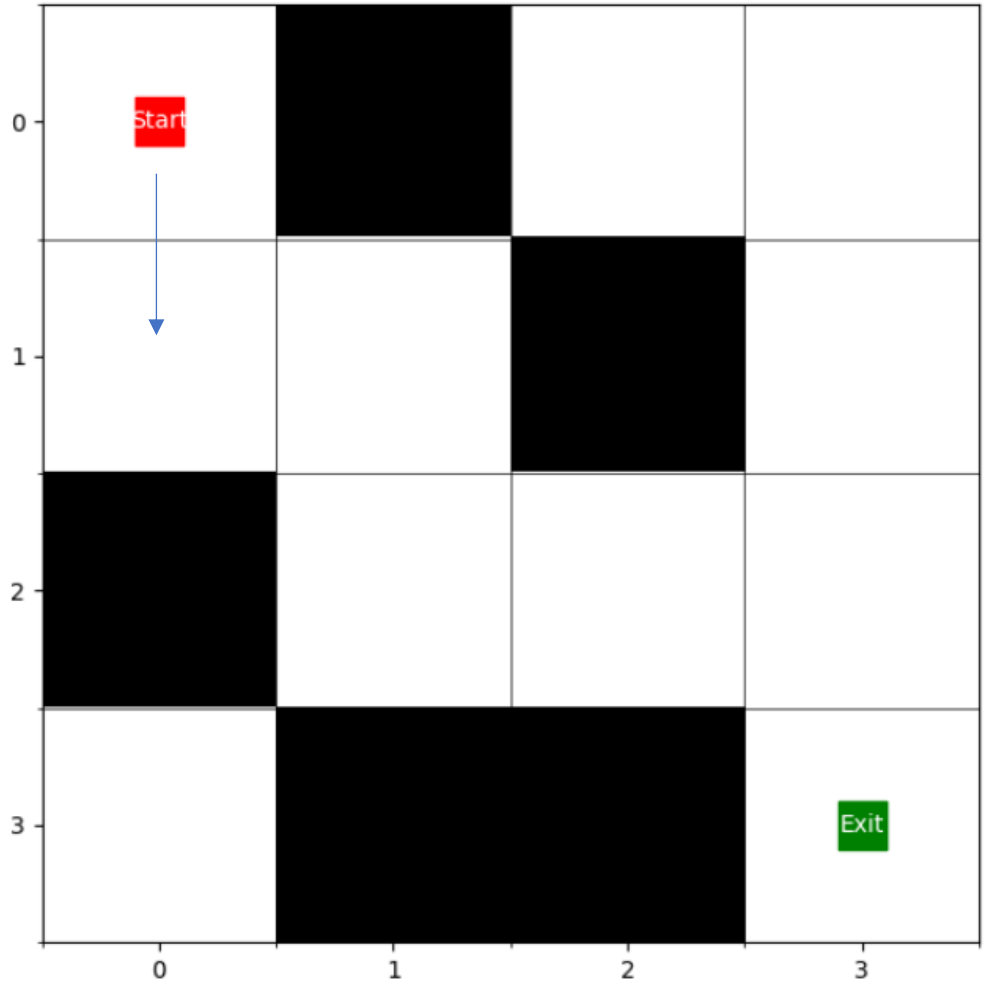
$$s_t = (0,0) + a_t = Down \rightarrow s_{t+1} = (0,1)$$



# The maze problem

**Reward value:**

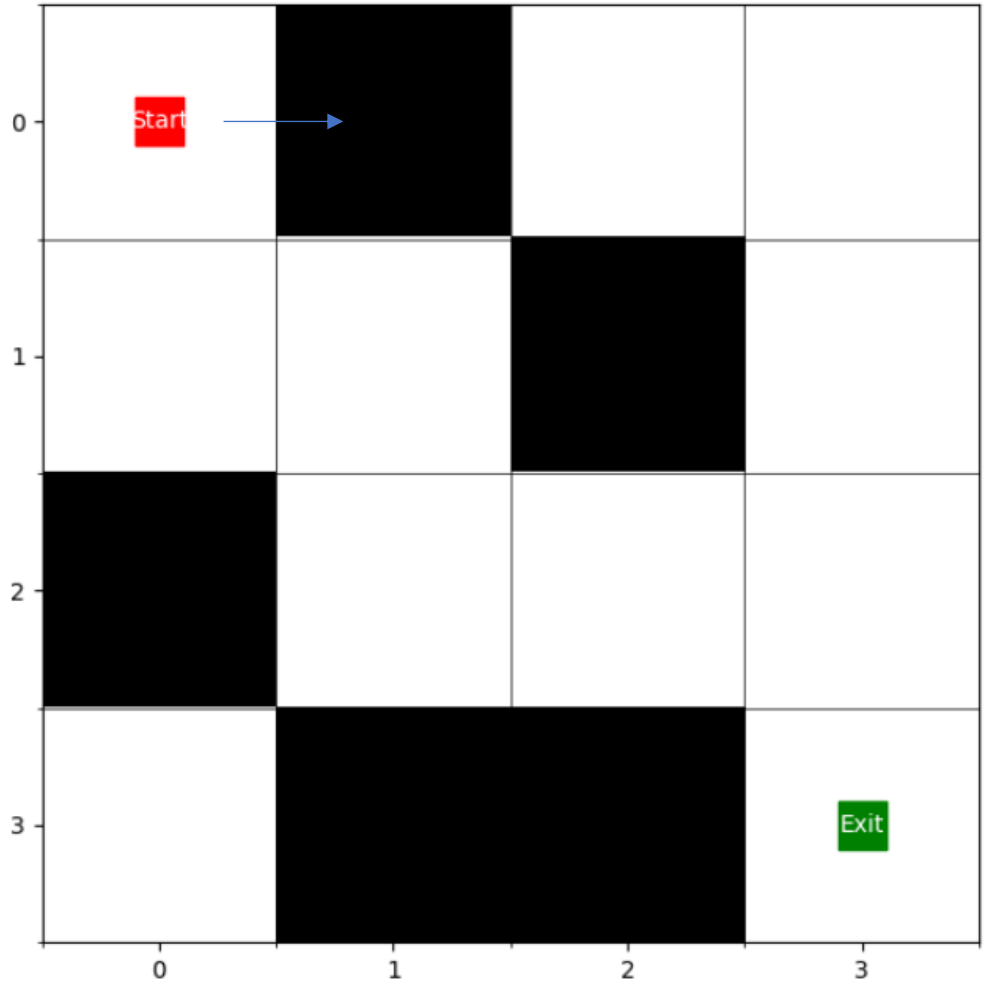
- For every **state** and every **action** of the agent, the **reward** $R_t$ is set to -0.05.
- Because of the negative reward applied on each action, the agent will try to minimize the number of actions taken.
- Or in other words, find the exit ASAP.



# The maze problem

**Reward value:**

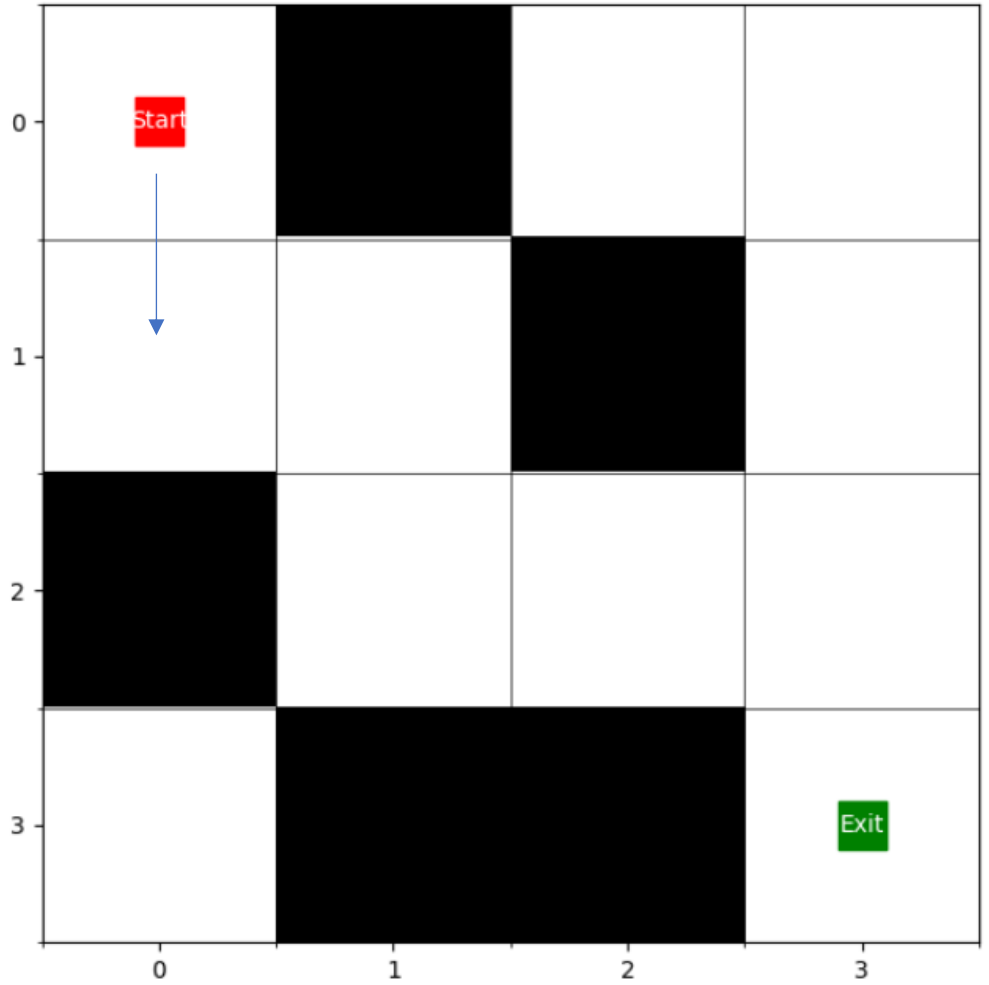
- In addition, the agent will be penalized by -0.75 if the **action** taken makes it bump into a wall.
- It will receive a negative penalty -0.25 if it revisits a cell it has already seen (i.e. comes back to a previously visited cell).
- These strong negative penalties will further encourage the agent to try to minimize the number of actions taken.



# The maze problem

**Reward value:**

- The game stops if the agent reaches the exit, that is **state (3,3)**. In this scenario, a large positive reward of +10 is given to encourage the agent to reach the exit.
- The game will also stop if the cumulated reward falls below a certain quantity (e.g. -10) to prevent the agent from aimlessly roaming in the maze for too long.



# The maze problem

**Reward value:**

- The **cumulated reward/gain** $\mathbf{G_t} = \sum_t \boldsymbol{R_t}$ will be calculated and updated after each **action** is taken.
- Maximizing $\mathbf{G_t} = \sum_t \boldsymbol{R_t}$ in our RL framework is then strictly equivalent to **finding the shortest path out of the maze**.
- This will be our **RL framework** for this task.

# The Maze environment

- Notebook 4 shows the implementation of the environment.
- Self-study the code and focus on the environment attributes and methods.

```python
class Maze:
    """
    A Maze environment where an agent must navigate from a start cell to an exit cell.
    """
    def __init__(self, maze, start_cell, exit_cell):
        # Initialize maze
        self.create_maze(maze, start_cell, exit_cell)
        # Define the RL parameters for agent (rewards, etc.)
        self.define_rl_params()
        # Reset
        self.reset(start_cell)

    def create_maze(self, maze, start_cell, exit_cell):
        # Maze definition
        self.maze = maze
        nrows, ncols = self.maze.shape
        self.cells = [(col, row) for col in range(ncols) for row in range(nrows)]
        self.empty = [(col, row) for col in range(ncols) for row in range(nrows) if self.maze[row, col] == Cell.EMPTY]
        self.exit_cell = exit_cell
        self.empty.remove(self.exit_cell)
        self.start_cell = start_cell

    def define_rl_params(self):
        # List of all 4 possible actions
        self.actions = [Action.MOVE_LEFT, Action.MOVE_RIGHT, Action.MOVE_UP, Action.MOVE_DOWN]
        # Reward for reaching the exit
        self.reward_exit = 10.0
        # Penalty applied everytime a move is taken (encourages the agent to find the exit ASAP)
        self.penalty_move = -0.05
        # Penalty for revisiting a cell (encourages the agent not to revisit cells it has already gone through)
        self.penalty_visited = -0.25
        # Penalty for invalid moves (encourages agent not to bump into walls)
        self.penalty_impossible_move = -0.75
        # Threshold to force game end, if cumulated reward falls below this value
        self.minimum_reward = -10

    def reset(self, start_cell):
        """
        Reset the maze to its initial state.
        """
        self.previous_cell = self.current_cell = start_cell
        self.total_reward = 0.0
        self.visited = set()
        return self.observe()
```

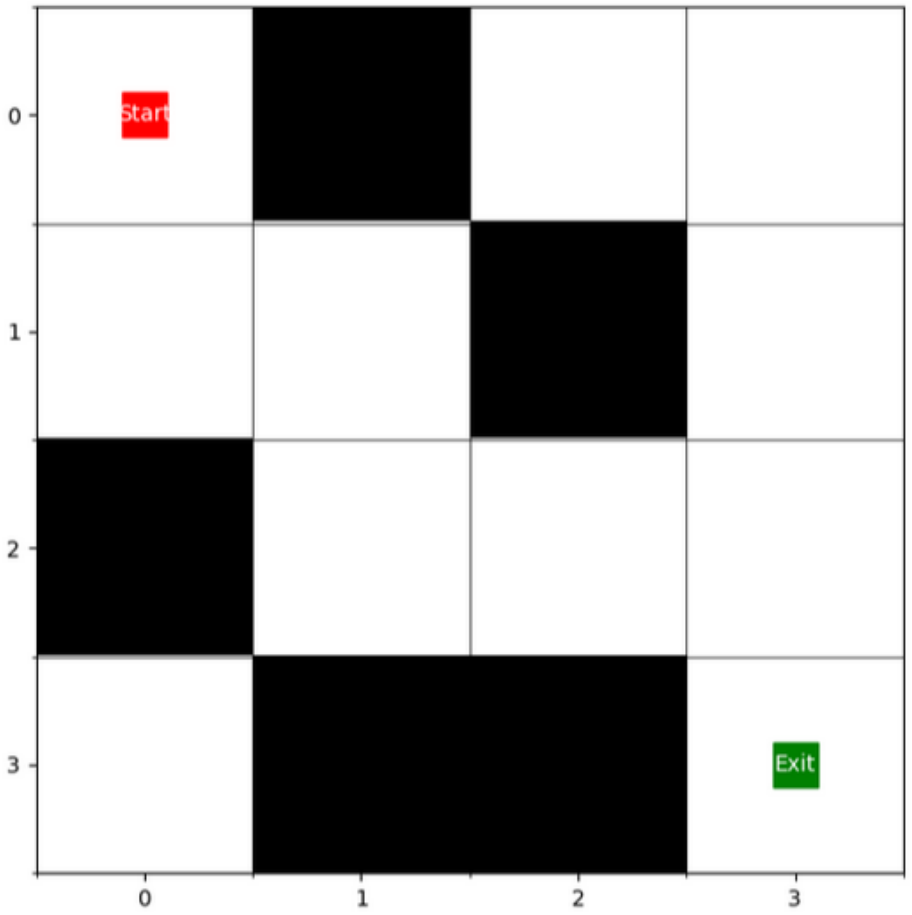
# The Maze environment

- Notebook 4 shows the implementation of the environment.
- Self-study the code and focus on the environment attributes and methods.

```
# Define a simple maze layout
maze_layout = np.array([[0, 1, 0, 0],
                        [0, 0, 1, 0],
                        [1, 0, 0, 0],
                        [0, 1, 1, 0]])
# Initialize the Maze environment
maze = Maze(maze = maze_layout, start_cell = (0, 0), exit_cell = (3, 3))
print("Maze initialized.\n")
# Reset the maze
state = maze.reset(start_cell = (0, 0))
print(f"Initial state: {state}\n")
# Display the maze
maze.draw_full_maze()
```

```
Maze initialized.

Initial state: [[0 0]]
```



# The Maze environment

- Notebook 4 shows the implementation of the environment.
- Self-study the code and focus on the environment attributes and methods.
- Pay special attention to the agent action methods (possible actions, observe, execute, visited, etc.)

```python
# Demonstrate taking a few actions
actions = [Action.MOVE_DOWN, Action.MOVE_DOWN, Action.MOVE_RIGHT, Action.MOVE_RIGHT, Action.MOVE_DOWN, \
           Action.MOVE_RIGHT, Action.MOVE_RIGHT, Action.MOVE_UP, Action.MOVE_DOWN, Action.MOVE_DOWN]

# Use the sequence of actions
print("Performing actions in the maze:\n")
for i, action in enumerate(actions):
    print("-----")
    print(f"Current state: {state}")
    print(f"Step {i + 1}: Executing action {action.name}")
    state, reward, status = maze.step(action)
    print(f"New state: {state}")
    print(f"Reward: {reward}")
    print(f"Status: {status.name}\n")

    if status != Status.PLAYING:
        print(f"Game ended with status: {status.name}")
        break
```

```
Performing actions in the maze:

-----
Current state: [[0 0]]
Step 1: Executing action MOVE_DOWN
New state: [[0 1]]
Reward: -0.05
Status: PLAYING

-----
Current state: [[0 1]]
Step 2: Executing action MOVE_DOWN
New state: [[0 1]]
Reward: -0.75
Status: PLAYING

-----
Current state: [[0 1]]
Step 3: Executing action MOVE_RIGHT
New state: [[1 1]]
Reward: -0.05
Status: PLAYING
```

# The Maze environment

- Notebook 4 shows the implementation of the environment.
- Self-study the code and focus on the environment attributes and methods.
- Pay special attention to the agent action methods (possible actions, observe, execute, visited, etc.)

```
# Show possible actions from a specific cell
specific_cell = (2, 2)
possible_actions = maze.possible_actions(cell = specific_cell)
act = [action.name for action in possible_actions]
print(f"Possible actions from cell {specific_cell}: {act}")

# Observe the current state
observation = maze.observe()
print(f"Current position for agent: {observation}")
```

```
Possible actions from cell (2, 2): ['MOVE_LEFT', 'MOVE_RIGHT']
Current position for agent: [[0 0]]
```

```
# Display visited cells
print("Agent has visited the following cells:", maze.visited)

# Display cumulated reward
print("Total reward for sequence of moves", round(maze.total_reward, 3))
```

```
Agent has visited the following cells: {(0, 1), (1, 2), (3, 1), (1, 1), (2, 2), (3, 2)}
Total reward for sequence of moves 7.95
```

# Value function V

**Definition (value function V):**

The **value function V** is a prediction/estimation of the future cumulated reward/gain, if in state $s_t$ at time $t$, for the policy $\pi$.

$$\forall t, \forall s_t, \qquad V_t^{\pi}(s_t) = E_{\pi}[R_t + R_{t+1} + R_{t+2} + \cdots | s_t]$$

In general, we like to add a parameter $\gamma \in [0,1]$, which gives more or less importance to the future or present rewards.

$$\forall t, \forall s_t, \qquad V_t^{\pi}(s_t) = E_{\pi}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \ldots | s_t]$$

# Value function V

**Definition (value function V):**

The **value function V** is a prediction/estimation of the future cumulated reward/gain, if in state $s_t$ at time $t$, for the policy $\pi$.

In general, we like to add a hyperparameter $\gamma \in [0,1]$, which decides on the importance to the future or present rewards.

$$\forall t, \forall s_t, \qquad V_t^{\pi}(s_t) = E_{\pi}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \dots | s_t]$$

This function is used to **evaluate the goodness/badness of a state $s_t$,** and **can help to identify the best action to use in a given state**.

# Optimization with value function V

The **value function V** can be used to rewrite our optimization problem, one step at a time.

Let us assume we are current at time $t$, in **state $s_t$**.

The best **action $a_t$** to use in this current state will simply maximize

- The **immediate reward** we will get after playing this action, that is $R_t$
- Plus, the **expected reward** we will get in the future, if we end up in a new **state $s_{t+1}$** after playing **action $a_t$**.

$$a_t = \arg\max_{a \in A} \left[ R_t(a, s_t) + V_{t+1}^{\pi}(s_{t+1}) \right]$$

# The state-action function Q

**Definition (state-action function Q):**

The **state-action function Q** is a function, which is used to quantify the goodness/badness of taking **action $a_t$** in **state $s_t$** at time $t$, according to our policy $\pi$.

It is closely related to the value function $V$, as it consists of our previous optimization function term, which was combining

- The **immediate reward** we will get after playing this action, that is $R_t$
- Plus, the **expected reward** we will get in the future, if we end up in a **new state $s_{t+1}$** after playing **action $a_t$**.

$$Q_t^{\pi}(a_t, s_t) = R_t(a_t, s_t) + V_{t+1}^{\pi}(s_{t+1})$$

# On the policy, V and Q functions relationship

**Theorem (Bellman principle of optimality):**

The policy $\pi$, $V$ and $Q$ functions are closely related. For instance, we have a clear relationship between $Q$ and $V$. The value of $Q$ can be used to compute the value of $V$ immediately, and vice-versa.

$$Q_t^{\pi}(a_t, s_t) = R_t(a, s_t) + V_{t+1}^{\pi}(s_{t+1})$$

The policy $\pi$, can then be derived from $Q$, as:

$$a_t = \pi_t(s_t) = \arg\max_{a \in A} Q_t^{\pi}(a, s_t)$$

These are called **Bellman's equations**.

# Defining a Q-table

- In our Maze problem, we have a finite number of **actions** and **states**.
- It is then simpler to write the $Q$ function as a table, which is then referred to as the ***Q*-table**.
- Later on, we will let our agent explore the maze and update the values of the $Q$-table.
- After training properly, looking at the $Q$-table gives us the best policy $\pi$ to use in any state.

Initialized

| Q-Table | | Actions | | | |
|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) |
| States | 0 | 0 | 0 | 0 | 0 |
| | . . . | . . . | . . . | . . . | . . . |
| | 327 | 0 | 0 | 0 | 0 |
| | . . . | . . . | . . . | . . . | . . . |
| | 499 | 0 | 0 | 0 | 0 |

Training

| Q-Table | | Actions | | | |
|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) |
| States | 0 | 0 | 0 | 0 | 0 |
| | . . . | . . . | . . . | . . . | . . . |
| | 328 | -2.30108105 | -1.97092096 | -2.30357004 | -2.20591839 |
| | . . . | . . . | . . . | . . . | . . . |
| | 499 | 9.96984239 | 4.02706992 | 12.96022777 | 29 |

# Updating the Q-table via Q-learning

- To update and learn the correct $Q$-table, we will let our agent play a few rounds of the game and "explore" the maze.
- At the end of each round, we obtain a finite **history of actions, states and rewards**, i.e. a sequence of values for run $k$

$$h_k = \{s_1, a_1, R_1, s_2, a_2, R_2, \dots\}$$

# Updating the Q-table via Q-learning

- This sequence can then be used, to update the Q-table, according to

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- In a sense, this formula is the "**equivalent**" of our **gradient descent update**, but in the case of reinforcement learning, and is commonly referred to as **Q-learning**.

# A bit of a mathematical explanation

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

- $\boldsymbol{Q(s_t, a_t)}$ **(Old Value):** This is the current estimate of the Q-value for the state-action pair $(s_t, a_t)$ according to our current understanding of the game that is stored in the Q-table.
- $\boldsymbol{r_t}$: The immediate reward received after taking action $a_t$ in state $s_t$.

# A bit of a mathematical explanation

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- $\boldsymbol{\gamma \max_a Q(s_{t+1}, a)}$ **(Estimate of Optimal Future Value):**
  This term estimates the best possible discounted future reward assuming the best action is taken at the next state $s_{t+1}$.

# A bit of a mathematical explanation

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

- $\boldsymbol{r_t + \gamma \max_a Q(s_{t+1}, a)}$ **(Temporal Difference Target):**
  This is the target for updating the Q-value. It consists of the immediate reward and the best possible future return.

# A bit of a mathematical explanation

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- $\boldsymbol{r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a)}$ **(Temporal Difference Error):**
  The difference between the estimated Q-value and the observed reward-based estimate (called TD error). If this difference is large, the update will be significant.

# A bit of a mathematical explanation

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big( \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- The learning rate $\alpha$ determines to what extent newly acquired information overrides old information in the $Q$-table.
- A value 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities). Use something in-between!

# A bit of a mathematical explanation

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big( \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- The discount factor $\gamma$ determines the importance of future rewards.
- A value 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, i.e. $R_t$ (in the update rule above).
- A value approaching 1 will make it strive for a long-term high reward.

# Updating the Q table via exploration

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big( \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

- This Q-learning update formula will eventually update the Q-values and makes them converge to their “real” values.
- It also implicitly assumes some initial values in the $Q$-table before the first update occurs. High initial values, also known as "optimistic initial conditions“ tend to encourage exploration, helping the agent learn through trial and error of actions it has never used before in all states.

# Training an agent

- Next, define a policy $\pi$, which will smoothly transition from exploration into exploitation, as in the random candy machines.
- On the first few rounds of the game, use lots of exploration moves in all states.
- Then, as our understanding of the game improves, policy $\pi$ should progressively transition from exploration to exploitation.

Initialized

| Q-Table | | Actions | | | |
|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) |
| States | 0 | 0 | 0 | 0 | 0 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 327 | 0 | 0 | 0 | 0 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 499 | 0 | 0 | 0 | 0 |

Training

| Q-Table | | Actions | | | |
|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) |
| States | 0 | 0 | 0 | 0 | 0 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 328 | -2.30108105 | -1.97092096 | -2.30357004 | -2.20591839 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 499 | 9.96984239 | 4.02706992 | 12.96022777 | 29 |

# Training an agent

- Upon seeing “convergence” on the values of the $Q$-table, we can then claim that the agent has been “properly” trained.
- From there, exploiting should then give the best strategy.

Initialized

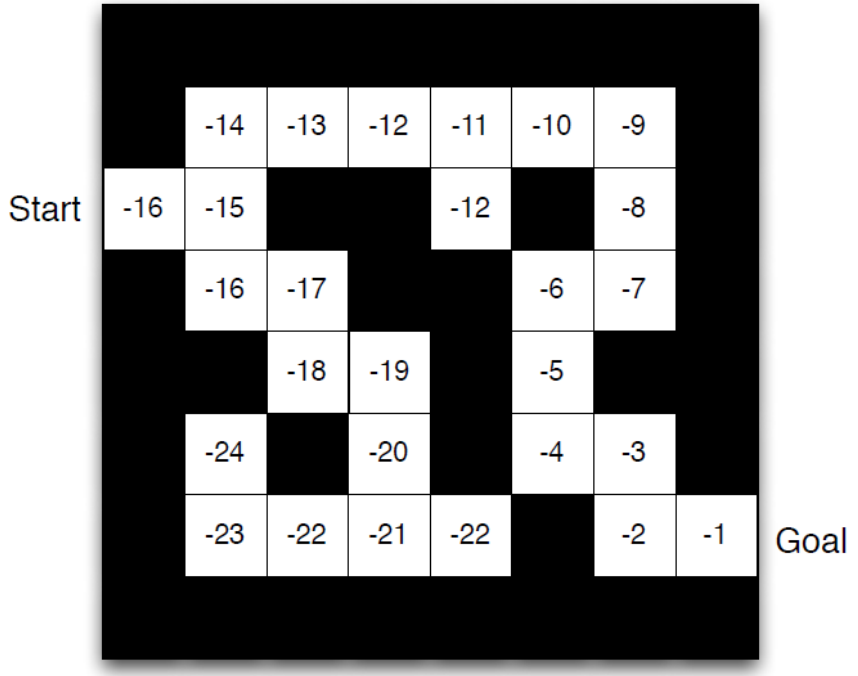
| Q-Table | | Actions | | | |
|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) |
| States | 0 | 0 | 0 | 0 | 0 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 327 | 0 | 0 | 0 | 0 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 499 | 0 | 0 | 0 | 0 |

Training

| Q-Table | | Actions | | | |
|---|---|---|---|---|---|
| | | South (0) | North (1) | East (2) | West (3) |
| States | 0 | 0 | 0 | 0 | 0 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 328 | -2.30108105 | -1.97092096 | -2.30357004 | -2.20591839 |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | 499 | 9.96984239 | 4.02706992 | 12.96022777 | 29 |

# Training an agent

- By successfully updating $V$ and $Q$ at the end of each round and playing the game for long enough, we will eventually obtain a good estimate of the "true" $V$ (or alternatively $Q$) values for our problem.
- That matches the maximal cumulated reward we could get for this game assuming we start from this current state.



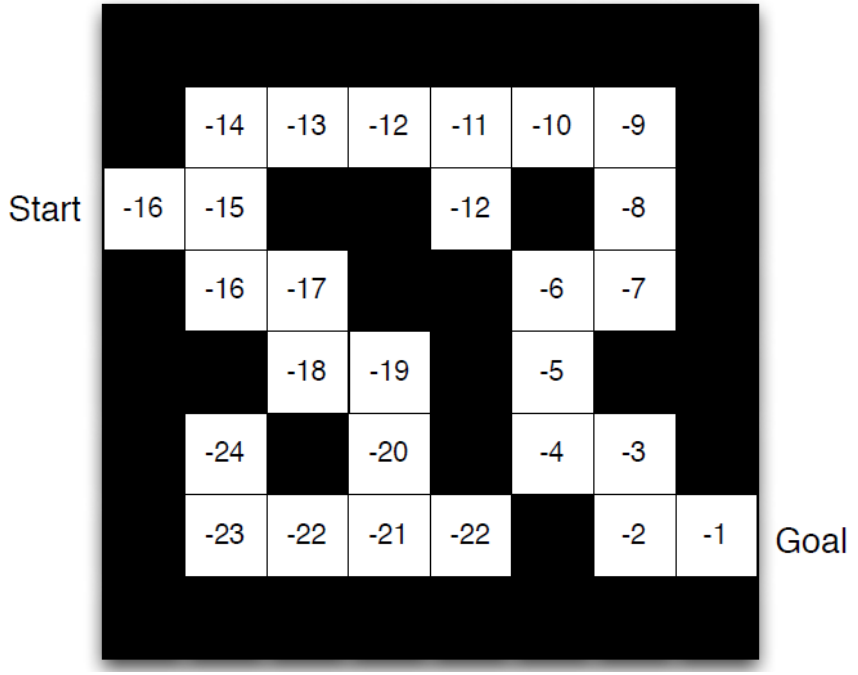

**Note:** the values above are arbitrary and used for illustrative purposes only, in practice they might be something else.

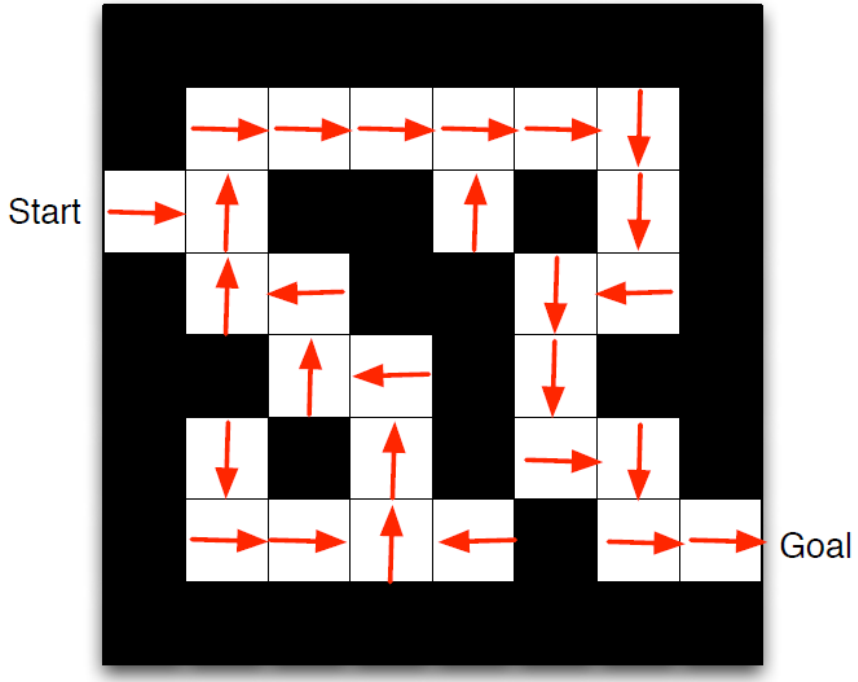
# Training an agent

- Reusing the $V$ (or $Q$) value allows to define the optimal policy $\pi$ to use in any given state.
- That is, the optimal direction in which we should go for any given square.

$$a_t = \pi_t(s_t) = \arg\max_{a \in A} Q_t^{\pi}(a, s_t)$$

$$Q_t^{\pi}(a_t, s_t) = R_t(a, s_t) + V_{t+1}^{\pi}(s_{t+1})$$



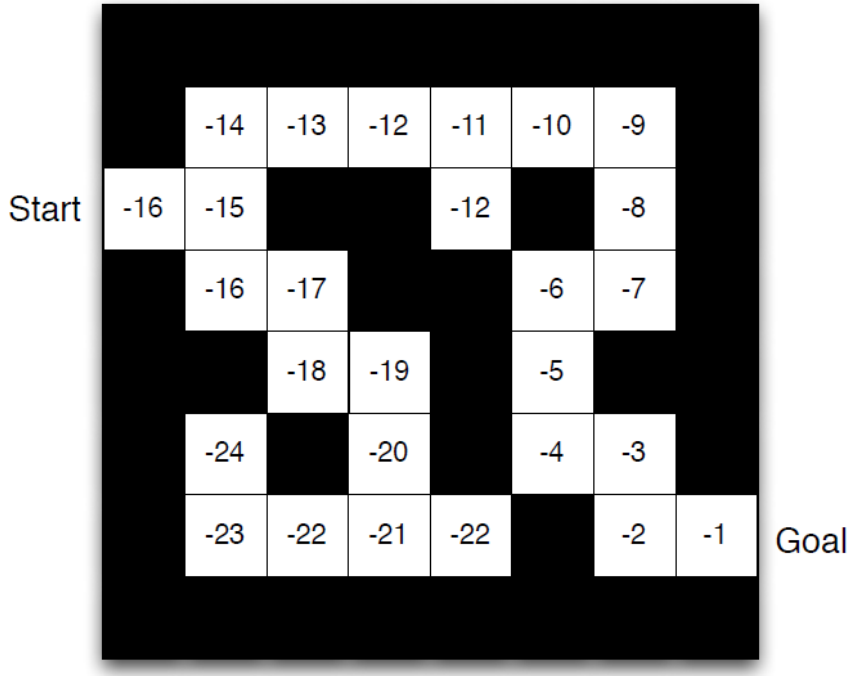
# Training an agent

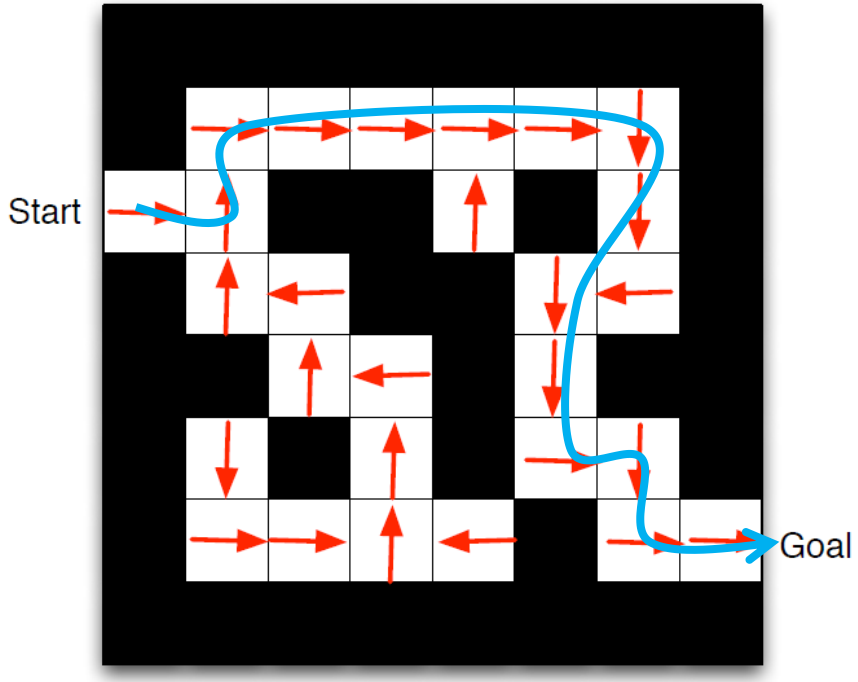
Later on, we can then replay the game, in full exploitation mode,

- by playing the best action according to our policy $\pi$ every time,
- without any exploration moves.

This gives us the shortest path.

Our agent has then learnt to recognize the maze and figured the shortest path through trial and error!



# Implementing a Q-table agent

Our QTableAgent class represents an AI agent that:

- Explores and exploits the environment.
- Uses a Q-table to store Q-values.
- Trains the Q-table using our Q-learning formula to learn the optimal Q-values and use them to find the optimal moves.
- Improves over time by balancing exploration and exploitation, with exploration decay.

```python
class QTableAgent:
    """
    Q-Learning agent using a Q-table.
    """
    def __init__(self, maze, learning_rate = 0.1, discount = 0.9, \
                 exploration_rate = 1.0, exploration_decay = 0.995):
        self.environment = maze
        self.lr = learning_rate
        self.gamma = discount
        self.epsilon = exploration_rate
        self.epsilon_decay = exploration_decay
        self.q_table = defaultdict(lambda: np.zeros(len(self.environment.actions)))

    def q(self, state):
        """
        Get Q-values for all actions for a given state.
        """
        return self.q_table[tuple(state.flatten())]

    def predict(self, state):
        """
        Policy: choose the action with the highest Q-value (exploitation).
        """
        return np.argmax(self.q(state))
```

# Implementing a Q-table agent

Our agent class has the attributes:

- **environment:** The Maze object that the agent interacts with.
- **learning_rate = 0.1:** the learning rate to use in the Q-learning formula.
- **discount = 0.95:** The discount factor ($\gamma$), controlling how much future rewards influence current decisions. A higher value (closer to 1) makes the agent focus more on long-term rewards.

```python
class QTableAgent:
    """
    Q-Learning agent using a Q-table.
    """
    def __init__(self, maze, learning_rate = 0.1, discount = 0.9, \
                 exploration_rate = 1.0, exploration_decay = 0.995):
        self.environment = maze
        self.lr = learning_rate
        self.gamma = discount
        self.epsilon = exploration_rate
        self.epsilon_decay = exploration_decay
        self.q_table = defaultdict(lambda: np.zeros(len(self.environment.actions)))

    def q(self, state):
        """
        Get Q-values for all actions for a given state.
        """
        return self.q_table[tuple(state.flatten())]

    def predict(self, state):
        """
        Policy: choose the action with the highest Q-value (exploitation).
        """
        return np.argmax(self.q(state))
```

# Implementing a Q-table agent

- **exploration_rate = 1.0:** The initial probability of taking random actions instead. Here, 1.0 means full exploration at the start.
- **exploration_decay = 0.995:** The rate at which exploration decreases over time. This ensures that the agent starts by exploring but gradually shifts to exploitation (following learned policies), by progressively shifting from 1 to 0 over the different episodes.

```python
class QTableAgent:
    """
    Q-Learning agent using a Q-table.
    """
    def __init__(self, maze, learning_rate = 0.1, discount = 0.9, \
                 exploration_rate = 1.0, exploration_decay = 0.995):
        self.environment = maze
        self.lr = learning_rate
        self.gamma = discount
        self.epsilon = exploration_rate
        self.epsilon_decay = exploration_decay
        self.q_table = defaultdict(lambda: np.zeros(len(self.environment.actions)))

    def q(self, state):
        """
        Get Q-values for all actions for a given state.
        """
        return self.q_table[tuple(state.flatten())]

    def predict(self, state):
        """
        Policy: choose the action with the highest Q-value (exploitation).
        """
        return np.argmax(self.q(state))
```

# Implementing a Q-table agent

Our agent has the following methods:

- **q():** Returns the Q-values for all possible actions in a given state. Uses the trained Q-table without modifying it.
- **predict():** Uses the Q-table to predict the best-known action (highest Q-value) in a given state.

```python
class QTableAgent:
    """
    Q-Learning agent using a Q-table.
    """
    def __init__(self, maze, learning_rate = 0.1, discount = 0.9, \
                 exploration_rate = 1.0, exploration_decay = 0.995):
        self.environment = maze
        self.lr = learning_rate
        self.gamma = discount
        self.epsilon = exploration_rate
        self.epsilon_decay = exploration_decay
        self.q_table = defaultdict(lambda: np.zeros(len(self.environment.actions)))

    def q(self, state):
        """
        Get Q-values for all actions for a given state.
        """
        return self.q_table[tuple(state.flatten())]

    def predict(self, state):
        """
        Policy: choose the action with the highest Q-value (exploitation).
        """
        return np.argmax(self.q(state))
```

# Trainer function

In the train() method, the training process consists of looping through multiple episodes, and on each episode, the game unfolds by following these steps:

- First, we start from a random valid position in the maze, not necessarily the start position (0, 0).
- In each state, we select actions using ε-greedy strategy. If an exploration move is selected, use a random action. If an exploitation move is selected, use the predict method to choose the best move to use given our current understanding of the game.
- After a move is selected, the environment changes.

```python
def train(self, episodes = 2000, stop_at_convergence = True):
    """
    Train the agent using Q-learning.
    """
    # Parameters for tracking training
    check_convergence_every = 5
    cumulative_reward_history = []
    win_history = []
    start_list = self.environment.empty.copy()

    for episode in range(1, episodes + 1):
        # Reset environment
        if not start_list:
            start_list = self.environment.empty.copy()
        start_cell = random.choice(start_list)
        start_list.remove(start_cell)
        state = self.environment.reset(start_cell)
        cumulative_reward = 0
        # Display
        if episode % 100 == 1 or episode == episodes:
            print(f"Episode: {episode} - Exploration rate: {self.epsilon}")
        while True:
            # Epsilon-greedy action selection
            if np.random.random() < self.epsilon:
                # Exploration move, randomly deciding
                action = random.choice(self.environment.actions)
            else:
                # Exploitation move, using the Q-table to decide on the best move
                action = self.predict(state)
            # Take action and observe result, update reward
            next_state, reward, status = self.environment.step(action)
            cumulative_reward += reward
            # Q-learning update rule
            best_next_q = np.max(self.q(next_state))
            target = reward + self.gamma * best_next_q if status == Status.PLAYING else reward
            self.q_table[tuple(state.flatten())][action] += self.lr * (target - self.q(state)[action])
            # Stop playing current episode if game has ended
            if status in (Status.WIN, Status.LOSE):
                break
            state = next_state
        # Record cumulative rewards
        cumulative_reward_history.append(cumulative_reward)
        # Check convergence
        if episode % check_convergence_every == 0:
            w_all, win_rate = self.environment.check_win_all(self)
            win_history.append((episode, win_rate))
            if w_all and stop_at_convergence:
                print("Won from all start cells. Stopped learning")
                break
        # Decay exploration rate
        self.epsilon *= self.epsilon_decay
    # Upon ending return all infos
    return cumulative_reward_history, win_history
```

# Trainer function

- After, we update Q-values using our Q-learning formula.
- While looping through multiple episodes, we track cumulative rewards and win rate over time.
- We also perform a decay on the exploration rate to progressively shift from exploring to exploiting.

```python
def train(self, episodes = 2000, stop_at_convergence = True):
    """
    Train the agent using Q-learning.
    """
    # Parameters for tracking training
    check_convergence_every = 5
    cumulative_reward_history = []
    win_history = []
    start_list = self.environment.empty.copy()

    for episode in range(1, episodes + 1):
        # Reset environment
        if not start_list:
            start_list = self.environment.empty.copy()
        start_cell = random.choice(start_list)
        start_list.remove(start_cell)
        state = self.environment.reset(start_cell)
        cumulative_reward = 0
        # Display
        if episode % 100 == 1 or episode == episodes:
            print(f"Episode: {episode} - Exploration rate: {self.epsilon}")
        while True:
            # Epsilon-greedy action selection
            if np.random.random() < self.epsilon:
                # Exploration move, randomly deciding
                action = random.choice(self.environment.actions)
            else:
                # Exploitation move, using the Q-table to decide on the best move
                action = self.predict(state)
            # Take action and observe result, update reward
            next_state, reward, status = self.environment.step(action)
            cumulative_reward += reward
            # Q-learning update rule
            best_next_q = np.max(self.q(next_state))
            target = reward + self.gamma * best_next_q if status == Status.PLAYING else reward
            self.q_table[tuple(state.flatten())][action] += self.lr * (target - self.q(state)[action])
            # Stop playing current episode if game has ended
            if status in (Status.WIN, Status.LOSE):
                break
            state = next_state
        # Record cumulative rewards
        cumulative_reward_history.append(cumulative_reward)
        # Check convergence
        if episode % check_convergence_every == 0:
            w_all, win_rate = self.environment.check_win_all(self)
            win_history.append((episode, win_rate))
            if w_all and stop_at_convergence:
                print("Won from all start cells. Stopped learning")
                break
        # Decay exploration rate
        self.epsilon *= self.epsilon_decay
    # Upon ending return all infos
    return cumulative_reward_history, win_history
```

Restricted

# Trainer function

- The training loops runs for a given number of episodes.
- Game stops upon reaching the exit, or the lowest penalty acceptable (timing out after too many moves taken)
- Training might stop early if the agent is able to win from all cells (early stopping?)

```python
def train(self, episodes = 2000, stop_at_convergence = True):
    """
    Train the agent using Q-learning.
    """
    # Parameters for tracking training
    check_convergence_every = 5
    cumulative_reward_history = []
    win_history = []
    start_list = self.environment.empty.copy()

    for episode in range(1, episodes + 1):
        # Reset environment
        if not start_list:
            start_list = self.environment.empty.copy()
        start_cell = random.choice(start_list)
        start_list.remove(start_cell)
        state = self.environment.reset(start_cell)
        cumulative_reward = 0
        # Display
        if episode % 100 == 1 or episode == episodes:
            print(f"Episode: {episode} - Exploration rate: {self.epsilon}")
        while True:
            # Epsilon-greedy action selection
            if np.random.random() < self.epsilon:
                # Exploration move, randomly deciding
                action = random.choice(self.environment.actions)
            else:
                # Exploitation move, using the Q-table to decide on the best move
                action = self.predict(state)
            # Take action and observe result, update reward
            next_state, reward, status = self.environment.step(action)
            cumulative_reward += reward
            # Q-learning update rule
            best_next_q = np.max(self.q(next_state))
            target = reward + self.gamma * best_next_q if status == Status.PLAYING else reward
            self.q_table[tuple(state.flatten())][action] += self.lr * (target - self.q(state)[action])
            # Stop playing current episode if game has ended
            if status in (Status.WIN, Status.LOSE):
                break
            state = next_state
        # Record cumulative rewards
        cumulative_reward_history.append(cumulative_reward)
        # Check convergence
        if episode % check_convergence_every == 0:
            w_all, win_rate = self.environment.check_win_all(self)
            win_history.append((episode, win_rate))
            if w_all and stop_at_convergence:
                print("Won from all start cells. Stopped learning")
                break
        # Decay exploration rate
        self.epsilon *= self.epsilon_decay
    # Upon ending return all infos
    return cumulative_reward_history, win_history
```

# Trainer function

```
# Create agent model
model = QTableAgent(maze, learning_rate = 0.1, discount = 0.9, exploration_rate = 1.0, exploration_decay = 0.99)
```

Using this RL framework, we are hoping that our Q-learning model will eventually learn the true action-value function $Q(s, a)$.

These are estimates of the future rewards and can be used to decide on the optimal action to use in any state of the maze.

```
# Train agent
h, w = model.train(episodes = 200, stop_at_convergence = True)
```

```
Episode: 1 - Exploration rate: 1.0
Episode: 11 - Exploration rate: 0.9043820750088043
Episode: 21 - Exploration rate: 0.8179069375972307
Episode: 31 - Exploration rate: 0.7397003733882802
Episode: 41 - Exploration rate: 0.6689717585696803
Episode: 51 - Exploration rate: 0.6050060671375365
Episode: 61 - Exploration rate: 0.5471566423907612
Episode: 71 - Exploration rate: 0.49483865960020695
Episode: 81 - Exploration rate: 0.44752321376381066
Episode: 91 - Exploration rate: 0.4047319726783239
Episode: 101 - Exploration rate: 0.36603234127322926
Episode: 111 - Exploration rate: 0.33103308832101386
Episode: 121 - Exploration rate: 0.29938039131233124
Episode: 131 - Exploration rate: 0.270754259511994
Episode: 141 - Exploration rate: 0.24486529903492946
Episode: 151 - Exploration rate: 0.22145178723886094
Episode: 161 - Exploration rate: 0.20027702685748935
Episode: 171 - Exploration rate: 0.18112695312597027
Episode: 181 - Exploration rate: 0.16380796970808745
Episode: 191 - Exploration rate: 0.1481449915475795
Episode: 200 - Exploration rate: 0.13533300490703207
```

# Trainer function

```
# Create agent model
model = QTableAgent(maze, learning_rate = 0.1, discount = 0.9, exploration_rate = 1.0, exploration_decay = 0.99)
```

Upon running the train() method:

- Notice how the exploration rate progressively shifts from 1 to 0.

```
# Train agent
h, w = model.train(episodes = 200, stop_at_convergence = True)
```

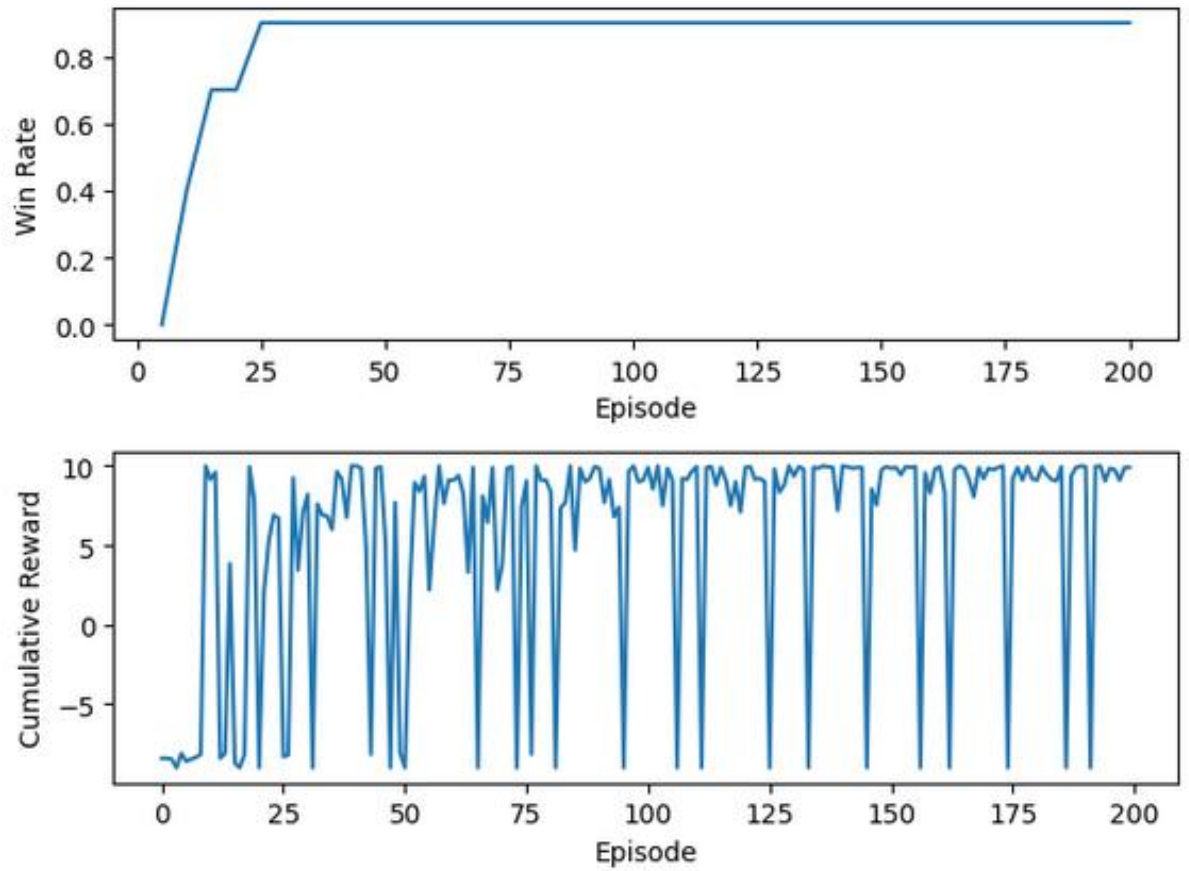
```
Episode: 1 - Exploration rate: 1.0
Episode: 11 - Exploration rate: 0.9043820750088043
Episode: 21 - Exploration rate: 0.8179069375972307
Episode: 31 - Exploration rate: 0.7397003733882802
Episode: 41 - Exploration rate: 0.6689717585696803
Episode: 51 - Exploration rate: 0.6050060671375365
Episode: 61 - Exploration rate: 0.5471566423907612
Episode: 71 - Exploration rate: 0.49483865960020695
Episode: 81 - Exploration rate: 0.44752321376381066
Episode: 91 - Exploration rate: 0.4047319726783239
Episode: 101 - Exploration rate: 0.36603234127322926
Episode: 111 - Exploration rate: 0.33103308832101386
Episode: 121 - Exploration rate: 0.29938039131233124
Episode: 131 - Exploration rate: 0.270754259511994
Episode: 141 - Exploration rate: 0.24486529903492946
Episode: 151 - Exploration rate: 0.22145178723886094
Episode: 161 - Exploration rate: 0.20027702685748935
Episode: 171 - Exploration rate: 0.18112695312597027
Episode: 181 - Exploration rate: 0.16380796970808745
Episode: 191 - Exploration rate: 0.1481449915475795
Episode: 200 - Exploration rate: 0.13533300490703207
```

# Trainer function

Upon running the train() method:

- Notice how the exploration rate progressively shifts from 1 to 0.
- Notice also how the average rewards are increasing (indicating that the agent is learning to improve at playing the game).
- Our win rate is also increasing and eventually plateauing at almost 100%.

```
# Display training curves
fig, (ax1, ax2) = plt.subplots(2, 1, tight_layout = True)
ax1.plot(*zip(*w))
ax1.set_xlabel("Episode")
ax1.set_ylabel("Win Rate")
ax2.plot(h)
ax2.set_xlabel("Episode")
ax2.set_ylabel("Cumulative Reward")
plt.show()
```

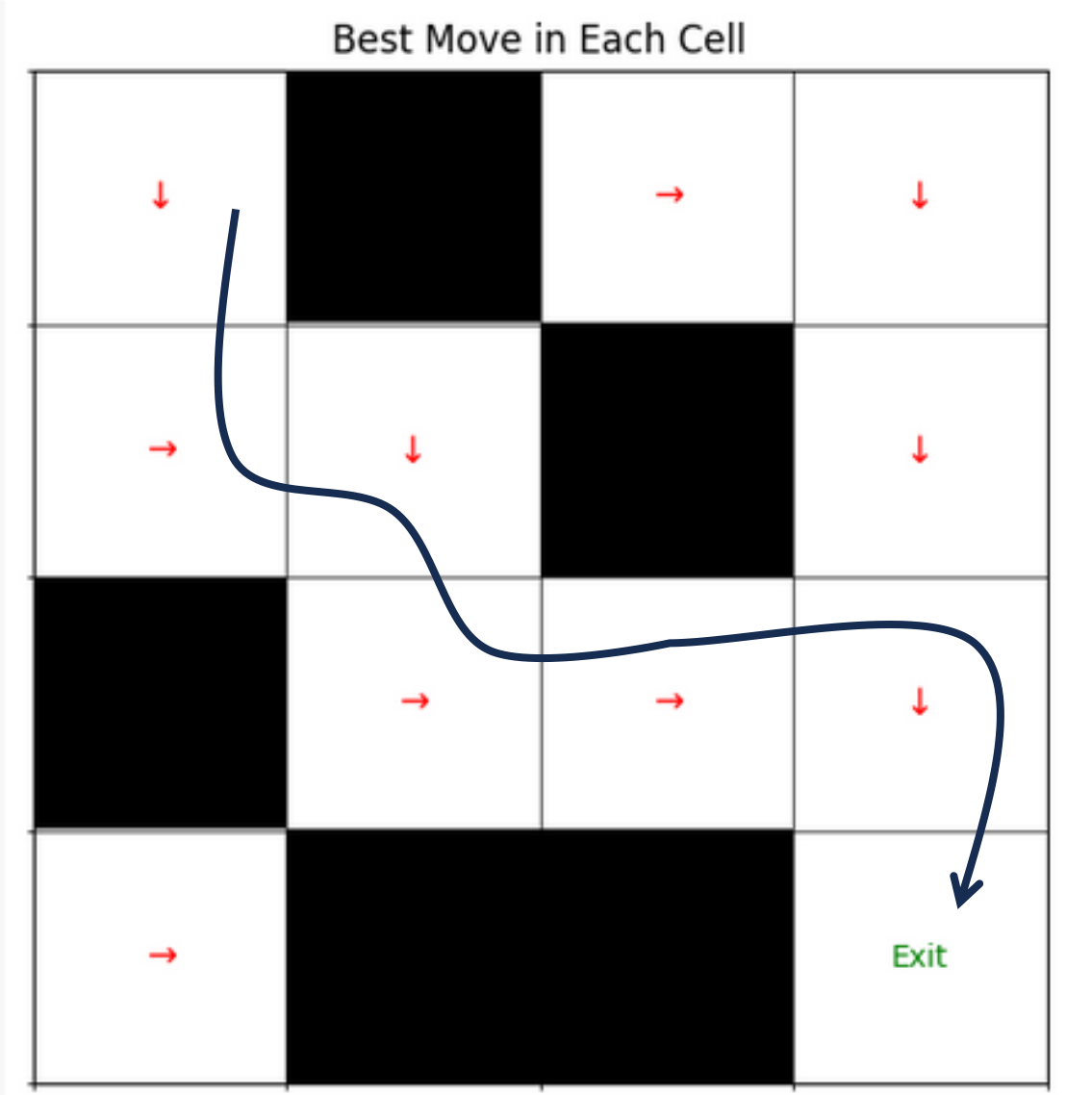


# Trainer function

Finally, we can visualize the entire maze and the best move to use in each walkable cell.

When starting in cell (0, 0), the agent is clearly able to identify the shortest path to exit the maze without hitting any walls!

```
# Visualize best moves in each cell after training
maze.visualize_best_moves(model)
```

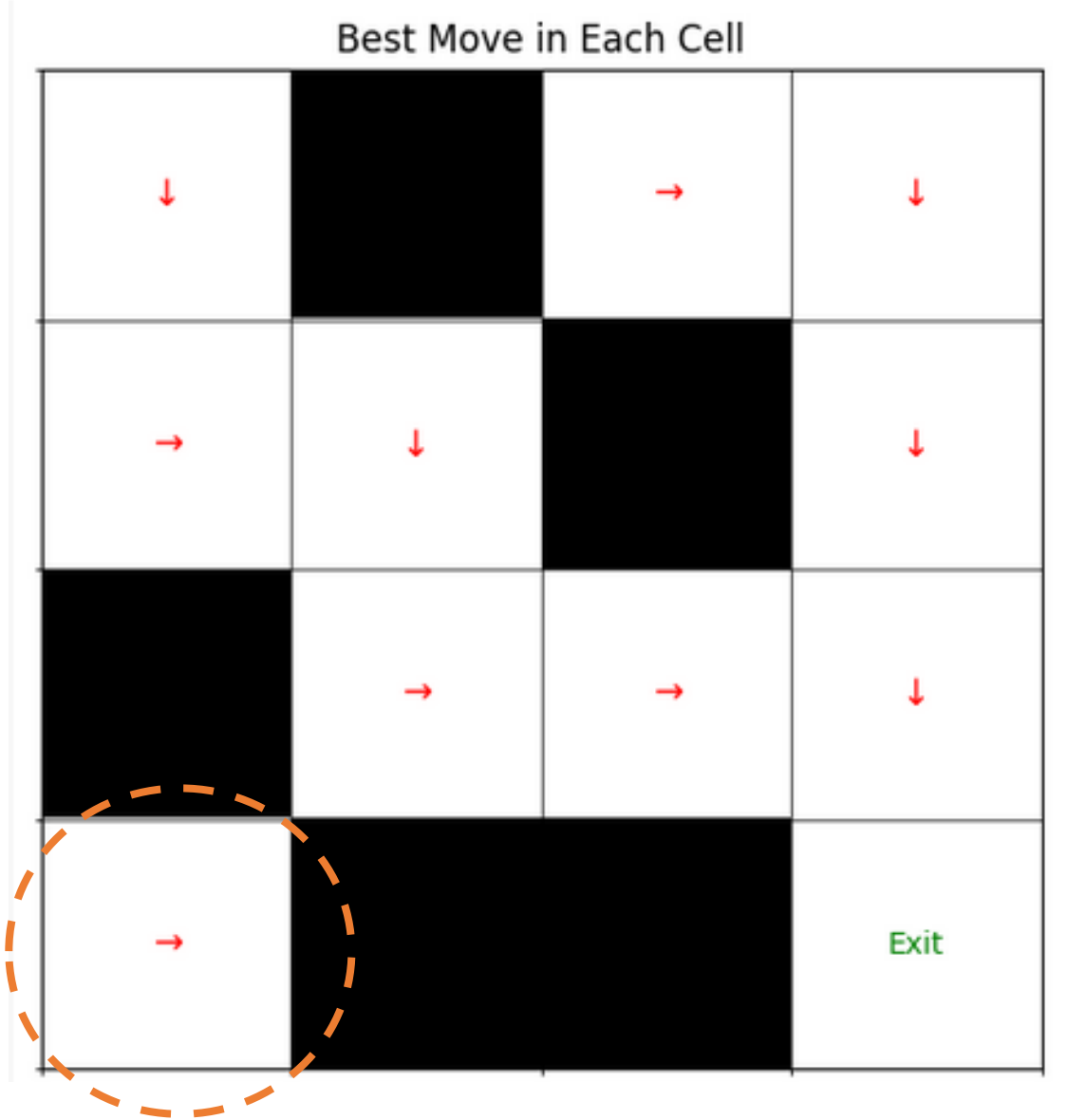


# Trainer function

There are however cells that are still problematic... For instance, cell (0, 3) suggests to go left, which is wrong…

But not the end of the world given that this cell is impossible to reach from (0, 0).

```
# Visualize best moves in each cell after training
maze.visualize_best_moves(model)
```

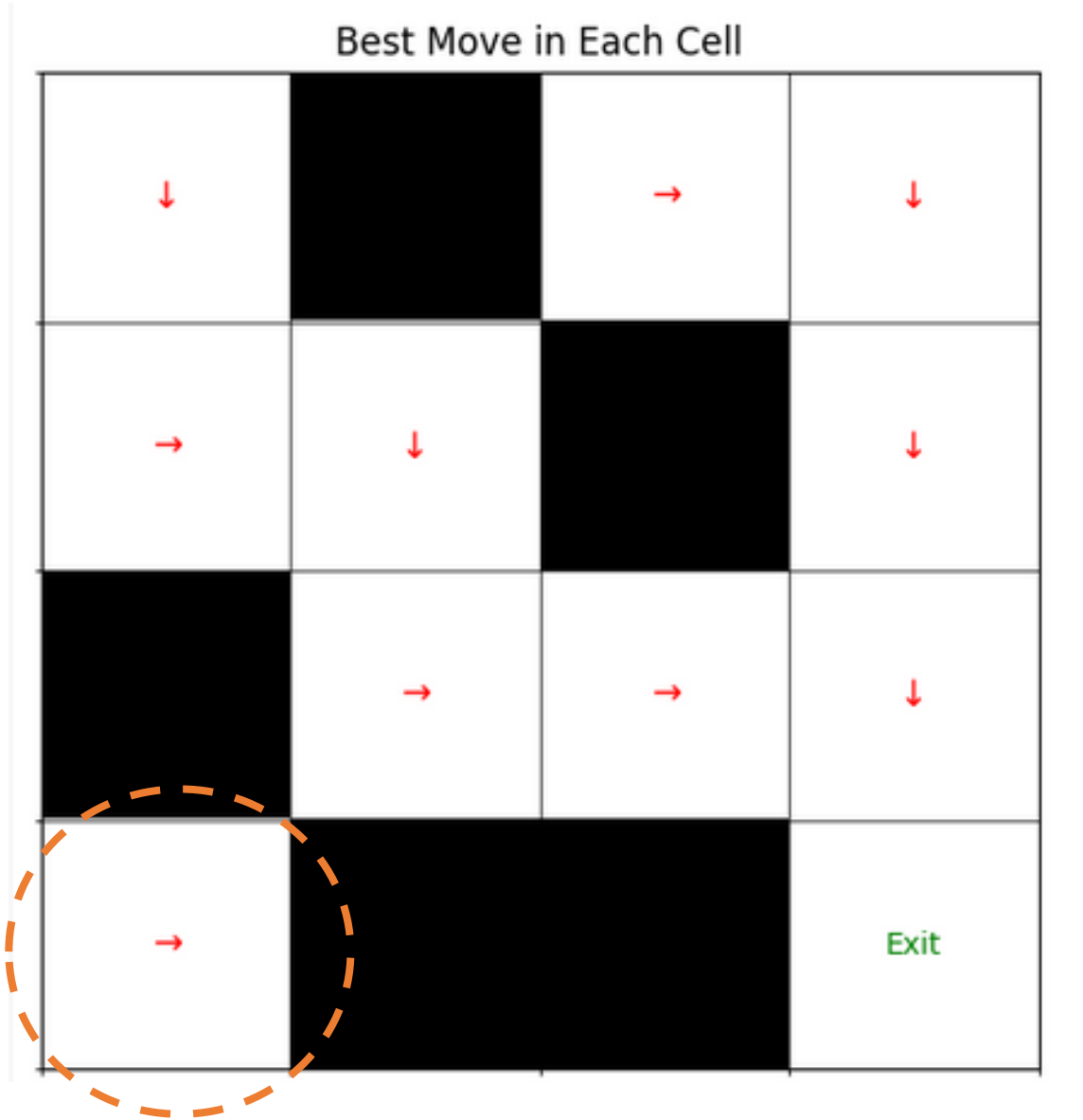


# Trainer function

While this suggests that the Q-Learning agent has yet more to learn about the maze and the best moves to use, this agent is, however, capable of learning the shortest path in a maze!

**This Q-learning approach can be used in any RL problem with a finite number of states, a finite number of actions and a clearly established reward system.**

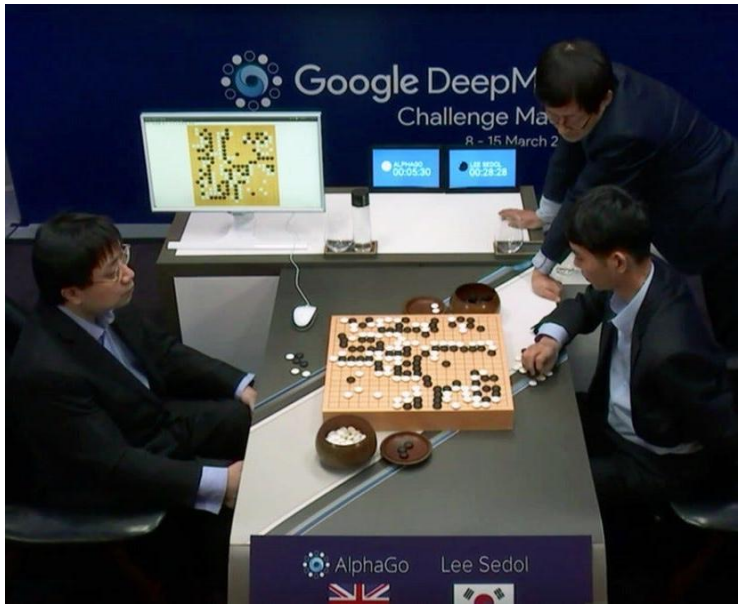
```
# Visualize best moves in each cell after training
maze.visualize_best_moves(model)
```

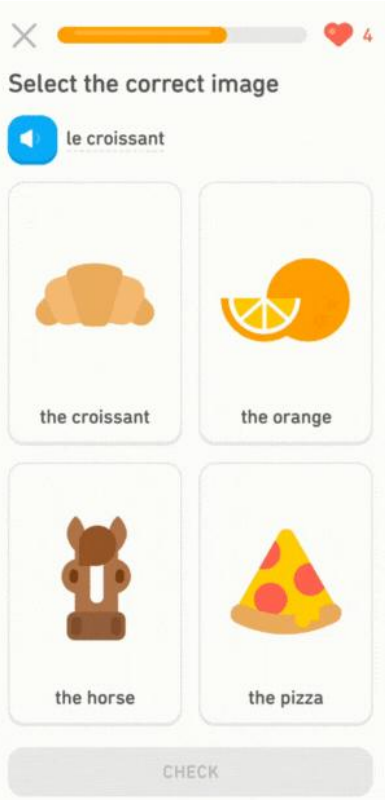


# Possible Applications of Q-learning

Q-tables and Q-learning can be used if the RL framework has a finite number of states and actions.

- Warehouse robotics navigation, using tiles drawn on the floor.
- Simple board games (maybe Go should not be considered a “simple” board game?)
- Quiz-based learning agents
  - States: Learner’s knowledge level (discretized).
  - Actions: Select next question or hint.
  - Goal: Maximize engagement and learning.



# Limitations of Q-learning

**Open question: What if the number of states and/or the number of actions is not finite?**

- We would not be able to use a Q-table to store values…

→ Idea: Replace the table with a Deep Learning model that predicts the value of Q for the given state and each possible action!

- Train that model to match the theoretical Q-value!
- Next lecture?

# Learn more about these topics

Out of class, for those of you who are curious

- [TheBibleOfRL] R. **Sutton** et al., “Reinforcement learning: An Introduction, 2nd edition”, 2018. http://www.incompleteideas.net/book/RLbook2020.pdf
- [Mnih2018] **Mnih** et al., “Playing Atari with Deep Reinforcement Learning”, 2018. https://arxiv.org/abs/1312.5602

# Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Richard Sutton: Professor** at **University of Alberta,** also **DeepMind**. Co-author of the Bible of RL (possibly most influential professor in the field of RL).
  https://scholar.google.ca/citations?user=6m4wv6gAAAAJ&hl=en
  https://www.ualberta.ca/admissions-programs/online-courses/reinforcement-learning/index.html
  http://www.incompleteideas.net/book/RLbook2020.pdf
- **Andrew Barto: Professor** at **University of Massachussets**. Co-author of the Bible of RL.
  https://people.cs.umass.edu/~barto/
  https://scholar.google.com/citations?user=CMIgrCgAAAAJ&hl=en