

# Thirteenth Session

**Alireza Moradi**

---



| [Linkedin](#)



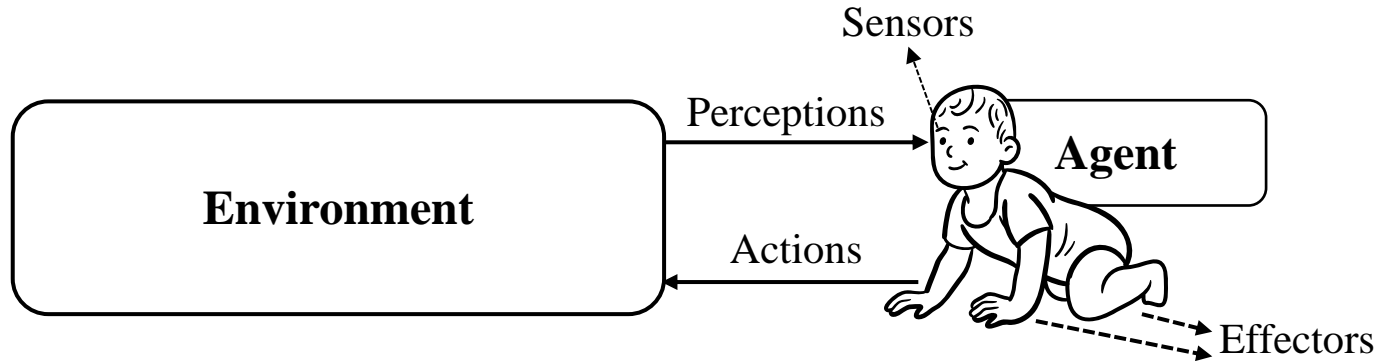
| [k](#)

# Reinforcement learning

---

# Reinforcement learning (RL)

- We saw that with **supervised learning**, an agent learns by **passively** observing example input/output pairs provided by a “teacher.”
- We will see how **agents can actively learn from their own experience**, without a teacher, by considering their own ultimate success or failure.



- In **RL**, the **training data is continuously gathered through interaction with the environment**.

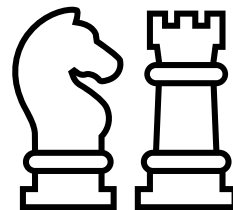
# Reinforcement learning (RL)

- ✓ Why is **Reinforcement Learning (RL)** necessary? Can't **Supervised Learning** adequately address the same tasks?
- Example: Chess

We have available databases of several million grandmaster games, each a sequence of positions and moves.

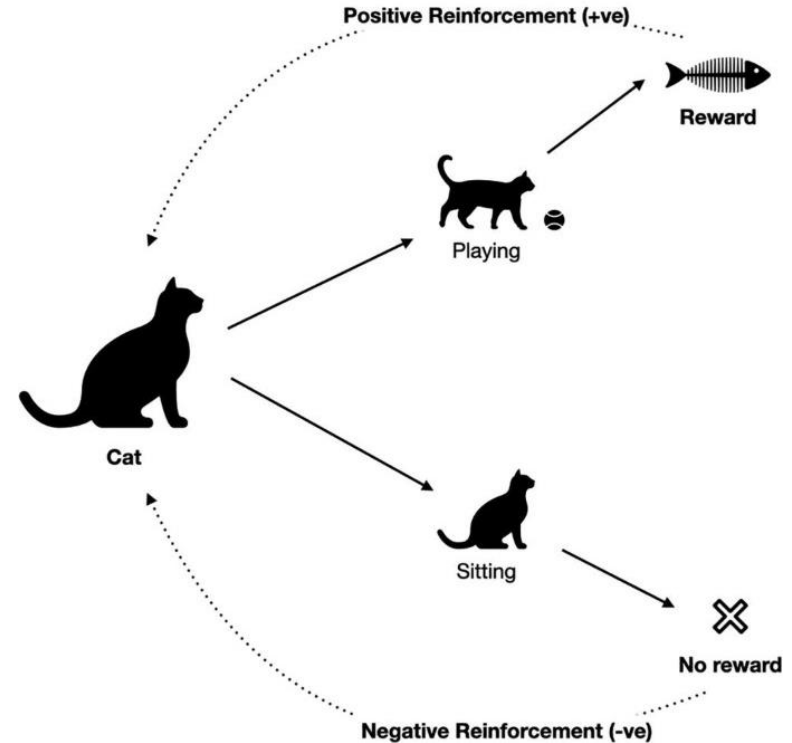
The problem is that there are **relatively few examples** (about  $10^8$ ) compared to the space of all possible chess positions (about  $10^{40}$ ).

In a new game, one soon encounters positions that are significantly different from those in the database.



# Reinforcement learning

- **RL** mimics the **trial-and-error learning** to achieve their goals. Software actions that work towards your goal are reinforced, while actions that detract from the goal are ignored.
- **RL algorithms** use a **reward-and-punishment** paradigm as they process data.
- They learn from the feedback of each action and **self-discover the best processing paths to achieve final outcomes.**



# Reinforcement learning

- We can summarize **reinforcement learning** as building an algorithm (or an AI agent) that **learns directly from its interaction with an environment**.
- The **environment** may be the real world, a computer game, a simulation or even a board game.
- ✓ How can we define reward and punishment here?
- The goal of the agent is **to maximize its cumulative reward**, called **return**.



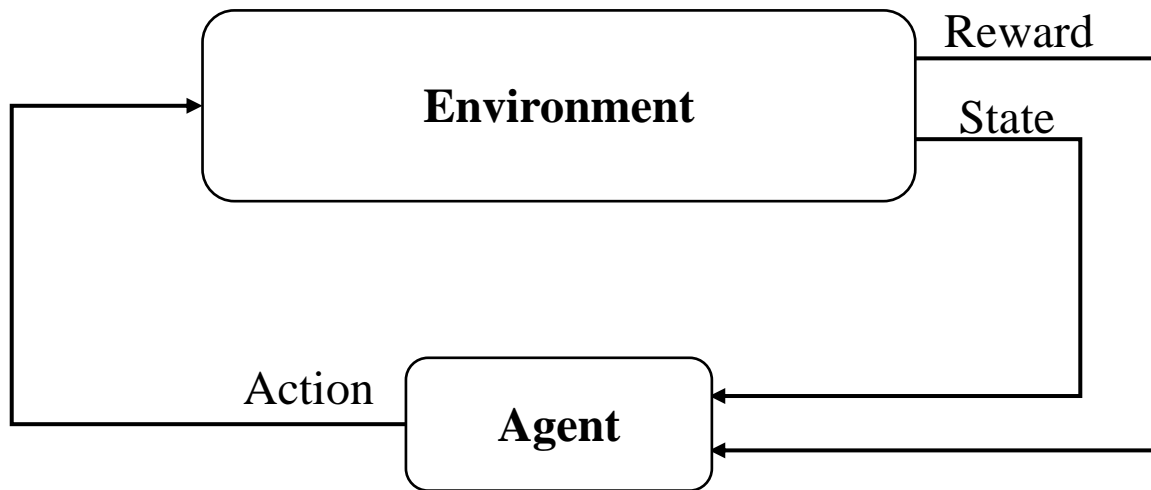
# Reinforcement learning

- **RL algorithms** are also capable of **delayed gratification**. The best overall strategy may require short-term sacrifices.
- **RL** inherently focuses on long-term reward maximization.
- It is particularly well-suited for real-world situations where feedback isn't immediately available for every step, since it can learn from delayed rewards.
- Example: Chess; win = +1, loss = -1, and draw = 0
- ✓ How can we program this?



# State, action, reward

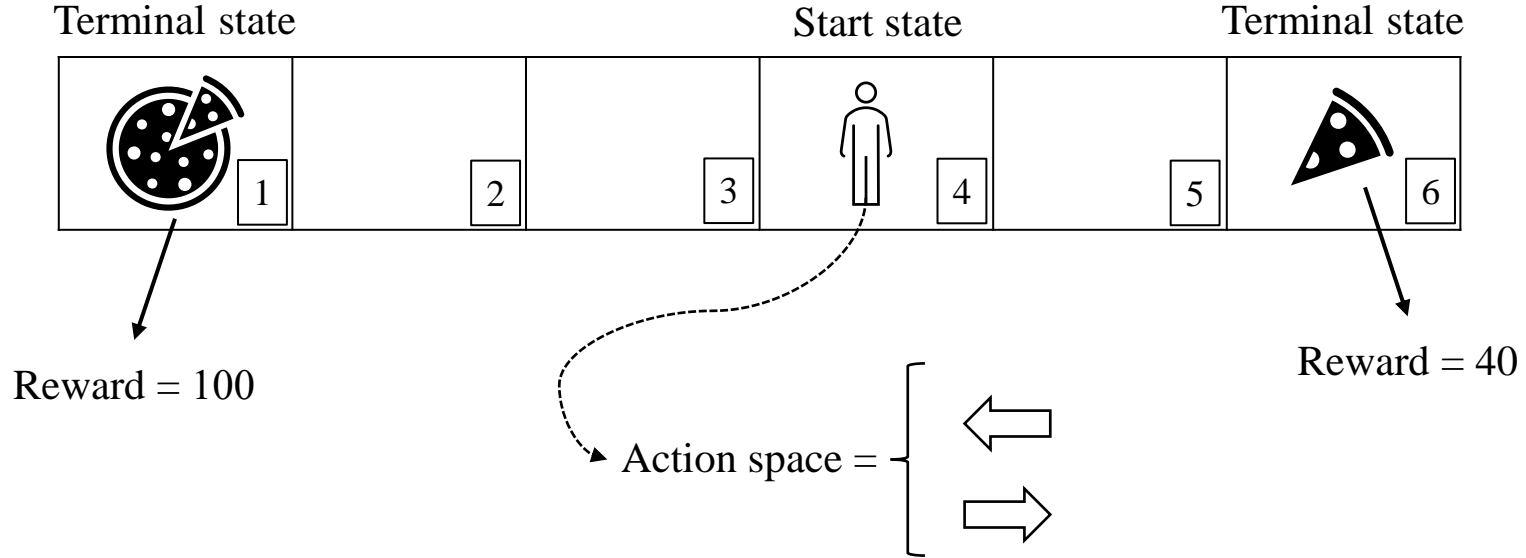
- The main characters of RL are the **agent** and the **environment**. The environment is the world that the agent lives in and interacts with.



- At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it but may also change on its own.



# Simple RL example

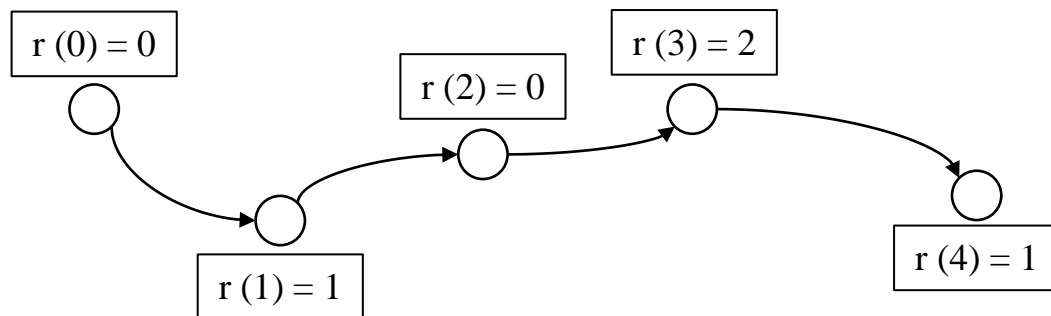


- Different environments allow different kinds of actions. The **set of all valid actions** in each environment is often called the **action space**.

# Return in RL

- As we said previously, the goal of the agent is **to maximize** its cumulative reward over a trajectory, called **return**. A **trajectory** is a sequence of states and actions.

- $r(t) \Rightarrow$  Reward in time  $t$
- $R \Rightarrow$  Return



- One kind of return is the **finite-horizon undiscounted return**, which is just the **sum of rewards obtained** in a fixed window of steps:

$$R = \sum_t r(t) \quad t = 0, 1, \dots, T \longrightarrow ?$$

# Return in RL

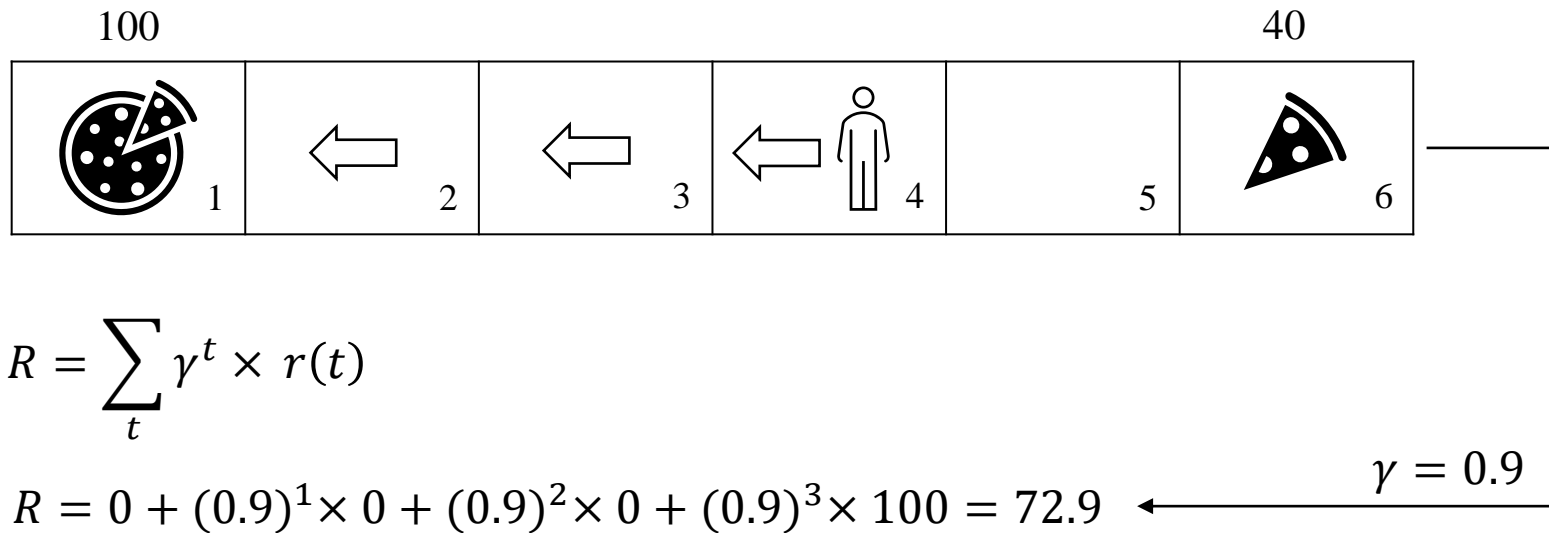
- Another kind of return is the **infinite-horizon discounted return**, which is the sum of all rewards ever obtained by the agent but **discounted by how far off in the future they're obtained**.
- This formulation of reward includes a discount factor  $\gamma \in (0,1)$ :

$$R = \sum_t \gamma^t \times r(t) \quad t = 0, 1, 2, \dots$$

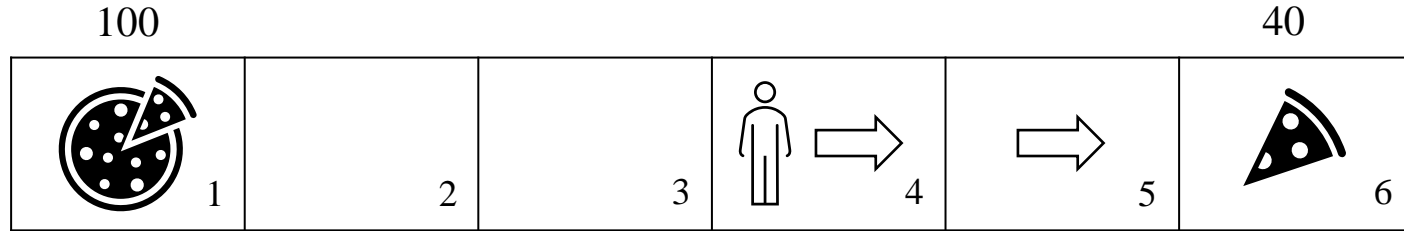
- ✓ What insights can we gain from this mathematical description?

# Simple RL example

- Computing infinite-horizon discounted return for one of the trajectories:



# Simple RL example

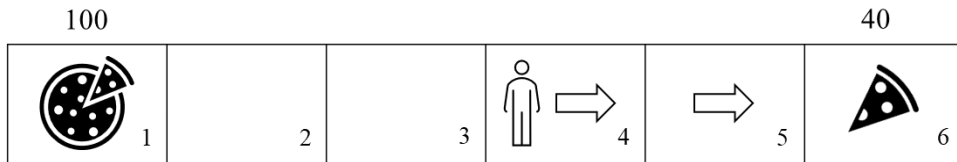


$$\boxed{\gamma = 0.9} \implies R = 0 + (0.9)^1 \times 0 + (0.9)^2 \times 40 = 32.8$$

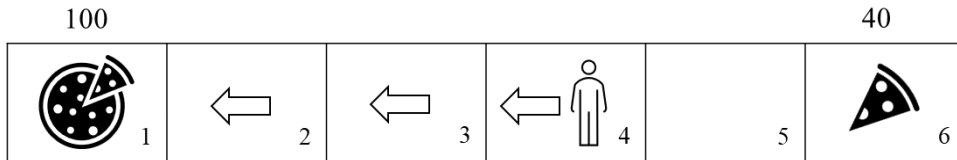
$$\boxed{\gamma = 0.5} \implies R =$$

# Simple RL example

- Let  $\gamma = 0.5$  :



$$R = 0 + (0.5)^1 \times 0 + (0.5)^2 \times 40 = 10$$







$$R = 0 + (0.5)^1 \times 0 + (0.5)^2 \times 0 + (0.5)^3 \times 100 = 12.5$$

- ✓ Starting in state 4, which action leads to optimal outcome?

# Simple RL example

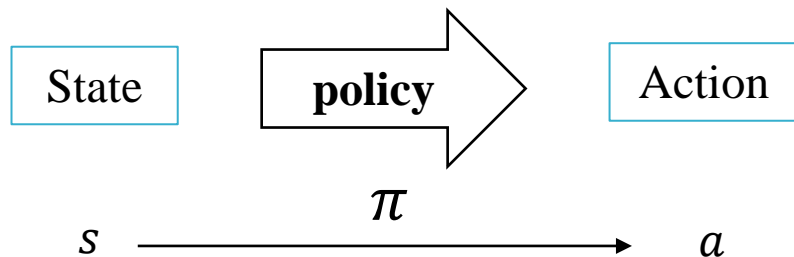
- If we assume we're going in one direction, let's compute the return for all initial states:

 100	50	25	12.5	6.25	40 	return
100	←	←	←	←	40	reward
1	2	3	4	5	6	

 100	2.5	5	10	20	40 	return
100	→	→	→	→	40	reward
1	2	3	4	5	6	

# Policy

- A **policy** is a rule used by an agent to decide what actions to take. It can be **deterministic** or **stochastic**.

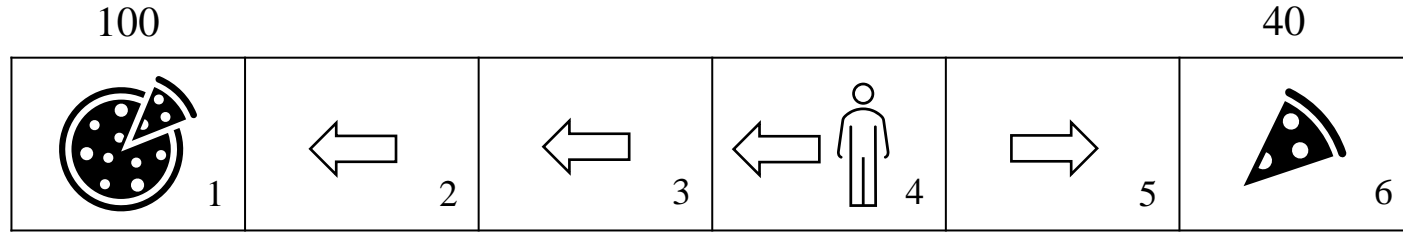


- Therefore, a policy is a function  $\pi(s) = a$  mapping from states to actions, that tells you what action  $a$  to take in each state  $s$ .
- Because the **policy** is essentially the agent's brain, it's not uncommon to substitute the word "policy" for "agent", e.g., saying "The policy is trying to maximize return."



# Policy

- The agent's policy in our example should determine to go left or right in each state.



$$\pi(s) = a \left\{ \begin{array}{l} \pi(2) = left \\ \pi(3) = left \\ \pi(4) = left \\ \pi(5) = right \end{array} \right.$$

# Value function

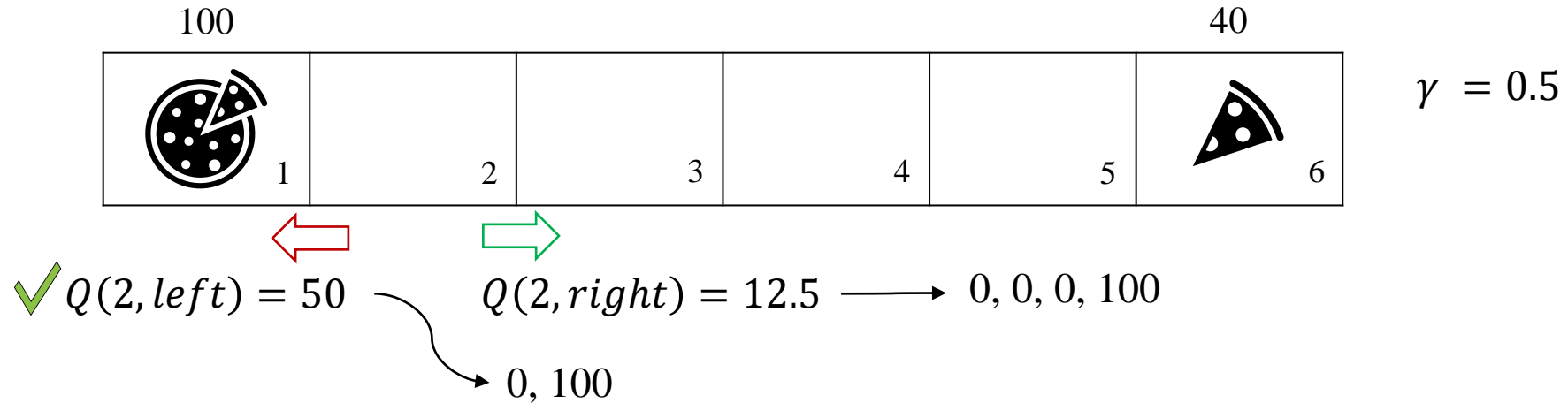
- We should find a policy  $\pi$  that tells us what action ( $a = \pi(s)$ ) to take in every state ( $s$ ) to **maximize the return**.
- It's often useful to know the **value of a state**, or state-action pair. By value, we mean the **expected return if you start in that state or state-action pair**, and then act according to a particular policy forever after.

$Q(s, a) = \text{Return if ;}$   
Start in state  $s$   
Take action  $a$   
Then behave optimally after that.

- Value functions are used, one way or another, in almost every RL algorithm.

# Value function

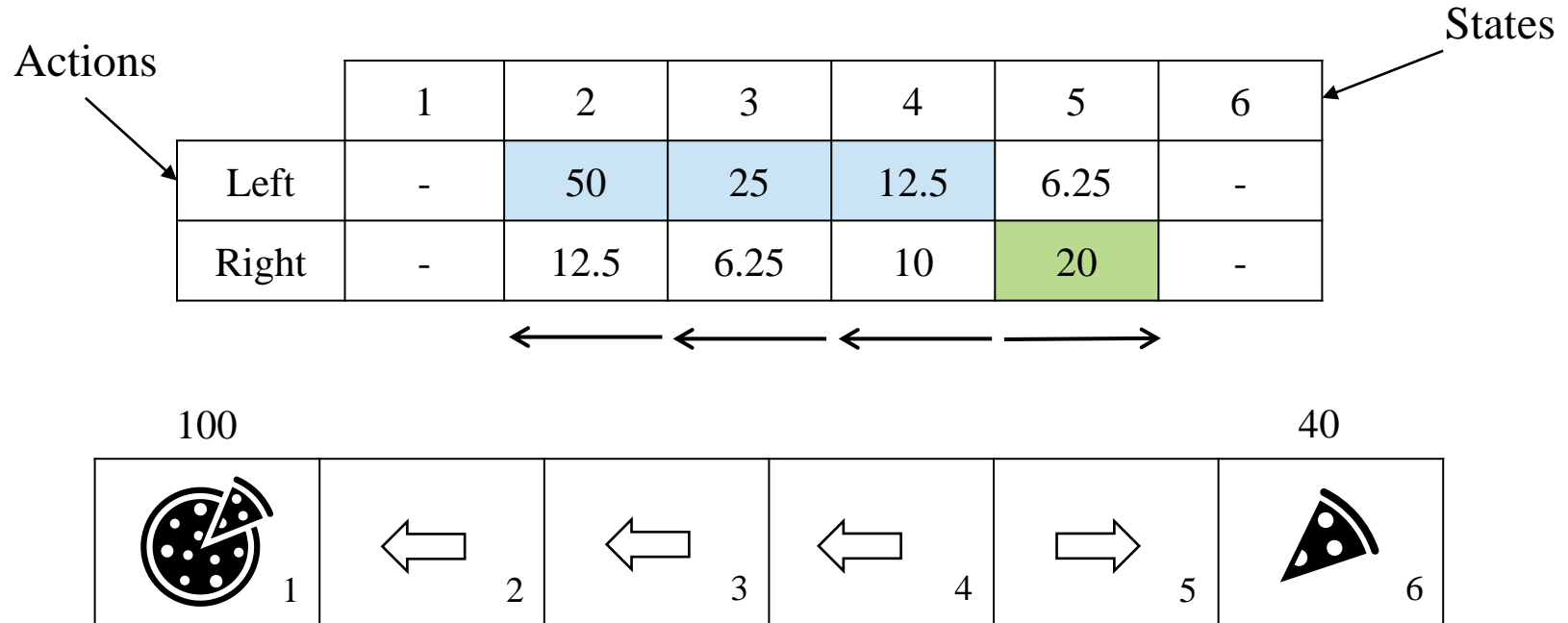
- At state 2, the agent can choose to go **left** or **right**. Each action has an associated value function  $Q(s, a)$  :



- ✓ Why did we choose left as the best action?

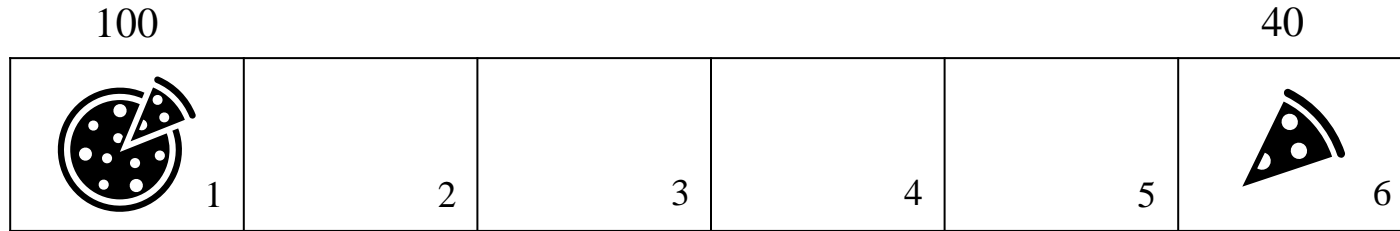
# Value function

- In our example:



# Value function

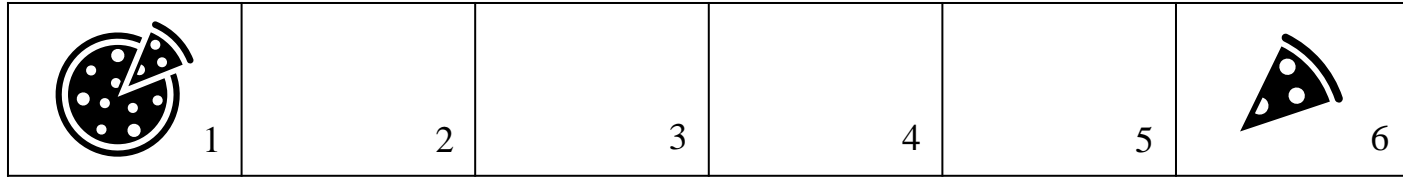
- ✓ What are simplifying assumptions we've made in our example?



- A state  $s$  is a complete description of the state of the world. There is no information about the world which is hidden from the state. An **observation**  $o$  is a **partial description of a state**, which may omit information.
- When the **agent can observe the complete state of the environment**, we say that the environment is **fully observed**. When the **agent can only see a partial observation**, we say that the environment is **partially observed**.

# Stochastic environment

- An environment is said to be stochastic when we **cannot determine the outcome based on the current state and chosen action**.
- For example, we never know what number will show up when throwing a dice.

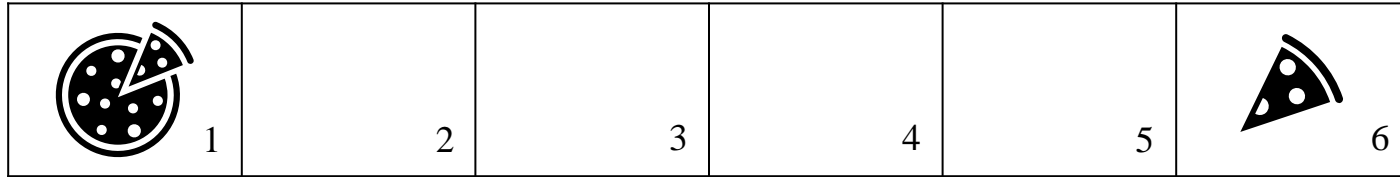


Go right:  $\left\{ \begin{array}{ll} 90\% & \Rightarrow \\ 10\% & \Rightarrow \end{array} \right.$

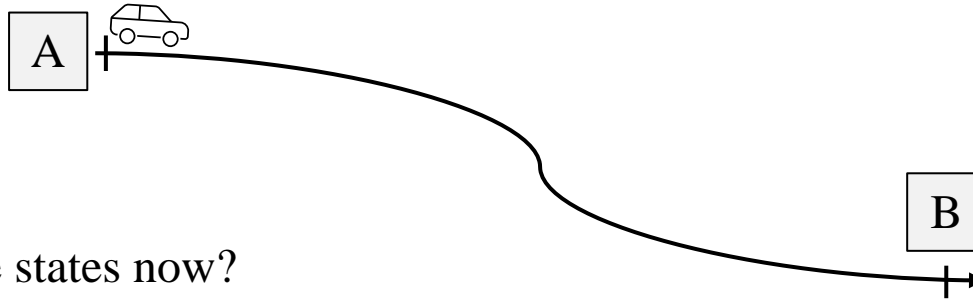
The 10% branch is crossed out with a red X, indicating a 0% probability of success in this stochastic environment.

# Discrete vs Continuous states

- Discrete states:



- Continuous states:

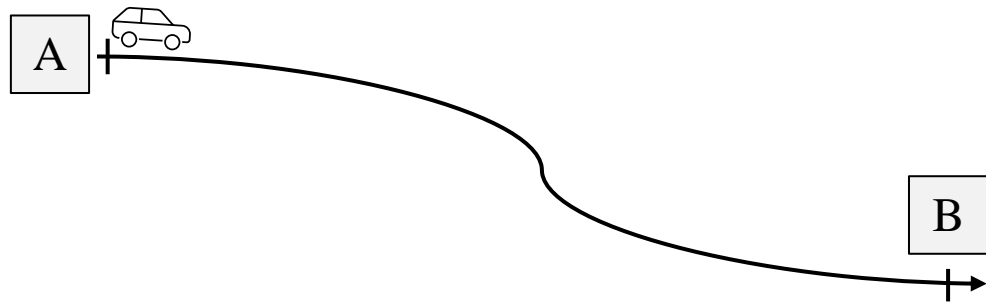


- ✓ What are the states now?

# Discrete vs Continuous states

- In RL, we almost always represent states and observations by a **real-valued vector**, **matrix**, or higher-order tensor.
- States in our discrete example:

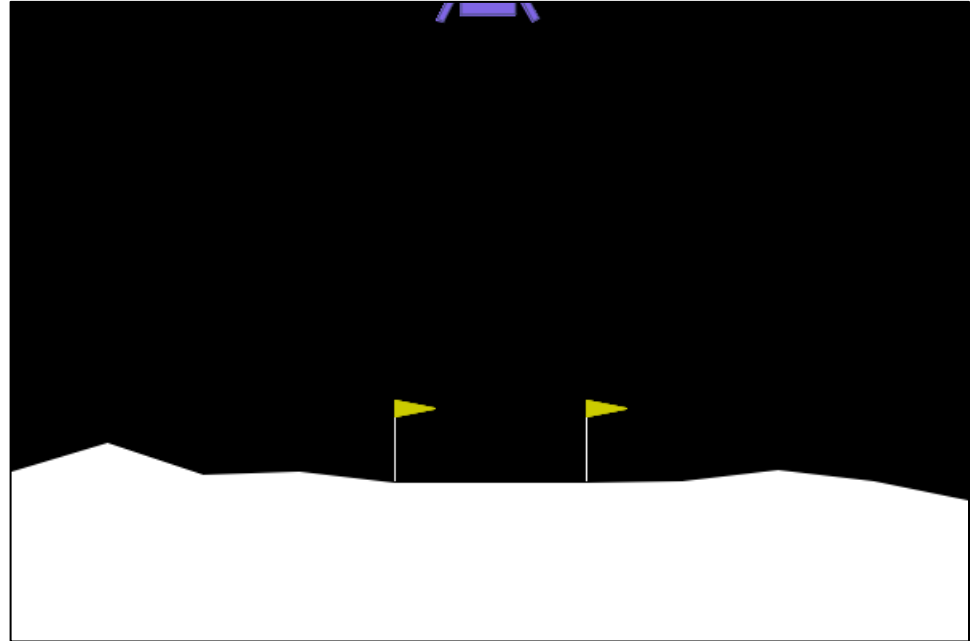
$$S = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$





# Example; Lunar Lander

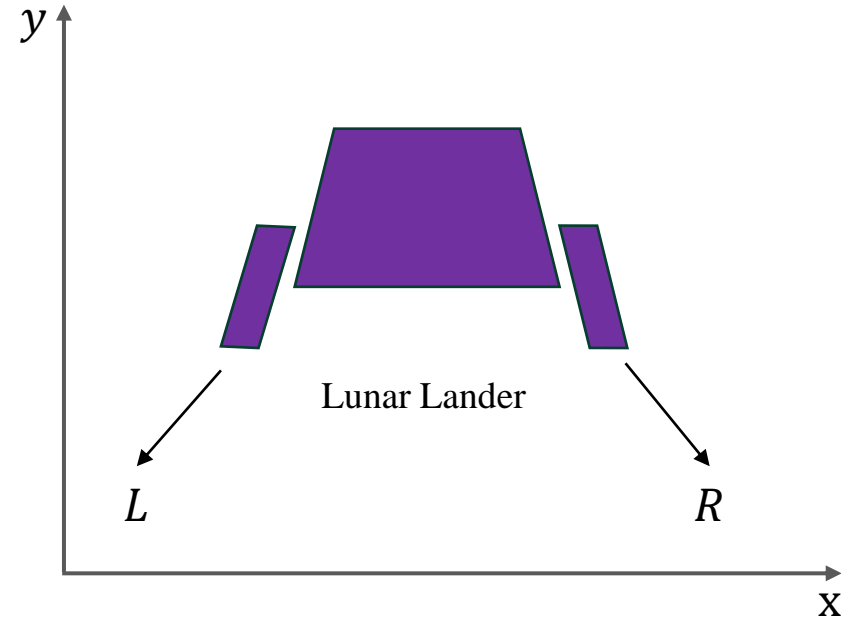
- This simulated environment deals with the problem of landing a lander on a landing pad.
- The lander starts at the top center of the viewport with a random initial force applied to its center of mass.
- A discrete action space :
  1. do nothing
  2. left thruster
  3. main thruster
  4. right thruster



# Example; Lunar Lander

- There is an 8-dimensional continuous state:

$$S = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \\ L \\ R \end{bmatrix} \left. \vphantom{\begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \\ L \\ R \end{bmatrix}} \right\} \begin{matrix} 0 \text{ or } 1 \end{matrix}$$



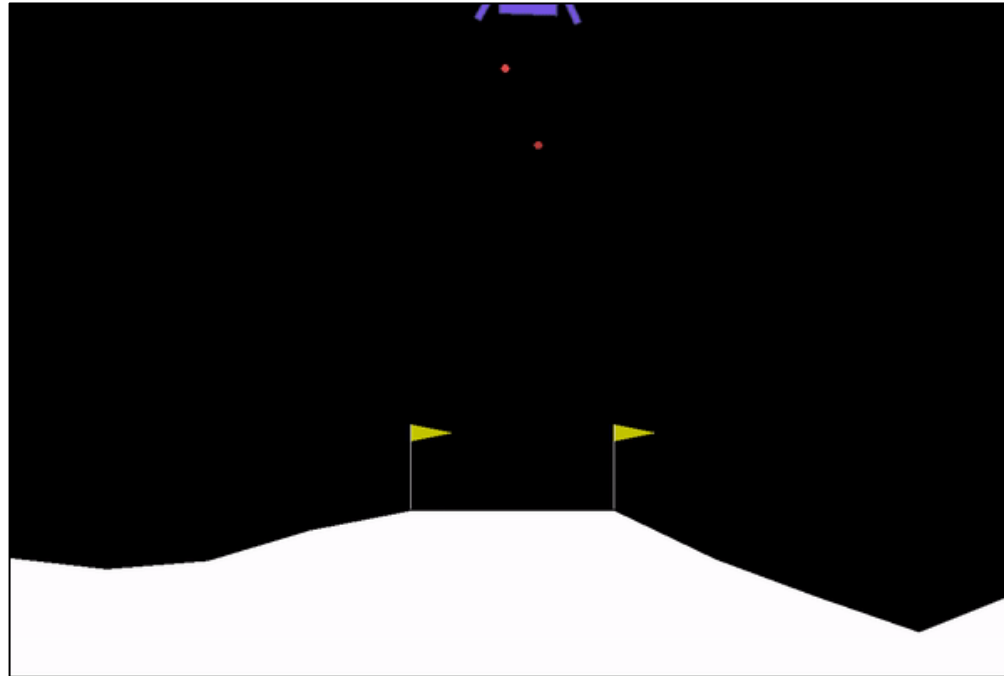
- How can we design an effective reward function?

# Example; Lunar Lander

- Rewards:
  - Getting to landing pad: 100 – 140
  - Additional reward for moving toward/away from pad.
  - Crash: -100
  - Soft landing: +100
  - Each leg with ground contact: +10
  - Fire main engine: -0.3 per frame
  - Fire side thruster: -0.03 per frame
- In this problem, we've chosen a **discount factor( $\gamma$ ) of 0.99**.
- This prioritizes long-term goals over immediate rewards, which may be suitable for this problem where the outcome(landing) is most important.

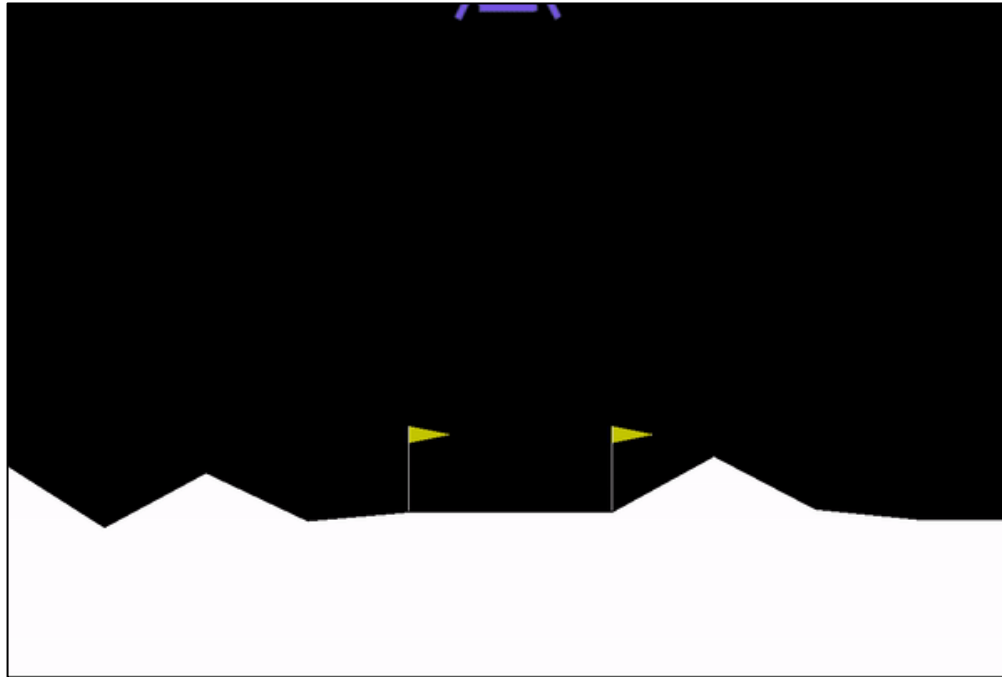
# Example; Lunar Lander

- Initially, as we can see below the agent is very bad at landing, it's basically taking random actions and receives the negative rewards for crashing the rocket.



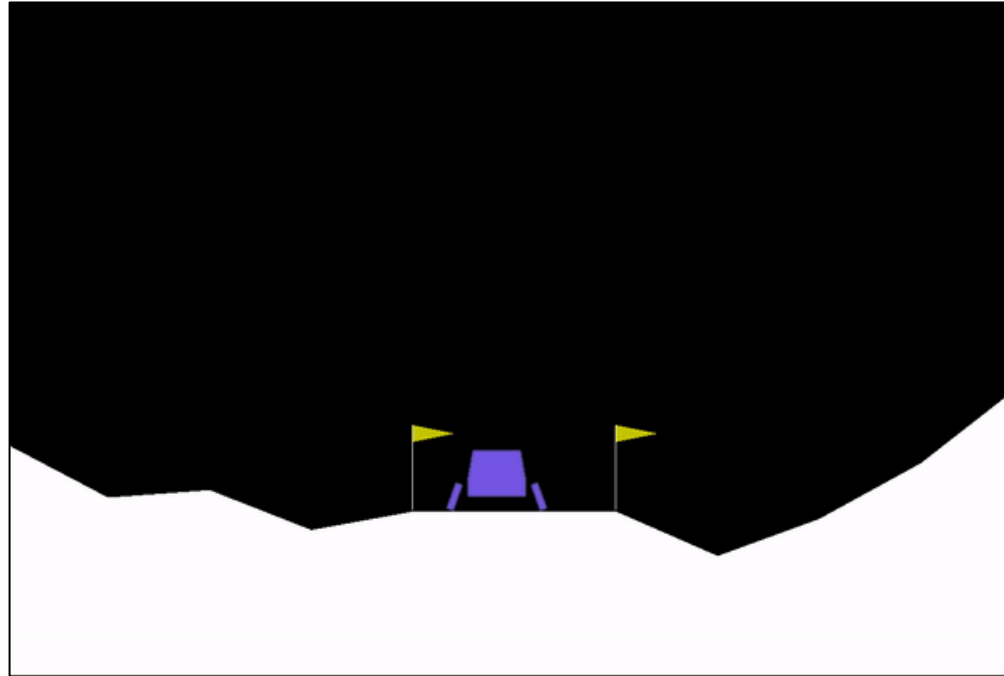
# Example; Lunar Lander

- After around 300 training episodes, it starts learning how to control and land.



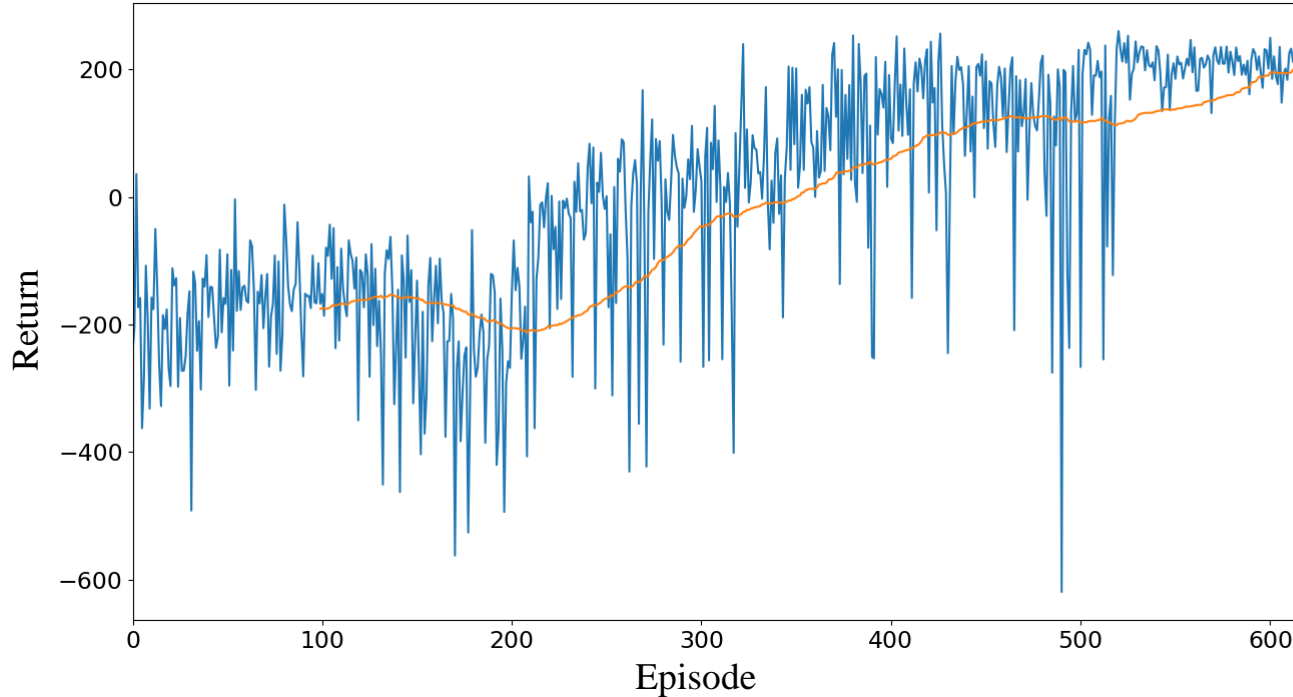
# Example; Lunar Lander

- After 600 episode, the agent is fully trained. It learns to handle the rocket and lands the rocket perfectly each time.



# Example; Lunar Lander

- The reward for each training episode:



# Example; Lunar Lander

- In each state we need to know value functions to decide what action to do. We pick the action  $a$  that maximizes  $Q(s, a)$ .

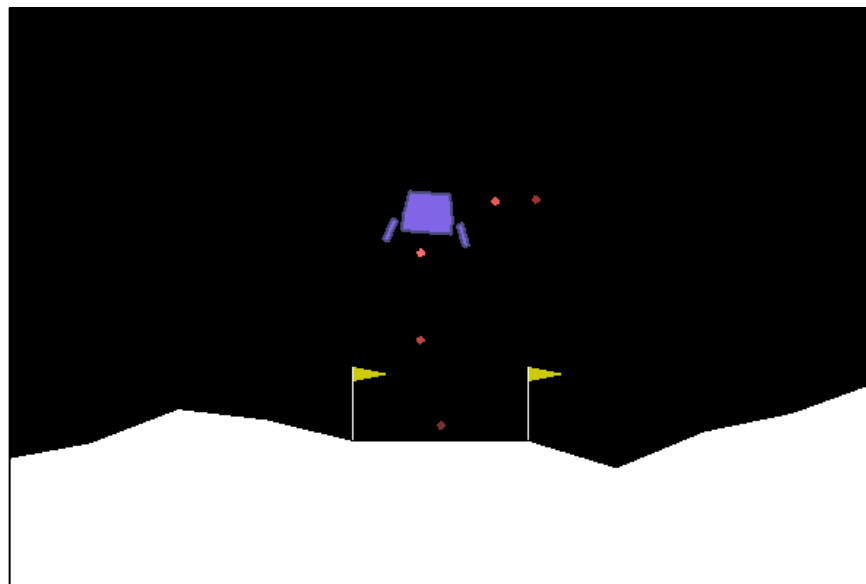
$Q(s, do\ nothing)$

$Q(s, left)$

$Q(s, right)$

$Q(s, main)$

- ✓ How to find value functions?

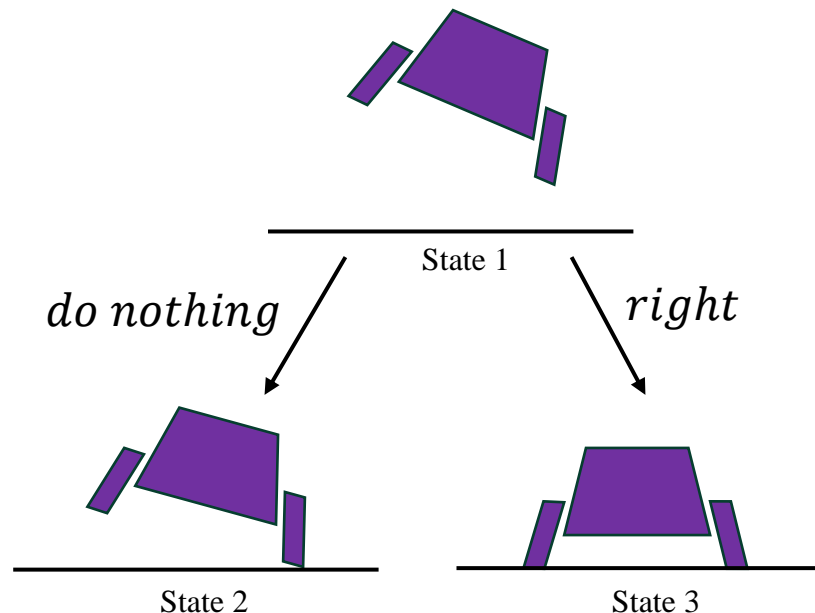


S



# Finding value functions

- There are several ways to find value functions in reinforcement learning (RL):
  1. Iterative methods:
    - Value Iteration
    - Q-Learning
  2. Model-based methods:
    - Dynamic Programming with a Model
  3. Deep Reinforcement Learning:
    - Deep Q-Networks (DQNs)



# Exploration vs. Exploitation

- **Exploration** is any action that lets the agent discover new features about the environment, while **exploitation** is capitalizing on knowledge already gained.
- If the agent continues to **exploit** only past experiences, it is likely to get stuck in a suboptimal policy.
- For optimal learning, we should **trade off** between exploration and exploitation
- Interactive visualization in [this article](#).

