

Data-intensive space engineering

Lecture 11

Carlos Sanmiguel Vila

Based on Deep RL Course from
Hugging Face



How to find optimal policy π^* ?

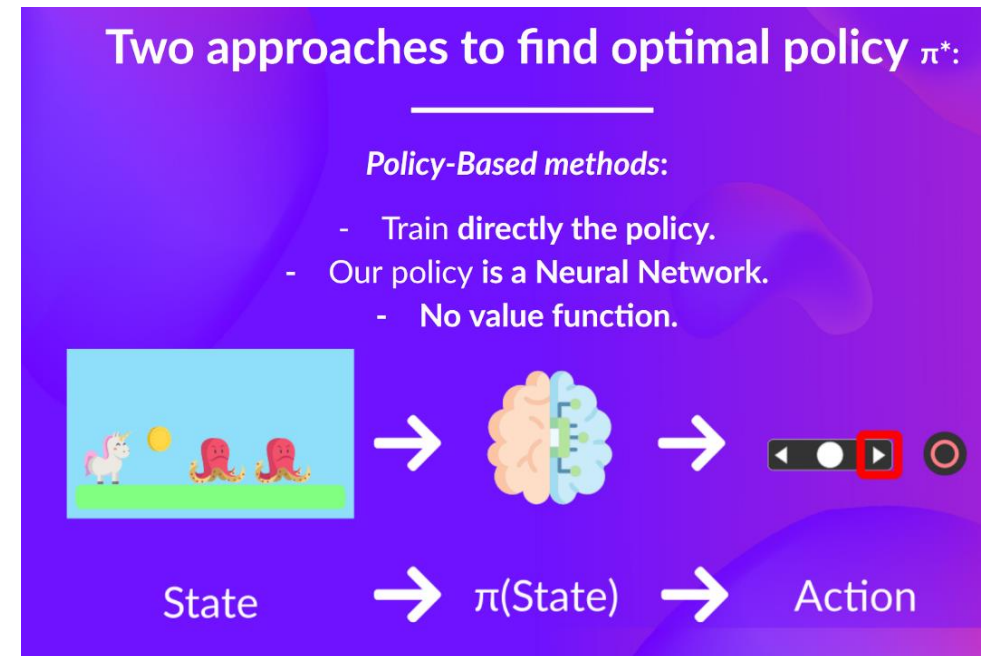
- In value-based methods, **we learn a value function that maps a state to the expected value of being at that state.**
- The value of a state is the **expected discounted return** the agent can get if it **starts at that state and then acts according to our policy.**

$$\underbrace{v_\pi(s)}_{\text{Value function}} = \underbrace{\mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots]}_{\text{Expected discounted return}} \mid \underbrace{S_t = s}_{\text{Starting at state } s}$$

But what does it mean to act according to our policy? After all, we don't have a policy in value-based methods since we train a value function and not a policy.

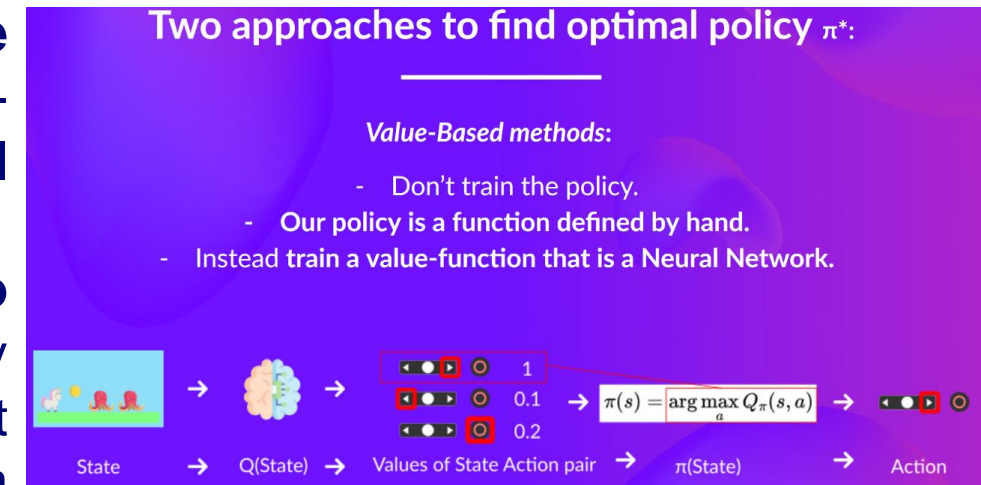
How to find optimal policy π^* ?

- To find the optimal policy, we learned about **two different methods**:
- *Policy-based methods*: **Directly train the policy** to select what action to take given a state (or a probability distribution over actions at that state). In this case, we **don't have a value function**.
- The policy takes a state as input and outputs what action to take at that state (deterministic policy: a policy that output one action given a state, contrary to stochastic policy that output a probability distribution over actions).
- And consequently, **we don't define by hand the behavior of our policy**; it's the training that will define it.



How to find optimal policy π^* ?

- To find the optimal policy, we learned about **two different methods**:
- *Value-based methods*: **Indirectly, by training a value function** that outputs the value of a state or a state-action pair. Given this value function, our policy **will take an action**.
- Since the policy is not trained/learned, **we need to specify its behavior**. For instance, if we want a policy that, given the value function, will take actions that always lead to the biggest reward, **we'll create a Greedy Policy**.
- Consequently, whatever method you use to solve your problem, **you will have a policy**. In the case of value-based methods, you don't train the policy: your policy **is just a simple pre-specified function** (for instance, the Greedy Policy) that uses the values given by the value-function to select its actions.



How to find optimal policy π^* ?

So the difference is:

- In policy-based training, **the optimal policy (denoted π^*) is found by training the policy directly.**
- In value-based training, **finding an optimal value function (denoted Q^* or V^*) leads to having an optimal policy.**

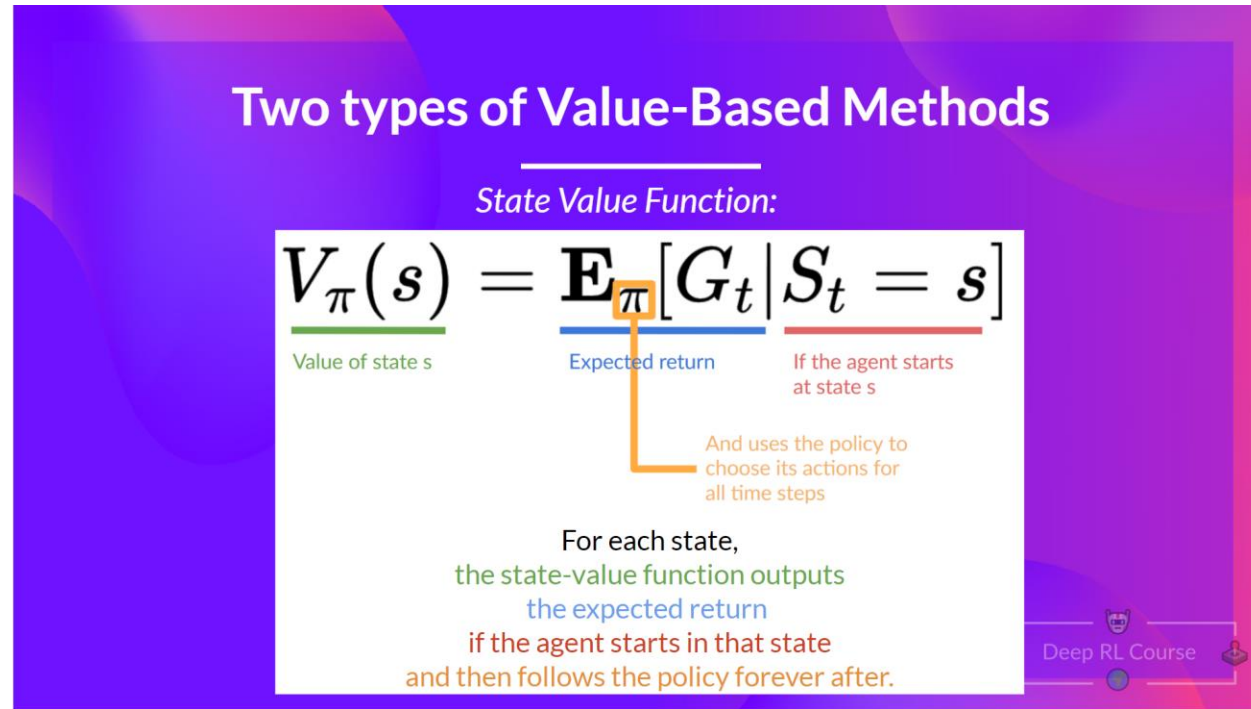
The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

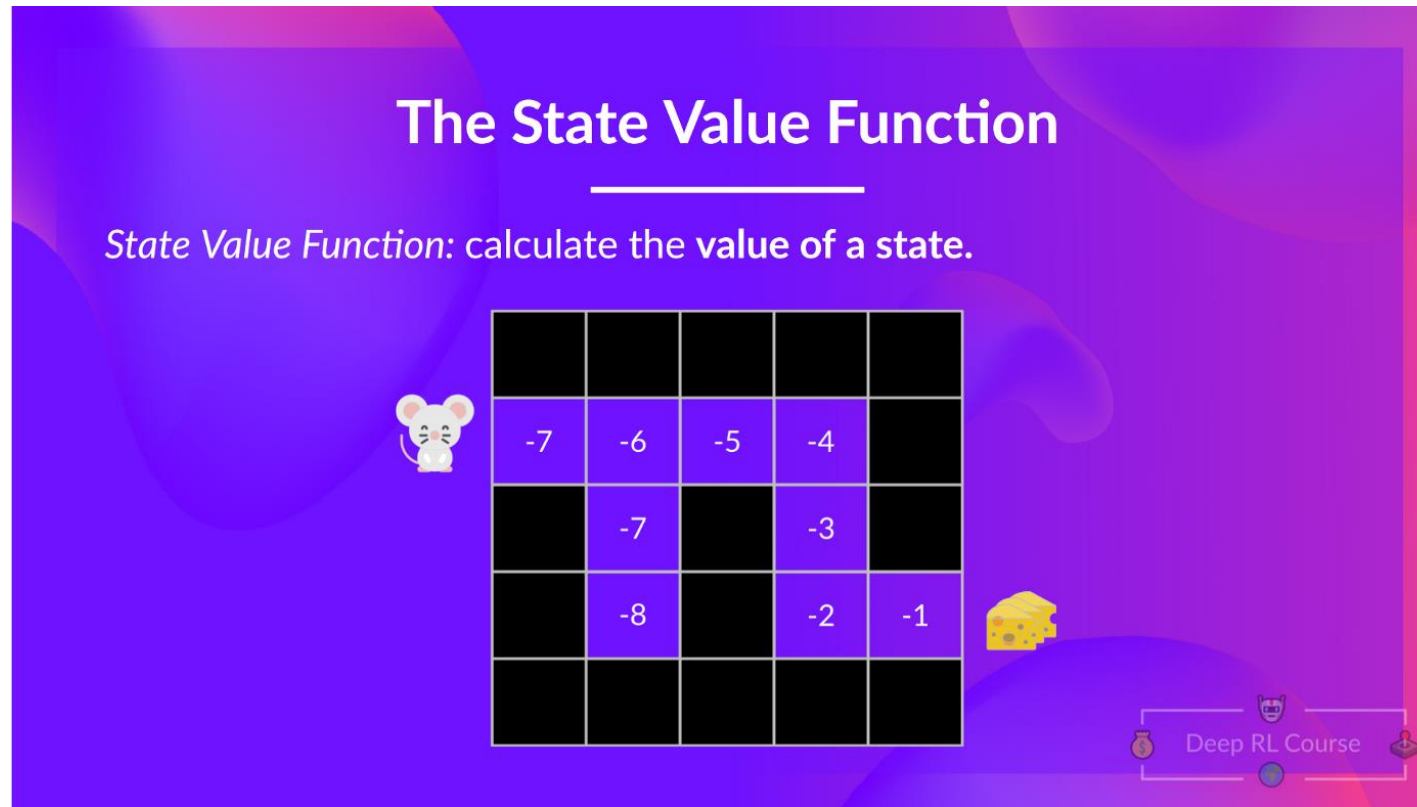
The state-value function

We write the state value function under a policy π like this:



For each state, the state-value function outputs the expected return if the agent **starts at that state** and then follows the policy forever afterward (for all future timesteps).

The state-value function



If we take the state with value -7: it's the expected return starting at that state and taking actions according to our policy (greedy policy), so right, right, right, down, down, right, right.

The action-value function

- In the action-value function, for each state and action pair, the action-value function **outputs the expected return** if the agent starts in that state, takes that action, and then follows the policy forever after.
- The value of taking action a in state s under a policy π is:

Two types of Value-Based Methods

Action Value Function:

$$Q_{\pi}(s, a) = \mathbf{E}_{\pi}[G_t | S_t = s, A_t = a]$$

Value of state-action pair s, a Expected return If the agent starts at state s and chooses action a

And then uses the policy to choose its actions for all time steps

For each state and action, the action-value function outputs the expected return if the agent starts in that state and takes the action and then follows the policy forever after.

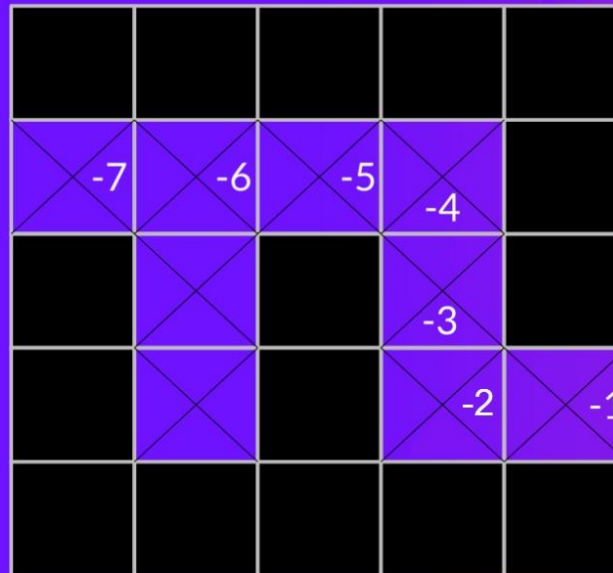
Deep RL Course

C. Samiriguel Vila

The action-value function

The Action Value Function

Action Value Function: calculate the value of state-action pair.



*We didn't fill all the state-actions pair for the example of Action-value function



Value-based methods

- For the state-value function, we calculate **the value of a state S_t**
- For the action-value function, we calculate **the value of the state-action pair (S_t, A_t) hence the value of taking that action at that state.**
- In either case, whichever value function we choose (state-value or action-value function), **the returned value is the expected return.**
- However, the problem is that **to calculate EACH value of a state or a state-action pair**, we need to sum all the rewards an agent can get if it starts at that state.

Two types of Value-Based Methods

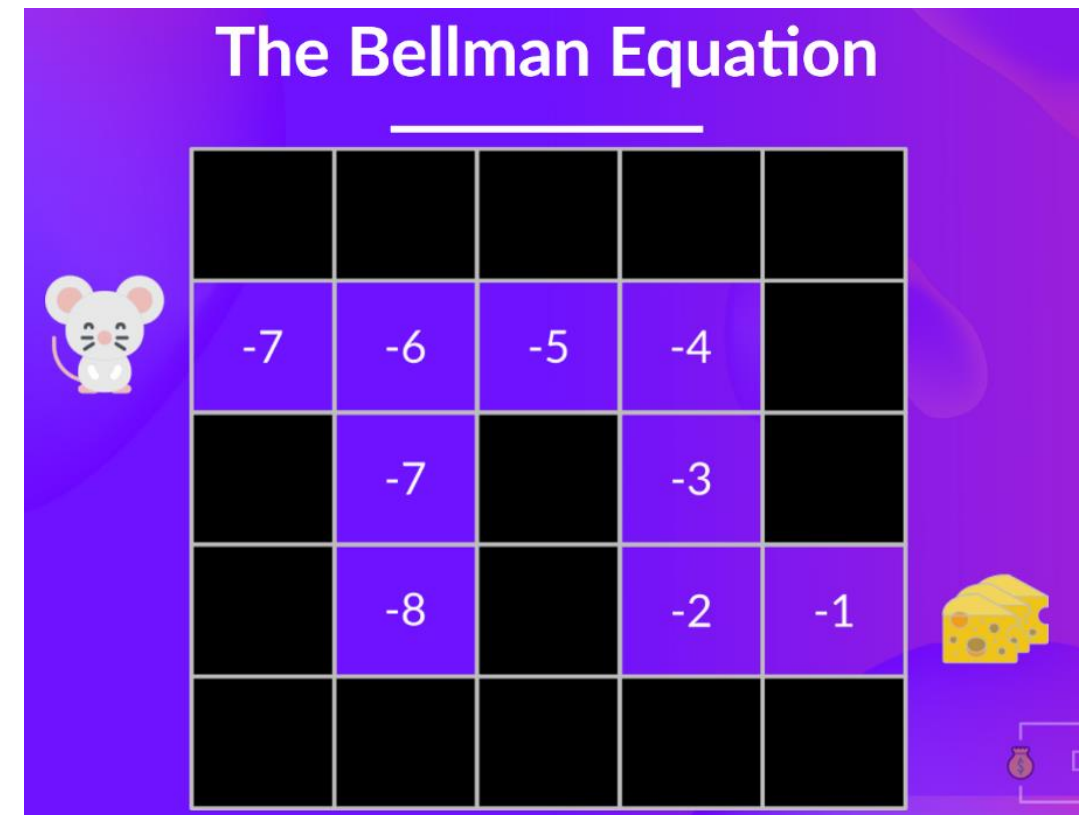
State Value Function:
calculate the value of a state.

Action Value Function:
calculate the value of state-action pair.

The Bellman Equation

The Bellman equation **simplifies our state value or state-action value calculation.**

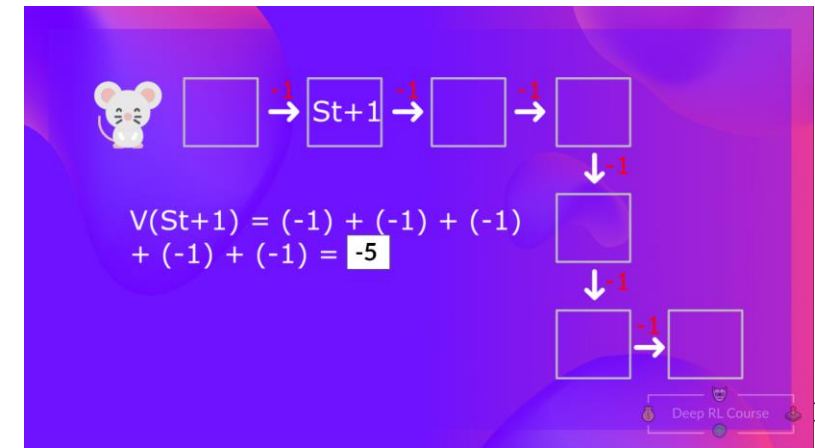
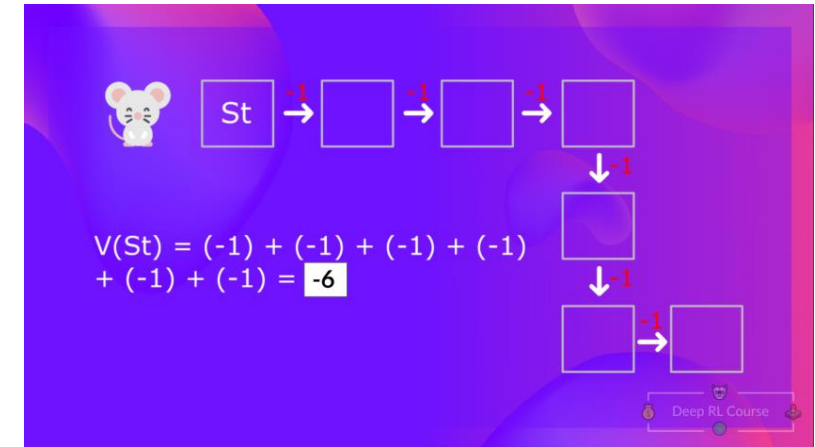
- With what we have learned so far, we know that if we calculate $V(S_t)$ (the value of a state), we need to calculate the return starting at that state and then follow the policy forever after. (The policy we defined in the following example is a Greedy Policy; for simplification, we don't discount the reward).
- A greedy algorithm reaches a problem solution using sequential steps where, at each step, it makes a decision based on the best solution at that time, without considering future consequences or implications.



The Bellman Equation

To calculate $V(S_t)$ we need to calculate the sum of the expected rewards. Hence:

- To calculate the value of State 1: the sum of rewards if the agent started in that state and then followed the greedy policy (taking actions that lead to the best state values) for all the time steps.
- To calculate the value of State 2: the sum of rewards if the agent started in that state, and then followed the policy for all the time steps..
- We're repeating the computation of the value of different states, which can be tedious if you need to do it for each state value or state-action value.
- Instead of calculating the expected return for each state or each state-action pair, we can use the **Bellman equation**.



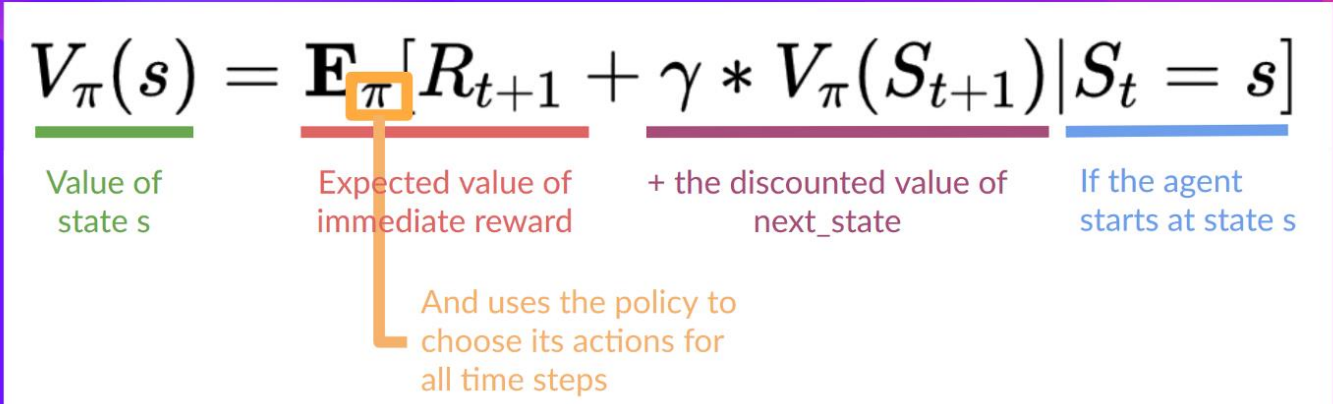
The Bellman Equation

The Bellman equation is a recursive equation that works like this: instead of starting for each state from the beginning and calculating the return, we can consider the value of any state as:


The immediate reward R_{t+1} + the discounted value of the state that follows ($\gamma * V(S_{t+1})$)

The Bellman Equation

$$V_{\pi}(s) = \mathbf{E}_{\pi} [R_{t+1} + \gamma * V_{\pi}(S_{t+1}) | S_t = s]$$



And uses the policy to choose its actions for all time steps



$V(S_t)$

$\xrightarrow{R_{t+1}}$

$V(S_{t+1})$

Deep RL Course

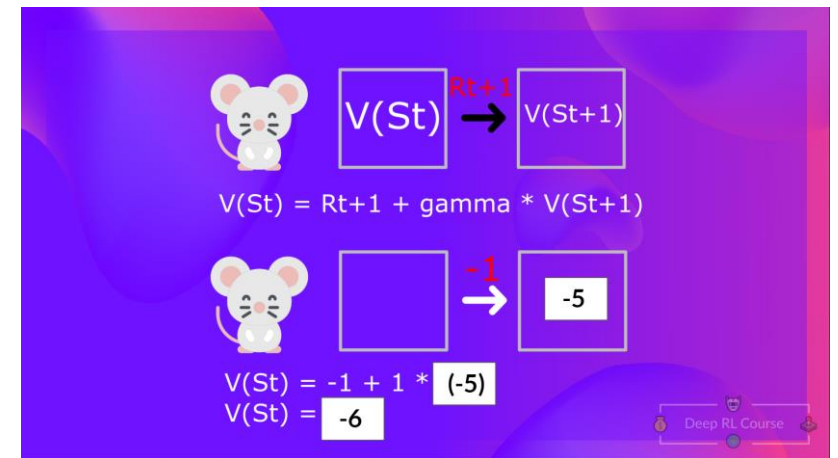
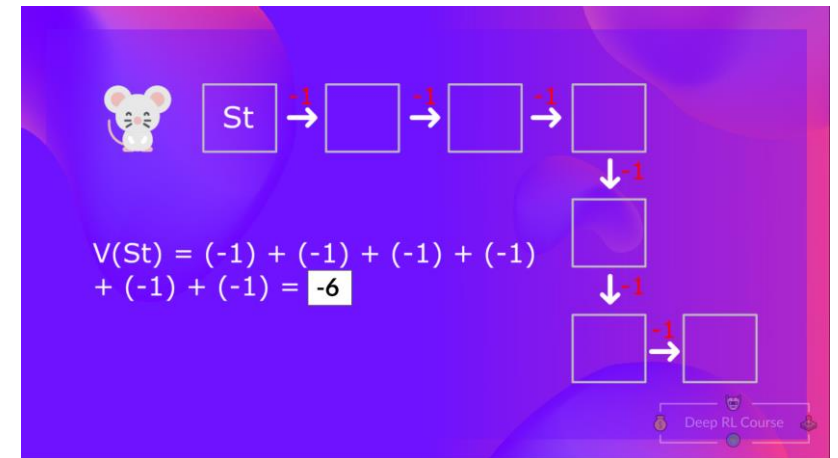
$V(S_t) = R_{t+1} + \gamma * V(S_{t+1})$

The Bellman Equation

If we go back to our example, we can say that the value of State 1 is equal to the expected cumulative return if we start at that state.

- To calculate the value of State 1: the sum of rewards if the agent started in that state 1 and then followed the policy for all the time steps.
- This is equivalent to $V(S_t) = \text{Immediate reward } R_{t+1} + \text{Discounted value of the next state } \gamma * V(S_{t+1})$ (In the interest of simplicity, here we don't discount, so $\gamma = 1$)
- The value of $V(S_{t+1}) = \text{Immediate reward } R_{t+2} + \text{Discounted value of the next state } \gamma * V(S_{t+2})$

To recap, the idea of the Bellman equation is that instead of calculating each value as the sum of the expected return, which is a long process, we calculate the value as the sum of immediate reward + the discounted value of the state that follows.



Monte Carlo vs Temporal Difference Learning

Remember that an RL agent **learns by interacting with its environment**. The idea is that **given the experience and the received reward, the agent will update its value function or policy**.

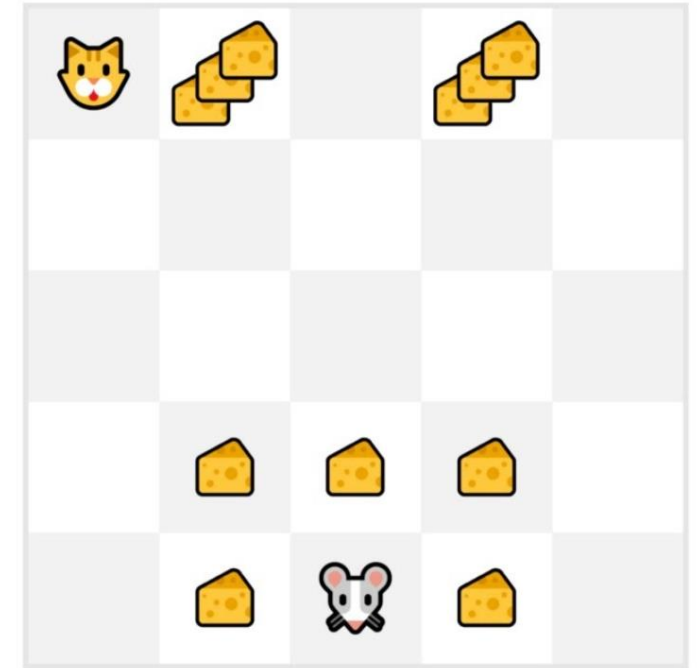
Monte Carlo and Temporal Difference Learning are two different **strategies on how to train our value function or our policy function**. Both **use experience to solve the RL problem**.

On one hand, Monte Carlo uses **an entire episode of experience before learning**. On the other hand, Temporal Difference uses **only a step ($S_t, A_t, R_{t+1}, S_{t+1}$) to learn**.

Monte Carlo

- We always start the episode **at the same starting point**.
- **The agent takes actions using the policy**. For instance, using an Epsilon Greedy Strategy, a policy that alternates between exploration (random actions) and exploitation.
- We get **the reward and the next state**.
- We terminate the episode if the cat eats the mouse or if the mouse moves > 10 steps.
- At the end of the episode, **we have a list of State, Actions, Rewards, and Next States tuples**
- **The agent will sum the total rewards G_t** (to see how well it did).
- It will then **update $V(S_t)$ based on the formula $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$**
- Then **start a new game with this new knowledge**

By running more and more episodes, **the agent will learn to play better and better.**



$$\text{Action at time}(t) \left\{ \begin{array}{ll} \max Q_t(a) & \text{with probability } 1-\epsilon \\ \text{any action } (a) & \text{with probability } \epsilon \end{array} \right.$$

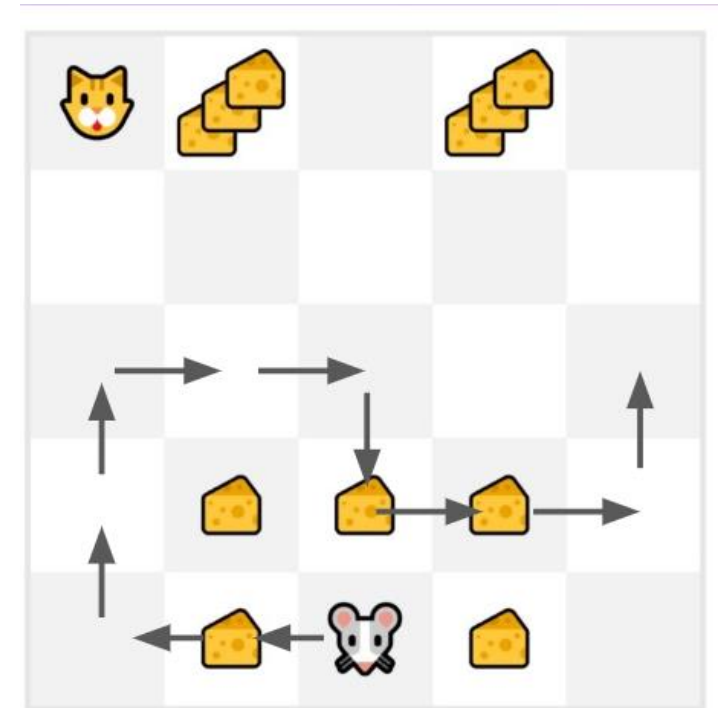
Monte Carlo

For instance, if we train a state-value function using Monte Carlo:

- We initialize our value function **so that it returns 0 value for each state**
- Our learning rate (lr) is 0.1 and our discount rate is 1 (= no discount)
- Our mouse **explores the environment and takes random actions**
- The mouse made more than 10 steps, so the episode ends .
- We have a list of state, action, rewards, next_state, **we need to calculate the return**
- $G_t = 0$ $G_t = R_{t+1} + R_{t+2} + R_{t+3} \dots$ (for simplicity, we don't discount the rewards) $G_0 = R_1 + R_2 + R_3 \dots G_0 = R_1 + R_2 + R_3 \dots G_0 = 3$
- We can now compute the **new $V(S_0)$** :

$$V(S_0) = V(S_0) + \alpha[G_0 - V(S_0)]$$

$$V(S_0) = 0 + 0.1[3 - 0] = 0.3$$



Temporal Difference Learning

- Temporal Difference, on the other hand, waits for **only one interaction (one step)** S_{t+1} to form a TD target and update $V(S_t)$ using R_{t+1} and $\gamma * V(S_{t+1})$.
- The idea with TD is to **update the $V(S_t)$ at each step**.
- But because we didn't experience an entire episode, we don't have G_t (expected return). Instead, **we estimate G_t by adding R_{t+1} and the discounted value of the next state**.
- This is called bootstrapping. It's called this **because TD bases its update in part on an existing estimate $V(S_{t+1})$. and not a complete sample G_t** .
- This method is called TD(0) or **one-step TD (update the value function after any individual step)**.

TD Learning Approach:

Temporal Difference Learning: learning at each time step.

$$\underbrace{V(S_t)}_{\text{New value of state } t} \leftarrow \underbrace{V(S_t)}_{\text{Former estimation of value of state } t} + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R_{t+1}}_{\text{Reward}} + \underbrace{\gamma V(S_{t+1})}_{\text{Discounted value of next state}} - \underbrace{V(S_t)}_{\text{TD Target}}]$$

Temporal Difference Learning

If we take the same example

- We initialize our value function so that it returns 0 value for each state.
- Our learning rate (α) is 0.1, and our discount rate is 1 (no discount).
- Our mouse begins to explore the environment and takes a random action: **going to the left**
- It gets a reward $R_{t+1} = 1$ since **it eats a piece of cheese**

We can now update $V(S_0)$:

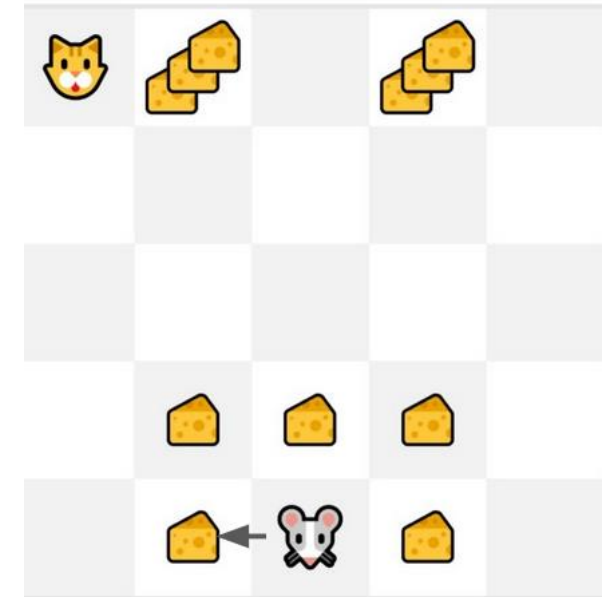
$$\text{New } V(S_0) = V(S_0) + \alpha[R_{t+1} + \gamma * V(S_1) - V(S_0)]$$

$$\text{New } V(S_0) = 0 + 0.1 * [1 + 1 * 0 - 0]$$

$$\text{New } V(S_0) = 0.1.$$

So we just updated our value function for State 0.

Now we **continue to interact with this environment with our updated value function.**



Monte Carlo vs Temporal Difference Learning

To summarize:

With Monte Carlo, we update the value function from a complete episode, and so we **use the actual accurate discounted return of this episode.**

With TD Learning, we update the value function from a step, and we replace G_t , which we don't know, with **an estimated return called the TD target.**

Monte Carlo:
$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

TD Learning:
$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

What is Q-Learning?

Q-Learning is an **off-policy value-based method** that uses a **TD approach** to train its **action-value function**:

- *Off-policy*: we'll talk about that at the end of this unit.
- *Value-based method*: finds the optimal policy indirectly by training a value or action-value function that will tell us **the value of each state or each state-action pair**.
- *TD approach*: **updates its action-value function at each step instead of at the end of the episode**.

Q-Learning is the algorithm we use to train our Q-function, an **action-value function** that determines the value of being at a particular state and taking a specific action at that state.

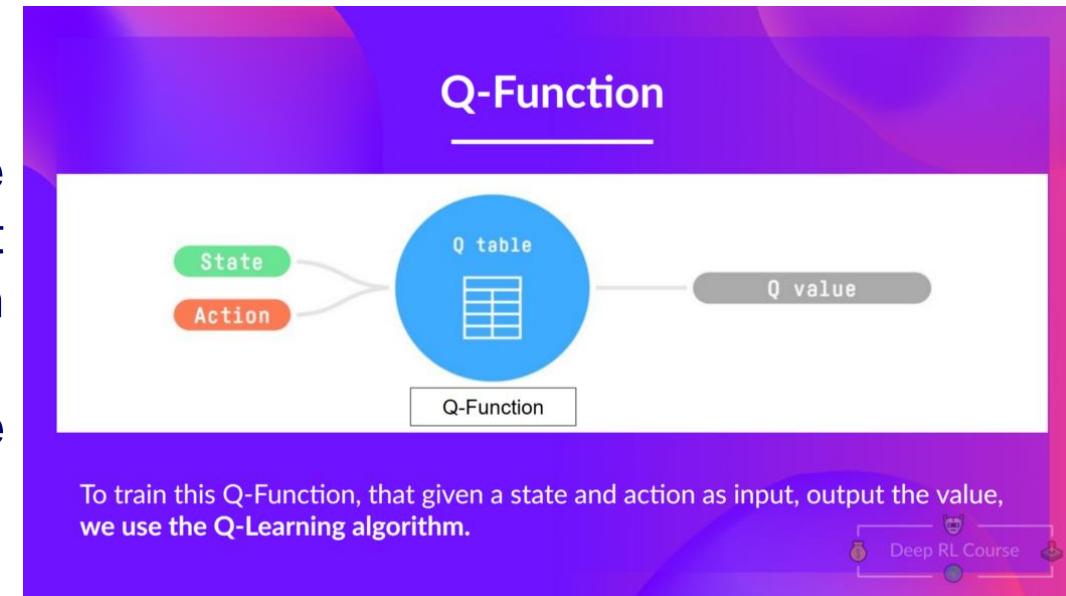
What is Q-Learning?

The **Q** comes from “the Quality” (the value) of that action at that state.

Let's recap the difference between value and reward:

- The *value* of a state, or a *state-action pair*, is the expected cumulative reward our agent gets if it starts at this state (or state-action pair) and then acts according to its policy.
- The *reward* is the **feedback I get from the environment** after performing an action at a state.

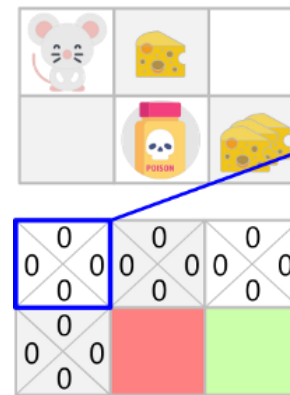
Internally, our Q-function is encoded by a **Q-table**, a table where each cell corresponds to a state-action pair value. Think of this Q-table as the memory or cheat sheet of our Q-function.



What is Q-Learning?

Let's go through an example of a maze.

The Q-table is initialized. That's why all values are = 0. This table **contains, for each state and action, the corresponding state-action values**. For this simple example, the state is only defined by the position of the mouse. Therefore, we have 2*3 rows in our Q-table, one row for each possible position of the mouse. In more complex scenarios, the state could contain more information than the position of the actor.




States

Actions				
	←	→	↑	↓
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

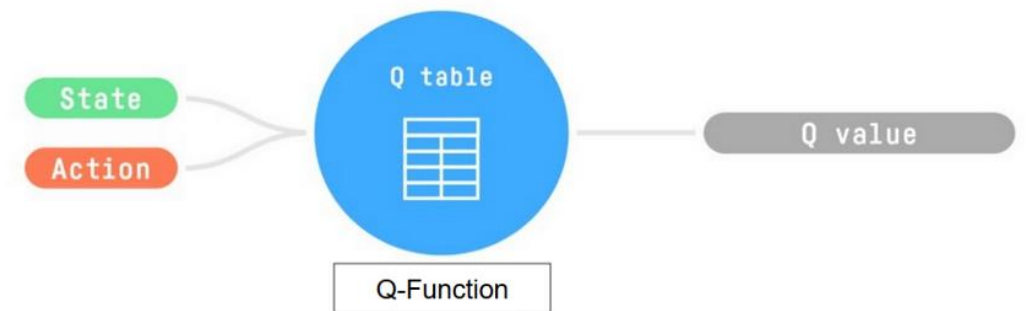
The RL Process

Here we see that the **state-action value of the initial state and going up is 0**:

So: the Q-function uses a Q-table **with the value of each state-action pair**. Given a state and action, **our Q-function will search inside its Q-table to output the value**.



		Actions			
		←	→	↑	↓
States		0	0	0	0
		0	0	0	0
		0	0	0	0
		0	0	0	0
		0	0	0	0
		0	0	0	0



The RL Process

If we recap, *Q-Learning* is the RL algorithm that:

- Trains a *Q-function* (an **action-value function**), which internally is a **Q-table** that **contains all the state-action pair values**.
- Given a state and action, our Q-function **will search its Q-table for the corresponding value**.
- When the training is done, **we have an optimal Q-function**, which means we have **optimal Q-table**.
- And if we **have an optimal Q-function**, we have an **optimal policy** since we **know the best action to take at each state**.

The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

The RL Process

In the beginning, **our Q-table is useless** since it gives **arbitrary values** for each **state-action pair** (most of the time, we initialize the Q-table to 0). As the agent **explores the environment** and **we update the Q-table**, it will give us a **better and better approximation** to the optimal policy.



The Q-Learning algorithm

Q-Learning

Algorithm 14: Sarsamax (Q-Learning)

Input: policy π , positive integer $num_episodes$, small positive fraction α , GLIE $\{\epsilon_i\}$

Output: value function Q ($\approx q_\pi$ if $num_episodes$ is large enough)

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

for $i \leftarrow 1$ to $num_episodes$ do

$\epsilon \leftarrow \epsilon_i$

 Observe S_0

$t \leftarrow 0$

 repeat

 Choose action A_t using policy derived from Q (e.g., ϵ -greedy) Step 2

 Take action A_t and observe R_{t+1}, S_{t+1} Step 3

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$ Step 4

$t \leftarrow t + 1$

 until S_t is terminal;

end

return Q

↖ Step 1

Step 2











Step 3

Step 4

The Q-Learning algorithm

Q-Learning, Step 1

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

				
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

We initialize the Q-Table

The Q-Learning algorithm

Q-Learning, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



Choose the action using ϵ -greedy policy

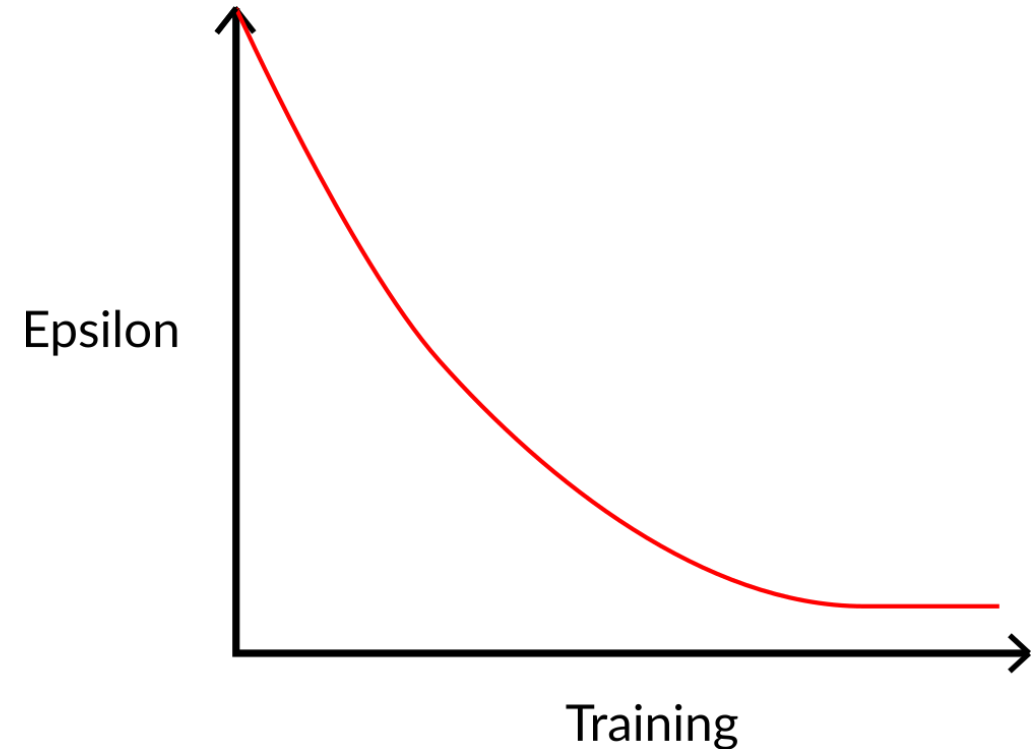
The Q-Learning algorithm

The epsilon-greedy strategy is a policy that handles the exploration/exploitation trade-off.

The idea is that, with an initial value of $\varepsilon = 1.0$:

- With probability $1 - \varepsilon$: we do **exploitation** (aka our agent selects the action with the highest state-action pair value).
- With probability ε : **we do exploration** (trying random action).

At the beginning of the training, **the probability of doing exploration will be huge since ε is very high, so most of the time, we'll explore.** But as the training goes on, and consequently our **Q-table gets better and better in its estimations, we progressively reduce the epsilon value** since we will need less and less exploration and more exploitation.



The Q-Learning algorithm

Step 3: Perform action A_t , get reward R_{t+1} and next state S_{t+1}

Q-Learning, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}

The Q-Learning algorithm

Step 4: Update $Q(S_t, A_t)$

Remember that in TD Learning, we update our policy or value function (depending on the RL method we choose) **after one step of the interaction**.

To produce our TD target, we used the **immediate reward R_{t+1} plus the discounted value of the next state**, computed by finding the action that maximizes the current Q-function at the next state. (We call that bootstrap).

$$\underbrace{V(S_t)}_{\text{New value of state } t} \leftarrow \underbrace{V(S_t)}_{\text{Former estimation of value of state } t} + \underbrace{\alpha}_{\text{Learning Rate}} \underbrace{[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]}_{\text{TD Target}}$$

The diagram illustrates the TD update equation with color-coded components: $V(S_t)$ (green), $V(S_t)$ (blue), α (red), R_{t+1} (orange), $\gamma V(S_{t+1}) - V(S_t)$ (purple), and the entire bracketed term (teal).

The Q-Learning algorithm

Step 4: Update $Q(S_t, A_t)$

Therefore, our $Q(S_t, A_t)$ **update formula goes like this:**

$$\underbrace{Q(S_t, A_t)}_{\text{New Q-value estimation}} \leftarrow \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}} + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R_{t+1}}_{\text{Immediate Reward}} + \underbrace{\gamma \max_a Q(S_{t+1}, a)}_{\text{Discounted Estimate optimal Q-value of next state}} - \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}}]$$

New
Q-value
estimation

Former
Q-value
estimation

Learning
Rate

Immediate
Reward

Discounted Estimate
optimal Q-value
of next state

Former
Q-value
estimation

TD Target

TD Error

The Q-Learning algorithm

Step 4: Update $Q(S_t, A_t)$

This means that to update our $Q(S_t, A_t)$:

- We need $S_t, A_t, R_{t+1}, S_{t+1}$.
- To update our Q-value at a given state-action pair, we use the TD target.

How do we form the TD target?

1. We obtain the reward R_{t+1} after taking the action A_t .
2. To get the **best state-action pair value** for the next state, we use a greedy policy to select the next best action. Note that this is not an epsilon-greedy policy, this will always take the action with the highest state-action value.

Then when the update of this Q-value is done, we start in a new state and select our action **using a epsilon-greedy policy again**.

This is why we say that Q Learning is an off-policy algorithm.

Off-policy vs On-policy

The difference is subtle:

- *Off-policy*: using a **different policy for acting (inference) and updating (training)**.

For instance, with Q-Learning, the epsilon-greedy policy (acting policy), is different from the greedy policy that is **used to select the best next-state action value to update our Q-value (updating policy)**.

Choose action A_t using policy derived from Q (e.g., ϵ -greedy) Acting Policy

$\gamma \max_a Q(S_{t+1}, a)$ Updating policy

- *On-policy*: using the **same policy for acting and updating**.

For instance, with Sarsa, another value-based algorithm, **the epsilon-greedy policy selects the next state-action pair, not a greedy policy**.

```
Choose action  $A_0$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
 $t \leftarrow 0$ 
repeat
    Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$ 
    Choose action  $A_{t+1}$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$ 
```

Epsilon Greedy Policy

Q-Learning example

- You're a mouse in this tiny maze. You always **start at the same starting point**.
- The goal is **to eat the big pile of cheese at the bottom right-hand corner** and avoid the poison. After all, who doesn't like cheese?
- The episode ends if we eat the poison, **eat the big pile of cheese**, or if we take more than five steps.
- The learning rate is 0.1
- The discount rate (gamma) is 0.99



Q-Learning example

The reward function goes like this:

- **+0**: Going to a state with no cheese in it.
- **+1**: Going to a state with a small cheese in it.
- **+10**: Going to the state with the big pile of cheese.
- **-10**: Going to the state with the poison and thus dying.
- **+0** If we take more than five steps.



To train our agent to have an optimal policy (so a policy that goes right, right, down), **we will use the Q-Learning algorithm.**

Q-Learning example











Step 1: Initialize the Q-table

So, for now, **our Q-table is useless**; we need to train our Q-function using the Q-Learning algorithm.

Let's do it for 2 training timesteps.

Example, Step 1

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\text{terminal-state}, \cdot) = 0$)

				
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

We initialize the Q-Table

Q-Learning example

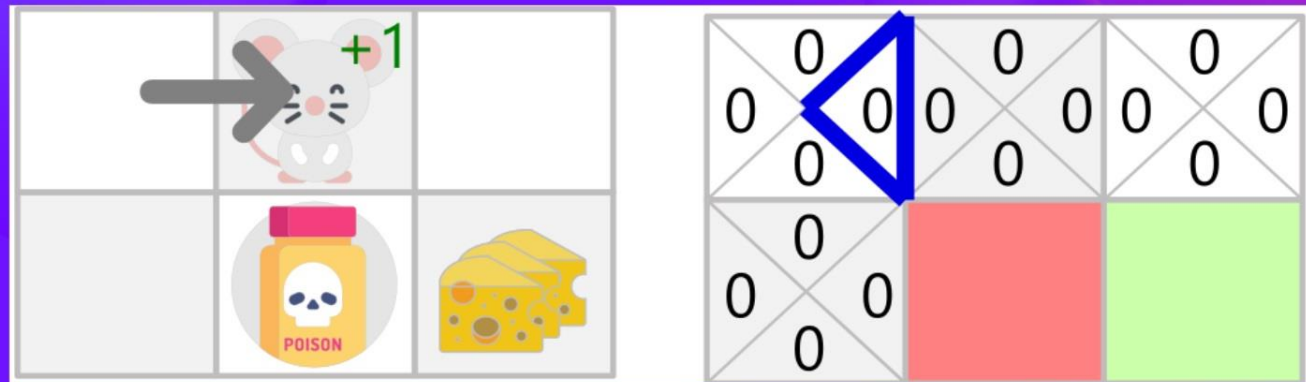
Training timestep 1:

Step 2: Choose an action using the Epsilon Greedy Strategy

Because epsilon is big (= 1.0), I take a random action. In this case, I go right.

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Q-Learning example

Training timestep 1:

Step 3: Perform action A_t , get R_{t+1} and S_{t+1}

By going right, I
get a small
cheese, so
 $R_{t+1} = 1$ and I'm
in a new state.

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Q-Learning example

Training timestep 1:

Step 4: Update $Q(S_t, A_t)$

We can now update $Q(S_t, A_t)$ using our formula.

Example, Step 4

$$\underbrace{Q(S_t, A_t)}_{\text{New Q-value estimation}} \leftarrow \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}} + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R_{t+1}}_{\text{Immediate Reward}} + \underbrace{\gamma \max_a Q(S_{t+1}, a)}_{\text{Discounted Estimate optimal Q-value of next state}} - \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}}]$$

New
Q-value
estimation

Former
Q-value
estimation

Learning
Rate

Immediate
Reward

Discounted Estimate
optimal Q-value
of next state

Former
Q-value
estimation

TD Target

TD Error

Update our Q-value estimation

Q-Learning example

Training timestep 1:

Step 4: Update $Q(S_t, A_t)$







We can now update $Q(S_t, A_t)$ using our formula.

Example, Step 4

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

$$Q(\text{Initial state, Right}) = 0 + 0.1 * [1 + 0.99 * 0 - 0]$$

$$Q(\text{Initial state, Right}) = 0.1$$

	←	→	↑	↓
	0	0.1	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Q-Learning example

Training timestep 2:

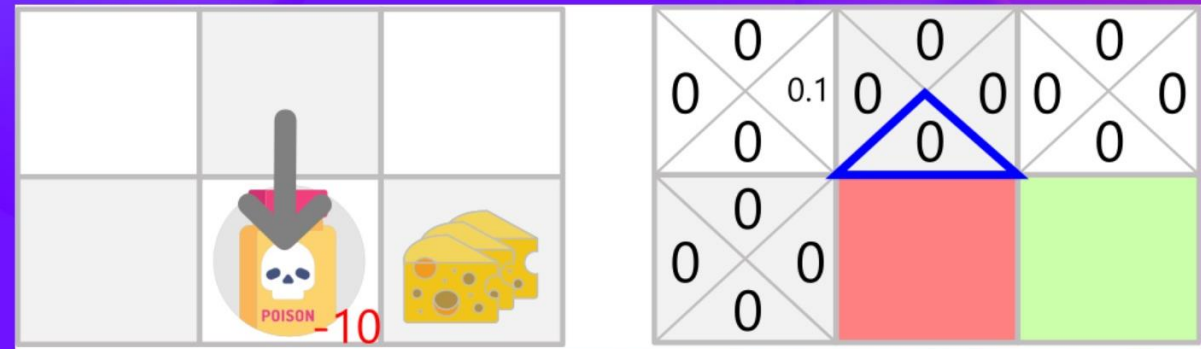
Step 2: Choose an action using the Epsilon Greedy Strategy

We take a random action again, since $\epsilon = 0.99$ is big. (Notice we decay epsilon a little bit because, as the training progress, we want less and less exploration).

I took the action 'down'. **This is not a good action since it leads me to the poison.**

Example, Step 2

Choose action A_t using policy derived from Q (e.g., ϵ -greedy)



We took a random action (exploration)

Q-Learning example

Training timestep 2:

Because we ate poison, I get $R_{t+1} = -10$, and we die.

Step 3: Perform action A_t , get R_{t+1} and S_{t+1}

Example, Step 3

Take action A_t and observe R_{t+1}, S_{t+1}



Q-Learning example

Training timestep 2:

We can now update $Q(S_t, A_t)$ using our formula.







Step 4: Update $Q(S_t, A_t)$

Example, Step 4

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

$$Q(\text{State 2, Down}) = 0 + 0.1 * [-10 + 0.99 * 0 - 0]$$

$$Q(\text{State 2, Down}) = -1$$

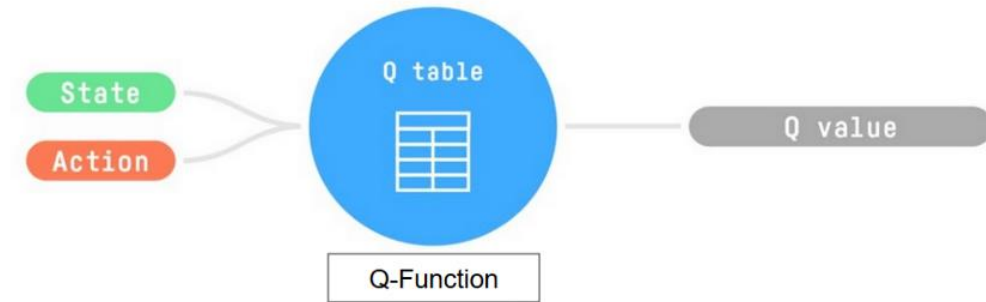
	←	→	↑	↓
	0	0.1	0	0
	0	0	0	-1
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Q-Learning example

Because we're dead, we start a new episode. But what we see here is that, **with two explorations steps, my agent became smarter.**

As we continue exploring and exploiting the environment and updating Q-values using the TD target, the **Q-table will give us a better and better approximation.** **At the end of the training, we'll get an estimate of the optimal Q-function.**

Q-Learning Recap



Q-Learning is the RL algorithm that :

- Trains a *Q-function*, an **action-value function** encoded, in internal memory, by a *Q-table* **containing all the state-action pair values**.
- Given a state and action, our Q-function **will search its Q-table for the corresponding value**.
- When the training is done, **we have an optimal Q-function, or, equivalently, an optimal Q-table**.
- And if we **have an optimal Q-function**, we have an optimal policy, since we **know, for each state, the best action to take**.

Q-Learning Recap

But, in the beginning, our **Q-table** is **useless since it gives arbitrary values for each state-action pair** (most of the time we initialize the Q-table to 0 values).

However, as we explore the environment and update our Q-table, we will see a better approximation.

The link between Value and Policy:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

Finding an optimal value function leads to having an optimal policy.

Q-Learning

	←	→	↑	↓
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

→
Training

	←	→	↑	↓
	0	10.8	0	0
	0	9.9	0	-10
	0	0	0	10
	0	-10	0	0
	0	0	0	0
	0	0	0	0

Q-Learning Recap

Strategies to find the optimal policy

Policy-based methods. The policy is usually trained with a neural network to select what action to take given a state. In this case it is the neural network which outputs the action that the agent should take instead of using a value function. Depending on the experience received by the environment, the neural network will be re-adjusted and will provide better actions.

Value-based methods. In this case, a value function is trained to output the value of a state or a state-action pair that will represent our policy. However, this value doesn't define what action the agent should take. In contrast, we need to specify the behavior of the agent given the output of the value function. For example, we could decide to adopt a policy to take the action that always leads to the biggest reward (Greedy Policy). In summary, the policy is a Greedy Policy (or whatever decision the user takes) that uses the values of the value-function to decide the actions to take.

Q-Learning Recap

Among the value-based methods, we can find two main strategies

- **The state-value function.** For each state, the state-value function is the expected return if the agent starts in that state and follows the policy until the end.
- **The action-value function.** In contrast to the state-value function, the action-value calculates for each state and action pair the expected return if the agent starts in that state, takes that action, and then follows the policy forever after.

Q-Learning Recap

Epsilon-greedy strategy:

- A common strategy used in reinforcement learning that involves balancing exploration and exploitation.
- Chooses the action with the highest expected reward with a probability of $1 - \epsilon$.
- Chooses a random action with a probability of ϵ .
- Epsilon typically decreases over time to shift focus towards exploitation.

Greedy strategy:

- Involves always choosing the action that is expected to lead to the highest reward, based on the current knowledge of the environment. (Only exploitation)
- Always choose the action with the highest expected reward.
- Does not include any exploration.
- Can be disadvantageous in environments with uncertainty or unknown optimal actions.

Q-Learning Recap

Off-policy vs on-policy algorithms

- **Off-policy algorithms:** A different policy is used at training time and inference time
- **On-policy algorithms:** The same policy is used during training and inference

Monte Carlo and Temporal Difference learning strategies

- **Monte Carlo (MC):** Learning at the end of the episode. With Monte Carlo, we wait until the episode ends and then we update the value function (or policy function) from a complete episode.
- **Temporal Difference (TD):** Learning at each step. With Temporal Difference Learning, we update the value function (or policy function) at each step without requiring a complete episode.