

# Return

When running a policy in an MDP, the sequence of state and action ( $S_0, A_0, S_1, A_1, \dots$ ) is called **trajectory** or **rollout**, and is denoted by  $\tau$ . In each trajectory, a sequence of rewards will be collected as a result of the actions. A function of these rewards is called **return** and in its most simplified version, it is defined as follows:

$$G(\tau) = r_0 + r_1 + r_2 + \dots = \sum_{t=0}^{\infty} r_t \quad (3.1)$$

At this point, the return can be analyzed separately for trajectories with infinite and finite horizons. This distinction is needed because in the case of interactions within an environment that do not terminate, the sum previously presented will always have an infinite value. This situation is dangerous because it doesn't provide any information. Such tasks are called continuing tasks and need another formulation of the reward. The best solution is to give more weight to the short-term rewards while giving less importance to those in the distant future. This is accomplished by using a value between 0 and 1 called the **discount factor** denoted with the symbol  $\lambda$ . Thus, the return  $G$  can be reformulated as follows:

$$G(\tau) = r_0 + \lambda r_1 + \lambda^2 r_2 + \dots = \sum_{t=0}^{\infty} \lambda^t r_t \quad (3.2)$$

This formula can be viewed as a way to prefer actions that are closer in time with respect to those that will be encountered in the distant future. Take this example—you win the lottery and you can decide when you would like to collect the prize. You would probably prefer to collect it within a few days rather than in a few years.  $\lambda$  is the value that defines how long you are willing to wait to collect the prize. If  $\lambda = 1$ , that means that you are not bothered about when you collect the prize. If  $\lambda = 0$ , that means that you want it immediately.

# Elements of RL

As we know, an agent interacts with their environment by the means of actions. This will cause the environment to change and to feedback to the agent a reward that is proportional to the quality of the actions and the new state of the agent. Through trial and error, the agent incrementally learns the best action to take in every situation so that, in the long run, it will achieve a bigger cumulative reward. In the RL framework, the choice of the action in a particular state is done by a **policy**, and the cumulative reward that is achievable from that state is called the **value function**. In brief, if an agent wants to behave optimally, then in every situation, the policy has to select the action that will bring it to the next state with the highest value. Now, let's take a deeper look at these fundamental concepts.

## Policy

The policy defines how the agent selects an action given a state. The policy chooses the action that maximizes the cumulative reward from that state, not with the bigger immediate reward. It takes care of looking for the long-term goal of the agent. For example, if a car has another 30 km to go before reaching its destination, but only has another 10 km of autonomy left and the next gas stations are 1 km and 60 km away, then the policy will choose to get fuel at the first gas station (1 km away) in order to not run out of gas. This decision is not optimal in the immediate future as it will take some time to refuel, but it will be sure to ultimately accomplish the goal.

An important distinction is between stochastic policies and deterministic policies. In the deterministic case, the policy provides a single deterministic action to take. On the other hand, in the stochastic case, the policy provides a probability for each action. The concept of the probability of an action is useful because it takes into consideration the dynamicity of the environment and helps its exploration.

RO

You

off policy, on policy

ChatGPT

"Off-policy" and "on-policy" are terminologies commonly used in the context of reinforcement learning, a type of machine learning where an agent learns to interact with an environment in order to maximize some notion of cumulative reward.

1. **On-policy learning:** In on-policy learning, the agent learns from the current policy that it is following. This means that the agent evaluates and improves its policy based on the data collected while executing that policy. The policy being updated and the policy being executed are the same. It's like learning from firsthand experience.
2. **Off-policy learning:** In contrast, off-policy learning refers to a scenario where the agent learns from a different policy than the one it uses to interact with the environment. This allows the agent to learn from data generated by following any policy, not necessarily the one it's currently following. Off-policy learning is particularly useful in situations where exploration is expensive or risky, as it allows the agent to learn from historical data or from a behavior policy while still following a different target policy.

These concepts are fundamental in reinforcement learning algorithms such as Q-learning (off-policy) and SARSA (on-policy). Each has its own advantages and use cases depending on the specific problem being addressed.

why is q-learning off policy?

ChatGPT

Q-learning is considered an off-policy reinforcement learning algorithm because it learns the value of the optimal policy (the policy that maximizes expected cumulative reward) independently of the agent's behavior policy.

In Q-learning, the agent learns an action-value function (Q-function), which estimates the expected cumulative reward of taking a particular action in a given state and following the optimal policy thereafter. The key to Q-learning being off-policy lies in its update rule, which uses the maximum Q-value of the next state regardless of the action actually taken. This update rule allows Q-learning to learn from experiences gathered by following any exploratory behavior policy, making it off-policy.

Here's a simplified version of the Q-learning update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

In this update rule:

In this update rule:

- $Q(s_t, a_t)$  is the current estimate of the Q-value for taking action  $a_t$  in state  $s_t$ .
- $r_{t+1}$  is the immediate reward received after taking action  $a_t$  in state  $s_t$ , transitioning to state  $s_{t+1}$ .
- $\gamma$
- $\gamma$  is the discount factor determining the importance of future rewards.
- $\max_a Q(s_{t+1}, a)$  represents the maximum Q-value of all possible actions in the next state  $s_{t+1}$ .

Notice that the update is based on the maximum Q-value of the next state, regardless of the action the agent actually takes. This characteristic allows Q-learning to learn from past experiences regardless of the behavior policy used to collect those experiences, making it off-policy.

To better understand these two concepts, we can think of someone who has to learn a new skill. If the person behaves as on-policy algorithms do, then every time they try a sequence of actions, they'll change their belief and behavior in accordance with the reward accumulated. In comparison, if the person behaves as an off-policy algorithm, they (the target policy) can also learn by looking at an old video of themselves (the behavior policy) doing the same skill—that is, they can use old experiences to help them to improve.

---

[ 17 ]

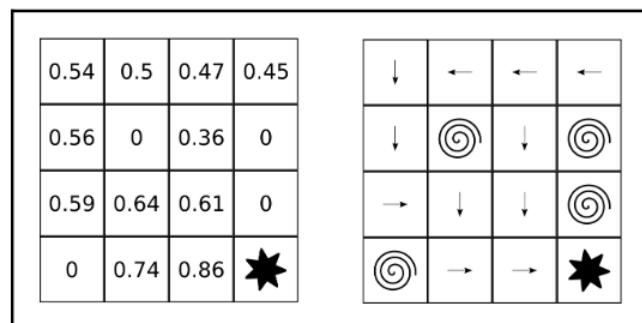
---

The **policy-gradient method** is a family of RL algorithms that learns a parametrized policy (as a deep neural network) directly from the gradient of the performance with respect to the policy. These algorithms have many advantages, including the ability to deal with continuous actions and explore the environment with different levels of granularity. They will be presented in greater detail in Chapter 6, *Learning Stochastic and PG Optimization*, Chapter 7, *TRPO and PPO Implementation*, and Chapter 8, *DDPG and TD3 Applications*.

# The value function

The **value function** represents the long-term quality of a state. This is the cumulative reward that is expected in the future if the agent starts from a given state. If the reward measures the immediate performance, the value function measures the performance in the long run. This means that a high reward doesn't imply a high-value function and a low reward doesn't imply a low-value function.

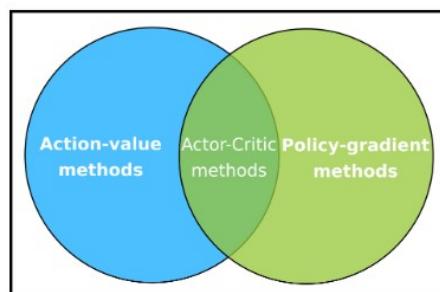
Moreover, the value function can be a function of the state or of the state-action pair. The former case is called a **state-value function**, while the latter is called an **action-value function**:



Here, the diagram shows the final state values (on the left side) and the corresponding optimal policy (on the right side).

Using the same gridworld example used to illustrate the concept of policy, we can show the state-value function. First of all, we can assume a reward of 0 in each situation except for when the agent reaches the star, gaining a reward of +1. Moreover, let's assume that a strong wind moves the agent in another direction with a probability of 0.33. In this case, the state values will be similar to those shown in the left-hand side of the preceding diagram. An optimal policy will choose the actions that will bring it to the next state with the highest state value, as shown in the right-hand side of the preceding diagram.

**Action-value methods** (or value-function methods) are the other big family of RL algorithms. These methods learn an action-value function and use it to choose the actions to take. Starting from *Chapter 3, Solving Problems with Dynamic Programming*, you'll learn more about these algorithms. It's worth noting that some policy-gradient methods, in order to combine the advantages of both methods, can also use a value function to learn the appropriate policy. These methods are called **actor-critic methods**. The following diagram shows the three main families of RL algorithms:



# Reward

At each timestep, that is, after each move of the agent, the environment sends back a number that indicates how good that action was to the agent. This is called a **reward**. As we have already mentioned, the end goal of the agent is to maximize the cumulative reward obtained during their interaction with the environment.

In literature, the reward is assumed to be a part of the environment, but that's not strictly true in reality. The reward can come from the agent too, but never from the decision-making part of it. For this reason and to simplify the formulation, the reward is always sent from the environment.

The reward is the only supervision signal injected into the RL cycle and it is essential to design the reward in the correct way in order to obtain an agent with good behavior. If the reward has some flaws, the agent may find them and follow incorrect behavior. For example, *Coast Runners* is a boat-racing game with the goal being to finish ahead of other players. During the route, the boats are rewarded for hitting targets. Some folks at OpenAI trained an agent with RL to play it. They found that, instead of running to the finish line as fast as possible, the trained boat was driving in a circle to capture re-populating targets while crashing and catching fire. In this way, the boat found a way to maximize the total reward without acting as expected. This behavior was due to an incorrect balance between short-term and long-term rewards.

/

The reward can appear with different frequencies depending on the environment. A frequent reward is called a **dense reward**; however, if it is seen only a few times during a game, or only at its end, it is called a **sparse reward**. In the latter case, it could be very difficult for an agent to catch the reward and find the optimal actions.

**Imitation learning** and **inverse RL** are two powerful techniques that deal with the absence of a reward in the environment. Imitation learning uses an expert demonstration to map states to actions. On the other hand, inverse RL deduces the reward function from an expert optimal behavior. Imitation learning and inverse RL will be studied in [Chapter 10, \*Imitation Learning with the DAgger Algorithm\*](#).

## Model

The model is an optional component of the agent, meaning that it is not required in order to find a policy for the environment. The model details how the environment behaves, predicting the next state and the reward, given a state and an action. If the model is known, planning algorithms can be used to interact with the model and recommend future actions. For example, in environments with discrete actions, potential trajectories can be simulated using look ahead searches (for instance, using the Monte Carlo tree search).

The model of the environment could either be given in advance or learned through interactions with it. If the environment is complex, it's a good idea to approximate it using deep neural networks. RL algorithms that use an already known model of the environment, or learn one, are called **model-based methods**. These solutions are opposed to model-free methods and will be explained in more detail in [Chapter 9, \*Model-Based RL\*](#).

A breakthrough in games occurred when, in 2015, AlphaGo beat Lee Sedol in the ancient game of Go. This win occurred in spite of the prediction that it wouldn't. At the time, it was thought that no computer would be able to beat an expert in Go for the next 10 years. AlphaGo used both RL and supervised learning to learn from professional human games. A few years after that match, the next version, named AlphaGo Zero, beat AlphaGo 100 games to 0. AlphaGo Zero learned to play Go in only three days through self-play.

An MDP expresses the problem of sequential decision-making, where actions influence the next states and the results. MDPs are general and flexible enough to provide a formalization of the problem of learning a goal through interactions, the same problem that is addressed with RL. Thus we can express and reason with RL problems in terms of MDPs.

An MDP is four-tuple  $(S, A, P, R)$ :

- $S$  is the state space, with a finite set of states.
- $A$  is the action space, with a finite set of actions.
- $P$  is a transition function, which defines the probability of reaching a state,  $s'$ , from  $s$  through an action,  $a$ . In  $P(s', s, a) = p(s'|s, a)$ , the transition function is equal to the conditional probability of  $s'$  given  $s$  and  $a$ .
- $R$  is the reward function, which determines the value received for transitioning to state  $s'$  after taking action  $a$  from state  $s$ .

By definition, the transition function and the reward function are determined only by the current state, and not from the sequence of the previous states visited. This property is called the **Markov property**, which means that the process is memory-less and the future state depends only on the current one, and not on its history. Thus, a state holds all the information. A system with such a property is called **fully observable**.

In many practical RL cases, the Markov property does not hold up, and for practicality, we can get around the problem by assuming it is an MDP and using a finite number of previous states (a finite history):  $S_t, S_{t-1}, S_{t-2}, \dots, S_{t-k}$ . In this case, the system is **partially observable** and the states are called **observations**. We'll use this strategy in the Atari games, where we'll use row pixels as the input of the agent. This is because the single frame is static and does not carry information about the speed or direction of the objects. Instead, these values can be retrieved using the previous three or four frames (it is still an approximation).

The final objective of an MDP is to find a policy,  $\pi$ , that maximizes the cumulative reward,

$$\sum_{t=0}^{\infty} R_{\pi}(s_t, s_{t+1}),$$

where  $R_{\pi}$  is the reward obtained at each step by following the policy,  $\pi$ .

A solution of an MDP is found when a policy takes the best possible action in each state of the MDP. This policy is known as the **optimal policy**.

# Policy

The policy chooses the actions to be taken in a given situation and can be categorized as deterministic or stochastic.

A deterministic policy is denoted as  $a_t = \mu(st)$ , while a stochastic policy can be denoted as  $a_t \sim \pi(\cdot | s_t)$ , where the tilde symbol ( $\sim$ ) means **has distribution**. Stochastic policies are used when it is better to consider an action distribution; for example, when it is preferable to inject a noisy action into the system.

Generally, stochastic policies can be categorical or Gaussian. The former case is similar to a classification problem and is computed as a softmax function across the categories. In the latter case, the actions are sampled from a Gaussian distribution, described by a mean and a standard deviation (or variance). These parameters can also be functions of states.

---

[ 53 ]

---

When using parameterized policies, we'll define them with the letter  $\theta$ . For example, in the case of a deterministic policy, it would be written as  $\mu_\theta(s_t)$ .

Policy, decision-maker, and agent are three terms that express the same concept, so, in this book, we'll use these terms interchangeably.



# Return

When running a policy in an MDP, the sequence of state and action ( $S_0, A_0, S_1, A_1, \dots$ ) is called **trajectory** or **rollout**, and is denoted by  $\tau$ . In each trajectory, a sequence of rewards will be collected as a result of the actions. A function of these rewards is called **return** and in its most simplified version, it is defined as follows:

$$G(\tau) = r_0 + r_1 + r_2 + \dots = \sum_{t=0}^{\infty} r_t \quad (3.1)$$

At this point, the return can be analyzed separately for trajectories with infinite and finite horizons. This distinction is needed because in the case of interactions within an environment that do not terminate, the sum previously presented will always have an infinite value. This situation is dangerous because it doesn't provide any information. Such tasks are called continuing tasks and need another formulation of the reward. The best solution is to give more weight to the short-term rewards while giving less importance to those in the distant future. This is accomplished by using a value between 0 and 1 called the **discount factor** denoted with the symbol  $\lambda$ . Thus, the return  $G$  can be reformulated as follows:

$$G(\tau) = r_0 + \lambda r_1 + \lambda^2 r_2 + \dots = \sum_{t=0}^{\infty} \lambda^t r_t \quad (3.2)$$

This formula can be viewed as a way to prefer actions that are closer in time with respect to those that will be encountered in the distant future. Take this example—you win the lottery and you can decide when you would like to collect the prize. You would probably prefer to collect it within a few days rather than in a few years.  $\lambda$  is the value that defines how long you are willing to wait to collect the prize. If  $\lambda = 1$ , that means that you are not bothered about when you collect the prize. If  $\lambda = 0$ , that means that you want it immediately.

RO

You

why do we need discount factor

RO

ChatGPT

The discount factor serves several important purposes in reinforcement learning (RL):

1. **Temporal Clarity:** RL often involves making decisions that have long-term consequences. The discount factor allows the agent to balance immediate rewards against future rewards. This helps provide clarity on the timing of rewards, ensuring that the agent understands the importance of rewards received at different time steps.
2. **Encouraging Convergence:** By discounting future rewards, the discount factor encourages the agent to converge on a policy or value function that maximizes the expected return over time. This encourages the agent to prioritize actions that lead to greater long-term rewards rather than focusing solely on immediate rewards.
3. **Handling Infinite Horizons:** In many RL problems, the number of time steps or episodes can be infinite. Without discounting, the sum of rewards may diverge, making it difficult to compute or reason about the agent's behavior. Discounting ensures that the sum of rewards converges to a finite value, facilitating computation and analysis.
4. **Modeling Uncertainty:** The discount factor can also be interpreted as a way to model uncertainty about the future. A lower discount factor implies greater uncertainty or impatience about future outcomes, while a higher discount factor implies more confidence or patience.

Overall, the discount factor is a crucial parameter in RL that helps agents effectively trade off immediate rewards against future rewards, facilitates convergence to optimal policies, and enables the handling of infinite time horizons. ↴

## Value functions

The return  $G(\tau)$  provides a good insight into the trajectory's value, but still, it doesn't give any indication of the quality of the single states visited. This quality indicator is important because it can be used by the policy to choose the next best action. The policy has to just choose the action that will result in the next state with the highest quality. The **value function** does exactly this: it estimates the **quality** in terms of the expected return from a state following a policy. Formally, the value function is defined as follows:

$$V_\pi(s) = E_\pi[G|s_0 = s] = E_\pi\left[\sum_{t=0}^k \lambda^t r_t | s_0 = s\right]$$

The **action-value function**, similar to the value function, is the expected return from a state but is also conditioned on the first action. It is defined as follows:

$$Q_\pi(s, a) = E_\pi[G|s_0 = s, a_0 = a] = E_\pi\left[\sum_{t=0}^k \lambda^t r_t | s_0 = s, a_0 = a\right]$$

The value function and action-value function are also called the **V-function** and **Q-function** respectively, and are strictly correlated with each other since the value function can also be defined in terms of the action-value function:

$$V_\pi(s) = E_\pi[Q_\pi(s, a)]$$

Knowing the optimal  $Q^*$ , the optimal value function is as follows:

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

That's because the optimal action is  $a^*(s) = \text{argmax}_a Q^*(s, a)$ .

The formula  $V(s) = \mathbb{E}[G_t | S_t = s]$  represents the state value function in reinforcement learning. Let's break down this formula:

- $V(s)$  represents the value of state  $s$ . It denotes the expected cumulative reward that an agent can achieve from state  $s$  onwards, following a certain policy.
- $\mathbb{E}[G_t | S_t = s]$  denotes the expected value of the return  $G_t$  given that the agent is in state  $s$  at time step  $t$ . This expectation is taken over all possible future returns starting from state  $s$ .

Now, let's break down the components:

- $G_t$  represents the return at time step  $t$ , which is the sum of rewards obtained from time step  $t$  onwards. It's calculated as  $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ , where  $R_{t+k+1}$  is the reward received at time step  $t + k + 1$ , and  $\gamma$  is the discount factor.
- $S_t = s$  specifies that the agent is in state  $s$  at time step  $t$ . This condition restricts the expectation to only consider returns when the agent is in state  $s$ .
- $\mathbb{E}[\cdot]$  denotes the expected value, which is the weighted average of all possible values of the random variable inside the square brackets, where the weights are the probabilities of the corresponding events.

Therefore,  $V(s) = \mathbb{E}[G_t | S_t = s]$  calculates the expected value of the return starting from state  $s$ , representing the value of being in that state under the given policy. It's a key concept in reinforcement learning, as it helps the agent evaluate and make decisions based on the potential rewards it can expect to receive from each state.

RO

You

not of discount factor. give me a simple numerical example of calculating the value function

ChatGPT

Let's consider a simple gridworld with 4 states arranged in a  $2 \times 2$  grid:

markdown

Copy code

```
-----  
| 1 | 2 |  
| - - | - - |  
| 3 | 4 |  
-----
```

Assume that each state transitions to its adjacent states deterministically:

- From state 1, you can move right to state 2 or down to state 3.
- From state 2, you can move left to state 1 or down to state 4.
- From state 3, you can move right to state 4 or up to state 1.
- From state 4, you can move left to state 3 or up to state 2.

Let's also assume that all transitions result in a reward of -1, except for transitions to state 4, which has a reward of +10.

Now, let's calculate the state values using dynamic programming, specifically the Bellman equation:



$$V(s) = R(s) + \sum_{a \in A} P(a|s)V(s')$$

Now, let's calculate the state values using dynamic programming, specifically the Bellman equation:

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s)V(s')$$

where:

- $V(s)$  is the value of state  $s$ .
- $R(s)$  is the immediate reward upon transitioning to state  $s$ .
- $\gamma$  is the discount factor (let's assume  $\gamma = 0.9$  for this example).
- $P(s'|s)$  is the probability of transitioning to state  $s'$  given that the agent is in state  $s$ .

Given our setup:

- $R(4) = 10$
- $R(1) = R(2) = R(3) = -1$
- $\gamma = 0.9$

We can calculate the value of each state using the Bellman equation:

1. For state 1:

$$V(1) = -1 + 0.9 \times (0.5 \times V(2) + 0.5 \times V(3))$$

$$V(1) = -1 + 0.9 \times (0.5 \times 0 + 0.5 \times 0) = -1$$

2. For state 2:

$$V(2) = -1 + 0.9 \times (0.5 \times V(1) + 0.5 \times V(4))$$

$$V(2) = -1 + 0.9 \times (0.5 \times (-1) + 0.5 \times 10) = 4.5$$

3. For state 3:

$$V(3) = -1 + 0.9 \times (0.5 \times V(1) + 0.5 \times V(4))$$

3. For state 3:

$$V(3) = -1 + 0.9 \times (0.5 \times V(1) + 0.5 \times V(4))$$

$$V(3) = -1 + 0.9 \times (0.5 \times (-1) + 0.5 \times 10) = 4.5$$

4. For state 4:

$$V(4) = 10 + 0.9 \times (0.5 \times V(2) + 0.5 \times V(3))$$

$$V(4) = 10 + 0.9 \times (0.5 \times 4.5 + 0.5 \times 4.5) = 14.05$$

So, the calculated state values are:

- $V(1) = -1$
- $V(2) = V(3) = 4.5$
- $V(4) = 14.05$

These values represent the expected cumulative rewards from each state onwards under the given policy and transition dynamics.

## Bellman equation

$\mathbf{V}$  and  $\mathbf{Q}$  can be estimated by running trajectories that follow the policy,  $\pi$ , and then averaging the values obtained. This technique is effective and is used in many contexts, but is very expensive considering that the return requires the rewards from the full trajectory.

Luckily, the Bellman equation defines the action-value function and the value function recursively, enabling their estimations from subsequent states. The Bellman equation does that by using the reward obtained in the present state and the value of its successor state. We already saw the recursive formulation of the return (in formula (5)) and we can apply it to the state value:

$$\begin{aligned} V_\pi(s) &= E_\pi[G_t | s_0 = s] = E_\pi[r_t + \gamma G_{t+1} | s_0 = s] \\ &= E_\pi[r_t + \gamma V_\pi(s_{t+1})] | s_t = s, a_t \sim \pi(s_t)] \end{aligned} \quad (3.6)$$

Similarly, we can adapt the Bellman equation for the action-value function:

$$\begin{aligned} Q_\pi(s, a) &= E_\pi[G_t | s_t = s, a_t = a] \\ &= E_\pi[r_t + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= E_\pi[r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})] | s_t = s, a_t = a \end{aligned} \quad (3.7)$$

Now, with (6) and (7),  $V_\pi$  and  $Q_\pi$  are updated only with the values of the successive states, without the need to unroll the trajectory to the end, as required in the old definition.



## ChatGPT

The Bellman equation is a fundamental concept in dynamic programming and reinforcement learning. It describes the relationship between the value of a state (or state-action pair) and the values of its successor states (or successor state-action pairs) under an optimal policy. The Bellman equation forms the basis for many algorithms used to solve reinforcement learning problems.

There are two main forms of the Bellman equation:

1. **State Value Bellman Equation (for deterministic policies):**

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s)V(s')$$

Where:

- $V(s)$  is the value of state  $s$ .
- $R(s)$  is the immediate reward upon transitioning to state  $s$ .
- $P(s'|s)$  is the probability of transitioning to state  $s'$  given that the agent is in state  $s$ .
- $\gamma$  is the discount factor, which balances immediate and future rewards.

This equation states that the value of a state is the immediate reward obtained upon entering the state plus the discounted value of the possible next states, weighted by their transition probabilities.

### 1. Action Value Bellman Equation (for deterministic policies):

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a)V(s')$$

Where:

- $Q(s, a)$  is the value of taking action  $a$  in state  $s$ .
- $R(s)$  is the immediate reward upon taking action  $a$  and transitioning to state  $s$ .
- $P(s'|s, a)$  is the probability of transitioning to state  $s'$  given that the agent is in state  $s$  and takes action  $a$ .
- $V(s')$  is the value of the successor state  $s'$ .

This equation states that the value of taking an action in a state is the immediate reward obtained upon taking that action plus the discounted value of the successor state, weighted by their transition probabilities.

These equations provide the foundation for solving reinforcement learning problems by iteratively updating the values of states (or state-action pairs) until convergence to the optimal values under the given policy. Various algorithms, such as value iteration, policy iteration, and Q-learning, leverage the Bellman equations to learn optimal policies in different settings.

```
def calculate_q_value(state, action):  
    next_state = transitions[state][action]  
    immediate_reward = rewards[next_state]  
    discounted_future_value = gamma * max(calculate_q_value(next_state, a) for a in range(n_actions))  
    q_value = immediate_reward + discounted_future_value  
    return q_value
```

The first distinction is between model-based and model-free algorithms. As the name suggests, the first requires a model of the environment, while the second is free from this condition. The model of the environment is highly valuable because it carries precious information that can be used to find the desired policies; however, in most cases, the model is almost impossible to obtain. For example, it can be quite easy to model the game tic-tac-toe, while it can be difficult to model the waves of the sea. To this end, model-free algorithms can learn information without any assumptions about the environment. A representation of the categories of RL algorithms is visible in figure 3.2.

Here the distinction is shown between model-based and model-free, and two widely known model-free approaches, namely policy gradient and value-based. Also, as we'll see in later chapters, a combination of those is possible:

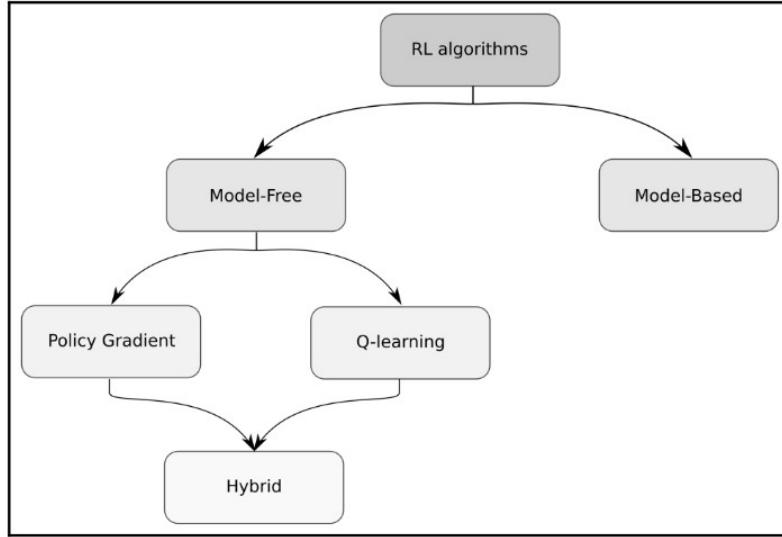


Figure 3.2. Categorization of RL algorithms

The first distinction is between model-free and model-based. Model-free RL algorithms can be further decomposed in policy gradient and value-based algorithms. Hybrids are methods that combine important characteristics of both methods.

## Model-free algorithms

In the absence of a model, **model-free (MF)** algorithms run trajectories within a given policy to gain experience and to improve the agent. MF algorithms are made up of three main steps that are repeated until a good policy is created:

1. The generation of new samples by running the policy in the environment. The trajectories are run until a final state is reached or for a fixed number of steps.
2. The estimation of the return function.
3. The improvement of the policy using the samples collected, and the estimation done in step 2.

## Value-based algorithms

Value-based algorithms, also known as **value function algorithms**, use a paradigm that's very similar to the one we saw in the previous section. That is, they use the Bellman equation to learn the Q-function, which in turn is used to learn a policy. In the most common setting, they use deep neural networks as a function approximator and other tricks to deal with high variance and general instabilities. To a certain degree, value-based algorithms are closer to supervised regression algorithms.

Typically, these algorithms are off-policy, meaning they are not required to optimize the same policy that was used to generate the data. This means that these methods can learn from previous experience, as they can store the sampled data in a replay buffer. The ability to use previous samples makes the value function more sample-efficient than other model-free algorithms.



### ChatGPT

In value function algorithms, such as Q-learning and SARSA, it's common for them to be off-policy.

This means that these algorithms are not required to optimize the same policy that was used to generate the data. There are a few reasons why this is often the case:

1. **Flexibility:** Off-policy algorithms offer more flexibility because they can learn from data generated by any behavior policy, not just the target policy that they are trying to optimize. This flexibility allows them to learn from diverse sets of experiences and potentially explore more efficiently.
2. **Data Reusability:** Off-policy algorithms can reuse data collected from any policy, making them more sample-efficient. This is particularly advantageous in scenarios where collecting new data is costly or time-consuming.
3. **Stability:** Off-policy algorithms often exhibit better stability during learning compared to on-policy algorithms. Since they are not directly affected by the changes in the behavior policy, they can maintain a stable estimate of the value function even when the behavior policy changes.
4. **Decoupling Exploration and Exploitation:** Off-policy algorithms decouple the exploration (behavior policy) from the exploitation (target policy) phases of learning. This allows them to explore more freely without being restricted by the target policy, which can lead to better exploration strategies.
5. **Value Estimation:** Off-policy algorithms can estimate the value of different policies, not just the one used for exploration. This is valuable for evaluating alternative policies or for learning from historical data collected by different agents or policies.

## Policy gradient algorithms

The other family of MF algorithms is that of the policy gradient methods (or policy optimization methods). They have a more direct and obvious interpretation of the RL problem, as they learn directly from a parametric policy by updating the parameters in the direction of the improvements. It's based on the RL principle that good actions should be encouraged (by boosting the gradient of the policy upward) while discouraging bad actions.

Contrary to value function algorithms, policy optimization mainly requires on-policy data, making these algorithms more sample inefficient. Policy optimization methods can be quite unstable due to the fact that taking the steepest ascent in the presence of surfaces with high curvature can easily result in moving too far in any given direction, falling down into a bad region. To address this problem, many algorithms have been proposed, such as optimizing the policy only within a trust region, or optimizing a surrogate clipped objective function to limit changes to the policy.

A major advantage of policy gradient methods is that they easily handle environments with continuous action spaces. This is a very difficult thing to approach with value function algorithms as they learn Q-values for discrete pairs of states and actions.

## Actor-Critic algorithms

**Actor-Critic (AC)** algorithms are on-policy policy gradient algorithms that also learn a value function (generally a Q-function) called a critic to provide feedback to the policy, the actor. Imagine that you, the actor, want to go to the supermarket via a new route. Unfortunately, before arriving at the destination, your boss calls you requiring you to go back to work. Because you didn't reach the supermarket, you don't know if the new road is actually faster than the old one. But if you reached a familiar location, you can estimate the time you'll need to go from there to the supermarket and calculate whether the new path is preferable. This estimate is what the critic does. In this way, you can improve the actor even though you didn't reach the final goal.

Combining a critic with an actor has been shown to be very effective and is commonly used in policy gradient algorithms. This technique can also be combined with other ideas used in policy optimization, such as trust-region algorithms.

## Hybrid algorithms

Advantages of both value functions and policy gradient algorithms can be merged, creating hybrid algorithms that can be more sample efficient and robust.

Hybrid approaches combine Q-functions and policy gradients to symbiotically and mutually improve each other. These methods estimate the expected Q-function of deterministic actions to directly improve the policy.



ChatGPT

Sure, let's break down the concepts of on-policy and off-policy in simple terms:

1. **On-Policy:**

- Imagine you're learning how to cook a specific recipe by following it step by step. You're actively cooking according to the recipe's instructions, and you're also using feedback from your own cooking experience to adjust and improve your cooking technique.
- In reinforcement learning, being on-policy means that the agent is learning and improving its policy while directly interacting with the environment. The agent's learning is based on the actions it takes and the feedback it receives from those actions.

2. **Off-Policy:**

- Now, consider a scenario where you're watching someone else cook a different recipe. You're observing their cooking process and the outcomes they achieve, but you're not actively involved in the cooking yourself. You may learn from their experiences and mistakes, even though you're not following the same recipe.
- In reinforcement learning, being off-policy means that the agent is learning from data collected by another policy or agent. The agent may be learning from past experiences or data generated by a different behavior policy, without actively following that policy during learning.

In simpler terms:

- **On-policy:** Learning by doing. The agent learns while actively following its own strategy and experiencing the outcomes firsthand.
- **Off-policy:** Learning by observing. The agent learns from data collected by other strategies or agents, even if it's not directly following those strategies during learning.

RO

You

what is convergence in learning



ChatGPT

In learning, convergence refers to the process by which a learning algorithm gradually improves its performance over time and eventually reaches a stable or optimal state. It indicates that the learning algorithm has reached a point where further iterations or updates do not significantly improve its performance or behavior.

Convergence can manifest in different ways depending on the specific learning algorithm and problem domain:

1. **Parameter Convergence:** In many machine learning algorithms, convergence refers to the convergence of model parameters. This means that the values of the parameters (e.g., weights in a neural network) stabilize or converge to certain values as the algorithm iterates through the training data. In supervised learning, this often means minimizing a loss function or error metric.
2. **Policy Convergence:** In reinforcement learning, convergence can refer to the convergence of the agent's policy. The policy represents the agent's strategy for selecting actions in different states. Convergence occurs when the agent's policy stabilizes and reaches an optimal or near-optimal strategy for maximizing rewards in the environment.
3. **Value Convergence:** In reinforcement learning, convergence can also refer to the convergence of value functions. Value functions estimate the expected cumulative rewards or values of states (or state-action pairs). Convergence occurs when these value estimates stabilize and accurately represent the true values of states or state-action pairs under the current policy.



Convergence is an essential property of learning algorithms because it indicates that the

## MODEL-BASED RL

Having a model of the environment means that the state transitions and the rewards can be predicted for each state-action tuple (without any interaction with the real environment). As we already mentioned, the model is known only in limited cases, but when it is known, it can be used in many different ways. The most obvious application of the model is to use it to plan future actions. Planning is a concept used to express the organization of future moves when the consequences of the next actions are already known. For example, if you know exactly what moves your enemy will make, you can think ahead and plan all your actions before executing the first one. As a downside, planning can be very expensive and isn't a trivial process.

---

[ 60 ]

---

A model can also be learned through interactions with the environment, assimilating the consequences (both in terms of the states and rewards) of an action. This solution is not always the best one because teaching a model could be terribly expensive in the real world. Moreover, if only a rough approximation of the environment is understood by the model, it could lead to disastrous results.

A model, whether known or learned, can be used both to plan and to improve the policy, and can be integrated into different phases of an RL algorithm. Well-known cases of model-based RL involve pure planning, embedded planning to improve the policy, and generated samples from an approximate model.

By definition, the value function of a policy is the expected return (that is, the sum of discounted rewards) of that policy starting from a given state:

$$V_\pi(s) = E_\pi[G|s_0 = s]$$

Following the reasoning of Chapter 3, *Solving Problems with Dynamic Programming*, DP algorithms update state values by computing expectations for all the next states of their values:

$$V_{k+1}(s) = E_\pi[r_t + \gamma V_k(s_{t+1})|s_t = s] = \sum_a \pi(s, a) \sum_{s', r} p(s'|s, a)[r + \gamma V_k(s')]$$

Unfortunately, computing the value function means that you need to know the state transition probabilities. In fact, DP algorithms use the model of the environment to obtain those probabilities. But the major concern is what to do when it's not available. The best answer is to gain all the information by interacting with the environment. If done well, it works because by sampling from the environment a substantial number of times, you should be able to approximate the expectation and have a good estimation of the value function.

## Policy evaluation

We just saw how using real experience to estimate the value function is an easy process. It is about running the policy in an environment until a final state is reached, then computing the return value and averaging the sampled return, as can be seen in equation (1):

$$V(s_t) = \frac{1}{N} \sum_{i=0}^N (G_t^i) \quad (1)$$

Thus the expected return of a state can be approximated from the experience by averaging the sampling episodes from that state. The methods that estimate the return function using (1) are called **Monte Carlo methods**. Until all of the state-action pairs are visited and enough trajectory has been sampled, Monte Carlo methods guarantee convergence to the optimal policy.

# The exploration problem

How can we guarantee that every action of each state is chosen? And why is that so important? We will first answer the latter question, and then show how we can (at least in theory) explore the environment to reach every possible state.

## Why explore?

The trajectories are sampled following a policy that can be stochastic or deterministic. In the case of a deterministic policy, each time a trajectory is sampled, the visited states will always be the same, and the update of the value function will take into account only this limited set of states. This will considerably limit your knowledge about the environment. It is like learning from a teacher that never changes their opinion on a subject—you will be stuck with those ideas without learning about others.

Thus the exploration of the environment is crucial if you want to achieve good results, and it ensures that there are no better policies that could be found.

Thus the exploration of the environment is crucial if you want to achieve good results, and it ensures that there are no better policies that could be found.

On the other hand, if a policy is designed in such a way that it explores the environment constantly without taking into consideration what has already been learned, the achievement of a good policy is very difficult, perhaps even impossible. This balance between exploration and exploitation (behaving according to the best policy currently available) is called the exploration-exploitation dilemma and will be considered in greater detail in Chapter 12, *Developing an ESBAS Algorithm*.

## How to explore

A very effective method that can be used when dealing with such situations is called  $\epsilon$ -greedy exploration. It is about acting randomly with probability  $\epsilon$  while acting greedily (that means choosing the best action) with probability  $1 - \epsilon$ . For example, if  $\epsilon = 0.8$ , on average, for every 10 actions, the agent will act randomly 8 times.

To avoid exploring too much in later stages when the agent is confident about its knowledge,  $\epsilon$  can decrease over time. This strategy is called **epsilon-decay**. With this variation, an initial stochastic policy will gradually converge to a deterministic and, hopefully, optimal policy.

There are many other exploration techniques (such as Boltzmann exploration) that are more accurate, but they are also quite complicated, and for the purpose of this chapter,  $\epsilon$ -greedy is a perfect choice.

# TD learning

Monte Carlo methods are a powerful way to learn directly by sampling from the environment, but they have a big drawback—they rely on the full trajectory. They have to wait until the end of the episode, and only then can they update the state values. Therefore, a crucial factor is knowing what happens when the trajectory has no end, or if it's very long. The answer is that it will produce terrifying results. A similar solution to this problem has already come up in DP algorithms, where the state values are updated at each step, without waiting until the end. Instead of using the complete return accumulated during the trajectory, it just uses the immediate reward and the estimate of the next state value. A visual example of this update is given in figure 4.2 and shows the parts involved in a single step of learning. This technique is called **bootstrapping**, and it is not only useful for long or potentially infinite episodes, but for episodes of any length. The first reason for this is that it helps to decrease the variance of the expected return. The variance is decreased because the state values depend only on the immediate next reward and not on all the rewards of the trajectory. The second reason is that the learning process takes place at every step, making these algorithms learn online. For this reason, it is called one-step learning. In contrast, Monte Carlo methods are offline as they use the information only after the conclusion of the episode. Methods that learn online using bootstrapping are called TD learning methods.

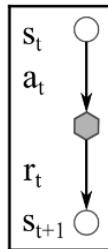


Figure 4.2. One-step learning update with bootstrapping

## TD update

From the previous chapter, *Solving Problems with Dynamic Programming* we know the following:

$$V_\pi(s) = E_\pi[G_t | s_t = s] \quad (2)$$

Empirically, the Monte Carlo update estimates this value by averaging returns from multiple full trajectories. Developing the equation further, we obtain the following:

$$\begin{aligned} &E_\pi[G_t | s_t = s] \\ &= E_\pi[r_t + \gamma G_{t+1} | s_t = s] \\ &= E_\pi[r_t + \gamma V_\pi(s_{t+1})] | s_t = s \end{aligned} \quad (3)$$

The preceding equation is approximated by the DP algorithms. The difference is that TD algorithms estimate the expected value instead of computing it. The estimate is done in the same way as Monte Carlo methods do, by averaging:

$$E_\pi[r_t + \gamma V_\pi(s_{t+1})] | s_t = s \approx \frac{1}{N} \sum_{i=0}^N \pi[r_t^i + \gamma V_\pi(s_{t+1}^i)] | s_t = s$$

In practice, instead of calculating the average, the TD update is carried out by improving the state value by a small amount toward the optimal value:

$$V(s_t) \leftarrow V(s_t) + \alpha[r + \gamma V(s_{t+1}) - V(s_t)] \quad (4)$$

$\alpha$  is a constant that establishes how much the state value should change at each update. If  $\alpha = 0$ , then the state value will not change at all. Instead, if  $\alpha = 1$ , the state value will be equal to  $r + \gamma V(s_{t+1})$  (called the **TD target**) and it will completely forget the previous value. In practice, we don't want these extreme cases, and usually  $\alpha$  ranges from 0.5 to 0.001.



ChatGPT

Q-learning is a reinforcement learning algorithm used to approximate action values (Q-values) in a Markov Decision Process (MDP). Q-values represent the expected return (cumulative future rewards) of taking a particular action in a given state and following an optimal policy thereafter.

Here's how Q-learning approximates Q-values:

1. **Initialization:** Initialize Q-values arbitrarily for all state-action pairs. Typically, Q-values are initialized to zero, but they can also be initialized randomly.
2. **Exploration vs. Exploitation:** Q-learning balances exploration (trying out different actions to learn about their values) and exploitation (taking the action that is believed to have the highest value based on current estimates).
3. **Action Selection:** At each time step, the agent selects an action according to an exploration-exploitation strategy, such as  $\epsilon$ -greedy. With probability  $\epsilon$ , the agent selects a random action (exploration), and with probability  $1-\epsilon$ , it selects the action with the highest Q-value for the current state (exploitation).
4. **Updating Q-values:** After selecting an action and observing the resulting reward and next state, the agent updates the Q-value of the previous state-action pair using the Q-learning update rule:  
$$Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Where:

- $Q(s, a)$  is the Q-value of taking action  $a$  in state  $s$ .
- $r$  is the immediate reward received after taking action  $a$  in state  $s$ .
- $s'$  is the next state observed after taking action  $a$ .
- $\gamma$  is the discount factor, representing the importance of future rewards.
- $\alpha$  is the learning rate, determining the step size of the update.

This update rule is derived from the Bellman equation for optimality and is a form of Temporal Difference (TD) learning.



5. **Convergence:** Through repeated interactions with the environment, Q-values converge towards

/

RO

You

can you give me a numerical example on `action = q_learning.select_action(state)`  
`next_state = transition_matrix[state, action]`  
`reward = reward_matrix[state, action]`  
`q_learning.update(state, action, next_state, reward)`  
state = next\_state with two iterations? elaborate on `q.learning.update`

ChatGPT

Certainly! Let's walk through two iterations of the Q-learning algorithm using the provided code snippet:

Suppose we have the following setup:

python

 Copy code

```
num_states = 4
num_actions = 2
transition_matrix = np.array([[1, 2],
                             [0, 3],
                             [1, 3],
                             [2, 0]])
reward_matrix = np.array([[0, 1],
                         [2, 3],
                         [4, 5],
                         [6, 7]])
```



And we initialize the Q-learning agent:

```
python
q_learning = QLearning(num_states, num_actions)
```

 Copy code

## Iteration 1:

### 1. Action Selection:

```
python
state = 0
action = q_learning.select_action(state)
# Suppose the agent selects action 1
action = 1
```

 Copy code

### 2. Transition to Next State:

```
python
next_state = transition_matrix[state, action]
next_state = transition_matrix[0, 1]
next_state = 2
```

 Copy code

### 3. Observe Reward:

```
python
reward = reward_matrix[state, action]
reward = reward_matrix[0, 1]    ↓
reward = 1
```

 Copy code

### 4. Update Q-value:

```
python
q_learning.update(state, action, next_state, reward)
```

 Copy code

Here's how the `update` method works:

```
python
max_next_q_value = np.max(q_learning.Q[2]) # Maximum Q-value of state 2
td_target = reward + q_learning.gamma * max_next_q_value
td_error = td_target - q_learning.Q[state, action]
q_learning.Q[state, action] += q_learning.alpha * td_error
```

 Copy code

 You

```
can you give me a numerical example on action = q_learning.select_action(state)
next_state = transition_matrix[state, action]
reward = reward_matrix[state, action]
q_learning.update(state, action, next_state, reward)
state = next_state with two iterations? elaborate on q.learning.update
```

 ChatGPT

Certainly! Let's walk through two iterations of the Q-learning algorithm using the provided code snippet:

Suppose we have the following setup:

python

 Copy code

```
num_states = 4
num_actions = 2
transition_matrix = np.array([[1, 2],
                             [0, 3],
                             [1, 3],
                             [2, 0]])
reward_matrix = np.array([[0, 1],
                         [2, 3],
                         [4, 5],
                         [6, 7]])
```



## Iteration 1:

### 1. Action Selection:

```
python  
  
state = 0  
action = q_learning.select_action(state)  
# Suppose the agent selects action 1  
action = 1
```

 Copy code

### 2. Transition to Next State:

```
python  
  
next_state = transition_matrix[state, action]  
next_state = transition_matrix[0, 1]  
next_state = 2
```

 Copy code

### 3. Observe Reward:

```
python  
  
reward = reward_matrix[state, action]  
reward = reward_matrix[0, 1]  
reward = 1
```

 Copy code

### 4. Update Q-value:

```
python  
  
q_learning.update(state, action, ↓ t_state, reward)
```

 Copy code

RO

You

the rule of learning rate on TD update and q-learning



ChatGPT

In both Temporal Difference (TD) learning and Q-learning, the learning rate (often denoted by  $\alpha$ ) plays a crucial role in controlling how much the estimated values are updated based on new information obtained from the environment. Let's delve into how the learning rate affects the update rules in TD learning and Q-learning:

### Temporal Difference (TD) Learning:

In TD learning, the update rule for estimating the value function involves adjusting the current estimate towards the target value, which is a combination of the immediate reward and the estimated value of the next state. The learning rate determines the step size of this adjustment.

Here's the TD update rule for state values:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (r + \gamma V(S_{t+1}) - V(S_t))$$

RO

You

so we first take action then update the q-value?

RO

ChatGPT

Yes, that's correct! In the Q-learning algorithm, the agent first selects an action based on its current policy, then observes the resulting reward and next state from the environment, and finally updates the Q-value of the previous state-action pair based on that observed reward and the estimated value of the next state.

Here's the typical sequence of steps in each iteration of Q-learning:

1. **Action Selection:** The agent selects an action according to its current policy. This action can be chosen using various strategies, such as  $\epsilon$ -greedy, softmax, or other exploration-exploitation techniques.
2. **Observation:** The agent executes the selected action in the environment, which results in a transition to the next state and the receipt of a reward.
3. **Q-Value Update:** Based on the observed reward and the estimated value of the next state, the agent updates the Q-value of the previous state-action pair using the Q-learning update rule.
4. **State Update:** The agent transitions to the next state and repeats the process for the next iteration.

This sequence ensures that the Q-values are updated iteratively based on the agent's experiences, gradually refining its action-value estimates towards the optimal values. By iteratively improving its Q-values, the agent can learn an optimal policy that maximizes long-term rewards in the environment.

## Role of Learning Rate:

1. **Convergence Speed vs. Stability:** A higher learning rate can lead to faster convergence but may also result in instability, oscillations, or divergence. Conversely, a lower learning rate may provide more stable updates but can slow down convergence.
2. **Exploration vs. Exploitation:** The learning rate influences the balance between exploration and exploitation in both TD learning and Q-learning. A higher learning rate allows the agent to update its estimates more aggressively, favoring exploitation, while a lower learning rate encourages more exploration by relying less on new information.
3. **Generalization:** The learning rate affects the generalization ability of the learned value function. Higher learning rates may lead to overfitting to the observed data, while lower learning rates can help prevent overfitting by smoothing out updates and allowing the model to generalize better to unseen states or actions.

## Monotone Convergence Theorem

---

Monotone Convergence Theorem is one of those mathematical gems that might seem a bit abstract at first, but once you grasp it, you'll see its beauty and its power.

Imagine you're on a journey, and with each step, you're either staying put or moving forward, never backward. You also know there is a boundary you can't cross. The Monotone Convergence Theorem is like a mathematical guarantee that you'll eventually reach your destination!

Let's break it down. The theorem applies to sequences of numbers. A sequence is just a list of numbers in a specific order. For example, the sequence of even numbers is 2, 4, 6, 8, and so on.

Now, a sequence is said to be monotonically increasing if each number in the sequence is greater than or equal to the one before it. It's like climbing a staircase where each step is the same height or higher than the previous one. You're either staying on the same step or moving up, but never down.

The Monotone Convergence Theorem states that if you have a monotonically increasing sequence that is also bounded above (meaning there's a limit to how high it can go), then that sequence will converge. In other words, it will settle down to a specific value.

Consider the sequence  $1/2, 3/4, 7/8, 15/16$ , and so on. This sequence is monotonically increasing because each number is larger than the one before it. It's also bounded above by 1, because no number in the sequence can be greater than 1.

Now, what does it mean for this sequence to converge? It means that as you go further and further along in the sequence, the numbers get closer and

## Formal Proof

We'll use the completeness property of the real numbers, which states that every non-empty set of real numbers that is bounded above has a least upper bound.

Let's consider a sequence  $(a_n)$  that is monotonically increasing and bounded above.

**1. Monotonicity:** For all natural numbers  $n$ , we have

$$a_n \leq a_{n+1}$$

This means each term in the sequence is less than or equal to the next term.

**2. Bounded above:** There exists a real number  $M$  such that for all natural numbers  $n$ , we have

$$a_n \leq M$$

This means there is a number  $M$  that is greater than or equal to every term in the sequence.

We want to show that this sequence converges to a limit  $L$ .

Let  $S$  be the set of all terms in the sequence, i.e.,

$$S = \{a_n : n \text{ is a natural number}\}$$

By the completeness property of the real numbers, S has a least upper bound, which we'll call L.

We claim that the sequence  $(a_n)$  converges to L. To prove this, we need to show that for any positive number  $\varepsilon$ , there exists a natural number N such that for all  $n \geq N$ ,  $|a_n - L| < \varepsilon$

$$\exists N \in \mathbb{N} : \forall n \geq N, |a_n - L| < \varepsilon.$$

*This is the formal definition of a sequence converging to a limit L. What it's saying is that we can get the terms of the sequence as close as we want to the limit L, by going far enough along in the sequence. The  $\varepsilon$  is a small positive number that represents how close we want to get to L. The N is the point in the sequence after which all terms are within  $\varepsilon$  of L.*

Given  $\varepsilon > 0$ , because  $L - \varepsilon$  is not an upper bound of S, there exists an element  $a_N$  in S such that  $a_N > L - \varepsilon$ . Since the sequence is monotonically increasing, for all  $n \geq N$ , we have  $a_n \geq a_N > L - \varepsilon$ .

$$\exists a_N \in S : a_N > L - \varepsilon$$

$$\forall n \geq N : a_n \geq a_N > L - \varepsilon$$

*Here, we're using the fact that L is the least upper bound of the set S. If we subtract a little bit from L (that's the  $L - \varepsilon$ ), then it's no longer an upper bound, so there must be some term in the sequence that's bigger than  $L - \varepsilon$ . We're calling that term  $a_N$ .*

*This is where we use the fact that the sequence is increasing. If  $a_N > L - \varepsilon$ , and all the terms after  $a_N$  are bigger than equal or to  $a_N$  (because the sequence is increasing), then all the terms after  $a_N$  are also bigger than  $L - \varepsilon$ .*

On the other hand, because L is an upper bound of S, we have  $a_n \leq L$  for all n.

$$a_n \leq L \text{ for all } n$$

*This is just restating the fact that L is an upper bound of the set S, which means all terms of the sequence are less than or equal to L.*

Therefore, for all  $n \geq N$ , we have  $L - \varepsilon < a_n \leq L$ , which implies  $|a_n - L| < \varepsilon$ .

$$L - \varepsilon < a_n \leq L \implies |a_n - L| < \varepsilon$$

*For all terms of the sequence after  $a_N$ , they are bigger than  $L - \varepsilon$  and less than or equal to L. In other words, they are within  $\varepsilon$  of L, which is exactly what we needed to show prove that the sequence converges to L.*

*let's understand this implication. The expression  $|a_n - L|$  represents the absolute difference between  $a_n$  and L, which is the distance between  $a_n$  and L on the number line.*

*If  $a_n$  is between  $(L - \varepsilon)$  and L, then the distance from  $a_n$  to L must be less than  $\varepsilon$ . This is because  $\varepsilon$  is the gap between L and  $(L - \varepsilon)$ , and  $a_n$  is somewhere in that gap.*

*So, if  $a_n$  is greater than  $(L - \varepsilon)$  and less than or equal to L, then the distance from  $a_n$  to L (which is  $|a_n - L|$ ) must be less than  $\varepsilon$ .*

.....

RO

### You

give an example of model free q-learning, how do we know what's the next state after taking a step?



### ChatGPT

In model-free Q-learning, the agent learns directly from experience without requiring explicit knowledge of the transition dynamics of the environment. Instead of relying on a transition model (like a transition matrix), the agent interacts with the environment by taking actions and observing the resulting states and rewards. Let's illustrate this with a simple example:

Suppose we have a simple grid-world environment represented as follows:

diff

Copy code

```
S - - -  
- - - -  
- - - G
```

Where:

- S is the starting state.
- G is the goal state.
- The agent can move up, down, left, or right, but cannot move outside the grid.
- Moving into a cell marked with "-" incurs a reward of -1.
- Moving into the goal state (G) results in a reward of +10 and terminates the episode.

```
# Grid world example

class GridWorld:

    def __init__(self):
        self.grid = np.array([[-1, -1, -1, -1],
                            [-1, -1, -1, -1],
                            [-1, -1, -1, 10]])
        self.num_states = np.prod(self.grid.shape)
        self.num_actions = 4 # Up, Down, Left, Right
        self.start_state = (0, 0)
        self.goal_state = (2, 3)

    def step(self, state, action):
        # Move according to action (0: Up, 1: Down, 2: Left, 3: Right)
        if action == 0:
            next_state = (max(state[0] - 1, 0), state[1])
        elif action == 1:
            next_state = (min(state[0] + 1, self.grid.shape[0] - 1), state[1])
        elif action == 2:
            next_state = (state[0], max(state[1] - 1, 0))
        else:
            next_state = (state[0], min(state[1] + 1, self.grid.shape[1] - 1))

        # Calculate reward
        reward = self.grid[next_state[0], next_state[1]]

        return next_state, reward
```

what do we mean by model free? isn't the gridworld class itself a model?



### ChatGPT

You're correct that in the provided example, the `GridWorld` class defines the environment, including the state space, action space, transition dynamics, and rewards. However, when we say "model-free" in the context of reinforcement learning algorithms like Q-learning, we mean that the agent does not explicitly know or use a model of the environment's dynamics (transition probabilities) to make decisions.

Let's clarify this concept:

#### 1. Model-Based vs. Model-Free:

- **Model-Based:** In a model-based approach, the agent has explicit knowledge of the transition dynamics of the environment, such as transition probabilities or the transition function. The agent uses this model to plan its actions by simulating future states and rewards.
- **Model-Free:** In a model-free approach, the agent learns directly from experience without relying on an explicit model of the environment's dynamics. Instead, it interacts with the environment, observes states and rewards, and learns a policy or value function based on these observations.

#### 2. GridWorld Class in the Example:

- In the provided example, the `GridWorld` class is used to define the environment's dynamics, such as the grid layout, possible actions, and the consequences of those actions (i.e., state transitions and rewards).
- However, the agent (implemented with Q-learning) does not have access to this information directly. It only interacts with the environment by selecting actions and observing the resulting states and rewards.
- The agent learns from these interactions and updates its Q-values based on observed