

CS 1810 Spring 2026 Section 10

Markov Decision Processes and Reinforcement Learning

1 Introduction

So far in CS 1810 we have mostly studied settings where the goal is to predict, represent, or model data. In this section, we move to a different learning problem: **sequential decision making**. Here, an agent repeatedly interacts with an environment, chooses actions over time, and tries to maximize long-run reward.

We begin with **Markov Decision Processes (MDPs)**, where the rules of the world are known. This allows us to plan using value functions and dynamic programming. We then turn to **reinforcement learning (RL)**, where those rules are no longer known in advance, so the agent must learn from experience.

The key questions of this section are:

- How do we formalize sequential decision making?
- How do value functions summarize exponentially many possible futures?
- How can we compute an optimal policy when the environment model is known?
- How can we learn a good policy when the model is unknown?

2 Markov Decision Processes

2.1 The sequential decision making goal

In many machine learning problems, we passively observe data and try to make predictions. In sequential decision making, the situation is more active: we choose actions, those actions affect future states, and we care about the long-run consequences of what we do.

For example, imagine treating a patient over time. The patient's condition is the state, the treatment decision is the action, and the resulting health improvement minus treatment cost is the reward. A good action may not maximize immediate reward, but may place the patient in a much better state later.

This kind of problem is naturally modeled by a Markov Decision Process.

2.2 MDP formulation

A **Markov Decision Process (MDP)** is a tuple

$$(S, \mathcal{A}, P, R, \gamma),$$

where

- S is the state space,
- $\mathcal{A}$ is the action space,

- P is the transition model, written as $p(s_{t+1} \mid s_t, a_t)$,
- R is the reward function, written as $r(s, a)$,
- $\gamma \in [0, 1)$ is the discount factor.

At each time step t , the agent observes a state s_t , chooses an action a_t , receives reward r_t , and then transitions to a new state s_{t+1} according to the environment dynamics.

2.3 The Markov property

A central assumption of an MDP is the **Markov property**:

$$p(s_{t+1} \mid s_{0:t}, a_{0:t}) = p(s_{t+1} \mid s_t, a_t).$$

This says that once we know the current state and action, the rest of the past does not matter for predicting the next state.

This assumption is restrictive, but it is what makes the problem tractable. Because the future depends only on the current state, an optimal way of acting can be written as a policy that depends only on the current state, rather than on the full history.

2.4 Policies

A **policy** is a rule for how the agent acts in each state. A deterministic policy outputs a single action:

$$a = \pi(s).$$

A stochastic policy specifies a distribution over actions:

$$\pi(a \mid s).$$

The important point is that a policy describes *how* the agent behaves, while the transition model describes *how the world responds* to the agent's action.

2.5 Agent-environment interaction

In an MDP, the interaction proceeds as follows:

- observe the current state s_t ,
- choose an action $a_t \sim \pi(\cdot \mid s_t)$,
- receive reward r_t ,
- transition to $s_{t+1} \sim p(\cdot \mid s_t, a_t)$.

So a policy together with the environment dynamics induces a distribution over trajectories

$$(s_0, a_0, r_0, s_1, a_1, r_1, \dots).$$

This trajectory viewpoint is useful because it makes clear that our policy choices shape the distribution of future states and rewards.

2.6 Rewards, returns, and the objective

In sequential decision making, we care about the total reward accumulated over time, not just the immediate reward of one action. The **return** from time t onward is

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

The discount factor γ downweights rewards far in the future. This can encode a preference for receiving rewards sooner rather than later, and it also helps keep the infinite sum finite and stable.

The overall goal in an MDP is to find a policy that maximizes expected return:

$$\pi^* \in \arg \max_{\pi} \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \right].$$

Exercise 2.1. Why do we discount future rewards? Give one intuitive reason and one mathematical reason.

2.7 Value functions

A direct search over all possible future action sequences is usually intractable, because the number of possibilities grows exponentially with the horizon. Value functions provide a compact summary of long-run quality.

The **state-value function** of a policy π is

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid s_t = s].$$

It tells us how good state s is if we follow policy π from that point onward.

The **action-value function** (or **Q-function**) of a policy π is

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

It tells us how good it is to take action a in state s and then continue following π .

These are related by

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(\cdot|s)}[Q^{\pi}(s, a)].$$

So the value of a state is the average of the action values under the policy's action distribution.

2.8 Bellman equations

The key recursive idea is that long-run return can be decomposed into a *now* term and a *later* term:

$$G_t = r_t + \gamma G_{t+1}.$$

Taking expectations yields the **Bellman equations**.

For the value function, start from the definition:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s].$$

Using the decomposition $G_t = r_t + \gamma G_{t+1}$, we get

$$V^\pi(s) = \mathbb{E}_\pi[r_t + \gamma G_{t+1} \mid s_t = s].$$

By linearity of expectation,

$$V^\pi(s) = \mathbb{E}_\pi[r_t \mid s_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} \mid s_t = s].$$

Now condition first on the action chosen at state s :

$$V^\pi(s) = \sum_a \pi(a \mid s) \mathbb{E}_\pi[r_t + \gamma G_{t+1} \mid s_t = s, a_t = a].$$

Then condition on the next state:

$$V^\pi(s) = \sum_a \pi(a \mid s) \left(r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) V^\pi(s') \right).$$

For the action-value function, we proceed similarly:

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

Again using $G_t = r_t + \gamma G_{t+1}$,

$$Q^\pi(s, a) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \sum_{a'} \pi(a' \mid s') Q^\pi(s', a').$$

These equations are powerful because they replace a long-horizon optimization problem with a local consistency condition.

2.9 Optimality and greedy improvement

The optimal value function satisfies the **Bellman optimality equation**

$$V^*(s) = \max_a \left(r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) V^*(s') \right).$$

The corresponding optimal action-value function satisfies

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a'} Q^*(s', a').$$

If we know the value function of a policy, we can try to improve that policy by acting greedily with respect to the value estimates:

$$\pi_{\text{new}}(s) = \arg \max_a \left(r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) V^\pi(s') \right).$$

Similarly, if we know Q^π , then acting greedily is especially simple:

$$\pi_{\text{new}}(s) = \arg \max_a Q^\pi(s, a).$$

2.10 Planning algorithms

When the transition model and reward function are known, we can solve for good policies directly.

Policy evaluation. Given a fixed policy π , compute $V^\pi(s)$ for every state.

Policy iteration.

1. Start with an initial policy π .
2. Evaluate the current policy to compute V^π .
3. Improve the policy by acting greedily with respect to V^π .
4. Repeat until convergence.

Value iteration. Instead of alternating between evaluation and improvement, we can update the value estimates directly:

$$V_{k+1}(s) = \max_a \left(r(s, a) + \gamma \sum_{s'} p(s' | s, a) V_k(s') \right).$$

This merges policy evaluation and policy improvement into a single recursive update.

Exercise 2.2. Consider an MDP with discount factor $\gamma = 0.9$ and terminal state T with value 0. There are three states A, B, T . At state A , action a_1 leads to T with reward $+1$, while action a_2 leads to B with probability 0.5 and to T with probability 0.5, both with reward 0. At state B , the only action a_3 leads to T with reward $+3$. Suppose policy π always takes a_2 at A and a_3 at B . Compute $V^\pi(B)$ and $V^\pi(A)$.

Exercise 2.3. Why is it useful to assume the environment is Markovian?

3 Reinforcement Learning

3.1 From known dynamics to unknown dynamics

The planning methods above assume that we know the transition dynamics $p(s' | s, a)$ and the reward function $r(s, a)$. In many real-world problems, this is unrealistic. An agent usually does not begin with a perfect model of the world.

This is the reinforcement learning setting. The agent must interact with the environment and learn from experience. Instead of being given the model, it observes sampled transitions of the form

$$(s_t, a_t, r_t, s_{t+1}),$$

and must use these experiences to improve its behavior.

3.2 Model-based and model-free learning

There are two broad ways to proceed.

Model-based RL. Estimate the missing transition model $\hat{p}(s' \mid s, a)$ and reward function $\hat{r}(s, a)$ from data, and then run planning algorithms such as policy iteration or value iteration on the estimated model.

Model-free RL. Bypass model learning entirely and directly estimate useful quantities such as value functions or Q-functions from sampled transitions.

Model-based methods can be appealing because they try to reconstruct the rules of the world. However, they may be unnecessarily complicated when the state space is large. The goal is not to perfectly learn the transition model itself; the goal is to learn how to act well.

Exercise 3.1. Would each of the following problems be more naturally solved by known-model planning or by reinforcement learning?

1. Finding the best route through a dangerous forest when you have a complete map.
2. Bringing a robot into a new obstacle course it has never seen before.

3.3 Model-free policy evaluation

Suppose we want to evaluate a fixed policy π but do not know the transition model.

One idea is **Monte Carlo evaluation**: run the policy, observe the full return G_t , and update the current estimate toward that realized return:

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha(G_t - V^\pi(s_t)).$$

This is conceptually simple, but it has two drawbacks:

- learning can be slow, because we may have to wait until the end of an episode to know G_t ;
- the variance can be high, because the total return may depend on many random events.

A more efficient idea is **temporal-difference (TD) learning**. Instead of waiting for the full return, we use the one-step approximation

$$G_t \approx r_t + \gamma V^\pi(s_{t+1}).$$

This yields the update

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha(r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)).$$

This is called **bootstrapping**, because we update the value of the current state using our current estimate of the next state's value. The quantity

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

is called the **TD error**.

3.4 Q-functions and acting

In the known-model setting, a value function V^π is enough to improve a policy, because the transition model lets us compare the downstream value of different actions. In the unknown-model setting, this is harder.

This is why Q-functions become especially useful in reinforcement learning. If we estimate $Q(s, a)$ directly, then acting greedily is easy:

$$a_{\text{greedy}}(s) = \arg \max_a Q(s, a).$$

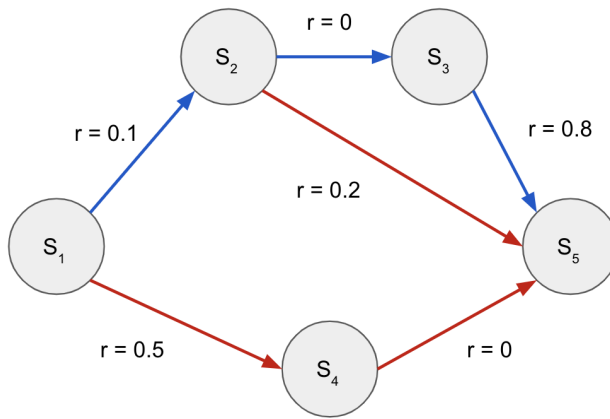
The optimal Q-function satisfies

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s' | s, a) \max_{a'} Q^*(s', a').$$

So if we can learn Q^* , then we also recover the optimal policy via

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

3.5 Q values example



Q* values

	Red action	Blue action
S ₁	0.5	0.9
S ₂	0.2	0.8
S ₃		0.8
S ₄	0	
S ₅		

$$Q^*(s_1, a_{\text{blue}}) = r(s_1, a_{\text{blue}}) + \max_a [Q^*(s_2, a)] = 0.1 + \max(0.2, 0.8) = 0.9$$

In the figure above, suppose that the discount factor is $\gamma = 1$ and the actions are deterministic. The agent starts from state s_1 and eventually ends in state s_5 , from which no more actions are taken. The table on the right contains converged Q-values corresponding to an optimal policy.

We can verify the Bellman optimality idea directly. For example,

$$Q^*(s_1, a_{\text{blue}}) = r(s_1, a_{\text{blue}}) + \max_a Q^*(s_2, a) = 0.1 + \max(0.2, 0.8) = 0.9.$$

The optimal policy can then be read off by choosing, at each state, the action with the highest Q-value.

3.6 Exploration versus exploitation

A central tension in reinforcement learning is the need to balance two goals:

- **Exploitation:** choose the action that currently appears best according to the present Q-estimates;
- **Exploration:** try actions that may currently look worse, but could lead to better estimates and ultimately better performance.

If the agent only exploits from the beginning, it may never discover actions whose true value is higher than the current estimates suggest.

A common strategy is **ϵ -greedy exploration**: with probability $1 - \epsilon$, choose the greedy action $\arg \max_a Q(s, a)$, and with probability ϵ , choose a random action.

Exercise 3.2. What is the main danger of using a purely exploitative strategy from the start?

3.7 On-policy RL: SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy TD control algorithm. Given current state s , action a , reward r , next state s' , and next action a' , it performs the update

$$Q(s, a) \leftarrow Q(s, a) + \alpha_t [r + \gamma Q(s', a') - Q(s, a)].$$

This is called **on-policy** because the next action a' is chosen by the same behavior policy that the agent is actually following. For example, if the behavior policy is ϵ -greedy, then SARSA learns the value of behaving in that exploratory way.

Intuitively, SARSA updates the current Q-value so that it becomes closer to the observed reward plus the discounted value of the next state-action pair that the policy will actually take.

Because SARSA is on-policy, its learned Q-values correspond to the behavior of the current policy. This means that if the policy keeps taking random exploratory actions, then the learned Q-values also reflect those exploratory actions.

3.7.1 Concept Question

What would SARSA learn if the policy π were fixed? What is the tension in reducing exploration ϵ in ϵ -greedy when trying to make SARSA converge to optimal behavior?

3.8 Off-policy RL: Q-learning

A second way to update Q-values is **Q-learning**, which uses

$$Q(s, a) \leftarrow Q(s, a) + \alpha_t [r + \gamma \max_{a'} Q(s', a') - Q(s, a)].$$

The key difference is that Q-learning does not use the next action actually taken by the behavior policy. Instead, it uses the best next action according to the current Q-function. This is what makes Q-learning **off-policy**.

Intuitively, SARSA learns “the value of what I actually do,” while Q-learning learns “the value of the best thing I could do next.”

Under suitable conditions, Q-learning converges to the optimal Q-function Q^* even if the behavior policy continues to explore.

3.8.1 Concept Question

If the behavior in SARSA were purely greedy rather than ϵ -greedy, would the SARSA update agree with the Q-learning update?

3.9 Exercise: Model-free RL

Consider the same gridworld-style setup from lecture. Suppose the agent is in a state S_1 , takes action Up , receives reward $r = 0$, and lands in state S_2 . Let the discount factor be $\gamma = 0.9$ and the learning rate be $\alpha = 0.5$. Suppose the current Q-table contains

$$Q(S_2, Left) = 2, \quad Q(S_2, Right) = 10, \quad Q(S_1, Up) = 5.$$

Exercise 3.3. Assume the behavior policy says that the next action will be `Left`.

1. Compute the SARSA update for $Q(S_1, Up)$.
2. Compute the Q-learning update for $Q(S_1, Up)$.
3. Why are they different?

3.10 The tabular limit

The methods we have discussed so far are **tabular**: they assume that we can store a value for every state or every state-action pair in a lookup table.

This is feasible when the state space is small and discrete, but it becomes impossible when the state is something like an image, a high-dimensional sensor vector, or a continuous quantity. In those cases, we cannot hope to explicitly store $Q(s, a)$ for every possible (s, a) pair.

3.11 A preview of deep reinforcement learning

This limitation motivates the next step: replacing the table by a function approximator, often a neural network.

At a conceptual level, two important ideas are:

- **Deep Q-Networks (DQN)**: approximate $Q(s, a)$ with a neural network instead of a table;
- **Policy gradients**: optimize the policy directly, rather than going through tabular Q-values.

The algorithms in this section provide the conceptual foundation for later deep RL methods that scale to much larger state and action spaces.