Reinforcement Learning (MDP)

Zbida Amine

September 2025

1 The Reinforcement Learning Problem

Reinforcement Learning (RL) is a sub eld of machine learning that focuses on training an agent, also referred to as the decision-maker, to e ectively control and manage a dynamic system, often known as the environment. This is achieved through a learning process in which the agent interacts with the environment over a speci c period of time, adjusting its behavior based on the data acquired during these interactions. This learning process is commonly described as trial-and-error learning. The control problem or decision-making problem tackled in reinforcement learning is typically reformulated as a Markov decision process (MDP)

1.1 MDP Definition

A Markov decision process (MDP) is a six-tuple $(S, A, P, R, \gamma, \rho_0)$, where:

- S is the set of all possible states of the environment,
- A is the set of all possible actions of the agent,
- $P : S \times A \times S \to [0, 1]$ is the transition model which specifies the next state of environment given the actual state and the action of the agent. That is
$$P(s_{t+1}|s_t, a_t) = P(s_t, a_t, s_{t+1}) \tag{1}$$
The transition model P is assumed to be stationary and Markovian.
- $R : S \times A \times S \to R$ is the reward function which is used to define the objective of the control process,
$$r_t = R(s_t, a_t, s_{t+1}) \tag{2}$$
- $\gamma \in]0, 1[$ is the discount factor, which offers both theoretical and practical advantages.
- ρ_0 is the initial state distribution.

Markov Hypothesis (Notion of Markovian)

$$P(s_t, a_t, s_{t+1}) = P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_0, a_0, s_1, a_1, \ldots, s_t, a_t) \tag{3}$$

The goal of the agent, and consequently of RL, is to find a function that prescribes the actions to take in every state. This function should be designed to maximize the expected discounted cumulative reward, as defined in (4).

$$G = E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{4}$$

Hence, solving an MDP essentially involves addressing a stochastic, time-discrete optimal control problem, and it will evolve some helper value functions.

$$\begin{aligned} \max_{\pi} \quad & E_\pi\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})\right] \\ \text{subject to} \quad & a_t \sim \pi(\cdot|s_t), \quad t = 0, 1, 2, \ldots \\ & s_{t+1} \sim P(s_t, a_t, \cdot), \quad t = 0, 1, 2, \ldots \\ & s_0 \sim \rho_0 \end{aligned} \tag{5}$$

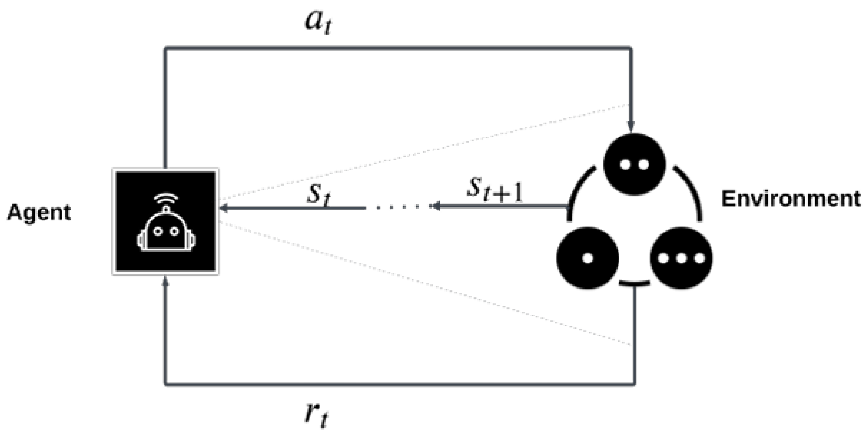


Figure 1: RL

Finite MDP is an MDP where S and A are finites.

2 Policy and Value functions

2.1 Policy

A policy of the agent is a function π that specifies the probability of selecting an action in a given state, that is

$$\begin{aligned} \pi : S \times A &\to [0,1] \\ (s,a) &\mapsto \pi(s,a) = P(a|s) \end{aligned} \tag{6}$$

A deterministic policy is a function π that specifies for any state $s \in S$ a single action $a \in A$, that is

$$\begin{aligned} \pi : S &\to A \\ s &\mapsto \pi(s) = a \end{aligned} \tag{7}$$

The agent tries to find optimal policy that maximizes G. Hence, we introduce the value functions that will lead us to this π^*

2.2 State-Value function

$v^\pi(s)$: Expected discounted cumulative reward when **starting** from state s.

$$v^\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right] \tag{8}$$

2.3 Action-Value function

$Q^\pi(s,a)$: Expected discounted cumulative reward when **starting** from state s and **taking** an action a.

$$Q^\pi(s,a) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right] \tag{9}$$

Theorem 1.1: Bellman equation for v^π and Q^π

Given an MDP, for all policies π:

$$v^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma v^\pi(s')) \tag{10}$$

$$Q^\pi(s,a) = \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')) \tag{11}$$

their relation:

$$Q^\pi(s,a) = \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma v^\pi(s')) \tag{12}$$

Proof

$$\begin{aligned}
v^\pi(s) &= E_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t \mid s_0 = s\right] \\
&= E_\pi\left[r_0 + \gamma\sum_{t=1}^{\infty}\gamma^{t-1} r_t \mid s_0 = s\right] \\
&= \sum_a \pi(s,a)\sum_{s'} P(s,a,s') E_\pi\left[r_0 + \gamma\sum_{t=1}^{\infty}\gamma^{t-1} r_t \mid s_0 = s, a_0 = a, s_1 = s'\right] \\
&= \sum_a \pi(s,a)\sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma E_\pi\left[\sum_{t=1}^{\infty}\gamma^{t-1} r_t \mid s_1 = s'\right]\right)
\end{aligned}$$

By the Markov Property: $\sum_{t=1}^{\infty}\gamma^{t-1}r_t$ is independent of s_0 and a_0 given s_1.

$$v^\pi(s) = \sum_a \pi(s,a)\sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma E_\pi\left[\sum_{t=0}^{\infty}\gamma^{t} r_t \mid s_0 = s'\right]\right)$$
$$v^\pi(s) = \sum_a \pi(s,a)\sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma v^\pi(s')\right)$$

Proof

$$\begin{aligned}
Q^\pi(s,a) &= E_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t \mid s_0 = s, a_0 = a\right] \\
&= E_\pi\left[r_0 + \gamma\sum_{t=1}^{\infty}\gamma^{t-1} r_t \mid s_0 = s, a_0 = a\right] \\
&= \sum_{s'} P(s,a,s') E_\pi\left[r_0 + \gamma\sum_{t=1}^{\infty}\gamma^{t-1} r_t \mid s_0 = s, a_0 = a, s_1 = s'\right] \\
&= \sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma E_\pi\left[\sum_{t=1}^{\infty}\gamma^{t-1} r_t \mid s_1 = s'\right]\right)
\end{aligned}$$

By the Markov Property: $\sum_{t=1}^{\infty}\gamma^{t-1}r_t$ is independent of s_0 and a_0 given s_1.

$$
\begin{aligned}
Q^\pi(s,a) &= \sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s' \right] \right) \\
&= \sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma \sum_{a'} \pi(s',a') E_\pi \left(\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s', a_0 = a' \right) \right) \\
Q^\pi(s,a) &= \sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a') \right)
\end{aligned}
$$

3 Optimal Policy and Optimal Value functions

For a finite MDP, an optimal policy π^* is the policy that is **equal** or **better** than all other policies. That is

$$v^{\pi^*}(s) \geq v^\pi(s), \quad \forall \pi, \forall s \in S$$

Solving MDP is equivalent to finding π^*

Theorem 1.2

In a finite MDP with a bounded reward functions, there is **always atleast** one π^* that is **deterministic**

We define optimal values as the following:

$$v^*(s) = \max_\pi v^\pi(s) \quad \forall s \in S \tag{13}$$

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \quad \forall s \in s, \ \forall a \in A \tag{14}$$

Theorem 1.3: Bellman Optimality equation

for a finite MDP, we define the bellman optimality as the following:

$$v^*(s) = \max_a \left(\sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma v^*(s') \right) \right) \tag{15}$$

$$Q^*(s,a) = \sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right) \tag{16}$$

their relation:

$$v^*(s) = \max_a Q^*(s,a) \tag{17}$$

Theorem 1.4

Let v^* and Q^* be the optimal value functions. The following policies are optimal

$$\pi^* = \arg\max_{a \in A} \sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma v^*(s')\right) \tag{18}$$

$$\pi^* = \arg\max_{a \in A} Q^*(s,a) \tag{19}$$

NB: Majority of RL algorithms seeks to estimate Q^* instead of v^*

4 Dynamic Programming for Solving MDP

4.1 Policy Evaluation

The objective is to compute v^π since it is the unique solution to equation (10) We define the Bellman operator:

$$\begin{aligned} B^\pi : F(S,R) &\to F(S,R) \\ f &\mapsto B^\pi(f) \end{aligned}$$

Theorem 1.5

If $\gamma < 1$, the Bellman operator B^π is a γ-contraction w.r.t. $\|\cdot\|_\infty$.

Proof

Let π be a policy. For all $f, g \in F(S,R)$ and $s \in S$:

$$\begin{aligned} |B^\pi(f)(s) - B^\pi(g)(s)| &= |\sum_a \pi(s,a) \sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma f(s')\right) \\ &\quad - \sum_a \pi(s,a) \sum_{s'} P(s,a,s')\left(R(s,a,s') + \gamma g(s')\right)| \\ &= \gamma |\sum_a \pi(s,a) \sum_{s'} P(s,a,s()\left(f(s') - g(s')\right)| \\ &\le \gamma \sum_a \pi(s,a) \sum_{s'} P(s,a,s') \max_{s'} |f(s') - g(s')| \\ &\le \gamma \max_{s'} |f(s') - g(s') \sum_a \pi(s,a) \sum_{s'} P(s,a,s') \\ &\le \gamma \|f - g\|_\infty \end{aligned}$$

Hence, for all $f, g \in F(S,R)$:

$$|B^\pi(f)(s) - B^\pi(g)(s)| \le \gamma \|f - g\|_\infty \quad \text{for all } s \in S$$

Taking the supremum over all $s \in S$:

$$\|B^\pi(f) - B^\pi(g)\|_\infty \leq \gamma \|f - g\|_\infty$$

By the Banach fixed-point theorem, v^π exists and is unique. Define the sequence $\{v^k\}_{k=0}^\infty \subset F(S, R)$ by:

$$\begin{cases} v^0 & \text{arbitrary} \\ v^k(s) & = (B^\pi v^{k-1})(s) \text{ for } k \geq 1, \forall s \in S \end{cases}$$

Algorithm 1 Policy Evaluation (Estimating v^π)

```
Require: $\pi$, $P$, $R$ the reward function, a small threshold $\theta > 0$ determining
    accuracy of estimation
Ensure: Value function approximation $V \approx v^\pi$
 1: Initialization: Arbitrary initialization of $V(s)$ for each $s \in S$, except that
    $V(\text{terminal}) = 0$
 2: Continue $\leftarrow$ True
 3: while Continue do
 4:     $\Delta \leftarrow 0$
 5:     for each state $s$ do
 6:         $v \leftarrow V(s)$
 7:         $V(s) \leftarrow \sum_{a \in A} \pi(s,a) \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \mu V(s')\right)$
 8:         $\Delta \leftarrow \max(\Delta, |V(s) - v|)$
 9:     end for
10:     if $\Delta < \theta$ then
11:         Continue $\leftarrow$ False
12:     end if
13: end while
14: return $V$
```

4.2 Policy Improvement

Theorem 1.6

Let π and π' be two policies. if

$$\forall s \in S, Q^{\pi}(s, \pi'(s)) \geq v^{\pi}(s) \tag{20}$$

then π' must be as good as, or better than π, meaning

$$\forall s \in S, v^{\pi'}(s) \geq v^{\pi}(s) \tag{21}$$

Given a policy π for which we already have computed the value function v^{π}. We consider the greedy policy w.r.t Q^{π} defined in Theorem 1.4.

Proposition

Given a policy π. The greedy policy w.r.t Q^{π} defined in Theorem 1.4 is an improvement of π. That is it is better or equal to π. Furthermore, if the greedy policy w.r.t Q^{π} is equal and not better than π, then both policies are optimal.

Proof. Let π' be the greedy policy with respect to Q^{π}. By construction, we have:

$$Q^{\pi}(s, \pi'(s)) = \arg\max_{a \in A} Q^{\pi}(s, a) \geq Q^{\pi}(s, a) \quad \forall a \in A$$

In particular:

$$Q^{\pi}(s, \pi'(s)) \geq Q^{\pi}(s, \pi(s)) = v^{\pi}(s)$$

By the Policy Improvement Theorem, π' is better than or equal to π.

Now assume that the greedy policy π' is equal to π, that is, $v^{\pi}(s) = v^{\pi'}(s)$ for all $s \in S$. Then we have:

$$\begin{aligned} v^{\pi'}(s) &= Q^{\pi'}(s, \pi'(s)) \\ &= Q^{\pi}(s, \pi'(s)) \\ &= \max_{a \in A} \sum_{s' \in S} P(s, a, s')\big(R(s, a, s') + \gamma v^{\pi}(s')\big) \\ &= \max_{a \in A} \sum_{s' \in S} P(s, a, s')\big(R(s, a, s') + \gamma v^{\pi'}(s')\big) \end{aligned} \tag{1.28}$$

Thus, for every state $s \in S$, we have:

$$v^{\pi'}(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s')\big(R(s, a, s') + \gamma v^{\pi'}(s')\big)$$

But this is precisely the Bellman optimality equation for v^*, which means that $v^* = v^{\pi'} = v^{\pi}$.

Therefore, both π' and π are optimal policies. □

4.3 Policy Iteration (PI)

Once we have improved a policy π_1 using v_{π_1} and obtained a better policy π_2, we can then improve π_2 to obtain another better policy π_3. Hence, we obtain a sequence of monotonically improving policies $\{\pi_n\}_{n\geq 0}$. Since a finite MDP has a finite number of deterministic policies and each improvement always yields a strictly better policy (unless the policy we have improved is already optimal), the sequence $\{\pi_n\}_{n\geq 0}$ will converge to an optimal policy. This approach is called the *policy iteration* algorithm. The main steps of this algorithm are given in Algorithm 2.

Algorithm 2 Main steps of policy iteration algorithm for obtaining π^*

```
Require: a small threshold $\theta > 0$ determining accuracy of estimation
 1: Initialization: arbitrary initialization of $V(s)$ and $\pi(s)$ for each $s \in S$

 2: Policy evaluation:
 3: Continue $\leftarrow$ True
 4: while Continue do
 5:     $\Delta \leftarrow 0$
 6:     for each state $s$ do
 7:         $v \leftarrow V(s)$
 8:         $V(s) \leftarrow \sum_{a\in A} \pi(s,a) \sum_{s'\in S} T(s,a,s') \left(R(s,a,s') + \mu V(s')\right)$
 9:         $\Delta \leftarrow \max(\Delta, |V(s) - v|)$
10:     end for
11:     if $\Delta < \theta$ then
12:         Continue $\leftarrow$ False
13:     end if
14: end while

15: Policy improvement:
16: Policy-stable $\leftarrow$ True
17: for each state $s$ do
18:     old-action $\leftarrow \pi(s)$
19:     $\pi(s) \leftarrow \arg\max_{a\in A} \sum_{s'\in S} T(s,a,s') \left(R(s,a,s') + \mu V_\pi(s')\right)$
20:     if $\pi(s) \neq$ old-action then
21:         Policy-stable $\leftarrow$ False
22:     end if
23: end for
24: if Policy-stable then
25:     Stop and return $\pi$
26: else
27:     return to Policy evaluation step
28: end if
```

4.4 Value Iteration (VI)

The policy iteration algorithm has a disadvantage in that it requires policy evaluation numerous times to achieve the required improvements, and this process could be a prolonged iterative computation that requires multiple passes through the state set. This problem can be alleviated by iteratively computing the optimal state-value function based on the optimal Bellman equation for the optimal state-value function. That is, we start with an arbitrary function $v_0 \in \mathcal{F}(S, R)$ and define a sequence $(v_k)_{k \in N}$ by

$$v_k(s) = B^* v_{k-1}(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \mu v_{k-1}(s') \right). \tag{22}$$

Theorem 1.7

If $\mu < 1$, then the Bellman optimal operator B^* defined in (1.30) is a μ-contraction with respect to the infinity norm:

$$B^*(f)(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \mu f(s') \right). \tag{23}$$

Proof. Let $f, g \in \mathcal{F}(S, R)$, then for all $s \in S$,

$$|B^*(f)(s) - B^*(g)(s)| = \left| \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \mu f(s') \right) - \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \mu g(s') \right) \right| \tag{24}$$

$$\le \max_{a \in A} \left| \sum_{s' \in S} T(s,a,s') \mu (f(s') - g(s')) \right| \tag{25}$$

$$\le \mu \max_{a \in A} \sum_{s' \in S} T(s,a,s') |f(s') - g(s')| \tag{26}$$

$$\le \mu \max_{s' \in S} |f(s') - g(s')| \tag{27}$$

$$= \mu \|f - g\|_\infty. \tag{28}$$

Furthermore,

$$\|B^*(f) - B^*(g)\|_\infty = \max_{s \in S} |B^*(f)(s) - B^*(g)(s)| \le \mu \|f - g\|_\infty. \tag{29}$$

Hence, B^* is a contraction if $\mu < 1$. $\square$

Since B^* is a μ-contraction for $\mu < 1$, the sequence $(v_k)_{k \in N}$ will converge to the fixed point of B^* according to the Banach fixed-point theorem. This fixed point is exactly the optimal state-value function. Once we have the optimal state-value function v^*, we can derive an optimal policy as in (1.19). This approach is called the *value iteration* algorithm [**?**]. The main steps of the value iteration algorithm are given in Algorithm 3.

Algorithm 3 Main steps of value iteration algorithm for estimating π^*

```
Require: a small threshold $\theta > 0$ determining accuracy of estimation
 1: Initialization: arbitrary initialization of $V(s)$ for each $s \in S$,
    except that $V(\text{terminal}) = 0$
 2: Continue ← True
 3: while Continue do
 4:     $\Delta \leftarrow 0$
 5:     for each state $s$ do
 6:         $v \leftarrow V(s)$
 7:         $V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} T(s, a, s') \left( R(s, a, s') + \mu V(s') \right)$
 8:         $\Delta \leftarrow \max(\Delta, |V(s) - v|)$
 9:     end for
10:     if $\Delta < \theta$ then
11:         Continue ← False
12:     end if
13: end while
14: Return: the policy
        $\pi(s) = \arg\max_{a \in A} \sum_{s' \in S} T(s, a, s') \left( R(s, a, s') + \mu V(s') \right)$
```

5 Monte Carlo

Monte Carlo (MC) methods attempt to achieve much the same effect as Dynamic Programming (DP) without relying on a perfect model of the environment. Monte Carlo methods require only experiential data, meaning sample sequences of actions, states, and rewards obtained from actual or simulated interaction with an environment. In this section, we define Monte Carlo methods only for episodic tasks to guarantee that the sum of cumulative rewards is well-defined. This implies that experiences are separated into episodes, with each episode eventually coming to an end regardless of the actions taken.

5.1 Monte Carlo Prediction

Monte Carlo (MC) methods estimate the value function $v_\pi(s)$ by averaging returns observed from sample episodes generated by following a fixed policy π, without requiring knowledge of the environment dynamics.

Let $E(s)$ denote the set of all episodes starting from state s, and let $G(e)$ be the return (discounted sum of rewards) of episode e. Then,

$$v^\pi(s) = E_\pi[G \mid S_0 = s] \approx \frac{1}{|E(s)|} \sum_{e \in E(s)} G(e).$$

Two common Monte Carlo prediction methods differ in *how returns are averaged*:

First-Visit Monte Carlo In First-Visit MC, the value of a state is updated using only the return following the *first time* the state appears in an episode.

Algorithm 4 First-Visit Monte Carlo for Estimating v_π

Input: A policy π to be evaluated.
Initialization: Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$, except $V(\text{terminal}) = 0$. For all $s \in \mathcal{S}$, set Returns$(s) \leftarrow \emptyset$.

```
1: while True do
2:     Generate an episode
           $e = (s_0, a_0 = \pi(s_0), r_0, s_1, a_1 = \pi(s_1), r_1, \ldots, s_T)$
       following $\pi$.
3:     for each state $s$ appearing in the episode do
4:         if $s$ is the first occurrence in $e$ then
5:             $G \leftarrow$ return following the first occurrence of $s$
6:             Append $G$ to Returns($s$)
7:             $V(s) \leftarrow$ average(Returns($s$))
8:         end if
9:     end for
10: end while
```

Every-Visit Monte Carlo In Every-Visit MC, the value of a state is updated using the returns following *every occurrence* of that state within an episode.

Algorithm 5 Every-Visit Monte Carlo for Estimating v_π

Input: A policy π to be evaluated.
Initialization: Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$, except $V(\text{terminal}) = 0$. For all $s \in \mathcal{S}$, set $\text{Returns}(s) \leftarrow \emptyset$.

```
1: while True do
2:     Generate an episode
           $e = (s_0, a_0 = \pi(s_0), r_0, s_1, a_1 = \pi(s_1), r_1, \dots, s_T)$
       following $\pi$.
3:     for each time step $t$ in the episode such that $s_t = s$ do
4:         $G_t \leftarrow$ return following time $t$
5:         Append $G_t$ to Returns($s$)
6:         $V(s) \leftarrow$ average(Returns($s$))
7:     end for
8: end while
```

Comparison First-Visit MC uses one return per episode for each state, while Every-Visit MC uses all occurrences. Both methods converge to $v^\pi(s)$ under standard assumptions, though Every-Visit MC may converge faster due to using more samples per episode.

5.2 Monte Carlo Control

Monte Carlo control aims to *learn an optimal policy* by interacting with the environment and improving the policy based on estimated action-value functions, rather than merely evaluating a fixed policy.

Instead of estimating the state-value function $v_\pi(s)$, Monte Carlo control estimates the *action-value function*

$$Q^\pi(s, a) = E_\pi[G_t \mid S_0 = s, A_0 = a],$$

and then improves the policy by acting greedily with respect to this estimate.

Policy Improvement Given an estimate $Q(s, a)$ of $q_\pi(s, a)$, a better policy π' can be obtained using

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

By alternating between policy evaluation (estimating Q) and policy improvement, MC control gradually converges to the optimal policy.

Exploration via ϵ-Greedy Policies Monte Carlo control requires continuous exploration to ensure all state-action pairs are visited infinitely often. This is commonly achieved using an ϵ-greedy policy:

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}, & \text{if } a = \arg\max_a Q(s,a), \\ \frac{\epsilon}{|\mathcal{A}|}, & \text{otherwise.} \end{cases}$$

Every-Visit Monte Carlo Control Algorithm Monte Carlo control typically uses the *every-visit* approach to estimate $Q(s,a)$.

Algorithm 6 Monte Carlo Control Using ϵ-Greedy Policy

```
Input: Exploration parameter $\epsilon$.
Initialization: Initialize $Q(s,a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$.
Initialize $\pi$ to be $\epsilon$-greedy with respect to $Q$.
For all $(s,a)$, set Returns$(s,a) \leftarrow \emptyset$.
 1: while True do
 2:     Generate an episode following $\pi$:
                    $e = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T)$
 3:     for each time step $t$ in the episode do
 4:         $G_t \leftarrow$ return following time $t$
 5:         Append $G_t$ to Returns$(s_t, a_t)$
 6:         $Q(s_t, a_t) \leftarrow$ average(Returns$(s_t, a_t)$)
 7:     end for
 8:     for each state $s$ appearing in the episode do
 9:         Update policy: $\pi(s) \leftarrow \arg\max_a Q(s,a)$
10:     end for
11: end while
```

Convergence Under the assumption that:

- all state-action pairs are visited infinitely often, and
- the policy remains exploratory (e.g., ϵ-greedy),

Monte Carlo control converges with probability 1 to the optimal action-value function $Q^*(s,a)$ and the optimal policy π^*.

Remarks Unlike Dynamic Programming, Monte Carlo control does not require a model of the environment. However, it only applies to episodic tasks and can be sample-inefficient because updates are made only after entire episodes are completed.

What to Retain: Theory & Practice in Dynamic Programming and RL

This summary organizes the key theoretical and practical lessons behind Dynamic Programming (DP) and Reinforcement Learning (RL), guided by the quizzes.

1. Core Operators and Convergence

True/False Highlights

- **T^* is a contraction with modulus γ. True.** This guarantees convergence of Value Iteration.

$$\|T_\pi V - T_\pi W\|_\infty \le \gamma \|V - W\|_\infty.$$

- **Residual-based stopping.** If

$$\rho^*(V) = \|T^*V - V\|_\infty \le (1-\gamma)\epsilon,$$

then

$$\|V - V^*\|_\infty \le \epsilon.$$

Proof.

$$\begin{aligned} \|V - V^*\|_\infty &= \|T^*V - T^*V^*\|_\infty \\ &= \|T^*V - V + V - T^*V^*\|_\infty \\ &\le \|T^*V - V\|_\infty + \|T^*V^* - T^*V\|_\infty, \end{aligned}$$

where we used the triangular inequality and the fact that $T^*V^* = V^*$.

By assumption, the residual satisfies

$$\|T^*V - V\|_\infty \le (1-\gamma)\varepsilon.$$

Moreover, since the Bellman optimality operator T^* is a γ-contraction,

$$\|T^*V^* - T^*V\|_\infty \le \gamma \|V^* - V\|_\infty.$$

Hence,

$$\|V - V^*\|_\infty \le (1-\gamma)\varepsilon + \gamma \|V - V^*\|_\infty.$$

Rearranging gives

$$(1-\gamma)\|V - V^*\|_\infty \le (1-\gamma)\varepsilon,$$

and therefore,

$$\|V - V^*\|_\infty \le \varepsilon.$$

□

Takeaway

- Contraction is the mathematical reason VI and policy evaluation converge.
- Residuals give *certificates of optimality.*

2. Backup Principle (One-Step Evaluation)

Bellman backup for policy evaluation:

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

Key Point Learning in RL and DP is built from one-step backups.

3. Bellman Equations and Linear Algebra

For a finite MDP:

$$V_\pi = r_\pi + \gamma P_\pi V_\pi, \qquad (I - \gamma P_\pi)V_\pi = r_\pi,$$

where:

$$P_\pi(s, s') = \sum_a \pi(a|s)P(s'|s, a), \qquad r_\pi(s) = \sum_a \pi(a|s)R(s, a).$$

Practice Tips

- Small MDP $\Rightarrow$ solve linear system.
- Large MDP $\Rightarrow$ use iterative methods.

4. Value Iteration (VI) vs Policy Iteration (PI)

Conceptual Difference

- **VI:** direct update toward optimal values.
- **PI:** alternate evaluation and improvement.

When to Prefer

- VI: large or continuous state spaces.
- PI: small MDP with exact evaluation.

5. Priority in Updates: Prioritized Sweeping

Effects

- Fewer useless updates.
- Same fixed point.
- Faster convergence.

6. Stopping Rules and Error Control

Given:

$$\delta_k = \|V_{k+1} - V_k\|_\infty,$$

then:

$$\|V_{k+1} - V^*\|_\infty \leq \frac{\delta_k}{1-\gamma}.$$

Proof.

$$\|V_{k+1} - V^*\|_\infty = \|T^* V_k - T^* V^*\|_\infty$$

since $V_{k+1} = T^* V_k$ and V^* is the unique fixed point of T^*, i.e. $V^* = T^* V^*$.

By contractivity of the Bellman optimality operator T^*,

$$\|V_{k+1} - V^*\|_\infty \leq \gamma \|V_k - V^*\|_\infty.$$

Now write

$$\begin{aligned} \|V_k - V^*\|_\infty &= \|V_k - T^* V_k + T^* V_k - T^* V^*\|_\infty \\ &\leq \|V_k - T^* V_k\|_\infty + \|T^* V_k - T^* V^*\|_\infty, \end{aligned}$$

by the triangle inequality. Therefore,

$$\|V_{k+1} - V^*\|_\infty \leq \gamma \|V_k - T^* V_k\|_\infty + \gamma \|T^* V_k - T^* V^*\|_\infty.$$

Using again the contraction property,

$$\|T^* V_k - T^* V^*\|_\infty \leq \gamma \|V_k - V^*\|_\infty,$$

hence

$$\|V_{k+1} - V^*\|_\infty \leq \gamma \delta_k + \gamma \|V_{k+1} - V^*\|_\infty,$$

where $\delta_k = \|V_{k+1} - V_k\|_\infty$.

Rearranging yields

$$(1-\gamma)\|V_{k+1} - V^*\|_\infty \leq \gamma \delta_k,$$

and finally

$$\|V_{k+1} - V^*\|_\infty \leq \frac{\gamma}{1-\gamma}\, \delta_k.$$

□

Example If $\gamma = 0.95$ and target error 10^{-2}:

$$\rho \leq (1-\gamma)10^{-2} = 5 \times 10^{-4}.$$

7. Bellman Optimality

Value form

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right].$$

Action-value form

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

8. Policy Improvement Theorem

Greedy Improvement If:

$$\pi'(s) \in \arg\max_a Q_\pi(s,a),$$

then:

$$V^{\pi'}(s) \geq V^\pi(s), \quad \forall s.$$

Meaning Greedy policies never degrade performance.

9. Monte Carlo Insight

- No model required.
- Unbiased estimates.
- High variance.
- Works only in episodic tasks.

5.3 Temporal Difference (TD)

In this section, we explore how Temporal-Difference (TD) methods can be used to evaluate a given policy π. We first introduce an *incremental implementation* of the First-Visit Monte Carlo (MC) method for estimating v_π.

As shown in Algorithm 4, the First-Visit MC method estimates the value function by averaging observed returns obtained by starting from a state s and following the policy π thereafter. Formally, let $\{G_i(s)\}_{i=1}^t$ be t independent samples of the return starting from state s under policy π. Then the MC estimate of $v_\pi(s)$ is given by:

$$V_t(s) = \frac{1}{t}\sum_{i=1}^{t} G_i(s). \tag{30}$$

Now suppose that a new return sample $G_{t+1}(s)$ becomes available. The updated estimate of $v_\pi(s)$ becomes:

$$\begin{aligned} V_{t+1}(s) &= \frac{1}{t+1}\sum_{i=1}^{t+1} G_i(s) \\ &= \frac{1}{t+1}\left(\sum_{i=1}^{t} G_i(s) + G_{t+1}(s)\right) \\ &= \frac{t}{t+1}\cdot\frac{1}{t}\sum_{i=1}^{t} G_i(s) + \frac{1}{t+1}G_{t+1}(s) \\ &= \frac{t}{t+1}V_t(s) + \frac{1}{t+1}G_{t+1}(s). \end{aligned}$$

Define the step-size parameter

$$\alpha_t = \frac{1}{t+1}.$$

Then the update rule can be written in the incremental form:

$$V_{t+1}(s) = (1-\alpha_t)V_t(s) + \alpha_t G_{t+1}(s) = V_t(s) + \alpha_t\big(G_{t+1}(s) - V_t(s)\big). \tag{31}$$

This yields the incremental First-Visit MC update:

$$V_{t+1}(s) = V_t(s) + \alpha_t\big(G_{t+1}(s) - V_t(s)\big), \qquad \text{where } \alpha_t = \frac{1}{t+1}. \tag{1.37}$$

Toward Temporal-Difference Learning Unlike Monte Carlo methods, which require waiting until the end of an episode to observe the full return $G_{t+1}(s)$, Temporal-Difference methods update value estimates after a single time step, using only the immediate reward r_t and the next state s_{t+1}.

TD methods can also be applied to *find an optimal policy.* Broadly, TD control methods fall into two categories:

- **On-policy methods:** Use the same policy for both generating data and improving it.
- **Off-policy methods:** Improve a policy that differs from the one used to generate data from the environment.

SARSA: On-Policy TD Control SARSA [?] is an on-policy, one-step TD control algorithm that learns the *action-value function* $Q(s, a)$ instead of the state-value function. It considers transitions from a state-action pair to a state-action pair. Given a sample $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, the SARSA update is:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)\Big[r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)\Big], \quad (32)$$

where $\alpha_t(s_t, a_t)$ is the learning rate.

The name *SARSA* comes from the experience tuple *State, Action, Reward, State, Action* used for updates.

Usage of SARSA

- **Policy evaluation:** Generate data using a fixed policy π and apply (32) iteratively to estimate Q_π.
- **Control (finding Q^*):** Interact with the environment using a policy that becomes increasingly greedy over time while ensuring sufficient exploration, e.g., an ϵ-greedy policy with decaying ϵ.

Algorithm Outline The main steps of SARSA for control are summarized in Algorithm 8. Its convergence properties depend on the exploration policy and the standard conditions for TD convergence, such as visiting all state-action pairs infinitely often and decaying step sizes.

Q-Learning: Off-Policy TD Control Q-Learning [?] is an *off-policy* TD control algorithm that directly estimates the *optimal action-value function* $Q^*(s, a)$, independently of the policy used to generate the data. Unlike SARSA, Q-Learning updates the Q-values toward the maximum over possible actions in the next state:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)\Big[r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)\Big], \quad (33)$$

where $\alpha_t(s_t, a_t)$ is the learning rate.

Key Features of Q-Learning

- **Off-policy:** Can learn the optimal policy regardless of the behavior policy used to explore the environment.
- **Greedy policy extraction:** Once Q^* is learned, the optimal policy is

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

- **Exploration:** While learning, a behavior policy (e.g., ϵ-greedy) ensures sufficient exploration of all state-action pairs.
- **Convergence:** Guaranteed to converge to Q^* under standard conditions: diminishing step sizes, sufficient exploration, and bounded rewards.

Comparison: SARSA vs Q-Learning

- SARSA is *on-policy*: updates follow the action actually taken by the current policy. It is sensitive to the exploration policy.
- Q-Learning is *off-policy*: updates use the greedy action in the next state regardless of the behavior policy. It often converges faster to Q^* but can be more risky in stochastic environments.

6 Function Approximation in Reinforcement Learning

Motivation Tabular methods for RL, like Value Iteration, Policy Iteration, SARSA, or Q-Learning, store value functions or Q-functions in a table for every state (or state-action) pair. This becomes infeasible for:

- Large or continuous state spaces.
- High-dimensional inputs, such as images.
- Environments where the state space is effectively infinite.

To address this, we introduce *function approximation*, where a parametric function $Q(s, a; \theta)$ is used to approximate the true action-value function $Q^*(s, a)$. The goal is to learn parameters θ such that $Q(s, a; \theta) \approx Q^*(s, a)$.

6.1 Deep Q-Networks (DQN)

Deep Q-Networks (DQN) combine Q-Learning with deep neural networks to handle large state spaces. The key idea is:

$$\text{Update: } \theta \leftarrow \theta + \alpha\big(y - Q(s, a; \theta)\big)\nabla_\theta Q(s, a; \theta), \tag{34}$$

where the *target* y is given by the one-step Q-Learning target:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-), \tag{35}$$

and θ^- are the parameters of a *target network*.

Target Network In DQN, the same network cannot be used to compute both $Q(s, a; \theta)$ and the target $r + \gamma \max_{a'} Q(s', a'; \theta)$ because it can lead to instability. DQN solves this by using a *target network*:

- A copy of the Q-network, with parameters θ^-.
- Updated periodically (every C steps) to match θ.
- Stabilizes training by providing a slowly moving target.

Experience Replay To decorrelate samples and improve learning stability:

- Experiences (s, a, r, s') are stored in a *replay buffer*.
- Mini-batches are randomly sampled from the buffer to update the network.

6.2 Variants of DQN

Double DQN (DDQN) Standard DQN tends to overestimate Q-values because it uses the same network to select and evaluate actions in $\max_{a'} Q(s', a'; \theta^-)$. Double DQN decouples these:

$$y_{\text{DDQN}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-), \tag{36}$$

where θ selects the greedy action and θ^- evaluates it. This reduces overestimation bias.

Dueling DQN Dueling networks decompose $Q(s, a)$ into:

$$Q(s, a) = V(s) + A(s, a), \tag{37}$$

where:

- $V(s)$ estimates the *state-value function*.
- $A(s, a)$ estimates the *advantage* of taking action a in state s relative to other actions.

This helps the network learn which states are valuable independently of the action choice, improving learning efficiency in environments with many similar-valued actions.

6.3 Summary

- Function approximation allows RL to scale beyond tabular methods.
- DQN uses neural networks with experience replay and target networks to stabilize training.
- Variants like Double DQN reduce overestimation, while Dueling DQN improves representation of value and advantage.

7 Policy Gradient Methods

Motivation While value-based methods like Q-Learning and DQN estimate value functions and derive a policy implicitly (via $\pi(s) = \arg\max_a Q(s,a)$), *policy gradient methods* optimize the policy $\pi_\theta(a|s)$ directly:

- Suitable for large or continuous action spaces where $\arg\max_a Q(s,a)$ is infeasible.
- Can represent stochastic policies naturally, which is useful for exploration.
- Optimize performance directly via gradient ascent on expected return.

7.1 Policy Gradient Theorem

The objective of a policy π_θ parameterized by θ is the expected discounted return:

$$J(\theta) = E_{\tau\sim\pi_\theta}\Big[R(\tau)\Big] = E_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T}\gamma^t r_t\Big], \tag{38}$$

where $\tau = (s_0, a_0, r_0, s_1, \dots)$ is a trajectory sampled from the environment.

The policy gradient theorem states that:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^{\pi_\theta}(s_t, a_t)\Big]. \tag{39}$$

Key Ideas

- The gradient is computed as the expectation of $\nabla_\theta \log \pi_\theta(a|s)$ weighted by returns or advantages.
- In practice, the *advantage function* $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ is often used to reduce variance.

7.2 Actor-Critic Methods

Concept Actor-Critic methods combine:

- **Actor:** Parameterized policy $\pi_\theta(a|s)$ updated using policy gradient.
- **Critic:** Value function $V_w(s)$ (or action-value function $Q_w(s, a)$) used to estimate advantages for the actor.

The actor is updated using:

$$\theta \leftarrow \theta + \alpha \, \nabla_\theta \log \pi_\theta(a|s) \, A_w(s, a), \tag{40}$$

while the critic is trained via TD learning:

$$w \leftarrow w + \beta \, \nabla_w \big(r + \gamma V_w(s') - V_w(s)\big)^2. \tag{41}$$

7.3 Asynchronous Advantage Actor-Critic (A3C)

A3C [?] improves training stability by:

- Running multiple parallel actor-learner threads to explore different parts of the environment simultaneously.
- Each thread updates the shared global parameters asynchronously, reducing correlation in updates.
- Uses advantage estimates $A(s, a) = r_t + \gamma V(s_{t+1}) - V(s_t)$.

Benefits of A3C

- No experience replay is needed, making it memory-efficient.
- Parallel exploration reduces variance and improves convergence speed.

7.4 Advantage Actor-Critic (A2C)

A2C [?] is the synchronous, batched version of A3C:

- Collect trajectories from multiple actors in parallel.
- Compute gradients over the batch and apply a single synchronous update to the global network.
- Simpler implementation and deterministic gradient updates compared to A3C.

7.5 Proximal Policy Optimization (PPO)

Philosophy and Paradigm Proximal Policy Optimization (PPO) [**?**] is built on the philosophy of *trust-region policy optimization* (TRPO), which seeks to maximize the expected return while constraining updates to the policy so that it does not change too abruptly.

The core idea is that large updates in policy parameters can destabilize learning because the agent may take actions that are drastically different from those observed during data collection, leading to high variance or collapse of training. PPO embraces this principle but implements it in a simpler, more practical way than TRPO:

- **Paradigm:** Safe, iterative policy improvement.
- **Goal:** Make small, conservative steps in policy space that improve expected return without exceeding a *trust region*.
- **Mechanism:** Instead of enforcing a hard constraint on the KL divergence between the old and new policy (as in TRPO), PPO introduces a *soft constraint* via a clipped surrogate objective.

Clipped Surrogate Objective Let π_θ be the current policy and $\pi_{\theta_{\text{old}}}$ the policy used to generate data. Define the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}.$$

The PPO objective is then:

$$L^{\text{CLIP}}(\theta) = E_t\Big[\min\big(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\big)\Big], \tag{42}$$

where A_t is the advantage estimate and ϵ is a hyperparameter controlling the maximum allowed policy deviation per update.

The objective ensures:

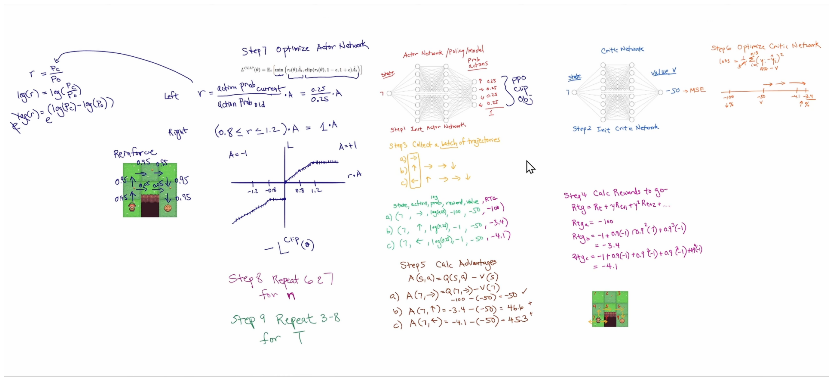
- If the policy update increases the probability of advantageous actions moderately ($|r_t - 1| < \epsilon$), the improvement is applied normally.
- If the update is too large ($|r_t - 1| > \epsilon$), the contribution to the objective is clipped, preventing excessive changes.

Advantages and Paradigm Shift

- **Stability:** PPO avoids the instability of vanilla policy gradient methods caused by large, uncontrolled updates.
- **Simplicity:** Unlike TRPO, PPO does not require solving a constrained optimization problem with second-order derivatives.
- **Versatility:** Can be applied to both discrete and continuous action spaces.

- **Philosophical Paradigm:** Iterative, conservative improvement; balance between exploration and stability; gradual policy adaptation ensures safe and reliable learning.

In practice, PPO has become a widely adopted standard in deep reinforcement learning due to this philosophy of careful, incremental policy improvement, combining the effectiveness of policy gradients with the robustness of trust-region methods.



Step 7 Optimize Actor Network
Left
Right
Reinforce
Step 8 Repeat 6 & 7 for n
Step 9 Repeat 3-8 for T
Actor Network / Policy / model
Step 1 Init Actor Network
Step 3 Collect a batch of trajectories
Step 5 Calc Advantages
Critic Network
Value V
Step 2 Init Critic Network
Step 4 Calc Rewards to go
Step 6 Optimize Critic Network
PPO clip obj

Figure 2: PPO

7.6 Summary of Policy Gradient Methods

- Policy gradients optimize policies directly; suitable for continuous/high-dimensional actions.
- Actor-Critic methods reduce variance using a learned value function.
- A3C introduces asynchronous updates for stability and faster exploration.
- A2C is the synchronous batch version of A3C.
- PPO improves gradient stability via a clipped surrogate objective, making it one of the most widely used policy gradient algorithms.