Reinforcement Learning (RL)

- Learning from Interaction with environment
  - to achieve some goal

- The first idea when we think about the nature of learning

- Examples:
  - Baby movements
  - Learning to drive car
    - Environment’s response affects our subsequent actions
    - We find out the effects of our actions later
Reinforcement Learning (RL)

- $S$: Set of states
- $A$: Set of actions

- Goal: Learning a mapping from states to actions in order to maximize a scalar reward (reinforcement signal)
Main characteristics of RL

- Delayed reward

- Opportunity for active exploration
  - Needs trade-off between exploration and exploitation
Environment properties

- Deterministic vs. stochastic
  - Stochastic: stochastic reward & transition

- Known vs. unknown
  - Unknown: Agent doesn't know the precise results of its actions before doing them

- Fully observable vs. partially observable
  - Partially observable: Agent doesn't necessarily know all about the current state
  - We discuss about only fully observable environments.
Reinforcement Learning: Example

- Chess game (deterministic game)
  - Learning task: choose move at arbitrary board states
  - Training signal: final win or loss
  - Training: e.g., played n games against itself

![Diagram of the agent-environment interaction](image-url)
Reinforcement Learning: Example

- Pacman (Stochastic game)
RL problem: deterministic environment

- **Deterministic**
  - Transition and reward functions

- **At time** $t$:
  - Agent observes state $s_t \in S$
  - Then chooses action $a_t \in A$
  - Then receives reward $r_t$, and state changes to $s_{t+1}$

- Learn to choose action $a_t$ in state $s_t$ that maximizes the discounted return:

  $$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \quad 0 < \gamma \leq 1$$
RL problem: stochastic environment

- Stochastic environment
  - Transition and/or reward function

- Learn to choose a policy that maximizes the expected discounted return:

\[ E[R_t] = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots] \]

starting from state \( s_t \)

\[ R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \]
Markov Decision Process (RL Setting)

- **Markov assumption:**
  \[
P(s_{t+1}, r_t | s_t, a_t, r_{t-1}, s_{t-1}, a_{t-1}, r_{t-2}, \ldots) = P(s_{t+1}, r_t | s_t, a_t)
  \]

- **Task:** for every possible state \(s \in S\) learn a policy \(\pi\) for choosing actions that maximizes
  \[
  E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, \pi], \quad 0 < \gamma \leq 1
  \]
RL: Autonomous Agent

- Execute actions in environment, observe results, and learn
  - Learn policy $\pi$ that maximizes $\sum_{k=0}^{\infty} \gamma^k E[r_{t+k} | s_t = s, \pi]$ for every state $s \in S$

- Example: Robot grid world
  - Deterministic reward and transition
RL: Autonomous Agent

- Execute actions in environment, observe results, and learn
  - Learn (perhaps stochastic) policy that maximizes \( \sum_{k=0}^{\infty} \gamma^k E[r_{t+k} | s_t = s, \pi] \) for every state \( s \in S \)

- Function to be learned is the policy \( \pi: S \times A \rightarrow [0,1] \) (when the policy is deterministic \( \pi: S \rightarrow A \))
  - But training examples are not of the form \( \langle s, a \rangle \)
  - They are instead of the form \( \langle \langle s, a \rangle, r \rangle \)
MDP: Recycling Robot example

- $S = \{\text{high, low}\}$
- $A = \{\text{search, wait, recharge}\}$
  - $A(\text{high}) = \{\text{search, wait}\}$  \[\text{Available actions in the 'high' state}\]
  - $A(\text{low}) = \{\text{search, wait, recharge}\}$
- $\mathbb{R}_\text{search} > \mathbb{R}_\text{wait}$

\[
P(s_{t+1} = \text{high}|s_t = \text{high}, a_t = \text{search})
\]
State-value function for policy $\pi$

Given a policy $\pi$, define the **value function**

$$
V^{\pi}(s) = E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, \pi \right\}
$$
Optimal policy

- Policy $\pi$ is better than (or equal to) $\pi'$ (i.e. $\pi \geq \pi'$) iff
  \[ V^\pi(s) \geq V^{\pi'}(s), \quad \forall s \in S \]

- **Optimal policy** $\pi^*$ is better than (or equal to) all other policies ($\forall \pi, \pi^* \geq \pi$)

- For any MDP, a deterministic optimal policy exists!
  - We’ll abbreviate $V^{\pi^*}(s)$ as $V^*(s)$

- **Optimal policy** $\pi^*$:
  \[ \pi^*(s) = \arg\max_{\pi} V^\pi(s), \quad \forall s \in S \]

- If we have $V^*(s)$ and $P(s_{t+1}|s_t, a_t)$ we can compute $\pi^*(s)$
MDP: Optimal policy state-value and action-value functions

- Optimal policies share the same optimal state-value function:

\[ V^*(s) = \max_\pi V^\pi(s), \quad \forall s \in S \]

- And the same optimal action-value function:

\[ Q^*(s, a) = \max_\pi Q^\pi(s, a), \quad \forall s \in S, a \in \mathcal{A}(s) \]
State-value function for policy $\pi$: example

- Deterministic example

$$V^\pi(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \quad s_t = s$$
State-value function for policy $\pi^*$: example

- Deterministic example

$$V^\pi(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

$$\pi^*(s) = \arg\max_{\pi} V^\pi(s)$$
Recursive definition for \( V^\pi(S) \)

\[
V^\pi(S) = E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, \pi \right\}
\]
\[
= E \left\{ r_t + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, \pi \right\}
\]

\[
= \sum_a \pi(s, a) \sum_{s'} p^a_{ss'} \left( R^a_{ss'} + \gamma E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_{t+1} = s', \pi \right\} \right)
\]

\[
p^a_{ss'} = p(s_{t+1} = s'|s_t = s, a_t = a)
\]
\[
R^a_{ss'} = E\{r_t|s_t = s, a_t = a, s_{t+1} = s'\}
\]

Bellman Equations
Action-value function for policy $\pi$

$$Q^\pi(s, a) = E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a, \pi \right\}$$

$$= \sum_{s'} P_{ss'}^a \left( R_{ss'}^a + \gamma E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_{t+1} = s', \pi \right\} \right)$$

$$V^\pi(s')$$
Backup diagram for $V^\pi$ and $Q^\pi$
Grid-world example

Actions that move the agent outside of the grid:
- location unchanged
- reward -1

In state $A$, all actions $\rightarrow A'$ and their rewards is 10
In state $B$, all actions $\rightarrow B'$ and their rewards is 5

Reward of all other actions is 0

$V^\pi$ for random policy
\[ \gamma = 0.9 \]
Recursive definition for $V^\pi(S)$

- Simpler case: $\pi$ deterministic function

$$V^\pi(S) = \sum_{s'} P_{ss'}^{\pi(s)} \left( R_{ss'}^{\pi(s)} + \gamma V^\pi(s') \right)$$
Bellman optimality equation

\[ V^*(s) = \max_{a \in \mathcal{A}(s)} Q^*(s, a) \]

\[ Q^*(s, a) = \sum_{s'} \mathcal{P}_{ss'}^a \left( R_{ss'}^a + \gamma V^*(s) \right) \]

\[ V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} \mathcal{P}_{ss'}^a \left( R_{ss'}^a + \gamma V^*(s') \right) \]

\[ Q^*(s, a) = \sum_{s'} \mathcal{P}_{ss'}^a \left( R_{ss'}^a + \gamma \max_{a'} Q^*(s', a') \right) \]
Backup diagrams for $V^*$ and $Q^*$

\[
V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} p^a_{ss'} \left( R^a_{ss'} + \gamma V^*(s') \right) \\
Q^*(s, a) = \sum_{s'} p^a_{ss'} \left( R^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right)
\]
Optimal policy: example 1
Optimal policy: example 2

$r(s, a)$ (immediate reward) values

$Q(s, a)$ values

$V^*(s)$ values

One optimal policy
Main steps in solving Bellman equations

- Two kinds of steps, which are repeated in some order for all the states until no further changes take place

\[ \pi(s) = \arg\max_a \left\{ \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V^\pi(s')) \right\} \]

\[ V^\pi(s) = \sum_{s'} P_{ss'}^{\pi(s)} (R_{ss'}^{\pi(s)} + \gamma V^\pi(s')) \]

- There are variants of algorithms
  - Order of the above steps
  - One can also do them for all states at once, or state by state, or more often to some states than others.
RL algorithms

- Model-based (passive)
  - Known environment model (transition and reward probabilities)

- Model-free (active)
  - Unknown environment model
Value iteration algorithm: model-based

1) Initialize $V(s)$ arbitrarily
2) Repeat until convergence
   for $s \in S$
     for $a \in A$
       $Q(s, a) \leftarrow \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V(s'))$
       $V(s) \leftarrow \max_a Q(s, a)$

$V(s)$ converges to $V^*(s)$
Dynamic programming

$V^*(s) = \max_{a \in A(s)} Q^*(s, a)$
Value Iteration

- Value iteration works even if we randomly traverse the environment instead of looping through each state and action
  - but we must still visit each state infinitely often

- If \( \max_{s \in S} |V^{old}(s) - V(s)| < \epsilon \), then the value of the greedy policy differs from the optimal policy by no more than \( \frac{2\epsilon \gamma}{1-\gamma} \)

Value Iteration

- Needs complete knowledge of the transitions and reward function
- It is time and memory expensive
Unknown transition model

- So far: learning optimal policy when we know $P_{ss'}^a$ and $R_{ss'}^a$

- What if we don’t?
Q learning

- If agent knows $Q(s, a)$, it can choose optimal action without knowing $P_{ss'}^a$, and $R_{ss'}^a$.
  - $\pi^*(s) = \arg\max_a Q(s, a)$

- And, it can learn $Q$ without knowing $P_{ss'}^a$, and $R_{ss'}^a$. 

Training rule to learn $Q$: deterministic environment

- We can write $Q$ recursively as:

\[
Q(s_t, a_t) = r(s_t, a_t, s_{t+1}) + \gamma V^*(s_{t+1}) \\
= r(s_t, a_t, s_{t+1}) + \gamma \max_{a'} Q(s_{t+1}, a')
\]

- Let $\hat{Q}$ denote learner’s current approximation to $Q$:

\[
\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')
\]

where $s'$ in the state resulting from applying action $a$ in the state $s$
Q-learning algorithm:
deterministic environments

Initialize Q table: $\hat{Q}(s, a) \leftarrow 0$
Repeat (for each episode):
  Initialize $s$
  Repeat (for each step of episode):
    Choose $a$ from $s$ using a policy derived from $\hat{Q}$
    Take action $a$, receive reward $r$, observe new state $s'$
    $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
    $s \leftarrow s'$
  until $s$ is terminal

Convergence: $\hat{Q}$ will converge to the true $Q$ function under certain conditions
Q-learning: Policy

- Greedy action selection:

\[ \pi(s) = \arg\max_a Q(s, a) \]

- \( \epsilon \)-greedy: be greedy most of the times, occasionally take a random action
Exploration/exploitation tradeoff

- Exploitation: High rewards from trying previously-well-rewarded actions

- Exploration: Which actions are best?
  - Must try ones not tried before
Q-learning: exploration strategy

- Give a higher probability to the actions that currently have better utility

\[
\pi(s, a) = \frac{k\hat{Q}(s,a)}{\sum_{a'} k\hat{Q}(s,a')} 
\]

- After learning \( Q^* \), the policy is greedy?
Q-learning: non-deterministic

- Q-learning generalizes to non-deterministic worlds
- Update rule:
  \[
  \hat{Q}_n(s, a) = \hat{Q}_{n-1}(s, a) + \alpha_n \left( r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a') - \hat{Q}_{n-1}(s, a) \right)
  \]
  \(\hat{Q}(s, a)\) after \(n\)-th iteration

- One of choices for \(\alpha_n\)
  \[
  \alpha_n = \frac{1}{\text{visits}_n(s, a)}
  \]

- Still can prove convergence of \(\hat{Q}\) to \(Q\) (under certain assumptions)
  \[
  \lim_{n \to \infty} \hat{Q}_n(s, a) = Q^*(s, a), \quad \forall s \in S, a \in A
  \]
**SARSA (State-Action-Reward-State-Action)**

Initialize $\hat{Q}(s, a)$ arbitrarily

Repeat (for each episode):

  - Initialize $s$
  - Choose $a$ from $s$ using a policy derived from $\hat{Q}$

Repeat (for each step of episode):

  - Take action $a$, receive reward $r$, observe new state $s'$
  - Choose $a'$ from $s'$ using policy derived from $\hat{Q}$

  \[ \hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha [r + \gamma \hat{Q}(s', a') - \hat{Q}(s, a)] \]

  - $s \leftarrow s'$
  - $a \leftarrow a'$

until $s$ is terminal
Q-learning vs. SARSA

- Q-learning is an off-policy learning algorithm
- SARSA is an on-policy learning algorithm
1-step Tabular Q-learning

\[ \hat{Q}_n(s, a) = \hat{Q}_{n-1}(s, a) + \alpha_n \left( r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a') - \hat{Q}_{n-1}(s, a) \right) \]

Temporal Difference error
Q-learning is a special case of temporal difference algorithms

- Q-learning iteratively reduces discrepancy between $Q$ estimates of a state and its immediate ancestors or descendants

Temporal Difference (TD) Learning

- Reduces discrepancy between $Q$ estimates of a state and more distant ancestors or descendants
Look farther into the future of a move

\[ Q(s_{t+1}, a_{t+1}) \]
\[ = E \left\{ r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \right\} \]
\[ = E \left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 Q(s_{t+2}, a_{t+2}) \right\} \]
\[ \vdots \]
\[ = E \left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^n Q(s_{t+n}, a_{t+n}) \right\} \]
\[ \vdots \]
\[ = E \left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \right\} \]
TD Learning

- Look farther into the future of a move
  - Update the $Q$ function after looking farther ahead
  - Speeds up the learning process

- Q-learning reduces discrepancy between successive $Q$ estimates:
  - One-step:
    $$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$
  - Two-step:
    $$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$
  - n-step:
    $$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$
TD Learning (Cont’d)

- Combines the estimates from various look-ahead distances:

\[
Q^\lambda(s_t, a_t) \equiv (1 - \lambda)[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \ldots]
\equiv (1 - \lambda) \sum_n \lambda^{n-1} Q^{(n)}(s_t, a_t)
\]

- TD(\lambda) sometimes converges faster than Q-learning
TD Learning: $\lambda$ parameter

- **TD(0):** $\lambda = 0$, we only look at the next reward

- **TD(1):** $\lambda = 1$, only observed reward values are considered with no consideration from the current $\hat{Q}$
  - Must wait until actual return is known to update weights (end of episode)

- As $\lambda$ is increased, more emphasis on discrepancies based on more distant lookaheads
TD(\(\lambda\)) applied to learning value function

- TD(\(\lambda\)) method can be similarly applied to the value function

- n-step return

\[
V^{(n)}(s_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{V}(s_{t+n})
\]

- TD(\(\lambda\)) is a weighted average of n-step returns:

\[
V^{\lambda}(s_t) \equiv (1 - \lambda) \sum_{n} \lambda^{n-1} V^{(n)}(s_t)
\]

- Similar TD(\(\lambda\)) approach applied to \(V^*\) function converges correctly for \(\forall 0 \leq \lambda \leq 1\)
TD(\(\lambda\)) motivation

- In some settings, training will be more efficient if more distant lookaheads are considered.

- TD(\(\lambda\)) updates state values (or action-state values) in a sequence leading up to each transition by an amount that drops off as \(\lambda^t\) for state \(t\) steps in the past.
Tabular methods: Problem

- All of the introduced methods maintain a table

- Table size can be very large for complex environments
  - We do not also estimate unseen values
Function Approximation

- Use an approximate functional representation to generalize over states.
  - Instead of huge tables for $V(s)$ and $Q(s, a)$, we approximate $V(s)$ and $Q(s, a)$ with any supervised learning methods
    - Neural Networks, SVM, Decision tree, kNN, ….
  
  $V_\theta(s) = \theta_1 f_1(s) + \cdots + \theta_m f_m(s)$
  
  or
  
  $Q_\theta(s, a) = \theta_1 f_1(s, a) + \cdots + \theta_m f_m(s, a)$

- We can generalize from visited states to unvisited ones.
  - In addition to the less space requirement
Adjusting function weights

\[ \theta \leftarrow \theta + \alpha \left[ r + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s', a') \right] \nabla_\theta \hat{Q}_\theta(s, a) \]

Tesauro used function approximation in his Backgammon playing temporal difference learning research.
Application

- **Game**
  - TD-Gammon plays at level of best human players (learn through self play)

- **Control & robotics**
  - Autonomous helicopter
    - self-reliant agent must do to learn from its own experiences.
    - eliminating hand coding of control strategies

- **Resource (time, memory, channel, ...) allocation**
  - E.g., Cell phone channel allocation