Reinforcement Learning (RL)

- How an autonomous agent that sense and act in the environment can learn to choose optimal actions to achieve its goals.
- Examples: mobile robot, optimization in process control, board games, etc.
- Ingredients: reward/penalty for each action, where the reinforcement signal can be significantly delayed.
- One approach: Q learning

Introduction: Agent

Terminology:
- State: state of the environment, obtained through sensors
- Action: alter the state
- Policy: choosing actions that achieve a particular goal, based on the current state.
- Goal: desired configuration (or state).

Desired policy:
- From any initial state, choose actions that maximize the reward accumulated over time by the agent.

Variations of RL Tasks

- Deterministic vs. nondeterministic action outcomes.
- With or without prior knowledge about the effect of action on environmental state.
- Partially or fully known environmental state.

Goal: learn to choose actions that maximize discounted, cumulative award:

\[ r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots, \text{ where } 0 \leq \gamma < 1. \]

That is, we want to learn a policy \( \pi : S \rightarrow A \) that maximizes the above, where \( S \) is the set of states, and \( A \) that of actions.
RL Compared to Other Learning Algorithms

- Planning (in AI)
- Function approximation: $\pi : S \rightarrow A$.

Differences:
- Delayed reward
- Exploration vs. exploitation
- Partially observable states
- Life-long learning: leveraging on existing knowledge, to make learning of a new complex task easier.

The Learning Task

Markov Decision Process: only immediate state matters.

- State $s_t$, action $a_t$ at time step $t$.
- Reward from environment: $r_t = r(s_t, a_t)$
- State transition by environment: $s_{t+1} = \delta(s_t, a_t)$
- $r(\cdot, \cdot)$ and $\delta(\cdot, \cdot)$ may be unknown to the agent!
- Task: learn $\pi : S \rightarrow A$ to select $a_t = \pi(s_t)$.
- Question: how to specify which $\pi$ to learn?

Discounted Cumulative Reward: $V^\pi(s_t)$

- Obvious approach is to find $\pi$ that maximizes the cumulative reward when $\pi$ is executed:
  
  $$V^\pi(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
  
  $$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i},$$

  where $0 \leq \gamma < 1$ is the discount rate.

- $\pi$ is repeatedly executed: $a_t = \pi(s_t), a_{t+1} = \pi(s_{t+1}), \ldots$

- When $\gamma = 0$, only the current reward is used.

- When $\gamma \rightarrow 1$, future rewards become more important.

Choosing a Policy

- Optimal policy $\pi^*$
  
  $$\pi^* = \arg\max_\pi V^\pi(s), \forall s$$

- Want a policy that does its best for all states.

- Cumulative reward under optimal policy $\pi^*$:
  
  $$V^*(s) \equiv V^{\pi^*}(s),$$

  for short.
Example: Grid World

- Immediate reward given only when entering the goal state $G$.
- Given any initial state, we want to generate an action sequence to maximize $V$.

$Q$ Learning

- Policy is hard to learn directly, because training experience does not provide $<s, a>$ pairs.
- Only available info: sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, ...$
- In this case, it is easier to learn an evaluation function and construct a policy based on that.

Optimal Policy using $V^*(s)$

- If reward $r(s, a)$, state transition $\delta(s)$, and evaluation function $V^*(s)$ are known the following gives an optimal policy:
  \[
  \pi^*(s) = \arg\max_a [r(s, a) + \gamma V^*(\delta(s, a))]
  \]
- For example, top middle state: move right $= 100 + \gamma 0 = 100$,
  move left $= 0 + \gamma 90 = 81$, move down $= 0 + \gamma 90 = 81$. 

Discount rate: $\gamma = 0.9$.
- Top middle: $100 + \gamma 0 + \gamma^2 0 + ... = 100$
- Top left: $0 + \gamma 100 + \gamma^2 0 + ... = 90$
- Bottom left: $0 + \gamma 0 + \gamma^2 100 + ... = 81$
- Note that these values are supposed to be obtained using the optimal policy $\pi^*$. 

Grid World: $V^*(s)$ Values

(a) $r(s, a)$ values
(b) $V^*(s)$ values
Problems with Policy Based on $V^*(s)$

- Requires perfect knowledge of $r(s, a)$ and $\delta(s, a)$, to exactly predict the outcome and reward of a particular action.
- In practice, the above is impossible.
- Thus, even when $V^*(s)$ is known, $\pi^*(s)$ cannot be found.

Refer to:

$$\pi^*(s) = \operatorname{argmax}_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

- Solution: use a surrogate – the $Q$ function.

### The $Q$ Function

Can we get by without explicit knowledge of $r(s, a)$ and $\delta(s, a)$?

- $Q(s, a)$: evaluation function whose value is the **maximum discounted cumulative reward** obtainable when action $a$ is taken in state $s$:
  $$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

- The derived policy is then:
  $$\pi^*(s) = \operatorname{argmax}_a Q(s, a)$$

Note that if $Q(s, a)$ can be learned without any reference to $r(s, a)$ and $\delta(s, a)$, we have solved our problem.

- Further problem: how to estimate $Q(s, a)$?

### Learning the $Q$ Function: Getting Rid of $V^*(\delta(s, a))$

- $Q(s, a)$ is defined over all possible actions $a$ from state $s$. But note that one of these actions is optimal for state $s$, and thus:
  $$V^*(s) = \max_a Q(s, a')$$

- With the above,
  $$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

can be rewritten as:
  $$Q(s, a) \equiv r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a'),$$

thus getting rid of $V^*(\delta(s, a))$.

### Learning the $Q$ Function: Getting Rid of $r$ and $\delta$

In state $s$, execute action $a$, and observe immediate reward $r$ and resulting state $s'$. Then, simply use those $r$ and $s'$ you got without worrying about $r(s, a)$ or $\delta(s, a)$.

- Initialize the estimate $\hat{Q}(s, a)$ to zero.
- Iteratively update, with estimated function $\hat{Q}(s, a)$:
  $$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a').$$
The Q Learning Algorithm

1. For each \( s, a \), initialize the table entry \( \hat{Q}(s, a) \) to zero.
2. Observe the current state \( s \).
3. Do forever:
   - Select action \( a \) and execute.
   - Receive immediate reward \( r \).
   - Observe resulting state \( s' \).
   - Update table entry for \( \hat{Q}(s, a) \) as:
     \[
     \hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a').
     \]
   - \( s \leftarrow s' \)

\[Q\] Learning Properties

- For deterministic Markov decision processes
- \( \hat{Q} \) converges to \( Q \), when
  - process is deterministic MDP,
  - \( r \) is bounded (and nonnegative), and
  - actions are chosen so that every state-action pair is visited infinitely often.

Example

\[
\begin{array}{ccc}
 s_1 & 73 & s_2 \\
 s_4 & 66 & s_5 \\
 s_4 & 81 & s_6 \\
\end{array}
\]

(a) Initial state, in \( s_1 \)

\[
\begin{array}{ccc}
 s_1 & 90 & s_2 \\
 s_4 & 66 & s_5 \\
 s_4 & 81 & s_6 \\
\end{array}
\]

(b) Next state, in \( s_2 \)

Arrows represent the \( \hat{Q} \) values.

- Move right \( (a = a_{right}) \) and get immediate reward \( r = 0 \), with discount rate \( \gamma = 0.9 \):
  \[
  \hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')
  \leftarrow 0 + 0.9 \max\{66, 81, 100\}
  \leftarrow 90
  \]

- Note that in (b), the \( \hat{Q}(s_1, a_{right}) \) value is updated from 73 to 90.

Exercise, from scratch

\[
\begin{array}{cccc}
 s_1 & t=0 & s_2 & t=1 \\
 s_4 & t=0 & s_5 & t=1 \\
 s_4 & t=0 & s_6 & t=1 \\
\end{array}
\]

(a) Initial state \( Q(s, a) = 0 \)

\[
\begin{array}{cccc}
 s_1 & t=1 & s_2 & t=2 \\
 s_4 & t=0 & s_5 & t=2 \\
 s_4 & t=0 & s_6 & t=2 \\
\end{array}
\]

(b) After one iteration

- Robot moved from \( s_4 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \).
- How do the various \( Q(s, a) \) values get updated?
  - For the first iteration?
  - For the next iteration of \( s_4 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \)?
For this domain, following actions that have max $Q(s, a)$ will lead you to the goal through an optimal path.

---

**Proof of Convergence: Sketch**

- The table entry $\hat{Q}(s, a)$ with the largest error must have its error reduced by a factor of $\gamma$ whenever it is updated.
- The updated $\hat{Q}(s, a)$ will be based on the error-prone $\hat{Q}(s, a)$ only partially. The accurate immediate reward $r$ used in the $Q$ update rule will help reduce the error.
- Proof: Define a full interval to be an interval during which each table entry $(s, a)$ is visited. During each full interval the largest error in $\hat{Q}$ table is reduced by factor of $\gamma$.

---

**Convergence of $\hat{Q}$ to $Q$**

- Properties (for non-negative rewards):
  \[
  \forall s, a, n : \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a) \\
  \forall s, a, n : 0 \leq \hat{Q}_n + (s, a) \leq Q_n(s, a)
  \]
- In general, convergence is guaranteed under three conditions:
  1. The system is a deterministic MDP.
  2. The reward is bounded ($\forall s, a$) $|r(s, a)| < c$ for a fixed constant $c$.
  3. All $(s, a)$ pairs are visited infinitely often.

---

**Proof of Convergence: Sketch**

Let $\hat{Q}_n$ be table after $n$ updates, and $\Delta_n$ be the maximum error in $\hat{Q}_n$; that is
\[
\Delta_n = \max_{s, a} |\hat{Q}_n(s, a) - Q(s, a)|
\]

For any table entry $\hat{Q}_n(s, a)$ updated on iteration $n + 1$, the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is
\[
|\hat{Q}_{n+1}(s, a) - Q(s, a)| = |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\
= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\
\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \\
\leq \gamma \Delta_n
\]
Convergence in $Q$

- Main result:
  \[ |\hat{Q}_{n+1}(s, a) - Q(s, a)| \leq \gamma \Delta_n \]
- That is, error in the updated $\hat{Q}(s, a)$ is less than $\gamma$ times the max error in the table before the update.
- Note that $\gamma < 1.0$.
- Given initial $\Delta_0$, after $k$ visits to $⟨s, a⟩$, the error will be at most $\gamma^k \Delta_0$, and as $k \to \infty$, $\Delta_k \to 0$.

Constructing the Policy from the Learned $Q$

1. Greedy: given state $s$, pick $\arg\max_a Q(s, a)$.
   - May cause the agent to exploit early successes and ignore interesting possibilities.
   - This would prevent the agent from visiting all $(s, a)$ pairs infinitely often.
2. Probabilistic: pick action $a_i$ with probability:
   \[ P(a_i | s) = \frac{k\hat{Q}(s, a_i)}{\sum_j k\hat{Q}(s, a_j)} \]
   where $k > 0$ controls exploration (low $k$) vs. exploitation (high $k$, greedy).

Updating Sequence

No specific order of $(s, a)$ visit is necessary for convergence. However, this can be inefficient.

1. Perform update in reverse order, once the goal has been reached.
2. Store past state-action transitions.

Nondeterministic Case

What if reward and next state are non-deterministic?

We redefine $V, Q$ by taking expected values

\[ V^\pi(s) \equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots] \]
\[ \equiv E \left[ \sum_{i=0}^{\infty} \gamma^i r_{t+i} \right] \]

\[ Q(s, a) \equiv E[r(s, a) + \gamma V^*(\delta(s, a))] \]
Nondeterministic Case

$Q(s, a)$ can be redefined as follows:

\[
Q(s, a) \equiv E[r(s, a) + \gamma V^*(\delta(s, a))]
\]
\[
= E[r(s, a)] + \gamma E[V^*(\delta(s, a))]
\]
\[
= E[r(s, a)] + \gamma \sum_{s'} P(s' \mid s, a) V^*(s')
\]

Finally, rewriting it recursively, we get:

\[
Q(s, a) = E[r(s, a)] + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q(s', a')
\]

Temporal Difference Learning

$Q$ learning reduces the difference between $\hat{Q}$ of a state and its immediate successor (one-step look ahead). This can be generalized to include more distant successors.

$Q$ learning reduces the difference between $\hat{Q}$ of a state

- $\hat{Q}(s_{t+1}, a_t)$ is estimated based $\hat{Q}(s_{t+1}, \cdot)$, where $s_{t+1} = \delta(s_t, a_t)$.
- One-step look ahead:
  \[
  Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)
  \]
- Two-step look ahead:
  \[
  Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)
  \]
- $n$-step look ahead:
  \[
  Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)
  \]

Non-deterministic Case: Learning

Using the original learning rule can result in oscillation in $\hat{Q}(s, a)$, and thus no convergence. Taking a decaying weighted average can solve the problem:

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

where

\[
\alpha_n = \frac{1}{1 + \text{visits}_s(s, a)}
\]

and $\alpha$ determines how much the old and new $\hat{Q}$ values will be used.

The $\alpha_n$ formula above is known to allow convergence (there can be other formulas).

Learning in TD

$TD(\lambda)$ for learning $Q$ using various lookaheads ($0 \leq \lambda \leq 1$):

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

which can be rewritten recursively:

\[
Q^\lambda(s_t, a_t) = (1 - \lambda) \left[ Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \ldots \right]
\]

\[
= r_t + \gamma (1 - \lambda) \max_a Q(s_t, a) + \gamma \left[ r_{t+1} + \gamma (1 - \lambda) \max_a Q(s_{t+1}, a) + \ldots \right]
\]

\[
= r_t + \gamma \left[ (1 - \lambda) \max_a Q(s_t, a) + \lambda Q^\lambda(s_{t+1}, a_{t+1}) \right]
\]
**TD(λ) Properties**

\[
Q^\lambda(s_t, a_t) = r_t + \gamma \left[ (1 - \lambda) \max_a \hat{Q}(s_t, a_t) + \lambda Q^\lambda(s_{t+1}, a_{t+1}) \right]
\]

- TD(0): same as \(Q^{(1)}\).
- TD(1): only observed \(r_{t+i}\) values are considered.
- When \(Q = \hat{Q}\), \(Q^\lambda\) values are the same for any \(0 \leq \lambda \leq 1\).

**TD(λ) Properties**

- Sometimes converges faster than \(Q\) learning
- Converges for learning \(V^*\) for any \(0 \leq \lambda \leq 1\) (Dayan, 1992)
- Tesauro’s TD-Gammon uses this algorithm

**Subtleties and Ongoing Research**

- Replace \(\hat{Q}\) table with neural net or other generalizer.
- Handle case where state is only partially observable (partially observable MDP, or POMDP).
- Design optimal exploration strategies.
- Extend to continuous action, state.
- Learn and use \(\hat{\delta}: S \times A \rightarrow S\).
- Relationship to dynamic programming.