Q-learning

- Q-learning is a temporal difference learning algorithm that learns optimal values for Q (instead of V, as value iteration did).
- The algorithm works in episodes, where the agent "practices" (aka *samples*) the MDP to learn which actions obtain the most rewards.
- Like value iteration, table of Q values eventually converge to Q*.

  (under certain conditions)

Initialize $Q[s,a]$ arbitrarily, e.g., $Q[s,a] = 0$ for all (s,a) pairs.
Repeat (for each episode):
    Set s to the start state
    Repeat (for each step of the episode):
        Choose action a from state s using policy derived from Q (see note below)
        Take action a, observe reward r, new state s'
        $Q[s,a] \leftarrow Q[s,a] + \alpha\,[r + \gamma \max_{a'} Q[s',a'] - Q[s,a]]$
        $s \leftarrow s'$
    until s is a final state
Output a policy π where $\pi(s) = \text{argmax}_a\, Q(s,a)$

- Notice the Q[s, a] update equation is very similar to the basic TD update equation.
  - (The extra $\gamma \max_{a'}$ Q[s', a'] piece is to handle future rewards.)
  - alpha ($0 < \alpha <= 1$) is called the learning rate; it controls how fast the algorithm learns. In stochastic environments, alpha is usually small, such as 0.1.

Initialize $Q[s,a]$ arbitrarily, e.g., $Q[s,a] = 0$ for all (s,a) pairs.
Repeat (for each episode):
    Set s to the start state
    Repeat (for each step of the episode):
        Choose action a from state s using policy derived from Q (see note below)
        Take action a, observe reward r, new state s'
        $Q[s,a] \leftarrow Q[s,a] + \alpha\left[r + \gamma \max_{a'} Q[s',a'] - Q[s,a]\right]$
        $s \leftarrow s'$
    until s is a final state
Output a policy π where $\pi(s) = \operatorname{argmax}_a Q(s,a)$

- Note: The "choose action" step does not mean you choose the best action according to your table of Q values.
- You must balance exploration and exploitation; like in the real world, the algorithm learns best when you "practice" the best policy often, but sometimes explore other actions that may be better in the long run.

Initialize $Q[s, a]$ arbitrarily, e.g., $Q[s, a] = 0$ for all (s, a) pairs.
Repeat (for each episode):
    Set s to the start state
    Repeat (for each step of the episode):
        Choose action a from state s using policy derived from Q (see note below)
        Take action a, observe reward r, new state s'
        $Q[s, a] \leftarrow Q[s, a] + \alpha \left[r + \gamma \max_{a'} Q[s', a'] - Q[s, a]\right]$
        $s \leftarrow s'$
    until s is a final state
Output a policy π where $\pi(s) = \operatorname{argmax}_a Q(s, a)$

- Often the "choose action" step uses policy that mostly exploits but sometimes explores.
- One common idea: (epsilon-greedy policy)
  - With probability 1 - ε, pick the best action (the "a" that maximizes Q[s, a].
  - With probability ε, pick a random action.
- Also common to start with large ε and decrease over time while learning.

Initialize $Q[s,a]$ arbitrarily, e.g., $Q[s,a] = 0$ for all (s,a) pairs.
Repeat (for each episode):
  Set s to the start state
  Repeat (for each step of the episode):
    Choose action a from state s using policy derived from Q (see note below)
    Take action a, observe reward r, new state s'
    $Q[s,a] \leftarrow Q[s,a] + \alpha\,[r + \gamma \max_{a'} Q[s',a'] - Q[s,a]]$
    $s \leftarrow s'$
  until s is a final state
Output a policy π where $\pi(s) = \operatorname{argmax}_a Q(s,a)$

- What makes Q-learning so amazing is that the Q-values still converge to the optimal Q* values even though the algorithm itself is not following the optimal policy!

Simple Blackjack

- Costs $5 to play.
- Infinite deck of shuffled cards, labeled 1, 2, 3.
  - (so equal prob of drawing each number at any time)
- You start with no cards. At every turn, you can either "hit" (take a card) or "stay" (end the game). Your goal is to get to a sum of 6 without going over, in which case you lose the game.
- You make all your decisions first, then the dealer plays the same game.
- If your sum is higher than the dealer's, you win $10 (your original $5 back, plus another $5).
  If lower, you lose (your original $5).
  If the same, draw (get your $5 back).

Simple Blackjack

- To set this up as an MDP, we need to automate the 2nd player (the dealer) in the MDP.
- Usually at casinos, dealers have simple rules they have to follow anyway about when to hit and when to stay.
- Is it ever optimal to "stay" from S0-S3?
- Assume that on average, if we "stay" from S4/S5/S6, and then the dealer plays, here's what happens:
  - Stay from S4, we win $3 (net $-2).
  - Stay from S5, we win $6 (net $1).
  - Stay from S6, we win $7 (net $2).
- Do you even want to play this game? (Does it make financial sense?)
- What should gamma be?

Q-learning with Blackjack

- Update formula:

$$Q[s,a] \leftarrow Q[s,a] + \alpha \left[r + \gamma \max_{a'} Q[s',a'] - Q[s,a] \right]$$

- Sample episodes (states and actions):

S0 ➔ Hit ➔ S3 ➔ Stay ➔ End

S0 ➔ Hit ➔ S3 ➔ Hit ➔ S6 ➔ Stay ➔ End

S0 ➔ Hit ➔ S3 ➔ Hit ➔ S5 ➔ Stay ➔ End

2-Player Q-learning

Normal update equation:

$$Q[s,a] \leftarrow Q[s,a] + \alpha \left[r + \gamma \max_{a'} Q[s',a'] - Q[s,a] \right]$$

Normally we always maximize our rewards. Consider ***2-player Q-learning*** with player A maximizing and player B minimizing (as in minimax).

Why does this break the update equation?

2-Player Q-learning

Player A's update equation:

$$Q[s,a] \leftarrow Q[s,a] + \alpha \left[r + \gamma \min_{a'} Q[s',a'] - Q[s,a] \right]$$

Player B's update equation:

$$Q[s,a] \leftarrow Q[s,a] + \alpha \left[r + \gamma \max_{a'} Q[s',a'] - Q[s,a] \right]$$

Player A's optimal policy output:

$$\pi(s) = \operatorname*{argmax}_{a} Q[s,a]$$

Player B's optimal policy output:

$$\pi(s) = \operatorname*{argmin}_{a} Q[s,a]$$