Reinforcement Learning

Environments

- Fully-observable vs partially-observable
- Single agent vs multiple agents
- Deterministic vs stochastic
- Episodic vs sequential
- Static or dynamic
- Discrete or continuous

What is reinforcement learning?

- Three machine learning paradigms:
  - Supervised learning
  - Unsupervised learning (overlaps w/ data mining)
  - Reinforcement learning
- In reinforcement learning, the agent receives incremental pieces of feedback, called rewards, that it uses to judge whether it is acting correctly or not.

Examples of real-life RL

- Learning to play chess.
- Animals (or toddlers) learning to walk.
- Driving to school or work in the morning.

- **Key idea**: Most RL tasks are *episodic*, meaning they repeat many times.
  - So unlike in other AI problems where you have one shot to get it right, in RL, it's OK to take time to try different things to see what's best.

n-armed bandit problem

- You have n slot machines.
- When you play a slot machine, it provides you a reward (negative or positive) according to some fixed probability distribution.
- Each machine may have a different probability distribution, and you don't know the distributions ahead of time.
- You want to maximize the amount of reward (money) you get.
- In what order and how many times do you play the machines?

RL problems

- Every RL problem is structured similarly.
- We have an ***environment***, which consists of a set of ***states***, and ***actions*** that can be taken in various states.
  - Environment is often stochastic (there is an element of chance).
  - Environment can be fully or partially observable (here, we will focus on fully observable).
- Our RL agent wishes to learn a ***policy***, π, a function that maps states to actions.
  - $\pi(s)$ tells you what action to take in a state s.

What is the goal in RL?

- In other AI problems, the "goal" is to get to a certain state. Not in RL!
- A RL environment gives feedback every time the agent takes an action. This is called a ***reward***.
  - Rewards are usually numbers.
  - Goal: Agent wants to maximize the amount of reward it gets over time.
  - Critical point: Rewards are given by the environment, not the agent.

Mathematics of rewards

- Assume our rewards are r_0, r_1, r_2, …
- What expression represents our total rewards?
- How do we maximize this? Is this a good idea?
- Use discounting: at each time step, the reward is discounted by a factor of γ (called the discount rate).
- Future rewards from time t =

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Markov Decision Processes

- An MDP has a set of states, S, and a set of actions, A(s), for every state s in S.
- An MDP encodes the probability of transitioning from state s to state s' on action a: **P(s' | s, a)**
- RL also requires a reward function, usually denoted by **R(s, a, s')** = reward for being in state s, taking action a, and arriving in state s'.
- An MDP is a Markov chain that allows for outside actions to influence the transitions.




- Grass gives a reward of 0.
- Monster gives a reward of -5.
- Pot of gold gives a reward of +10 (and ends game).
- Two actions are always available:
  - Action A: 50% chance of moving right 1 square,
    50% chance of staying where you are.
  - Action B: 50% chance of moving right 2 squares,
    50% chance of moving left 1 square.
  - Any movement that would take you off the board moves you as far in that direction as possible or keeps you where you are.

Value functions

- Almost all RL algorithms are based around computing, estimating, or learning ***value functions***.
- A value function represents the ***expected future reward*** from either a state, or a state-action pair.
  - $V^{\pi}(s)$: If we are in state s, and follow policy π, what is the total future reward we will see, on average?
  - $Q^{\pi}(s, a)$: If we are in state s, and take action a, then follow policy π, what is the total future reward we will see, on average?

Optimal policies

- Given an MDP, there is always a "best" policy, called π^*.
- The point of RL is to discover this policy by employing various algorithms.
  - Some algorithms can use sub-optimal policies to discover π^*.
- We denote the value functions corresponding to the optimal policy by $V^*(s)$ and $Q^*(s, a)$.

Bellman equations

- The V*(s) and Q*(s, a) functions always satisfy certain recursive relationships for any MDP.
- These relationships, in the form of equations, are called the ***Bellman equations.***

Recursive relationship of V* and Q*:

$$V^*(s) = \max_a Q^*(s, a)$$

The expected future rewards from a state s is equal to the expected future rewards obtained by choosing the best action from that state.

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

The expected future rewards obtained by taking an action from a state is the weighted average of the expected future rewards from the new state.

Bellman equations

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

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

- No closed-form solution in general.
- Instead, most RL algorithms use these equations in various ways to estimate V* or Q*. An optimal policy can be derived from either V* or Q*.

RL algorithms

- Goal of RL algorithm: estimate V* or Q* (and then derive optimal policy π^* from that.
- One way of classifying RL algorithms by whether or not the algorithm requires a full model of the environment.
- In other words, do we know P(s' | s, a) and R(s, a, s') for all combinations of s, a, s'?
  - If we have this information (uncommon in the real world), we can estimate V* or Q* directly with very good accuracy.
  - If we don't have this information, we can estimate V* or Q* from experience or simulations.

Value iteration

- ***Value iteration*** is an algorithm that computes an optimal policy, given a full model of the environment.
- Algorithm is derived directly from the Bellman equations (usually for V*, but can use Q* as well).

Value iteration

- ***Two steps:***
- Estimate V(s) for every state.
  - For each state:
    - Simulate taking every possible action from that state and examine the probabilities for transitioning into every possible successor state. Weight the rewards you would receive by the probabilities that you receive them.
    - Find the action that gave you the most reward, and remember how much reward it was.
- Compute the optimal policy by doing the first step again, but this time remember the actions that give you the most reward, not the reward itself.

Value iteration

- Value iteration maintains a table of V values, one for each state. Each value V[s] eventually converges to the true value V*(s).

Initialize V arbitrarily, e.g., $V[s] = 0$ for all states s.
Repeat
    for each state s:
        $V_{\text{new}}[s] \leftarrow \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V[s']\right]$
    $V \leftarrow V_{\text{new}}$ (copy new table over old)
until the maximum difference in new and old values is smaller than some threshold
Output a policy π where $\pi(s) = \text{argmax}_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V[s']\right]$




- Grass gives a reward of 0.
- Monster gives a reward of -5.
- Pot of gold gives a reward of +10 (and ends game).
- Two actions are always available:
  - Action A: 50% chance of moving right 1 square,
    50% chance of staying where you are.
  - Action B: 50% chance of moving right 2 squares,
    50% chance of moving left 1 square.
  - Any movement that would take you off the board moves you as far in that direction as possible or keeps you where you are.
- γ (gamma) = 0.9

V[s] values converge to:




6.47 7.91 8.56 0

How do we use these to compute π(s)?

Computing an optimal policy from V[s]

- Last step of the value iteration algorithm:

$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V[s']]$$

- In other words, run one last time through the value iteration equation for each state, and pick the action a for each state s that maximizes the expected reward.

V[s] values converge to:




6.47 7.91 8.56 0

Optimal policy:

A B B ---

Review

- Value iteration requires a perfect model of the environment.
  - You need to know P(s' | s, a) and R(s, a, s') ahead of time for all combinations of s, a, and s'.
  - Optimal V or Q values are computed directly from the environment using the Bellman equations.
- Often impossible or impractical.

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?)

Simple Blackjack

- What should gamma be?
- Assume we have finished one round of value iteration.
- Complete the second round of value iteration for S1—S6.

Learning from experience

- What if we don't know the exact model of the environment, but we are allowed to ***sample*** from it?
  - That is, we are allowed to "practice" the MDP as much as we want.
  - This echoes real-life experience.
- One way to do this is ***temporal difference learning (TD learning).***

Temporal difference learning

- We want to compute V(s) or Q(s, a).
- TD learning uses the idea of taking lots of samples of V or Q (from the MDP) and averaging them to get a good estimate.
- Let's see how TD learning works.

Example: Rolling a die

- Basic TD equation:
- $V(s) = V(s) + \alpha(\text{reward} - V(s))$
- But what if our reward comes in pieces, not all at once?
- total reward = one step reward + rest of reward
- total reward = $r_t + \gamma V(s')$
- $V(s) = V(s) + \alpha[r_t + \gamma V(s') - V(s)]$

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 driving time 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'
        $Q[s, a] \leftarrow Q[s, a] + \alpha \left[r + \gamma \max_{a'} Q[s', a'] - Q[s, a] \right]$
        $s \leftarrow s'$
    until s is a final state
Output a policy π where $\pi(s) = \operatorname{argmax}_a Q(s, a)$

- 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!

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:

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