Policies and value functions

- Almost all RL algorithms are based around computing, estimating, or learning ***policies*** and ***value functions***.
- A policy (usually π) is a function from states to actions that tells you what action you should do in each state.
- 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.

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

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

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

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

- 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).

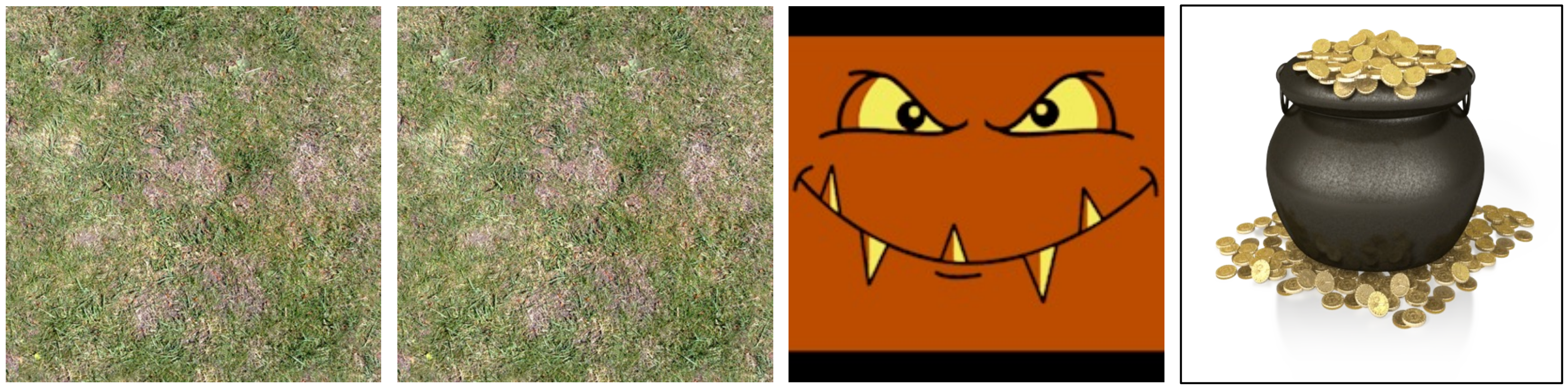
Value iteration

- ***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

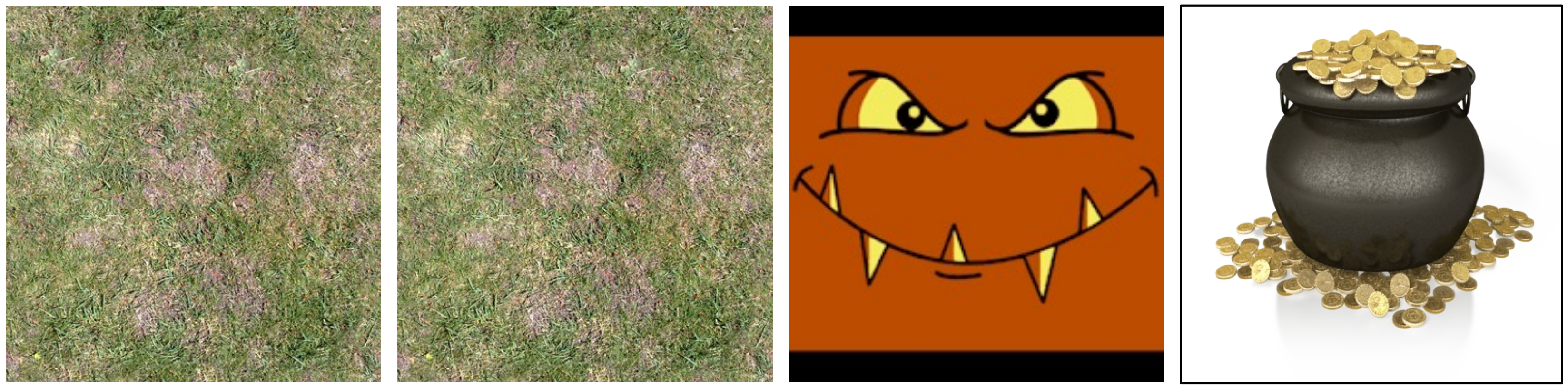
- Value iteration maintains a table of V values, one for each state. Each value V[s] eventually converges to the true value V*(s).

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



- 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

Running value iteration:



initial:	0	0	0	0
after rd 1:	0	5	5	0
after rd 2:	2.25	5	7.25	0

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:

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

- 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	---

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 = $r_t + \gamma V(s')$
- $V(s) = V(s) + \alpha[r_t + \gamma V(s') - V(s)]$