

Lecture 17: Markov Decision Processes

- Value functions for given policies
- Optimal value functions
- Bellman equations for value functions
- Dynamic programming methods (DP)
- Monte Carlo methods

Recall from last time: Markov Decision Processes (MDPs)



Assume:

- Set of states S
- Set of actions $A(s)$ available in each state s
- γ = discount factor for later rewards (between 0 and 1, usually close to 1)
- Markov assumption: s_{t+1} and r_{t+1} depend only on s_t , a_t and not on anything that happened before t

Models for MDPs

- r_s^a = expected value of the immediate reward if the agent is in s and does action a

$$r_s^a = E_{r_{t+1}} \{s_t = s, a_t = a\}$$

- $p_{ss'}^a$ = probability of going from s to s' when doing action a

$$p_{ss'}^a = E_{s_{t+1}=s'} \{s_t = s, a_t = a\}$$

These form the model of the environment, and are *usually unknown*

Value Functions

For each possible policy π that the agent might adopt, we can define an evaluation function over states.

The **value of a state** is the expected return starting from that state, when following the policy:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s\right\}$$

The value of taking action a in state s and following policy π afterwards is the expected return when starting in that state, taking the action and following π afterwards:

$$Q^\pi(s, a) = E_\pi\{R_t \mid s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s, a_t = a\right\}$$

Bellman Equation for Policy π

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

Based on this observation, V^π becomes:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\}$$

Without the expectation:

$$V^\pi(s) = \sum_a \pi(s, a) \left(r_s^a + \gamma \sum_{s'} P_{ss'}^a V^\pi(s') \right)$$

This is a system of *linear equations* whose unique solution is V^π .

Iterative Policy Evaluation

Main idea: turn Bellman equation into an update rule.

1. Start with some initial guess V_0
2. During every iteration k , perform a **full backup of the value function**:

$$V_{k+1}(s) \leftarrow \sum_a \pi(s, a) \left(r_s^a + \gamma \sum_{s'} P_{ss'}^a V_k(s') \right)$$

3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)

Key idea: bootstrapping!

The value of one state is updated based on the values of the other states

Optimal Value Functions

- Policies can be partially ordered: $\pi \succeq \pi'$ iff $V^{\pi}(s) \geq V^{\pi'}(s) \forall s$
- In an MDP there always exists at least one policy better than all others. This is called the **optimal policy**, π^* .
- The **optimal state-value function** is the value function shared by all optimal policies:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \forall s \in S$$

- Similarly, we can define the **optimal action-value function**:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a), \forall s \in S, \forall a \in A$$

This is the expected value for taking action a in state s and following an optimal policy afterwards

Bellman Optimality Equation for V^*

The value of a state under the optimal policy must be equal to the expected return for the best action in the state:

$$\begin{aligned} V^*(s) &= \max_a Q^*(s, a) \\ &= \max_a E_{r_{t+1} + \gamma V^*(s_{t+1})} \{s_t = s, a_t = a\} \\ &= \max_a r_s^a + \gamma \sum_{s'} p_{ss'}^a V^*(s') \end{aligned}$$

V^* is the **unique solution** of this system of non-linear equations

Bellman Optimality Equation for Q^*

$$\begin{aligned} Q^*(s, a) &= E_{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a')} \{s_t = s, a_t = a\} \\ &= r_s^a + \gamma \sum_{s'} P_{ss'}^a \max_{a'} Q^*(s', a') \end{aligned}$$

Q^* is the **unique solution** of this system of non-linear equations.

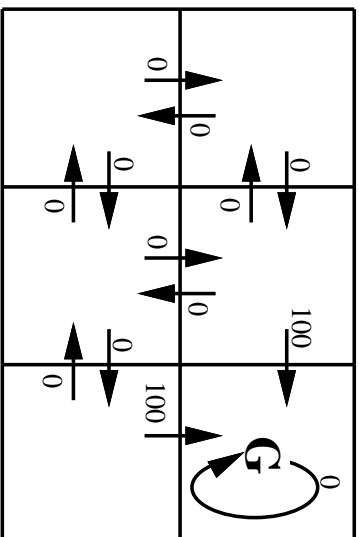
Why Optimal Value Functions are Useful

Any policy that is greedy with respect to V^* is an optimal policy!

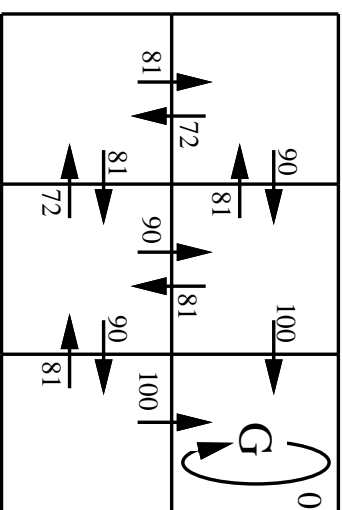
- If we know V^* and the model of the environment, one step of look-ahead will tell us what the optimal action is
- If we know Q^* , look-ahead is not even needed!

$$\pi^*(s) = \arg \max_a Q^*(s, a), \forall s$$

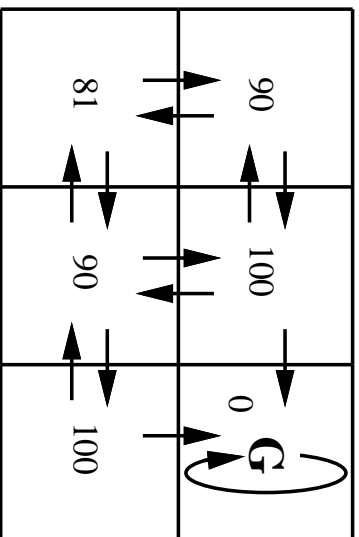
Illustration: A Deterministic Gridworld



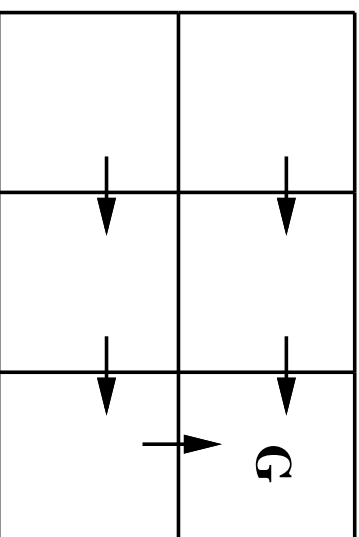
$r(s, a)$ (immediate reward) values



$Q^*(s, a)$ values ($\gamma = 0.9$)

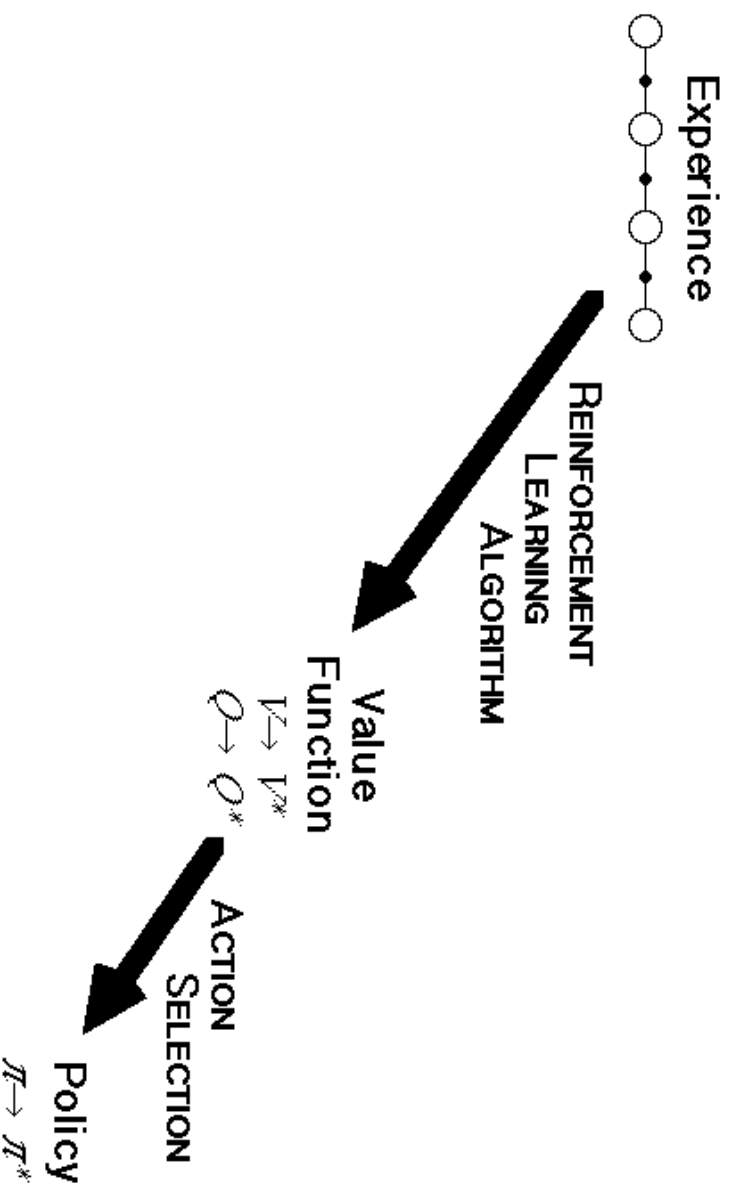


$V^*(s)$ values



One optimal policy

What RL Algorithms Do



Continual, on-line learning

Many RL methods can be understood as trying to solve the Bellman optimality equations in an approximate way.

Policy Improvement

Suppose we have computed V^π for some deterministic policy π

When is it better to do an action $a \neq \pi(s)$?

$$Q^\pi(s, a) > V^\pi(s)$$

If we make the change at all states, we get a policy π' which is

greedy with respect to Q^π :

$$\pi'(s) = \arg \max_a Q^\pi(s, a) = \arg \max_a r_s^a + \gamma \sum_{s'} p_{ss'}^a V^\pi(s')$$

Then $V^{\pi'}(s) \geq V^\pi(s), \forall s$

Policy Improvement (continued)

What if at some point $V^{\pi'} = V^{\pi}$?

Then we have:

$$V^{\pi}(s) = \max_a r_s^a + \gamma \sum_{s'} P_{ss'}^a V^{\pi}(s')$$

But this is the Bellman optimality equation!

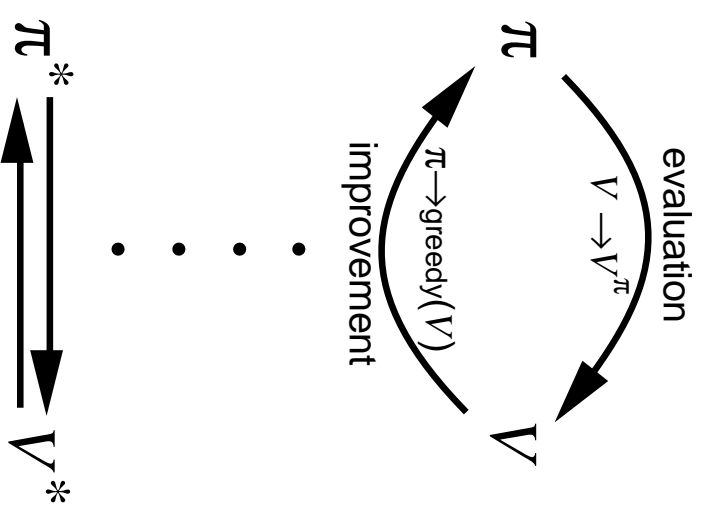
So if the value does not change at some point, both π and π' are optimal.

Policy Iteration

1. Start with an initial policy π_0
2. Repeat:
 - (a) Compute V^{π_i} using policy evaluation
 - (b) Compute a new policy π_{i+1} that is greedy with respect to V^{π_i}
until $V^{\pi_i} = V^{\pi_{i+1}}$

Generalized Policy Iteration

Any combination of policy evaluation and policy improvement steps, even if they are not complete



Value Iteration

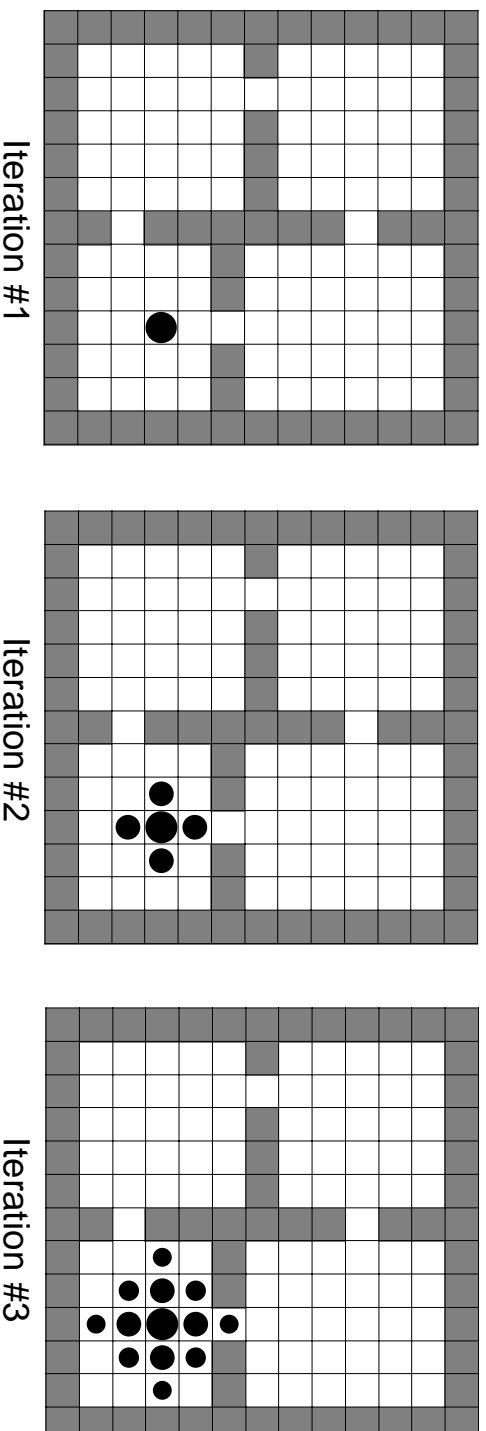
Main idea: Turn the Bellman optimality equation into an update rule
(same as done in policy evaluation):

1. Start with an arbitrary initial approximation V_0
2. $V_{k+1}(s) \leftarrow \max_a r_s^a + \gamma \sum_{s'} P_{ss'}^a V_k(s), \forall s$

Illustration: Rooms Example

Four actions, fail 30% of the time

No rewards until the goal is reached, $\gamma = 0.9$.



How do we tie learning with dynamic programming?

- Observe transitions in the environment, learn an *approximate*

$$\text{model } \hat{r}_s^a, \hat{P}_{ss'}^a$$

Note that this is just a supervised learning problem!

- Pretend the approximate model is correct and use it for any dynamic programming method
- This approach is called **model-based RL**
- Many believers, especially in the robotics community

Asynchronous dynamic programming

- All the methods described so far require sweeps over the entire state space
 - A more efficient idea: repeatedly pick states at random, and apply a backup, until some convergence criterion is met
 - How should states be selected?
- Based on the agent's experience!** I.e. along trajectories.
- Still needs lots of computation, but does not get locked into very long sweeps

Efficiency of DP

- Good news: finding an optimal policy is polynomial in the number of states

- Bad news: finding an optimal policy is polynomial in the number of states!

Number of states is often astronomical; typically number of states is exponential in the number of state variables

- In practice, classical DP can be applied to problems with a few millions states
- Asynchronous DP can be applied even to larger problems, and is appropriate for parallel computation
- It is surprisingly easy to find problems for which DP methods are not feasible

Monte Carlo Methods

- Suppose we have an episodic task (trials terminate at some point)
- The agent behave according to some policy π for a while, generating several trajectories. How can we compute V^π ?
- Compute $V^\pi(s)$ by **averaging the observed returns** after s on the trajectories in which s was visited.
- Two main approaches:
 - Every-visit: average returns for every time a state is visited in a trial
 - First-visit: average returns only for the first time a state is visited in a trial

Implementation of Monte Carlo Policy Evaluation

$$\begin{aligned} V_{n+1} &= \frac{1}{n+1} \sum_{i=1}^{n+1} R_i = \frac{1}{n+1} \left(\sum_{i=1}^n R_i + R_{n+1} \right) \\ &= \frac{n}{n+1} \frac{1}{n} \sum_{i=1}^n R_i + \frac{1}{n+1} R_{n+1} \\ &= \frac{n}{n+1} V_n + \frac{1}{n+1} R_{n+1} \end{aligned}$$

If we do not want to keep counts of how many times states have been visited, we can use a *learning rate* version:

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$$

Monte Carlo Estimation of Q values

- We use the same idea: $Q^{\pi}(s, a)$ is the average of the returns obtained by starting in state s , doing action a and then following π
- Like the state-value version, it converges asymptotically *if every state-action pair is visited*
- But π might not choose every action in every state!
- *Exploring starts*: Every state-action pair has a non-zero probability of being the starting pair

Dynamic Programming vs. Monte Carlo

	DP	MC
Need model	yes	no (+)
Bootstrapping	yes (+)	no
Learn directly from interaction	no	yes (+)
Focus on visited states	no	yes (+)

Can we combine the advantages of both methods?