Decision Making and Reinforcement Learning

Module 4: Dynamic Programming

Tony Dear, Ph.D.

Department of Computer Science

School of Engineering and Applied Sciences

Topics

- Dynamic programming for solving MDPs
- Value iteration
- Policy iteration
- Algorithm implementation and properties

Learning Objectives

- **Use** the idea of dynamic programming to solve the nonlinear Bellman equations
- **Implement** the value iteration and policy iteration algorithms
- **Understand** properties of algorithm complexity and convergence
- **Compare & contrast** dynamic programming approaches

MDPs and Bellman Equations

- **MDPs**: Stochastic, sequential decision problems
- **Policy** $\pi: S \rightarrow A$, assignment of action to each state
- **Value** $V^{\pi}: S \rightarrow \mathbb{R}$, expected state utilities from following π

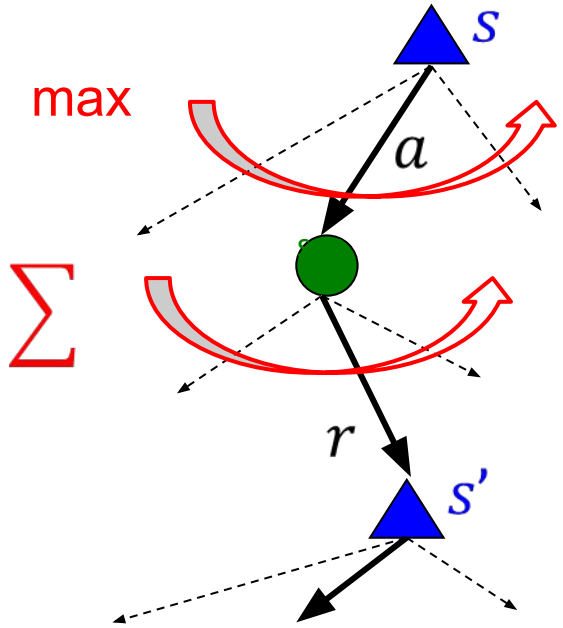
MDPs and Bellman Equations

- **MDPs**: Stochastic, sequential decision problems
- **Policy** $\pi: S \rightarrow A$, assignment of action to each state
- **Value** $V^{\pi}: S \rightarrow \mathbb{R}$, expected state utilities from following π

- **Bellman optimality equations**:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$



Time-Limited Values

- Problem: V^* is a function of infinitely many successor state values

Time-Limited Values

- Problem: V^* is a function of infinitely many successor state values

- Idea #1: We can *approximate* a discounted infinite sequence with a sufficiently long discounted *finite* sequence

Time-Limited Values

- Problem: V^* is a function of infinitely many successor state values
- Idea #1: We can *approximate* a discounted infinite sequence with a sufficiently long discounted *finite* sequence
- Idea #2: The utility of a sequence with $i + 1$ rewards can be computed if we have the utility of its *subsequence* with i rewards

Time-Limited Values

- Problem: V^* is a function of infinitely many successor state values
- Idea #1: We can *approximate* a discounted infinite sequence with a sufficiently long discounted *finite* sequence
- Idea #2: The utility of a sequence with $i + 1$ rewards can be computed if we have the utility of its *subsequence* with i rewards
- To solve for V^*, we will compute the **time-limited values** of successively larger *finite-horizon* MDPs

Time-Limited Values

- Let i be the number of transitions taken starting from a state s

Time-Limited Values

- Let i be the number of transitions taken starting from a state s

- Base case: $i = 0$. If no transitions allowed, $V_0(s) = 0 \; \forall s$
- No actions taken, no rewards received

Time-Limited Values

- Let i be the number of transitions taken starting from a state s

- Base case: $i = 0$. If no transitions allowed, $V_0(s) = 0 \; \forall s$
- No actions taken, no rewards received

- $i = 1$: $V_1(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_0(s')]$

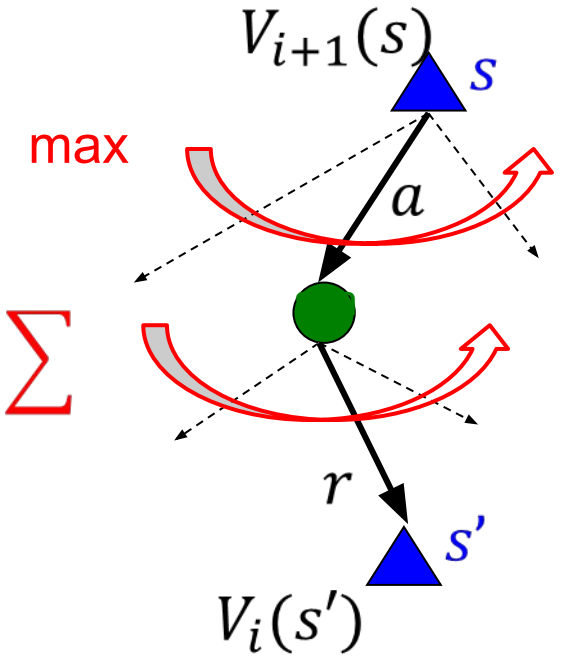
Time-Limited Values

- Let i be the number of transitions taken starting from a state s
- Base case: $i = 0$. If no transitions allowed, $V_0(s) = 0 \; \forall s$
- No actions taken, no rewards received
- $i = 1$: $V_1(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_0(s')]$
- $i = 2$: $V_2(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_1(s')]$

Time-Limited Values

- Given time-limited values V_i, we can compute V_{i+1} using the **Bellman update** using *discounted* V_i:

$$V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



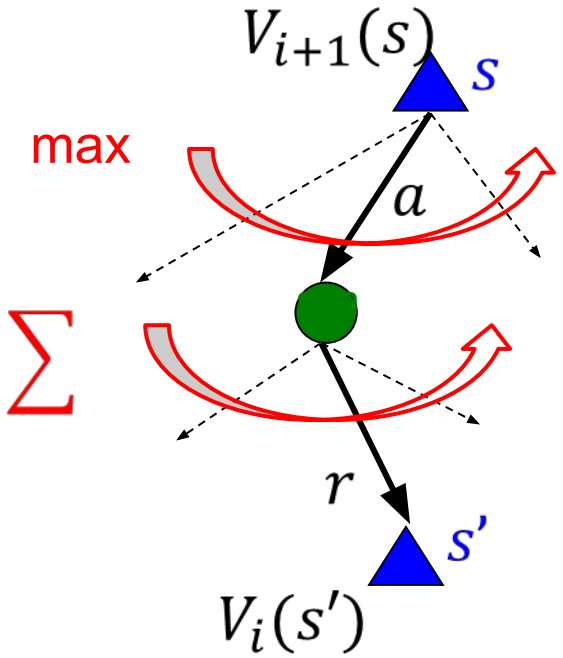
Time-Limited Values

- Given time-limited values V_i, we can compute V_{i+1} using the **Bellman update** using *discounted* V_i:

$$V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$

- The function V_i approaches V^* as $i \to \infty$

- Practically, we only need to find V_i for "sufficiently large" i assuming $0 < \gamma < 1$



Value Iteration

- **Value iteration** is a *dynamic programming* approach to solving for V^*
- Idea: Compute time-limited values V_i for all states, then use them to compute V_{i+1}
- Continue doing so for successively increasing i until *convergence*

Value Iteration

- **Value iteration** is a *dynamic programming* approach to solving for V^*
- Idea: Compute time-limited values V_i for all states, then use them to compute V_{i+1}
- Continue doing so for successively increasing i until *convergence*

- Initialize: $V_0(s) \leftarrow 0$ for all states s
- **Loop** from $i = 0$:

Value Iteration

- **Value iteration** is a *dynamic programming* approach to solving for V^*
- Idea: Compute time-limited values V_i for all states, then use them to compute V_{i+1}
- Continue doing so for successively increasing i until *convergence*

- Initialize: $V_0(s) \leftarrow 0$ for all states s
- **Loop** from $i = 0$:
  - **For** each state $s \in S$:

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_i(s')]$$

Value Iteration

- **Value iteration** is a *dynamic programming* approach to solving for V^*
- Idea: Compute time-limited values V_i for all states, then use them to compute V_{i+1}
- Continue doing so for successively increasing i until *convergence*

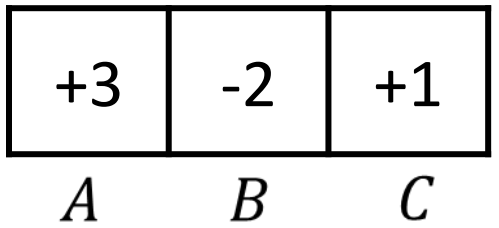
- Initialize: $V_0(s) \leftarrow 0$ for all states s
- **Loop** from $i = 0$:
  - **For** each state $s \in S$:

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_i(s')]$$

- **Until** $\max_s |V_{i+1}(s) - V_i(s)| < \epsilon$ (small threshold)

Example: Mini-Gridworld

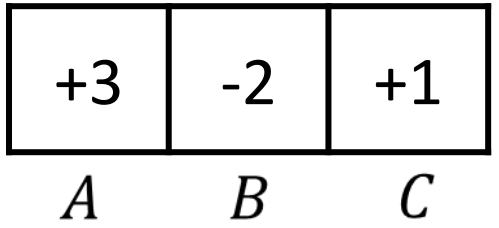
$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



- States, actions, rewards as shown; $\gamma = 0.5$
- Transitions: $\Pr(\text{intended direction}) = 0.8$, $\Pr(\text{opposite direction}) = 0.2$
- Initialize: $\big(V_0(A), V_0(B), V_0(C)\big) = (0,0,0)$

Example: Mini-Gridworld

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



- States, actions, rewards as shown; $\gamma = 0.5$
- Transitions: $\Pr(\text{intended direction}) = 0.8$, $\Pr(\text{opposite direction}) = 0.2$
- Initialize: $\big(V_0(A), V_0(B), V_0(C)\big) = (0,0,0)$

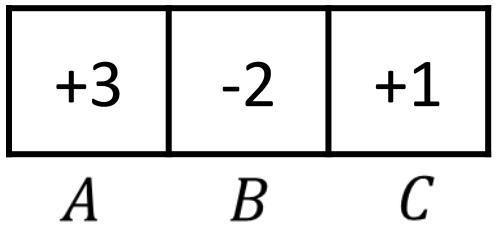
L action

$$V_1(A) = \max\big[0.8\big(3 + 0.5(0)\big) + 0.2\big(-2 + 0.5(0)\big), \quad\big]$$

$s' = A$ $\qquad$ $s' = B$

Example: Mini-Gridworld

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



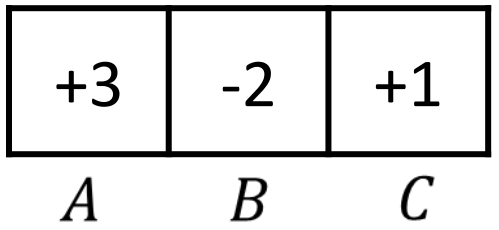
- States, actions, rewards as shown; $\gamma = 0.5$
- Transitions: $\Pr(\text{intended direction}) = 0.8$, $\Pr(\text{opposite direction}) = 0.2$
- Initialize: $\big(V_0(A), V_0(B), V_0(C)\big) = (0,0,0)$

$$V_1(A) = \max\Big[\underbrace{0.8\big(3 + 0.5(0)\big) + 0.2\big(-2 + 0.5(0)\big)}_{L \text{ action}}, \underbrace{0.8\big(-2 + 0.5(0)\big) + 0.2\big(3 + 0.5(0)\big)}_{R \text{ action}}\Big]$$

$s' = A$ $\quad$ $s' = B$ $\quad$ $s' = B$ $\quad$ $s' = A$

Example: Mini-Gridworld

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



- States, actions, rewards as shown; $\gamma = 0.5$
- Transitions: $\Pr(\text{intended direction}) = 0.8$, $\Pr(\text{opposite direction}) = 0.2$
- Initialize: $\big(V_0(A), V_0(B), V_0(C)\big) = (0,0,0)$

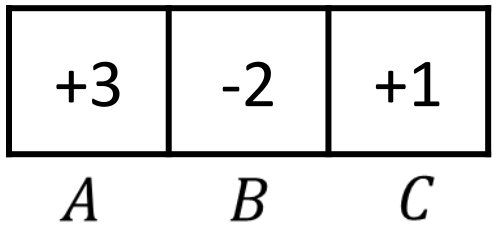
L action $\qquad\qquad$ R action

$$V_1(B) = \max\big[0.8\big(3 + 0.5(0)\big) + 0.2\big(1 + 0.5(0)\big), 0.8\big(1 + 0.5(0)\big) + 0.2\big(3 + 0.5(0)\big)\big]$$

$s' = A$ $\qquad$ $s' = C$ $\qquad$ $s' = C$ $\qquad$ $s' = A$

Example: Mini-Gridworld

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



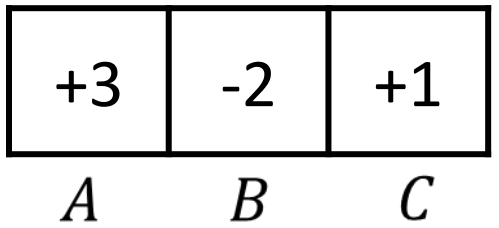
- States, actions, rewards as shown; $\gamma = 0.5$
- Transitions: $\Pr(\text{intended direction}) = 0.8$, $\Pr(\text{opposite direction}) = 0.2$
- Initialize: $\big(V_0(A), V_0(B), V_0(C)\big) = (0,0,0)$

$$V_1(C) = \max\Big[\underbrace{0.8\big(-2 + 0.5(0)\big)}_{s'=B} + \underbrace{0.2\big(1 + 0.5(0)\big)}_{s'=C}, \underbrace{0.8\big(1 + 0.5(0)\big)}_{s'=C} + \underbrace{0.2\big(-2 + 0.5(0)\big)}_{s'=B}\Big]$$

(L action: first two terms; R action: last two terms)

Example: Mini-Gridworld

- $V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$

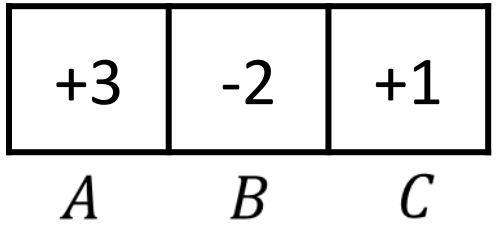


- Time-limited values V_1: $\begin{pmatrix} V_1(A) \\ V_1(B) \\ V_1(C) \end{pmatrix} = \begin{pmatrix} 2 \\ 2.6 \\ 0.4 \end{pmatrix}$

- From each state, we choose the *best action* that produces the *maximum expected value* due to *one transition*

Example: Mini-Gridworld

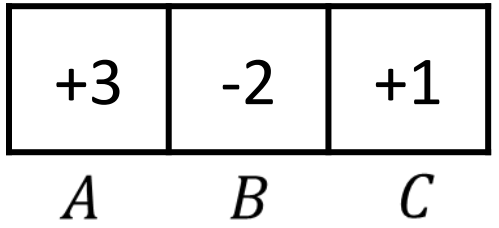
- $V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$



- Time-limited values V_1: $\begin{pmatrix} V_1(A) \\ V_1(B) \\ V_1(C) \end{pmatrix} = \begin{pmatrix} 2 \\ 2.6 \\ 0.4 \end{pmatrix}$
- From each state, we choose the *best action* that produces the *maximum expected value* due to *one transition*
- Try solving for V_2!

Example: Mini-Gridworld

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$



- Time-limited values V_1: $\begin{pmatrix} V_1(A) \\ V_1(B) \\ V_1(C) \end{pmatrix} = \begin{pmatrix} 2 \\ 2.6 \\ 0.4 \end{pmatrix}$
- From each state, we choose the *best action* that produces the *maximum expected value* due to *one transition*
- Try solving for V_2! $\begin{pmatrix} V_2(A) \\ V_2(B) \\ V_2(C) \end{pmatrix} = \begin{pmatrix} 3.06 \\ 3.44 \\ 0.82 \end{pmatrix}$

Value Iteration Implementation

- Value iteration can be implemented following the provided pseudocode
- The time-limited values V_i can be stored in a mutable array or list

Value Iteration Implementation

- Value iteration can be implemented following the provided pseudocode
- The time-limited values V_i can be stored in a mutable array or list
- When computing V_{i+1}, a temporary array should be created, since V_i should not be overwritten until all values are computed

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$

Value Iteration Implementation

- Value iteration can be implemented following the provided pseudocode
- The time-limited values V_i can be stored in a mutable array or list
- When computing V_{i+1}, a temporary array should be created, since V_i should not be overwritten until all values are computed

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_i(s')]$$

- Stop when values do not change by more than ϵ, a predefined threshold (e.g., a specified fraction of the values themselves)

Checkpoint: Value Iteration Implementation

- Some questions to think about:

- Does the *order* in which we loop over the states in each iteration of value iteration matter? Do different orderings lead to different values?

Checkpoint: Value Iteration Implementation

- Some questions to think about:
- Does the *order* in which we loop over the states in each iteration of value iteration matter? Do different orderings lead to different values?
- What happen if we set $\epsilon = \infty$?
- What happens if we set $\epsilon = 0$?

Convergence of Value Iteration

- Value iteration is *guaranteed* to converge and return V^*, or an approximation within the range of ϵ

Convergence of Value Iteration

- Value iteration is *guaranteed* to converge and return V^*, or an approximation within the range of ϵ

- The Bellman update is a **contraction mapping**
- Time-limited value are always guaranteed to move toward V^*

Convergence of Value Iteration

- Value iteration is *guaranteed* to converge and return V^*, or an approximation within the range of ϵ

- The Bellman update is a **contraction mapping**
- Time-limited value are always guaranteed to move toward V^*

- Fact 1: Bellman update does not change optimal values V^* (*fixed point*)

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Convergence of Value Iteration

- Define the *max norm* (error) between V_i and true values V^*:

$$\|V_i - V^*\| = \max_s |V_i(s) - V^*(s)|$$

Convergence of Value Iteration

- Define the *max norm* (error) between V_i and true values V^*:

$$\|V_i - V^*\| = \max_s |V_i(s) - V^*(s)|$$

- Fact 2: The max norm satisfies $\|V_{i+1} - V^*\| \leq \gamma \|V_i - V^*\|$

- Each update shrinks max "error" in V by factor of γ—exponentially fast!

Convergence of Value Iteration

- Define the *max norm* (error) between V_i and true values V^*:

$$\|V_i - V^*\| = \max_s |V_i(s) - V^*(s)|$$

- Fact 2: The max norm satisfies $\|V_{i+1} - V^*\| \leq \gamma \|V_i - V^*\|$

- Each update shrinks max "error" in V by factor of γ—exponentially fast!

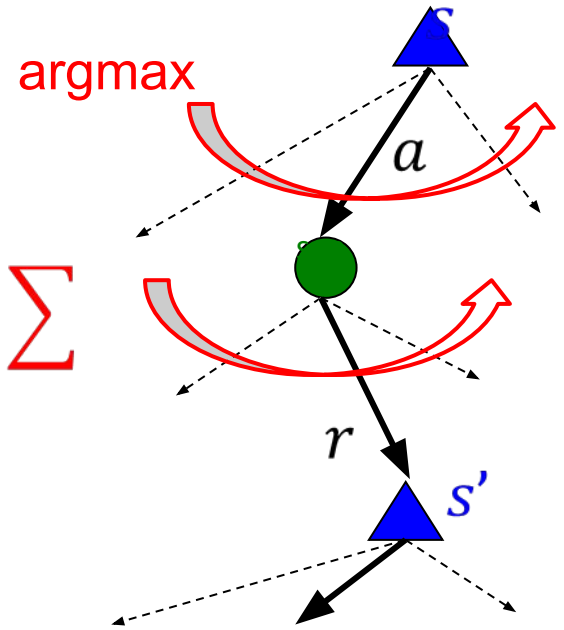
- For example, suppose largest true value, and max initial error, is $\frac{|r_{\max}|}{1-\gamma}$
- After k passes of value iteration, max error is bounded by $\gamma^k \frac{|r_{\max}|}{1-\gamma}$

Policy Extraction

- The goal of value iteration is to eventually extract an optimal policy

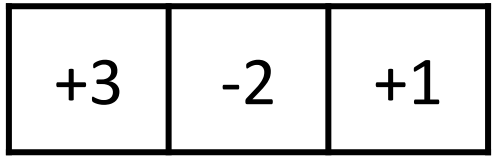
Policy Extraction

- The goal of value iteration is to eventually extract an optimal policy
- Bellman equation: $\pi^*(s) = \underset{a}{\text{argmax}} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
- Given V^*, we can solve for π^* in $O(|S|^2|A|)$ time



Example: Mini-Gridworld

- Given $V^*(A), V^*(B), V^*(C), \gamma = 0.5$
- Find $\pi^*(B)$:



Example: Mini-Gridworld

- Given $V^*(A), V^*(B), V^*(C), \gamma = 0.5$
- Find $\pi^*(B)$:

+3	-2	+1

- $V^{left}(B) = 0.8\big(3 + 0.5V^*(A)\big) + 0.2\big(1 + 0.5V^*(C)\big)$
- $V^{right}(B) = 0.8\big(1 + 0.5V^*(C)\big) + 0.2(3 + 0.5V^*(A))$

Example: Mini-Gridworld

- Given $V^*(A), V^*(B), V^*(C), \gamma = 0.5$
- Find $\pi^*(B)$:

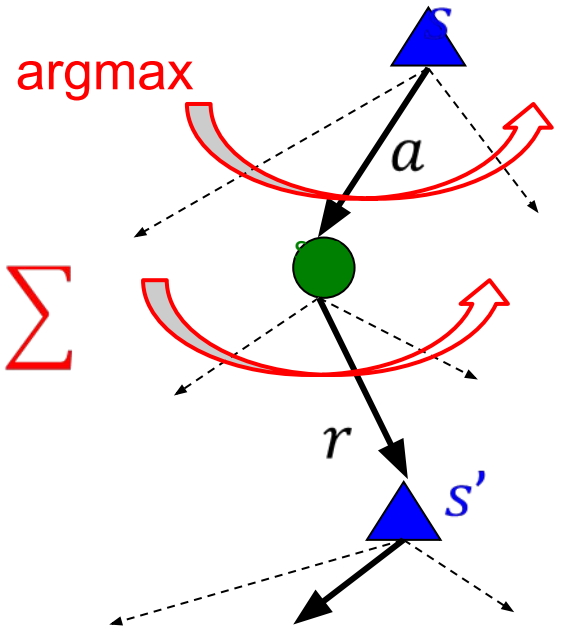
+3	-2	+1

- $V^{left}(B) = 0.8\big(3 + 0.5V^*(A)\big) + 0.2\big(1 + 0.5V^*(C)\big)$
- $V^{right}(B) = 0.8\big(1 + 0.5V^*(C)\big) + 0.2(3 + 0.5V^*(A))$

- $\pi^*(B) = \text{argmax}\big(V^{left}, V^{right}\big)$

Policy Extraction

- The goal of value iteration is to eventually extract an optimal policy
- Bellman equation: $\pi^*(s) = \underset{a}{\text{argmax}} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
- Given V^*, we can solve for π^* in $O(|S|^2|A|)$ time
- Observation: Only *relative*, not absolute, V^* required
- We may not need to wait for V^* to converge



Policy Iteration

- Idea: A policy can be computed *at any point* during value iteration

- We can improve on policy *directly*, leading to better values, leading to a better policy, and so on…

Policy Iteration

- Idea: A policy can be computed *at any point* during value iteration

- We can improve on policy *directly*, leading to better values, leading to a better policy, and so on…

- Initialize $\pi_0(s)$ arbitrarily for all states s
- **Loop** from $i = 0$:

Policy Iteration

- Idea: A policy can be computed *at any point* during value iteration

- We can improve on policy *directly*, leading to better values, leading to a better policy, and so on…

- Initialize $\pi_0(s)$ arbitrarily for all states s
- **Loop** from $i = 0$:
  - **Policy evaluation**: Compute V^{π_i} for policy π_i

Policy Iteration

- Idea: A policy can be computed *at any point* during value iteration

- We can improve on policy *directly*, leading to better values, leading to a better policy, and so on…

- Initialize $\pi_0(s)$ arbitrarily for all states s
- **Loop** from $i = 0$:
  - **Policy evaluation**: Compute V^{π_i} for policy π_i
  - **Policy improvement**: Given V^{π_i}, find new policy π_{i+1}

Policy Iteration

- Idea: A policy can be computed *at any point* during value iteration

- We can improve on policy *directly*, leading to better values, leading to a better policy, and so on…

- Initialize $\pi_0(s)$ arbitrarily for all states s
- **Loop** from $i = 0$:
  - **Policy evaluation**: Compute V^{π_i} for policy π_i
  - **Policy improvement**: Given V^{π_i}, find new policy π_{i+1}
- **Until** $\pi_{i+1} = \pi_i$

Iterative Policy Evaluation

- Given a policy π, the value function V^{π} can be found by solving a linear system
- In practice, an *iterative* approach is used to solve for V^{π}
- Just like value iteration, but with a fixed policy!

Iterative Policy Evaluation

- Given a policy π, the value function V^{π} can be found by solving a linear system
- In practice, an *iterative* approach is used to solve for V^{π}
- Just like value iteration, but with a fixed policy!

- Initialize V_0^{π}, e.g. all equal to 0 or values from a similar policy

Iterative Policy Evaluation

- Given a policy π, the value function V^π can be found by solving a linear system
- In practice, an *iterative* approach is used to solve for V^π
- Just like value iteration, but with a fixed policy!

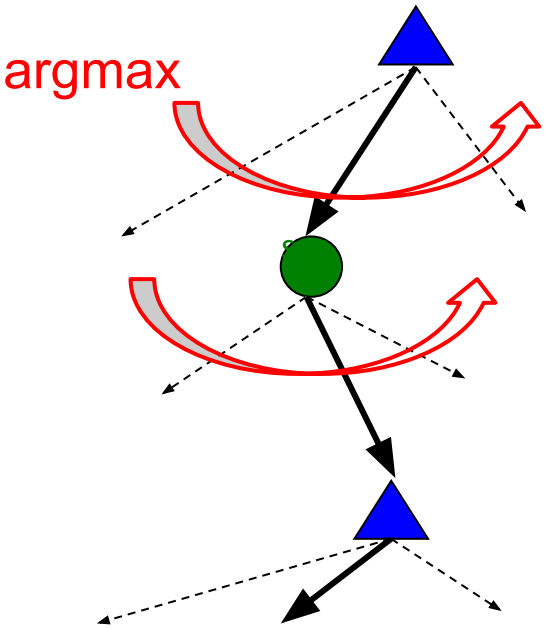
- Initialize V_0^π, e.g. all equal to 0 or values from a similar policy
- **Loop** from $i = 0$:
  - **For** each state $s \in S$:

$$V_{i+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^\pi(s')]$$

- **Until** $\max_s |V_{i+1}^\pi(s) - V_i^\pi(s)| < \epsilon$ (small threshold)

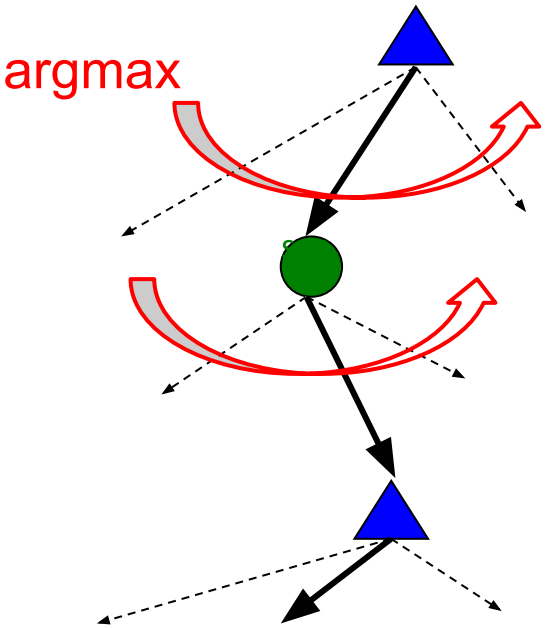
Policy Improvement

- Given values for a policy π_i, how can we improve it?
- Consider taking "greediest" action at each state:



Policy Improvement

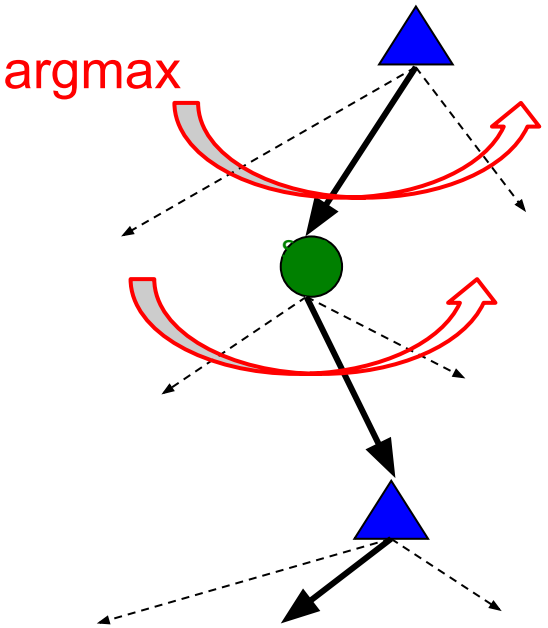
- Given values for a policy π_i, how can we improve it?
- Consider taking "greediest" action at each state:



- If π_i already optimal, then $V^{\pi_i} = V^*$ and $\pi_{i+1} = \pi_i$

Policy Improvement

- Given values for a policy π_i, how can we improve it?
- Consider taking "greediest" action at each state:

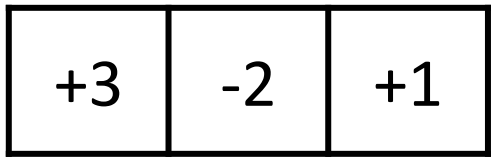


- If π_i already optimal, then $V^{\pi_i} = V^*$ and $\pi_{i+1} = \pi_i$

- Otherwise, V^{π_i} can be moved closer to V^* by changing some actions
- Full proof of convergence shown by *policy improvement theorem*

Example: Mini-Gridworld

- Suppose we initialize $\left(\pi_0(A), \pi_0(B), \pi_0(C)\right) = (R, R, R)$



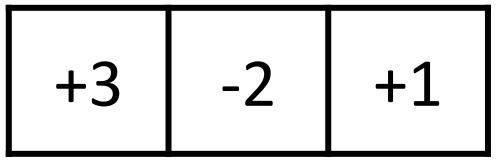
$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$V_{i+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^{\pi}(s')]$$

Example: Mini-Gridworld

- Suppose we initialize $\left(\pi_0(A), \pi_0(B), \pi_0(C)\right) = (R, R, R)$



$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$V_{i+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^{\pi}(s')]$$

$$V_{i+1}^{\pi_0}(A) \leftarrow 0.8\left(-2 + 0.5V_i^{\pi_0}(B)\right) + 0.2\left(3 + 0.5V_i^{\pi_0}(A)\right)$$

Example: Mini-Gridworld

- Suppose we initialize $\left(\pi_0(A), \pi_0(B), \pi_0(C)\right) = (R, R, R)$

+3	-2	+1

$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$V_{i+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^{\pi}(s')]$$

$$V_{i+1}^{\pi_0}(A) \leftarrow 0.8\left(-2 + 0.5V_i^{\pi_0}(B)\right) + 0.2\left(3 + 0.5V_i^{\pi_0}(A)\right)$$

$$V_{i+1}^{\pi_0}(B) \leftarrow 0.8\left(1 + 0.5V_i^{\pi_0}(C)\right) + 0.2\left(3 + 0.5V_i^{\pi_0}(A)\right)$$

$$V_{i+1}^{\pi_0}(C) \leftarrow 0.8\left(1 + 0.5V_i^{\pi_0}(C)\right) + 0.2\left(-2 + 0.5V_i^{\pi_0}(B)\right)$$

Example: Mini-Gridworld

- Suppose we initialize $\left(\pi_0(A), \pi_0(B), \pi_0(C)\right) = (R, R, R)$

+3	-2	+1

$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$V_{i+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^{\pi}(s')]$$

$$V_{i+1}^{\pi_0}(A) \leftarrow 0.8\left(-2 + 0.5V_i^{\pi_0}(B)\right) + 0.2\left(3 + 0.5V_i^{\pi_0}(A)\right)$$

$$V_{i+1}^{\pi_0}(B) \leftarrow 0.8\left(1 + 0.5V_i^{\pi_0}(C)\right) + 0.2\left(3 + 0.5V_i^{\pi_0}(A)\right)$$

$$V_{i+1}^{\pi_0}(C) \leftarrow 0.8\left(1 + 0.5V_i^{\pi_0}(C)\right) + 0.2\left(-2 + 0.5V_i^{\pi_0}(B)\right)$$

$$\begin{pmatrix} V^{\pi_0}(A) \\ V^{\pi_0}(B) \\ V^{\pi_0}(C) \end{pmatrix} = \begin{pmatrix} -.333 \\ 1.75 \\ .958 \end{pmatrix}$$

Example: Mini-Gridworld

- Suppose we initialize $\big(\pi_0(A), \pi_0(B), \pi_0(C)\big) = (R, R, R)$

+3	-2	+1

$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$\big(V_0(A), V_0(B), V_0(C)\big) = (-0.333, 1.75, 0.958)$$

- **Improve** policy:

$$\pi_1(A) = \text{argmax}\big[0.8\big(3 + 0.5V_0(A)\big) + 0.2\big(-2 + 0.5V_0(B)\big), -0.333\big]$$

Example: Mini-Gridworld

- Suppose we initialize $\big(\pi_0(A), \pi_0(B), \pi_0(C)\big) = (R, R, R)$

+3	-2	+1

$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$\big(V_0(A), V_0(B), V_0(C)\big) = (-0.333, 1.75, 0.958)$$

- **Improve** policy:

$$\pi_1(A) = \text{argmax}\big[0.8\big(3 + 0.5V_0(A)\big) + 0.2\big(-2 + 0.5V_0(B)\big), -0.333\big]$$

$$\pi_1(B) = \text{argmax}\big[\big(0.8\big(3 + 0.5V_0(A)\big) + 0.2\big(1 + 0.5V_0(C)\big), 1.75\big]$$

$$\pi_1(C) = \text{argmax}\big[\big(0.8\big(-2 + 0.5V_0(B)\big) + 0.2\big(1 + 0.5V_0(C)\big), 0.958\big]$$

Example: Mini-Gridworld

- Suppose we initialize $\big(\pi_0(A), \pi_0(B), \pi_0(C)\big) = (R, R, R)$

+3	-2	+1

$\gamma = 0.5$

- **Evaluate** policy using iterative policy evaluation:

$$\big(V_0(A), V_0(B), V_0(C)\big) = (-0.333, 1.75, 0.958)$$

- **Improve** policy: $\begin{pmatrix} \pi_1(A) \\ \pi_1(B) \\ \pi_1(C) \end{pmatrix} = \begin{pmatrix} L \\ L \\ R \end{pmatrix}$

- **Repeat** the previous steps for π_1

Algorithm Complexity

- Value iteration: Each sweep consists of one full round of iterative policy evaluation (sum) followed by policy improvement (max)

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')] \qquad O(|S|^2|A|) \text{ time}$$

Algorithm Complexity

- Value iteration: Each sweep consists of one full round of iterative policy evaluation (sum) followed by policy improvement (max)

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')] \qquad O(|S|^2|A|) \text{ time}$$

- Policy iteration: Each sweep consists of *many* rounds of iterative policy evaluation (sum) followed by policy improvement (argmax)

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s')[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')]$$

$$\pi_{i+1}(s) = \operatorname*{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_i}(s')]$$

Algorithm Complexity

- Value iteration: Each sweep consists of one full round of iterative policy evaluation (sum) followed by policy improvement (max)

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$

$O(|S|^2|A|)$ time

- Policy iteration: Each sweep consists of *many* rounds of iterative policy evaluation (sum) followed by policy improvement (argmax)

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s')[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')]$$

$$\pi_{i+1}(s) = \operatorname*{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_i}(s')]$$

$O(|S|^3 + |S|^2|A|)$ time

Algorithm Complexity

- Per sweep, policy iteration requires more computations than value iteration

Algorithm Complexity

- Per sweep, policy iteration requires more computations than value iteration

- Value iteration: Number of sweeps depends on γ and error threshold ϵ
- Increases dramatically for high γ

Algorithm Complexity

- Per sweep, policy iteration requires more computations than value iteration

- Value iteration: Number of sweeps depends on γ and error threshold ϵ
- Increases dramatically for high γ

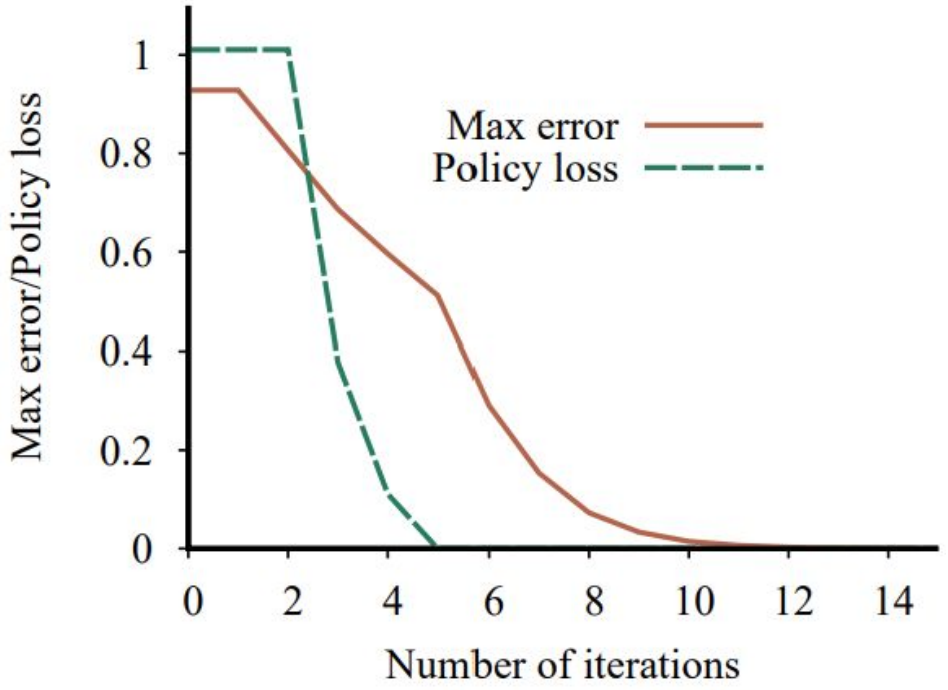
- Policy iteration: Iterative policy evaluation is typically very efficient
- Fewer sweeps needed overall to converge

Algorithm Complexity

- Per sweep, policy iteration requires more computations than value iteration

- Value iteration: Number of sweeps depends on γ and error threshold ϵ
- Increases dramatically for high γ

- Policy iteration: Iterative policy evaluation is typically very efficient
- Fewer sweeps needed overall to converge



Example comparison: Max error (VI) requires more iterations to converge than does policy loss (PI)

Asynchronous Dynamic Programming

- Recall that we make a temporary copy of the values in each sweep of value iteration
- We do the same in iterative policy evaluation (new values computed using old values)
- This is called **synchronous dynamic programming**

Asynchronous Dynamic Programming

- Recall that we make a temporary copy of the values in each sweep of value iteration
- We do the same in iterative policy evaluation (new values computed using old values)
- This is called **synchronous dynamic programming**

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s')[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')]$$

Asynchronous Dynamic Programming

- Recall that we make a temporary copy of the values in each sweep of value iteration
- We do the same in iterative policy evaluation (new values computed using old values)
- This is called **synchronous dynamic programming**

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i(s')]$$

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s')[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')]$$

- In practice, it is also correct to make all updates *in-place* in the same array
- Can lead to faster convergence since we use more accurate information
- This is called **asynchronous dynamic programming**

Generalized Policy Iteration

- **Generalized policy iteration**: Interleave policy evaluation and improvement arbitrarily

Generalized Policy Iteration

- **Generalized policy iteration**: Interleave policy evaluation and improvement arbitrarily

- E.g., update values of *some* rather than *all* states in each iteration
- E.g., run policy improvement *before* current policy values converge
- E.g., run *more or fewer* iterative policy evaluation sweeps on different states
- E.g., improve policy on *some* rather than *all* states

Generalized Policy Iteration

- **Generalized policy iteration**: Interleave policy evaluation and improvement arbitrarily

- E.g., update values of *some* rather than *all* states in each iteration
- E.g., run policy improvement *before* current policy values converge
- E.g., run *more or fewer* iterative policy evaluation sweeps on different states
- E.g., improve policy on *some* rather than *all* states

- Convergence in values in some or all states drives convergence in policy
- Convergence in policy in some or all states drives convergence in values
- Both processes stabilize at joint optimal solution

Summary

- Dynamic programming solves MDPs by exploiting recursive relationships among the state values in the Bellman equations
- Bellman updates push both values and policies toward optimal solution
- Value iteration: Compute and converge toward optimal values for all states, then extract policy
- Policy iteration: Alternative between evaluating and improving a current policy