CS 188: Artificial Intelligence

Markov Decision Processes

Instructor: Saagar Sanghavi – UC Berkeley

[Slides adapted from Dan Klein, Pieter Abbeel, Ketrina Yim, Stuart Russell, Satish Rao, and many others.]
## Deep Reinforcement Learning

<table>
<thead>
<tr>
<th>Year</th>
<th>Game</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>Atari (DQN)</td>
<td><img src="image" alt="Atari (DQN)" /></td>
</tr>
<tr>
<td></td>
<td>[Deepmind]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pong</td>
<td><img src="image" alt="Pong" /></td>
</tr>
<tr>
<td></td>
<td>Enduro</td>
<td><img src="image" alt="Enduro" /></td>
</tr>
<tr>
<td></td>
<td>Beamrider</td>
<td><img src="image" alt="Beamrider" /></td>
</tr>
<tr>
<td></td>
<td>Q*bert</td>
<td><img src="image" alt="Q*bert" /></td>
</tr>
</tbody>
</table>
Deep Reinforcement Learning

2013
Atari (DQN) [Deepmind]

2015
AlphaGo [Deepmind]

AlphaGo Silver et al, Nature 2015
AlphaGoZero Silver et al, Nature 2017
AlphaZero Silver et al, 2017
Tian et al, 2016; Maddison et al, 2014; Clark et al, 2015
Deep Reinforcement Learning

<table>
<thead>
<tr>
<th>Year</th>
<th>Project</th>
<th>Agency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>Atari (DQN)</td>
<td>Deepmind</td>
</tr>
<tr>
<td>2015</td>
<td>AlphaGo</td>
<td>Deepmind</td>
</tr>
<tr>
<td>2016</td>
<td>3D locomotion (TRPO+GAE)</td>
<td>Berkeley</td>
</tr>
</tbody>
</table>

[Schulman, Moritz, Levine, Jordan, Abbeel, ICLR 2016]
# Deep Reinforcement Learning

<table>
<thead>
<tr>
<th>Year</th>
<th>Project</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>Atari (DQN)</td>
<td>Deepmind</td>
</tr>
<tr>
<td>2015</td>
<td>AlphaGo</td>
<td>Deepmind</td>
</tr>
<tr>
<td>2016</td>
<td>3D locomotion (TRPO+GAE)</td>
<td>Berkeley</td>
</tr>
<tr>
<td>2016</td>
<td>Real Robot Manipulation (GPS)</td>
<td>Berkeley</td>
</tr>
</tbody>
</table>

[Levine*, Finn*, Darrell, Abbeel, JMLR 2016]
## Deep Reinforcement Learning

<table>
<thead>
<tr>
<th>Year</th>
<th>Project Description</th>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>Atari (DQN)</td>
<td>Deepmind</td>
</tr>
<tr>
<td>2015</td>
<td>AlphaGo</td>
<td>Deepmind</td>
</tr>
<tr>
<td>2016</td>
<td>3D locomotion (TRPO+GAE)</td>
<td>Berkeley</td>
</tr>
<tr>
<td>2016</td>
<td>Real Robot Manipulation (GPS)</td>
<td>Berkeley</td>
</tr>
<tr>
<td>2019</td>
<td>Rubik’s Cube (PPO+DR)</td>
<td>OpenAI</td>
</tr>
</tbody>
</table>
Non-Deterministic Search
Example: Grid World

- **A maze-like problem**
  - The agent lives in a grid
  - Walls block the agent’s path

- **Noisy movement: actions do not always go as planned**
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put

- **The agent receives rewards**
  - Small “living” reward each step (can be negative)
  - Big rewards come at the end (good or bad)

- **Goal**: maximize sum of rewards
Grid World Actions

Deterministic Grid World

Stochastic Grid World
Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s, a, s')$
    - Probability that $a$ from $s$ leads to $s'$, i.e., $P(s' | s, a)$
    - Also called the model or the dynamics
  - A reward function $R(s, a, s')$
    - Sometimes just $R(s)$ or $R(s')$
  - A start state
  - Maybe a terminal state
Video of Demo Gridworld Manual Intro
What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent.

- For Markov decision processes, “Markov” means action outcomes depend only on the current state.

  \[
  P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0) = P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
  \]

- This is just like search, where the successor function could only depend on the current state (not the history).

Andrey Markov (1856-1922)
In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal.

For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$.
- A policy $\pi$ gives an action for each state.
- An optimal policy is one that maximizes expected utility if followed.
- An explicit policy defines a reflex agent.

Optimal policy when $R(s, a, s') = -0.03$ for all non-terminals $s$. 
Optimal Policies

- $R(s) = -0.01$
- $R(s) = -0.03$
- $R(s) = -0.4$
- $R(s) = -2.0$
Example: Racing
Example: Racing

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward

```
<table>
<thead>
<tr>
<th>State</th>
<th>Slow</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cool</td>
<td>0.5</td>
<td>+1</td>
</tr>
<tr>
<td>Warm</td>
<td>0.5</td>
<td>+2</td>
</tr>
<tr>
<td>Overheated</td>
<td>0.5</td>
<td>-10</td>
</tr>
</tbody>
</table>
```
Racing Search Tree
MDP Search Trees

- Each MDP state projects an expectimax-like search tree

(s,a,s') called a transition

\( T(s,a,s') = P(s' \mid s,a) \)

\( R(s,a,s') \)
Utilities of Sequences
Utilities of Sequences

- What preferences should an agent have over reward sequences?

- More or less?  [1, 2, 2]  or  [2, 3, 4]

- Now or later?  [0, 0, 1]  or  [1, 0, 0]
Discounting

- It’s reasonable to maximize the sum of rewards
- It’s also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially
Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once

- Why discount?
  - Reward now is better than later
  - Can also think of it as a $1 - \gamma$ chance of ending the process at every step
  - Also helps our algorithms converge

- Example: discount of 0.5
  - $U([1,2,3]) = 1 \times 1 + 0.5 \times 2 + 0.25 \times 3$
  - $U([1,2,3]) < U([3,2,1])$
Stationary Preferences

- Theorem: if we assume stationary preferences:

\[
[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]
\]

\[\Downarrow\]

\[
[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]
\]

- Then: there is only ways to define utilities
  - Additive discounted utility:

\[
U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \ldots
\]
Quiz: Discounting

- Given:
  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

- Quiz 1: For $\gamma = 1$, what is the optimal policy? 
  
- Quiz 2: For $\gamma = 0.1$, what is the optimal policy? 
  
- Quiz 3: For which $\gamma$ are West and East equally good when in state d?
  
$1_{\gamma} = 10 \gamma^3$
Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?

- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies ($\pi$ depends on time left)
  
  - Discounting: use $0 < \gamma < 1$
    $$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1 - \gamma)$$
    - Smaller $\gamma$ means smaller "horizon" – shorter term focus
  
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)
Recap: Defining MDPs

- Markov decision processes:
  - Set of states $S$
  - Start state $s_0$
  - Set of actions $A$
  - Transitions $P(s' \mid s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)

- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards
Solving MDPs
Recall: Racing MDP

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward
Racing Search Tree
Racing Search Tree
We’re doing way too much work with expectimax!

Problem: States are repeated
- Idea: Only compute needed quantities once

Problem: Tree goes on forever
- Idea: Do a depth-limited computation, but with increasing depths until change is small
- Note: deep parts of the tree eventually don’t matter if $\gamma < 1$
Optimal Quantities

- The value (utility) of a state $s$:
  \[ V^*(s) \text{ = expected utility starting in } s \text{ and acting optimally} \]

- The value (utility) of a q-state $(s,a)$:
  \[ Q^*(s,a) \text{ = expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \]

- The optimal policy:
  \[ \pi^*(s) \text{ = optimal action from state } s \]
Gridworld V* Values

Noise = 0.2
Discount = 0.9
Living reward = 0
Gridworld Q* Values

Noise = 0.2
Discount = 0.9
Living reward = 0
Values of States: Bellman Equation

- Recursive definition of value:

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

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

\[
V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]
\]
Key idea: time-limited values

Define $V_k(s)$ to be the optimal value of $s$ if the game ends in $k$ more time steps

Equivalently, it’s what a depth-$k$ expectimax would give from $s$
$k=0$

VALUES AFTER 0 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=1

VALUES AFTER 1 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=2$

VALUES AFTER 2 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=3

VALUES AFTER 3 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
\[ k=4 \]

VALUES AFTER 4 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=5$

VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
\( k = 6 \)

VALUES AFTER 6 ITERATIONS

Noise = 0.2  
Discount = 0.9  
Living reward = 0
$k=7$

VALUES AFTER 7 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=8

VALUES AFTER 8 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
\( k = 9 \)

VALUES AFTER 9 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=10

VALUES AFTER 10 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=11$

VALUES AFTER 11 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=12$

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=100$

VALUES AFTER 100 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
Computing Time-Limited Values

$V_4(\mathcal{V}) \quad V_4(\mathcal{V}) \quad V_4(\mathcal{V})$

$V_3(\mathcal{V}) \quad V_3(\mathcal{V}) \quad V_3(\mathcal{V})$

$V_2(\mathcal{V}) \quad V_2(\mathcal{V}) \quad V_2(\mathcal{V})$

$V_1(\mathcal{V}) \quad V_1(\mathcal{V}) \quad V_1(\mathcal{V})$

$V_0(\mathcal{V}) \quad V_0(\mathcal{V}) \quad V_0(\mathcal{V})$
Solving MDPs
The Bellman Equations

How to be optimal:
Step 1: Take correct first action
Step 2: Keep being optimal
The value (utility) of a state $s$:

$V^*(s) = \text{expected utility starting in } s \text{ and acting optimally}$

The value (utility) of a q-state $(s,a)$:

$Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally}$

The optimal policy:

$\pi^*(s) = \text{optimal action from state } s$
Value Iteration
Value Iteration

- Bellman equations **characterize** the optimal values:
  \[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

- Value iteration **computes** them:
  \[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
  
  "Bellman Update"

- Value iteration is just a fixed point solution method
  - ... though the \( V_k \) vectors are also interpretable as time-limited values
Similarly, can have Q-Value Iteration

- **Bellman Equation:** recursive **definition** of Q-values
  \[ Q^*(s, a) = \sum_{s', T(s, a, s')} [R(s, a, s') + \gamma \max_{a'} Q^*(s', a')] \]

- **Q-Value Iteration:** Dynamic Programming
  \[ Q_{k+1}(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')] \]
Value Iteration: Dynamic Programming

- Start with \( V_0(s) = 0 \): no time steps left means an expected reward sum of zero

- Given vector of \( V_k(s) \) values, do one ply of expectimax from each state:
  
  \[
  V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]
  \]

- \( V = B(V) \) Where B is the Bellman update operator

- Repeat until convergence, which yields \( V^* \)

- Complexity of each iteration: \( O(S^2A) \)

- Theorem: Value Iteration will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do
Example: Value Iteration

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
Example: Value Iteration

$V_0$

$V_1$

$V_2$

$V_k+1(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]$
Example: Value Iteration

<table>
<thead>
<tr>
<th></th>
<th>V₀</th>
<th>V₁</th>
<th>V₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Assume no discount!

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
Example: Value Iteration

\[ V_2 \]
- S: 1+2=3
- F: \(0.5(2+2)+0.5(2+1)=3.5\)

\[ V_1 \]
- 2 1 0

\[ V_0 \]
- 0 0 0

Assume no discount!

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
**Example: Value Iteration**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_1$</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$V_2$</td>
<td>3.5</td>
<td>2.5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Assume no discount!**

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]$$
How do we know the $V_k$ vectors are going to converge? (assuming $0 < \gamma < 1$)

Proof Sketch:
- For any state $V_k$ and $V_{k+1}$ can be viewed as depth $k+1$ expectimax results in nearly identical search trees
- The difference is that on the bottom layer, $V_{k+1}$ has actual rewards while $V_k$ has zeros
- That last layer is at best all $R_{\text{MAX}}$
- It is at worst $R_{\text{MIN}}$
- But everything is discounted by $\gamma^k$ that far out
- So $V_k$ and $V_{k+1}$ are at most $\gamma^k \max |R|$ different
- So as $k$ increases, the values converge
Convergence of Value Iteration: Contraction

- New concept: contraction
  - If some operator $F$ is a contraction by a factor, it brings any pair of objects closer to each other (according to some metric $d(, )$)
    - For any $x, y$, we have $d(F(x), F(y)) < cd(x, y)$ where $c < 1$
    - If $F$ is a contraction it has a unique fixed point $z$ (i.e., $F(z)=z$)
- Since Value iteration is just $V_{k+1} = B(V_k)$, the Bellman update $B$ is a contraction by $\gamma$
- Metric is the max norm: $||V - W|| = \max_s |V(s) - W(s)|$
- What’s the fixed point for $B$?
  - $BV^* = V^*$
Speed of Convergence

- Look at what happens to the distance between $V_k$ and $V^*$
- $||V_{k+1} - V^*||$
- $= ||BV_k - V^*||$ (definition of $V_{k+1}$ from VI update)
- $= ||BV_k - BV^*||$ ($V^*$ is the fixed point of $B$)
- $\leq \gamma ||V_k - V^*||$ ($B$ is a contraction by $\gamma$)
- I.e., the error is reduced by at least a factor $\gamma$ on every iteration
- Exponentially fast convergence!
Correctness of Convergence

- Don’t usually converge exactly; stops when change < $\frac{\epsilon(1-\gamma)}{\gamma}$

- I.e. $||V_{k+1} - V_k|| < \frac{\epsilon(1-\gamma)}{\gamma}$

- What about $||V_{k+1} - V^*||$ when $||V_{k+1} - V_k|| < \frac{\epsilon(1-\gamma)}{\gamma}$

- Useful properties:
  - Contraction: $||V_{k+1} - V^*|| \leq \gamma ||V_k - V^*||$
  - Triangle inequality: $||V_k - V^*|| \leq ||V_{k+1} - V_k|| + ||V_{k+1} - V^*||$
Correctness of Convergence

- Value Iteration: stop when $||V_{k+1} - V_k|| < \frac{\epsilon (1-\gamma)}{\gamma}$
- What about $||V_{k+1} - V^*||$ when $||V_{k+1} - V_k|| < \frac{\epsilon (1-\gamma)}{\gamma}$?
- Triangle inequality: $||V_k - V^*|| \leq ||V_{k+1} - V_k|| + ||V_{k+1} - V^*||$
- $\frac{1}{\gamma} ||V_{k+1} - V^*|| \leq ||V_{k+1} - V_k|| + ||V_{k+1} - V^*||$ B(V) is contraction by $\gamma$
- $\left(\frac{1}{\gamma} - 1\right) ||V_{k+1} - V^*|| \leq ||V_{k+1} - V_k||$
- $\left(\frac{1}{\gamma} - 1\right) ||V_{k+1} - V^*|| \leq \frac{\epsilon (1-\gamma)}{\gamma}$ when we have converged
- $||V_{k+1} - V^*|| \leq \epsilon$
Policy Extraction
Computing Actions from Values

- Let’s imagine we have the optimal values $V^*(s)$
- How should we act?
  - It’s not obvious!
- We need to do a mini-expectimax (one step)

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

- This is called policy extraction, since it gets the policy implied by the values
Computing Actions from Q-Values

- Let's imagine we have the optimal q-values:

- How should we act?
  - Completely trivial to decide!

\[ \pi^*(s) = \arg \max_a Q^*(s, a) \]

- Important lesson: actions are easier to select from q-values than values!
Problems with Value Iteration

- Value iteration repeats the Bellman updates:

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Problem 1: It’s slow – \( O(S^2A) \) per iteration

- Problem 2: The “max” at each state rarely changes

- Problem 3: The policy often converges long before the values
k=12

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=100$

VALUES AFTER 100 ITERATIONS

0.64 0.74 0.85 1.00
0.57 0.57 -1.00
0.49 0.43 0.48 0.28

Noise = 0.2
Discount = 0.9
Living reward = 0
Policy Iteration

- Alternative approach for optimal values:
  - **Step 1: Policy Evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - **Step 2: Policy Improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is **Policy Iteration**
  - It’s still optimal!
  - Can converge (much) faster under some conditions
Policy Evaluation
Fixed Policies

- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state
  - ... though the tree's value would depend on which policy we fixed
Another basic operation: compute the utility of a state $s$ under a fixed (generally non-optimal) policy

Define the utility of a state $s$, under a fixed policy $\pi$:

$$V^\pi(s) = \text{expected total discounted rewards starting in } s \text{ and following } \pi$$

Recursive relation (one-step look-ahead / Bellman equation):

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$
Policy Evaluation

- How do we calculate the V's for a fixed policy $\pi$?

  - Idea 1: Turn recursive Bellman equations into updates (like value iteration)
    
    \[
    V_0^\pi(s) = 0 \\
    V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')] 
    \]

  - Efficiency: $O(S^2)$ per iteration

  - Idea 2: Without the maxes, the Bellman equations are just a linear system
    - Solve the system of equations

- $s$, $\pi(s)$
- $s'$, $\pi(s), s'$
- $s$, $\pi(s)$
Example: Policy Evaluation

Always Go Right

Always Go Forward
Example: Policy Evaluation

Always Go Right

Always Go Forward
Policy Iteration
Policy Iteration

- **Evaluation:** For fixed current policy $\pi$, find values with policy evaluation:
  - Iterate until values converge:
    $$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$$

- **Improvement:** For fixed values, get a better policy using policy extraction
  - One-step look-ahead:
    $$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_i}(s') \right]$$
Both value iteration and policy iteration compute the same thing (all optimal values)

In value iteration:
- Every iteration updates both the values and (implicitly) the policy
- We don’t track the policy, but taking the max over actions implicitly recomputes it

In policy iteration:
- We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
- After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
- The new policy will be better (or we’re done)

Both are dynamic programming approaches for solving MDPs
Summary: MDP Algorithms

- So you want to….
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions
Double Bandits
Double-Bandit MDP

- Actions: Blue, Red
- States: Win, Lose

No discount
100 time steps
Both states have the same value
Offline Planning

- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play Red</td>
<td>150</td>
</tr>
<tr>
<td>Play Blue</td>
<td>100</td>
</tr>
</tbody>
</table>

No discount
100 time steps
Both states have the same value
Let’s Play!

$2$ $2$ $0$ $2$ $2$

$2$ $2$ $0$ $0$ $0$
Online Planning

- Rules changed! Red’s win chance is different.
Let’s Play!

$0  $0  $0  $2  $0

$2  $0  $0  $0  $0
What Just Happened?

- That wasn’t planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn’t solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP