Recap: MDPs and planning under uncertainty
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
- 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 each time step

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

Goal: Maximum sum of (discounted) rewards
Markov decision processes:

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

Set of *states* $S$

*Start state* $s_0 \in S$

Set of *actions* $A$

*Transitions* $P(s' \mid s, a)$ or $T(s, a, s')$

*Rewards* $R(s, a, s')$ and discount $\gamma$

Quantities:

*Policy*: Map of states to actions

*Utility*: Sum of discounted rewards

*Values*: Expected future utility from a state (max node)

*Q-Values*: Expected future utility from a q-state (chance node)
Markov decision processes:

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

Quantities:

- **Policy**: Map of states to actions
- **Utility**: Sum of discounted rewards
- **Values**: Expected future utility from a state (max node)
- **$Q$-Values**: Expected future utility from a q-state (chance node)
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:

$$\pi^*(s) = \text{optimal action from state } s$$
Grid World: $V^*$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.64</td>
<td>0.74</td>
<td>0.85</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.57</td>
<td></td>
<td>0.57</td>
<td>-1.00</td>
<td></td>
</tr>
<tr>
<td>0.49</td>
<td>0.43</td>
<td>0.48</td>
<td>0.28</td>
<td></td>
</tr>
</tbody>
</table>

VALUES AFTER 100 ITERATIONS
Grid World: $Q^*$

$Q$-VALUES AFTER 100 ITERATIONS
The Bellman equations
How to be optimal:

*Step 1*: Take the correct first action.
How to be optimal:

*Step 1*: Take the correct first action.

*Step 2*: Keep being optimal.
How to be optimal:

*Step 1*: Take the correct first action.

And do the next right thing.

*Step 2*: Keep being optimal.
The definition of “optimal utility” via the expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values.
The definition of “optimal utility” via the expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values.

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

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

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

These are the Bellman equations, which characterize optimal values in a way we’ll use over and over.
Time-limited values
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*: The tree goes on forever.

*Idea*: Do a depth-limited computation, but with increasing depths until change is small. Deep parts of the tree eventually don’t matter if $\gamma < 1$. 
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 $s$. 
**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 $s$. 
**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 $s$. 
\( k = 0 \)

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

Noise = 0.2
Discount = 0.9
Living reward = 0

VALUES AFTER 1 ITERATIONS
\( k = 2 \)

**Gridworld Display**

VALUES AFTER 2 ITERATIONS

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

Noise = 0.2  
Discount = 0.9  
Living reward = 0
$k = 4$

<table>
<thead>
<tr>
<th></th>
<th>0.37</th>
<th>0.66</th>
<th>0.83</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td>0.00</td>
<td></td>
<td>-1.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.31</td>
<td></td>
<td>0.00</td>
</tr>
</tbody>
</table>

VALUES AFTER 4 ITERATIONS

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

Noise = 0.2  
Discount = 0.9  
Living reward = 0
$k = 6$

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

No noise = 0.2
Discount = 0.9
Living reward = 0
$k = 8$

Noise = 0.2
Discount = 0.9
Living reward = 0
$k = 9$

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

Noise = 0.2
Discount = 0.9
Living reward = 0

VALUES AFTER 10 ITERATIONS
$k = 11$

VALUES AFTER 11 ITERATIONS

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

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

Noise = 0.2
Discount = 0.9
Living reward = 0
Computing time-limited values
Computing time-limited values
Computing time-limited values
Computing time-limited values

\[ V_0(\cdot) \quad V_0(\cdot) \quad V_0(\cdot) \]
Computing time-limited values
Computing time-limited values
Computing time-limited values
Computing time-limited values
Value iteration
Bellman equations characterize the optimal values:

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

Value iteration computes them.
Value iteration

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 $s$:

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

Repeat until convergence.
Bellman equations \textit{characterize} the optimal values:

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

Value iteration \textit{computes} them.

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

Value iteration is just a fixed-point solution method.

...though the $V_k$ vectors are also interpretable as time-limited values
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

Assume no discount; \( \gamma = 1 \).

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

Assume no discount; $\gamma = 1$.

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

Assume no discount; $\gamma = 1$.

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

Assume no discount; \( \gamma = 1 \).

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

Assume no discount; $\gamma = 1$.

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

Assume no discount; $\gamma = 1$.

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

Assume no discount; \( \gamma = 1 \).

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

Assume no discount; \( \gamma = 1 \).

\[
V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]
\]
Policy extraction
Imagine we have the optimal values, $V^*(s)$:

How should we act?
Imagine we have the optimal values, $V^*(s)$:

How should we act?

It's not obvious from the values themselves! So we need to do a mini-expectimax (one step):

$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]$$
Imagine we have the optimal values, $V^*(s)$:

How should we act?

It’s not obvious from the values themselves! So we need to do a mini-expectimax (one step):

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

This is called \textit{policy extraction}, since it gets the policy implied by the values.
Imagine we have the optimal q-values:

How should we act?
Imagine we have the optimal q-values:

How should we act?

Easy to decide!

\[ \pi^*(s) = \arg \max_a Q^*(s, a) \]
Imagine we have the optimal q-values:

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

How should we act?

Easy to decide!

*Important lesson*: Actions are easier to select from q-values than values!
Policy methods
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.
\[ k = 12 \]

Noise = 0.2
Discount = 0.9
Living reward = 0
\[ k = 100 \]

Noise = 0.2
Discount = 0.9
Living reward = 0

VALUES AFTER 100 ITERATIONS
Problems with value iteration

Value iteration repeats the Bellman updates:

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

**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.
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.

Do the optimal action
Fixed policies

If we fixed some policy \( \pi(s) \), then the tree would be simpler – only one action per state.

Expectimax trees max over all actions to compute the optimal values.

Do the optimal action

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.

Do what \( \pi \) says to do
Utilities for a fixed policy

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 $\pi$:

$V_\pi(s) = \text{expected total discounted rewards, starting in } s \text{ and following policy } \pi$
Utilities for a fixed policy

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 \( \pi \):

\[ V^\pi(s) = \text{expected total discounted rewards, starting in } s \text{ and following policy } \pi \]

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

\[
V^\pi(s) = \sum_{s'} T(s, \pi(s), s') \left[ R(s, \pi(s), s') + \gamma V^\pi(s') \right]
\]
Example: Policy evaluation
Example: Policy evaluation

Always go right
Example: Policy evaluation

Always go right

Always go forward
Example: Policy evaluation

Always go right

Always go forward
Policy evaluation

How do we calculate the utility values ($V_s$) for a fixed policy $\pi$?
Policy evaluation

How do we calculate the utility values ($V_s$) for a fixed policy $\pi$?

Idea 1: Turn recursive Bellman equations into updates (like value iteration):

$$V^\pi_0(s) = 0$$

$$V^\pi_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s') \left[ R(s, \pi(s), s') + \gamma V^\pi_k(s') \right]$$
Policy evaluation

How do we calculate the utility values ($V_s$) for a fixed policy $\pi$?

**Idea 1:** Turn recursive Bellman equations into updates (like value iteration):

$$V_0^\pi(s) = 0$$

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

**Efficiency:** $O(S^2)$ per iteration
Policy evaluation

How do we calculate the utility values ($V_s$) for a fixed policy $\pi$?

**Idea 1:** Turn recursive Bellman equations into updates (like value iteration):

\[
V^\pi_0(s) = 0
\]

\[
V^\pi_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s') \left[ R(s, \pi(s), s') + \gamma V^\pi_k(s') \right]
\]

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

**Idea 2:** Without the maxes, the Bellman equations are just a linear system of equations.

Solve with Matlab or your favorite linear system solver.
Policy iteration
Policy iteration

*Evaluation*: For fixed current policy $\pi$, find values with policy evaluation:

Iterate until values converge:
Policy iteration

**Evaluation**: For fixed current policy \( \pi \), find values with policy evaluation:

Iterate until values converge:

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

**Evaluation**: For fixed current policy \( \pi \), find values with policy evaluation:

Iterate until values converge:

\[
V^\pi_k(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V^\pi_k(s') \right]
\]

**Improvement**: For fixed values, get a better policy using policy extraction

One-step look-ahead:
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:

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_i}(s') \right]$$
Comparison

Both value iteration and policy iteration compute the same thing (all optimal values).
Comparison

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.
Comparison

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).
Comparison

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 programs 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:
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’re 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*. 
Next time: Reinforcement learning!
Acknowledgments

The lecture incorporates material from:

Pieter Abbeel, Dan Klein, et al., University of California, Berkeley;
ai.berkeley.edu
Ketrina Yim (illustrations)