Solving MDPs

- Value Iteration
- Policy Iteration
- Reinforcement Learning
Policy Evaluation
Fixed Policies

Do the optimal action

Do what $\pi$ says to do

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):

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

Always Go Right

Always Go Forward
Example: Policy Evaluation

### Always Go Right

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-10.00</td>
<td><strong>100.00</strong></td>
<td>-10.00</td>
<td></td>
</tr>
<tr>
<td>-10.00</td>
<td>1.09</td>
<td>-10.00</td>
<td></td>
</tr>
<tr>
<td>-10.00</td>
<td>-7.88</td>
<td>-10.00</td>
<td></td>
</tr>
<tr>
<td>-10.00</td>
<td>-8.69</td>
<td>-10.00</td>
<td></td>
</tr>
</tbody>
</table>

### Always Go Forward

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-10.00</td>
<td>100.00</td>
<td>-10.00</td>
<td></td>
</tr>
<tr>
<td>-10.00</td>
<td>70.20</td>
<td>-10.00</td>
<td></td>
</tr>
<tr>
<td>-10.00</td>
<td>48.74</td>
<td>-10.00</td>
<td></td>
</tr>
<tr>
<td>-10.00</td>
<td>33.30</td>
<td>-10.00</td>
<td></td>
</tr>
</tbody>
</table>
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 with Matlab (or your favorite linear system solver)
Policy Iteration

Alternative approach for optimal values:

- **Step 1: Policy evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence

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

- **Step 2: Policy improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values

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

- Repeat steps until policy converges

This is policy iteration

- It’s still optimal! Can converge (much) faster under some conditions
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

To 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
Manipulator Control

Arm with two joints (workspace)

Configuration space
Manipulator Control Path

Arm with two joints (workspace)

Configuration space
Manipulator Control Path

Arm with two joints (workspace)

Configuration space
Double Bandits
Double-Bandit MDP

Actions: Blue, Red
States: Win, Lose

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

$2  $2  $0  $2  $2
$2  $2  $0  $0  $0
Rules changed! Red’s win chance is different.
Let’s Play!

![Slot Machine 1](image1.png)

![Slot Machine 2](image2.png)
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
Next Time: Reinforcement Learning!