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
  - Values = expected future utility from a state (max node)
  - Q-Values = expected future utility from a q-state (chance node)
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
Solving MDPs
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:
  \[ \pi^*(s) = \text{optimal action from state } s \]
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:

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

Repeat until convergence, which yields $V^*$.

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

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

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

- Value iteration is just a **fixed point solution method**
  - ... though the \( V_k \) vectors are also interpretable as time-limited values
Value Iteration (again 😊)

- **Init:**
  \[ \forall s: \ V(s) = 0 \]

- **Iterate:**

  \[ \forall s: \ V_{new}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V(s')] \]

  \[ V = V_{new} \]

Note: can even directly assign to \( V(s) \), which will not compute the sequence of \( V_k \) but will still converge to \( V^* \)
The Bellman Equations

How to be optimal:

Step 1: Take correct first action

Step 2: Keep being optimal
\[ k=0 \]

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
$k=6$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.59</td>
<td>0.73</td>
<td>0.85</td>
<td>1.00</td>
</tr>
<tr>
<td>0.41</td>
<td></td>
<td>0.57</td>
<td>-1.00</td>
</tr>
<tr>
<td>0.21</td>
<td>0.31</td>
<td>0.43</td>
<td>0.19</td>
</tr>
</tbody>
</table>

VALUES AFTER 6 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
\[ k = 7 \]

VALUES AFTER 7 ITERATIONS

\[
\begin{array}{cccc}
0.62 & 0.74 & 0.85 & 1.00 \\
0.50 & \text{gray} & 0.57 & -1.00 \\
0.34 & 0.36 & 0.45 & 0.24 \\
\end{array}
\]

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
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)

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

- 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!
Let’s think.

- Take a minute, think about value iteration.
- Write down the biggest question you have about it.
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

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
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

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):

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

How to be optimal:

Step 1: Take correct first action

Step 2: Keep being optimal
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  $0
$2  $2  $0  $0  $0
Rules changed! Red’s win chance is different.
Let’s Play!

$1$ Winner

$0 \ 0 \ 0 \ 2 \ 0$

$2 \ 0 \ 0 \ 0 \ 0 \ 0$
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
Next Time: Reinforcement Learning!