Dynamic Programming

Model-Based Planning — Lecture 2/4

Etienne Chassaing

CentraleSupélec

Plan

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration
4. Asynchronous Dynamic Programming
5. Convergence and Contraction

Where we are

Lecture 1 — MDPs & Bellman

- Formalized MDP as (S, A, p, r, γ)
- Bellman equations: $v_\pi(s)$, $q_\pi(s, a)$
- Optimal equations: $v_*(s)$, $q_*(s, a)$

Lecture 2 — Dynamic Programming and Model-Based Planning

- We know the model $p(s'|s, a)$
- We know $r(s, a)$
- **Goal:** Compute v_* or π_* exactly

Today's key insight: Bellman equations are systems of linear equations, instead of solving them directly, we can solve them iteratively by repeatedly applying the Bellman operator until convergence.
Why this matters: Later on, we'll be sutdying model-free control, and re-use this concept of iterative solution computation.

Plan

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration
4. Asynchronous Dynamic Programming
5. Convergence and Contraction

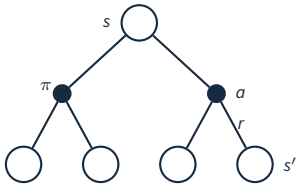
Policy Evaluation - Recall

- A **policy** π says to you how to act : It is a mapping from states to actions .
- The **value function** $v^{\pi}(s)$ says to you how good being a state is: It gives the expected return when starting in state s and following policy π thereafter.

⚠ : The value function **depends on the policy** π.

- **Bellman equation** is a way to link the value of a state to the values of its successor states. It is a recursive equation :

$$v_{\pi}(s) = \sum_{a} \pi(a \mid s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi}(s')]$$



Backup diagram for v^{π}

Policy Evaluation - Principle

Goal: Given a policy π, compute $v_\pi(s)$ for all states s.
For each state $s \in S$, we have one Bellman equation. With known dynamics, **this is** $|S|$ **equations in** $|S|$ **unknowns:** the value functions $v^\pi(s_1), v^\pi(s_2), \ldots, v^\pi(s_{|S|})$.

In principle: We could solve this system directly (e.g., Gaussian elimination), but for large state spaces this is **computationally expensive**.

Better approach: Use **iterative methods**. Repeatedly apply the Bellman expectation operator until convergence.

$$v_\pi(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

Key idea: Repeatedly apply the Bellman expectation operator. Each iteration refines the value estimates, and the sequence converges to v^π.

Policy Evaluation — Pseudocode

Policy Evaluation

Input: Policy π, threshold $\epsilon > 0$
  Initialize $v(s) = 0$ for all $s \in S$
  repeat
    $\Delta \leftarrow 0$
    for each $s \in S$:
      $v_{\text{old}}(s) \leftarrow v(s)$
      $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
      $\Delta \leftarrow \max(\Delta, |v(s) - v_{\text{old}}(s)|)$
  until $\Delta < \epsilon$
Output: Value function v^π

In-place updates of $v(s)$ use the most recent values, creating a bootstrapping effect that accelerates convergence.

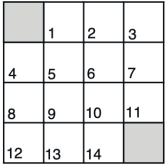
Policy Evaluation — Gridworld Example

Setup: A simple gridworld with 14 non-terminal states $S = \{1, 2, \ldots, 14\}$ arranged in a grid.

- **Actions:** $A = \{\text{up, down, right, left}\}$, deterministic transitions, except at boundaries (action unchanged) $p(5, -1 \mid 1, \text{down}) = 1$, $p(10, -1 \mid 5, \text{down}) = 0$, $p(5, 5 \mid 1, \text{down}) = 0$
- **Reward:** $r(s, a, s') = -1$ for all transitions (cost per step)
- **Task:** Undiscounted, episodic (one terminal state)
- **Policy:** Equiprobable random: $\pi(a|s) = 0.25$ for all a, s

Result: Iterative policy evaluation converges to $v^\pi(s)$ = *(negative of expected steps to termination)*

Convergence: Each iteration refines estimates using the Bellman equation. Values propagate inward from boundaries.



$R_t = -1$
on all transitions

Policy Evaluation - Value Function Evolution

Equiprobable random policy, $\gamma = 1$**,** $r = -1$ **per step.** Values converge to v^π as $k \to \infty$.

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 0$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 1$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 2$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 3$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = 10$

0.0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0.0

v^π **(**$k = \infty$**)**

$s = 1$ (orange state)

$$v_1(1) = \tfrac{1}{4}\left[1\cdot(-1+v_0(1)) + 1\cdot(-1+v_0(5)) + 1\cdot(-1+v_0(0)) + 1\cdot(-1+v_0(2))\right] = -1$$

$s = 6$ (cyan state)

$$v_2(6) = \tfrac{1}{4}\left[1\cdot(-1+v_1(2)) + 1\cdot(-1+v_1(10)) + 1\cdot(-1+v_1(5)) + 1\cdot(-1+v_1(7))\right] = -2.0$$

Question : At which k do the realtionships between surrounding blocks stop evolving?

Plan

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration
4. Asynchronous Dynamic Programming
5. Convergence and Contraction

From Evaluation to Improvement

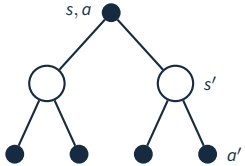
- **Key insight:** Once we know $v^\pi(s)$ for all states, we can measure the impact of taking a **different action** than the policy recommends.
- **Recall:** the action-value function under policy π is

$$\begin{aligned} q^\pi(s,a) &= \mathbb{E}[R_t \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma v^\pi(s')] \end{aligned}$$

  the **expected return** if we take action a in state s, then follow π forever after.

- **Relationship:**

$$v^\pi(s) = \sum_a \pi(a|s) q^\pi(s,a)$$



Backup diagram for q^π

Improvement question

If $\exists a \neq \pi(s)$ such that $q^\pi(s,a) > v^\pi(s)$, taking a is **better than following** π at state s.

This motivates the policy improvement step: greedify with respect to q^π.

Policy Improvement Theorem

Theorem: Policy Improvement Theorem

Let π and π' be two policies. If

$$q^{\pi}(s, \pi'(s)) \geq v^{\pi}(s) \quad \forall s \in S$$

then π' is **at least as good as** π: $v^{\pi'}(s) \geq v^{\pi}(s)$ for all s.

Greedy policy improvement: We apply this by computing the greedy policy for each state:

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^{\pi}(s')]$$

This is equivalent to $\pi'(s) = \arg\max_a q^{\pi}(s, a)$.
Convergence: If $\pi' = \pi$ (policy is stable), then:

$$v^{\pi}(s) = \max_a q^{\pi}(s, a)$$

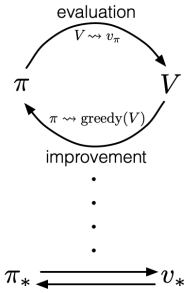
which means that π satisfied the Bellman **optimality** equation. Hence $v^{\pi} = v_*$ and $\pi = \pi_*$.

Policy Iteration

Key idea: Greedily improving with respect to v^π is guaranteed to make the policy at least as good, and continuing should reach the optimal policy.

Policy Iteration = Full Evaluation + Greedy Improvement:

1. **Policy Evaluation:** Compute v^π (iterate to convergence or near-convergence)
2. **Policy Improvement:** Greedify: $\pi'(s) = \arg\max_a q^\pi(s, a)$
3. Repeat outer loop until policy stabilizes: $\pi' = \pi$



The iteration chain: (E: Evaluation, I: Improvement)

$$\pi_0 \xrightarrow{\text{E}} v^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*, \pi_*$$

Because a finite MDP has a finite number of policies, this process is **guaranteed to converge** to the optimal value function (unique) and optimal policy (might be non-unique) in a **finite** number of iterations.

Policy Iteration — Pseudocode

Policy Iteration

Initialize π arbitrarily (e.g., random)
repeat
  Policy Evaluation:
    Compute $v^{\pi}(s)$ for all s via iterative Bellman:
    $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$ until convergence
  Policy Improvement:
    $\pi_{\text{old}} \leftarrow \pi$
    for each $s \in S$:
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
  if $\pi = \pi_{\text{old}}$ **then break**
until policy is stable

Full policy evaluation (itself iterative) can be slow. Each outer loop guarantees improvement or convergence.

Policy Iteration — Evolution of the Policy (1/2)

Idea: greedify w.r.t. v_k (the partial sweeps of π_0's evaluation) at each k, without waiting for full convergence.

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

v_0

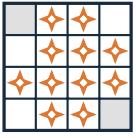
0.0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0.0

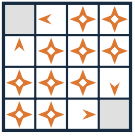
v_1

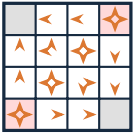
0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

v_2

greedy(v_k):







Policy Iteration — Evolution of the Policy (2/2)

Continuing further: the greedy policy stabilizes well before v_k itself converges.

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

v_3

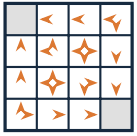
0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

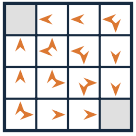
v_{10}

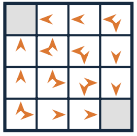
0.0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0.0

v^{π_0} $(k = \infty)$

greedy(v_k):







Result: by $k = 3$ the greedy policy already matches π_*; from $k = 10$ onward it settles into one consistent (valid) tie-break and never changes again.

Case Study: Jack's Car Rental

Sutton & Barto, Example 4.2: the textbook's worked example for policy iteration.

Setup: Jack manages two rental locations.

- **State:** number of cars at each location (up to 20 each)
- **Action:** move up to 5 cars between locations **overnight**
- **Reward:** \$10 per car rented, $-$\$2 per car moved
- **Dynamics:** requests and returns follow **Poisson distributions** (known rates), so $p(s'|s,a)$ is fully known $\Rightarrow$ this **is** a DP problem

What the book shows: starting from π_0 = "never move cars", policy iteration is run to convergence. Each iteration's policy is plotted as a heatmap over the state space (s_1, s_2); the boundaries between "move n cars" regions visibly shift and stabilize by π_4.

[Figure: policy heatmaps $\pi_0 \to \pi_4$ + final value function surface]

Plan

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration
4. Asynchronous Dynamic Programming
5. Convergence and Contraction

Value Iteration

Idea: Skip full policy evaluation. Instead, just apply the **Bellman optimality operator**:

$$v_{k+1}(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

Value Iteration = Direct optimization without explicit policy:

1. Initialize $v(s) = 0$ for all s
2. **repeat** (value update):
   - For each state s, Until $\|v_{\text{new}} - v\| < \epsilon$ (convergence)

$$v_{\text{new}}(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

Extract (pseudo) optimal policy (if needed):

$$\pi_*(s) \approx \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

Advantage: Single, simple update. **Tradeoff:** May need more iterations than policy iteration.

Value Iteration — Pseudocode

Value Iteration

Initialize $v(s) = 0$ for all $s \in S$
repeat
  $\Delta \leftarrow 0$
  for each $s \in S$:
    $v_{\text{old}}(s) \leftarrow v(s)$
    $v(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
    $\Delta \leftarrow \max(\Delta, |v(s) - v_{\text{old}}(s)|)$
until $\Delta < \epsilon$
return optimal policy: $\pi_*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$

Single update rule per iteration. Typically requires more iterations than policy iteration but simpler to implement.

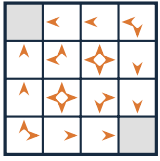
Value Iteration — Values Converging, Then the Policy

Bellman optimality update $v_{k+1}(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$, from $v_0 = 0$:

Extract the policy with one greedy sweep on $v_* = v_3$:

$$\pi_*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

Same π_* as the one Policy Iteration converged to (previous slide): both algorithms reach the same v_*, hence the same optimal actions.



π_* (all optimal actions shown)

Policy Iteration vs Value Iteration

	Policy Iteration	Value Iteration
Convergence guarantee	Yes (π_* in finite steps)	Yes (v_* asymptotically)
Speed (typical)	Fewer outer iterations	More iterations but simpler update
Computation per step	One full PE sweep	One Bellman max
When to use	Medium-sized MDPs	Large state spaces

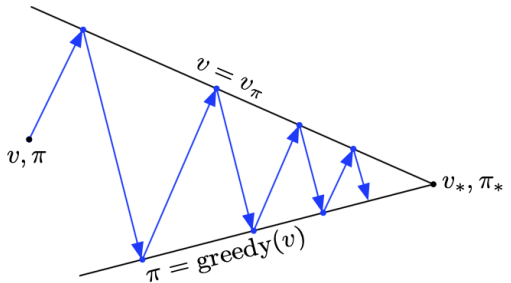
In practice: Most modern RL uses value iteration or variants (TD learning, etc.)

Generalized Policy Iteration

Core idea: Both Policy Iteration and Value Iteration are instances of the same framework. **Generalized Policy Iteration** is a generic name to describe the interaction of evaluation and improvement.

- **Policy Evaluation:** Estimate v^π.
  Any policy evaluation method
- **Policy Improvement:** Greedify w.r.t. v^π to get a new policy π'.
  Any policy improvement method

Value iteration can be seen as a special case of policy iteration where the evaluation step is truncated to a single update per state.



Plan

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration
4. Asynchronous Dynamic Programming
5. Convergence and Contraction

The Problem: Sweeps Don't Scale

Drawback of the algorithms so far: Every iteration requires a **sweep** over the **entire** state set S. If $|S|$ is very large, a single sweep can be prohibitively expensive.

Example: Backgammon

- Backgammon has over 10^{20} states
- Even at 10^6 state updates per second…
- …a single sweep would take **over a thousand years**

Question: Do we really need to update *every* state before making progress?

Asynchronous DP

Idea: In-place iterative DP algorithms that are **not** organized as systematic sweeps of the state set.

- States are updated **in any order**, using whatever values of other states happen to be available
- Some states may be updated several times before others are updated once
- **Great flexibility** in selecting which states to update, and when

One constraint remains: to converge correctly, the algorithm must keep updating **all** states; it cannot ignore any state forever after some point.

Example (asynchronous value iteration): update the value of only **one** state s_k at each step k, using the usual value iteration update.

Convergence guarantee: If $0 \leq \gamma < 1$, this converges to v_* provided every state appears in the sequence $\{s_k\}$ **infinitely often** (the sequence can even be stochastic).

In the undiscounted episodic case, some update orderings may fail to converge, but such orderings are easy to avoid.

Plan

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration
4. Asynchronous Dynamic Programming
5. Convergence and Contraction

The Bellman Operator as a Contraction

Why do these algorithms converge?

Define the **Bellman optimality operator**:

$$(\mathcal{T}^* v)(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

Theorem (Contraction Mapping): For discount factor $\gamma < 1$, $\mathcal{T}^*$ is a γ-contraction in sup-norm:

$$\|\mathcal{T}^* v_1 - \mathcal{T}^* v_2\|_\infty \leq \gamma \|v_1 - v_2\|_\infty$$

Consequence (Banach Fixed-Point Theorem):

- $\mathcal{T}^*$ has a unique fixed point v_*
- Repeated application: $v_k = \mathcal{T}^* v_{k-1}$ converges to v_*
- Convergence rate: $\|v_k - v_*\| \leq \gamma^k \|v_0 - v_*\|$ (exponential)

Sources : Full proof: Bertsekas & Tsitsiklis, *Neuro-Dynamic Programming* (1996), or `https://cfml.se/bellman-operators-are-contractions/` for a self-contained write-up.

Why $\gamma < 1$ Matters

With $\gamma < 1$:

- Bellman operator is a **contraction**
- Iterations provably converge
- Error shrinks exponentially
- Mathematical guarantee

With $\gamma = 1$ (undiscounted):

- May diverge (no contraction)
- Works *only* for episodic tasks (finite horizon)
- Or with bounded rewards + finite state space

Rule of thumb: Always use $\gamma < 1$ for infinite-horizon problems. Set $\gamma \approx 0.99$ by default.

Where This Is Going: AlphaZero

Go also has a fully known model: the rules are exact, so in principle it *is* a planning problem, just like Jack's Car Rental.

The catch: the state space is about 10^{170} states. Sweep-based DP (and even async DP) is hopeless at this scale.

AlphaZero's answer:

- Replace exhaustive backups with **Monte Carlo Tree Search**, a form of *planning* that expands only the most promising states, guided by a value estimate (conceptually a Bellman backup done selectively, not exhaustively)
- Replace the exact value/policy tables with a **neural network** $(\hat{v}, \hat{\pi})$ trained by self-play

Same idea, different scale: Bellman-style backups + policy improvement, but **approximated** and **selective** instead of exact and exhaustive.

This is exactly the tension motivating the rest of the course: exact DP $\rightarrow$ sample-based, approximate methods (Monte Carlo, TD, function approximation).

Key Takeaways

- **Dynamic Programming** solves MDPs when the model $p(s'|s, a)$ is known
- **Policy Evaluation:** Solve for v^π given π
- **Policy Iteration:** Repeat evaluation + greedy improvement → converges to π_*
- **Value Iteration:** Apply Bellman max directly → simpler, converges to v_*
- **Convergence guarantee:** Bellman operator is a γ-contraction ($\gamma < 1$)
- **Next:** Model-free learning (no need to know p) via Monte Carlo, TD, Q-learning

Questions?

Your feedback on this lecture is welcome

etienne.chassaing@centralesupelec.fr

Survey form (anonymous feedback):
`forms.gle/your-form-link`

Next: Kahoot quiz at start of Lecture 3!

Notes on AI and LLM

About this course: I do use LLMs/AI to create this course, with the following uses:

- Format LaTeX slides and exercises
- Find relevant sources
- Format equations and notations

However, plan and ideas are human-based, and everything content has been carefully reviewed. If you find some errors, or inconsistencies, please report them to me.

What about your own LLM usage?

Of course you also have the right to use LLMs, but don't forget you're in a learning step. Learning process goes through effort and the newer the subject, the more one needs empty-sheet recall.

* See: `[Source to be added]`