DS-GA 3001 001 | Lecture 3

Reinforcement Learning

Jeremy Curuksu, PhD
NYU Center for Data Science
jeremy.cur@nyu.edu

February 19, 2025

DS-GA 3001 RL Curriculum

Reinforcement Learning:

- Introduction to Reinforcement Learning
- Multi-armed Bandit
- **Dynamic Programming on Markov Decision Process**
- Model-free Reinforcement Learning
- Value Function Approximation (Deep RL)
- Policy Function Approximation (Actor-Critic)
- Planning from a Model of the Environment (AlphaZero)
- Aligning AI systems to Human Preferences (ChatGPT)
- Examples of Industrial Applications
- Advanced Topics

Dynamic Programming on Markov Decision Process

Last week:

- Multi-armed Bandit with action values (ϵ-greedy)
- Upper Confidence Bound
- Bayesian Bandit model

Today:

- **Markov Decision Process**
- **Value Functions and Bellman Equations**
- **Dynamic Programming**

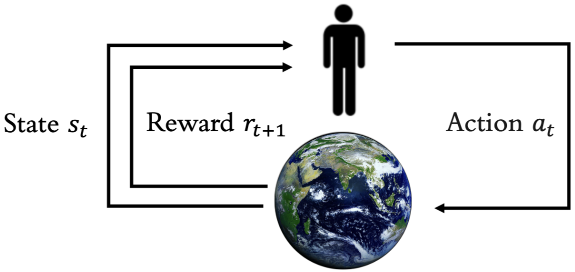
Generalization to Sequential RL

Sequential Goal-Directed Reinforcement Learning

- Bandit problems have only one state. But often the agent must learn different actions in different situations (states)
- Actions in turn may influence subsequent states, and through those states may influence future rewards
- To learn to make good decisions, we need assign credit for long term consequences to individual actions

Markov Decision Process

Markov Decision Process (MDP)



At each step, the agent:

- Finds itself in state s_t (from o_t)
- Executes action a_t
- Receives reward r_{t+1}

The environment:

- Receives action a_t
- Send reward r_{t+1}
- Send observation o_{t+1}

Markov Decision Process (MDP)

An MDP is a mathematical idealization of sequential goal-directed learning from interaction with an environment

- Simulating a MDP produces a sequence of n tuples (trajectory)

$$(s_t, a_t, r_{t+1}, s_{t+1})_n = (s_0, a_0, r_1, s_1, a_1, r_2, ..., s_n)$$

- The enviroment dynamics is fully characterized by the joint probability of each possible s_{t+1} and r_{t+1} as a function of the immediately preceding state and action, s_t and a_t

$$p(s', r|s, a) = p(s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a)$$

- **Markov property**: *The state must include all information from past agent–environment interactions that influence the future*

$$p\left(s, r \mid s_t, a_t\right) = p\left(s, r \mid H_t, a_t\right)$$

Goals and Rewards

RL applies the reward hypothesis

- The purpose of an RL agent is formalized in term of a signal called *reward* $r_t \in \mathbb{R}$ passing from the environment to the agent
- The agent goal is to maximize the amount of reward it receives

Reward:

$$r_t$$

Optimal Policy:

$$\pi_* = \arg\max_a \left(\sum r_t \right)$$

Agent goal is to maximize return G_t

G_t is the total accumulated reward from time-step t

- Acting in a MDP results in returns G_t that depend on the policy:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$$

- G_t can be discounted by factor $\gamma \in [0, 1]$ to account for present value of future rewards (in episodic or continuing tasks)

$$G_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k}$$

  - $\gamma < 1$ weights up immediate rewards vs. delayed rewards
  - γ close to 0 $\Rightarrow$ "Myopic" agent
  - γ close to 1 $\Rightarrow$ "Far-sighted" agent

Value Functions and Bellman Equations

State Value Function $v_\pi(s)$

Expected return when starting in s and following π

- Rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting (policies)

$$\forall s \in \mathcal{S}, \qquad v_\pi(s) \doteq \mathop{\mathbb{E}}_{\pi}(G_t \mid s)$$

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}(r_{t+1} + \gamma\, G_{t+1} \mid s)$$

$$v_\pi(s) = \mathbb{E}(r_{t+1} + \gamma\, v_\pi(s_{t+1}) \mid s)$$

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

- $v_\pi(s)$ indicates how good it is to be in s when following π

Action Value Function $q_\pi(s, a)$

Expected return when selecting a in s and following π

- The action value more directly informs on which action to take

$$\forall s \in \mathcal{S}, \qquad q_\pi(s, a) \doteq \underset{\pi}{\mathbb{E}}(G_t \mid s, a)$$

$$q_\pi(s, a) = \mathbb{E}(r_{t+1} + \gamma\, q_\pi(s_{t+1}, a_{t+1}) \mid s, a)$$

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

- $q_\pi(s, a)$ indicates how good it is to select a in s under π
- Note that $\sum_a \pi(a \mid s)\, q_\pi(s, a) = \underset{a}{\mathbb{E}}(q_\pi(s, a)) = v_\pi(s) \quad \forall s$

Optimal Value Functions v_* and q_*

Bellman Optimality equations

- $v_*(s)$ is the maximum state-value function over all policies:

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s)$$

$$v_*(s) = \max_{a} \mathbb{E}(r_{t+1} + \gamma\, v_*(s_{t+1}) \,|\, s, a)$$

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

- $q_*(s, a)$ is the maximum action-value function over all policies:

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a)$$

$$q_*(s, a) = \mathbb{E}(r_{t+1} + \gamma\, \max_{a'} q_*(s_{t+1}, a') \,|\, s, a)$$

$$q_*(s, a) = \sum_{s',\, r} p(s', r \,|\, s, a) \left[r + \gamma\, \max_{a'}\, q_*(s', a') \right]$$

Summary of Bellman equations

There are four main Bellman equations:

$$v_\pi(s) = \mathbb{E}(r_{t+1} + \gamma\, v_\pi(s_{t+1}) \mid s) \tag{1}$$

$$v_*(s) = \max_a \mathbb{E}(r_{t+1} + \gamma\, v_*(s_{t+1}) \mid s, a) \tag{2}$$

$$q_\pi(s,a) = \mathbb{E}(r_{t+1} + \gamma\, q_\pi(s_{t+1}, a_{t+1}) \mid s, a) \tag{3}$$

$$q_*(s,a) = \mathbb{E}(r_{t+1} + \gamma\, \max_{a'} q_*(s_{t+1}, a') \mid s, a) \tag{4}$$

- **There are equivalences between state and action values:**

$$v_\pi(s) = \sum_a \pi(a \mid s)\, q_\pi(s,a) = \mathbb{E}_a\left(q_\pi(s,a)\right)$$

$$v_*(s) = \max_a q_*(s,a)$$

Policy Evaluation and Optimization

Bellman equations are used for prediction and control

- **Policy Evaluation:** Evaluate a policy π by estimating v_π or q_π

$$v_{\pi'}(s) \geq v_\pi(s) \iff \pi' \geq \pi \quad \forall s$$

- **Policy Improvement:** Improve a policy by selecting actions that correspond to the highest values under the current policy

- **Example of greedy policy:** $\pi' = \begin{cases} 1, & \text{if } a = \arg\max_a(q_\pi(s,a)) \\ 0, & \text{otherwise} \end{cases}$

- **Theorem**: *For any MDP, there exists at least one optimal policy π_* that is better than or equal to all other policies:* $\pi_* \geq \pi, \forall \pi$

Policy Evaluation and Optimization

Bellman equations are used for prediction and control

- **Policy Evaluation:** Evaluate a policy π by estimating v_π or q_π

$$v_{\pi'}(s) \geq v_\pi(s) \iff \pi' \geq \pi \quad \forall s$$

- **Policy Improvement:** Improve a policy by selecting actions that correspond to the highest values under the current policy

- **Example of greedy policy:** $\pi_* = \begin{cases} 1, & \text{if } a = \arg\max_a(q_*(s,a)) \\ 0, & \text{otherwise} \end{cases}$

- **Theorem**: *For any MDP, there exists at least one optimal policy π_* that is better than or equal to all other policies:* $\pi_* \geq \pi, \ \forall \pi$

Solving Bellman Equations

Solving the RL problem algebraically

- For n states, the Bellman equation is a system of n equations in n unknowns, so can be solved algebraically:

$$V = R + \gamma P^{\pi} V$$

$$(I - \gamma P^{\pi}) V = R$$

$$V = (I - \gamma P^{\pi})^{-1} R$$

where: $v_i = v(s_i)\,, \quad r_i = \mathbb{E}_{\pi}[r_t | s_i]\,, \quad P^{\pi}_{ij} = \sum_a \pi(a \,|\, s_i)\, p(s_j \,|\, s_i, a)$

- Solving Bellman equations algebraically is akin to exhaustive search ($O(n^3)$), thus can be computed only for small problems

- This method assumes (1) Markov property, (2) MDP dynamics is known, (3) we have enough ressources to compute the solution

Solving Bellman Equations

Solving the RL policy evaluation problem

- Bellman equations for the evaluation problem are linear:

$$V = (I - \gamma P^{\pi})^{-1} R$$

Solving the RL policy optimization problem

- Bellman optimality equations are non-linear, thus can't be solved directly (require methods for non-linear equations)
- RL optimization often relies on iterative solution methods:
  - Dynamic Programming (use a model)
  - Monte-Carlo, Temporal Difference (use samples)

Dynamic Programming

Dynamic Programming

- **DP refers to a collection of algorithms to evaluate and/or improve policies given a model of the environment as a MDP**
- DP is an essential foundation: all RL methods can be viewed as attempts to achieve the same effect as DP, but with less computation and without a perfect model of the environment
- Key idea of DP is the use of value functions to organize the search for good policies
- All DP methods consist of two different parts: policy evaluation and (optionally) policy improvement
- All DP methods update estimates of the values of states based on estimates of the values of successor states (*bootstrapping*)

Policy Evaluation for a Given Policy

Estimate $v_\pi(s)$ of a given policy π

- Turn the Bellman equation

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

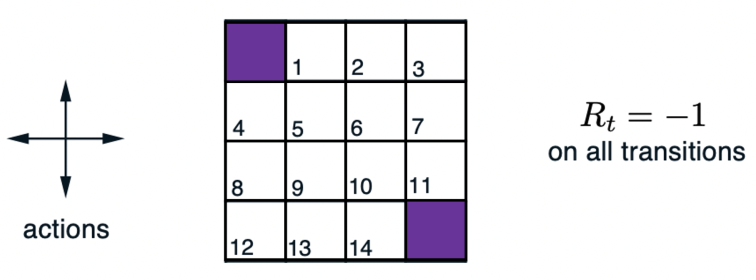
...into an update function:

- **Initialize v_0 e.g., to zero, then iterate:**

$$\forall s \,, \; v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma \, v_k(s') \right]$$

- **Whenever $v_{k+1}(s) = v_k(s)$, for all s, we have found v_π**
- It can be shown that $\lim_{k \to \infty} v_k = v_\pi$ (demonstration out of scope)

Example of Policy Evaluation



Evaluate random policy π_{random} in this 4×4 gridworld

- At each iteration k, loop through all states and update value estimate $v_k(s)$ of every state s for π_{random}

Example of Policy Evaluation

k = 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	0.0

k = 3

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 = 1

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 = 10

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 = 2

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

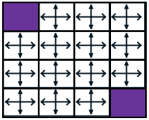
$k = \infty$

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

Example of Policy Evaluation

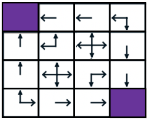
k = 0

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	0.0



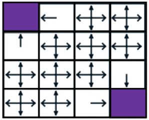
k = 3

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



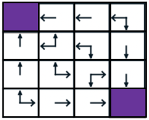
k = 1

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



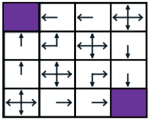
k = 10

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



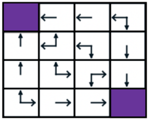
k = 2

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 =∞

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



Policy Improvement

Find a better policy π' given $v_\pi(s)$

1. **For a given policy π, take:**

$$\forall s: \;\; \pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma\, v_\pi(s')\right]$$

2. **Evaluate $v_{\pi'}(s)$ as in previous slides (policy evaluation)**

3. **Repeat**

Policy Improvement Theorem:

$\forall s,\; \pi'(s) = \arg\max_a q_\pi(s,a) \Rightarrow v_{\pi'}(s) \geq v_\pi(s)$ i.e., π' better or same as π

Dem.: $\pi'(s) = \arg\max_a q_\pi(s,a) \Leftrightarrow v_{\pi'}(s) = \max_a q_\pi(s,a) \geq \mathbb{E}_a\left[q_\pi(s,a)\right] = v_\pi(s)$ $\square$

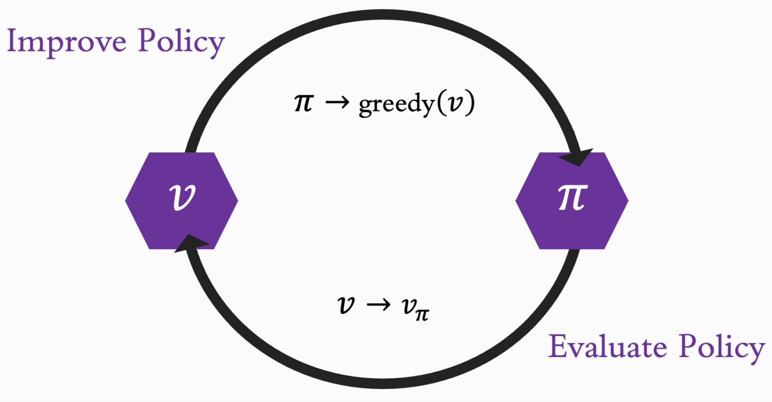
Implication: When $\forall s,\; v_{\pi'}(s) = v_\pi(s)$, then $\forall s,\; v_\pi(s) = \max_a q_\pi(s,a)$

This is the Bellman optimality equality, thus π is optimal: $v_\pi = v_* = v_{\pi'}$

$\implies \pi' = \arg\max_a q_\pi(s,a)$ either is an improvement with respect to π, or is optimal.

Generalized Policy Iteration

All RL methods are Generalized Policy Iteration methods



Practice: Policy Iteration Algorithm

Policy Iteration iterates multiple loops over all states to evaluate v, then loops over all states once to improve π, then repeats:

Initialize $v(s)$ and $\pi(s)$ arbitrarily for all s

1. Loop:

    $\Delta = 0$

    For each s:

    $v_{\text{old}} = v(s)$

    $v(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[(r + \gamma\, v(s')\right]$

    $\Delta = \max(\Delta, |v_{\text{old}} - v(s)|)$

    Stop when $\Delta < \xi$

2. For each s:

    $\pi_{\text{old}}(s) = \pi(s)$

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

Stop if $\pi_{\text{old}} \iff \pi(s)$, else go to step 1

Practice: Value Iteration Algorithm

Policy improvement with truncated policy evaluation

- Policy iteration involves policy evaluation at each iteration, which may itself require multiple loops through all states
- Is exact convergence needed, or can we stop sooner? When?
- Policy evaluation can be truncated in several ways without losing the convergence guarantees of policy iteration
- **A special case is when policy evaluation is stopped after just one loop (one update of each state). It is equivalent to turning the Bellman optimality equation into an update function:**

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_k(s') \right]$$

Practice: Value Iteration Algorithm

Value Iteration truncates policy evaluation to only 1 step for each step of policy improvement, and performs both simultaneously

Initialize $v(s)$ arbitrarily for all s

Loop:

$\Delta = 0$

For each s:

$v_{\text{old}} = v(s)$

$v(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[(r + \gamma\, v(s') \right]$

$\Delta = \max(\Delta, |v_{\text{old}} - v(s)|)$

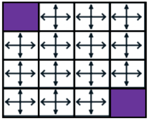
Stop when $\Delta < \xi$

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

Example of Value Iteration

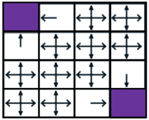
k = 0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0



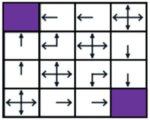
k = 1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0



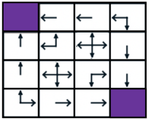
k = 2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-1
-2	-2	-1	0



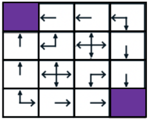
k = 3

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0



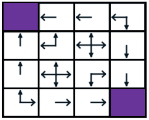
k = 10

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0



k =∞

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0



Asynchronous Dynamic Programming

Update values in any order whatsoever...

- DP algorithms described so far loop over all states, but in practice this is often impossible (e.g., Chess has 10^{40} states)
- Asynchronous DP backs up states in any order, and still converges if it continues to udpate values of all states
- **Asynchronous DP makes it possible to focus DP updates onto parts of the state space that are most relevant to the agent:**
  - **Prioritised Sweeping**: States with largest Bellman Error:

$$\max \left| [r_{t+1} + \gamma \hat{v}(s_{t+1}) \mid s] - \hat{v}(s) \right|$$

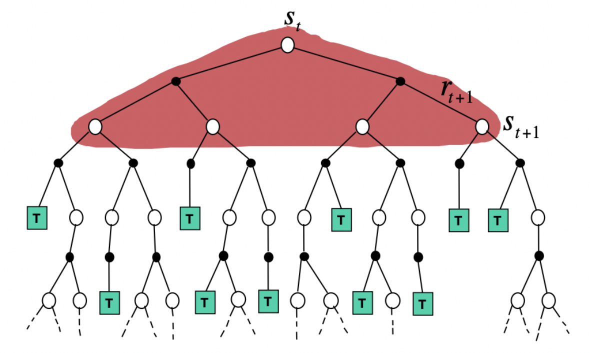
  - **Real-time DP**: Agent's real experience determines states to update, while latest values guide its decision making

Efficiency of Dynamic Programming

DP provides a well defined notion of optimality, but is often an ideal that AI agents can only approximate

- ✓ Asynchronous DP often exponentially faster than direct search
- ✓ ...in particular if agent starts with good initial values or policies
- ✓ DP is iterative so can learn with limited compute resources
- ✓ With today's computers, DP can solve MDPs with millions of state (assuming a small number of actions)
- × In most cases of practical interest, a perfect MDP model of state transitions and rewards is not available
- × In most cases of practical interest, there are far more states that there could possibly be entries in a look-up table

Efficiency of Dynamic Programming



DP often suffers from the curse of dimensionality

- DP uses full-width backups
- Even one full-depth backup can be too expensive
- Need to sample (next lecture)

Today's Takeaways

Dynamic Programming provides a foundation for RL by numerically solving the Bellman equations with two processes taking place in parallel:

- **Process of prediction** which can learn values online at every step by bootstrapping estimates on the basis of previous estimates until they are consistent with a policy followed.
- **Process of optimization** which can improve the policy followed by making it greedy with respect to the latest value estimates.
- **Both processes stabilize only when a policy has been found that is greedy with respect to its own value function.**
- But Dynamic Programming needs a model of the environment.

Thank you!