Top of Form

Pieter Abbeel's "Foundations of Deep Reinforcement Learning" series begins with **Lecture 1: MDPs, Exact Solution Methods, Max-ent RL**, laying the groundwork for understanding advanced DRL techniques.

**1. Motivation for Deep Reinforcement Learning (DRL)** The excitement around DRL largely stems from significant breakthroughs:

• **DeepMind's Atari Games (2013):** Neural network agents, using DRL, learned to play a wide range of Atari games from visual inputs, a major advance from previous small-scale "toy" problems in RL.

• **Robotics Applications:** Abbeel's lab at Berkeley applied DRL to simulated robots (swimmer, hopper, 2D/3D walkers) and later real robots (e.g., BRETT putting blocks into openings). These robots learned complex skills through **trial-and-error reinforcement learning**, rather than careful manual programming. A key takeaway is that the same piece of code could train different robots or different tasks, highlighting the generalizability of the approach.

• **AlphaGo (2015):** DeepMind's AlphaGo, powered by RL, defeated the human world champion at Go, a feat many thought years away.

• **OpenAI's Dota 2 (2017) and Rubik's Cube (2019):** OpenAI demonstrated DRL's capability in mastering complex video games like Dota 2 and enabling a robot hand to solve a Rubik's cube, even after being trained entirely in simulation. These examples illustrate that DRL allows agents and robots to **acquire skills from their own trial-and-error learning**, indicating a powerful learning capability that can transfer to new tasks. At the core of all this is **Reinforcement Learning** and its underlying framework: **Markov Decision Processes** (MDPs).

**2. Markov Decision Processes (MDPs)** An MDP provides a formal structure for an agent's interaction with an environment:

• **Agent-Environment Interaction:** An agent chooses an action, the environment changes, and the agent observes the new state, repeating the cycle.

• **Reward:** A scalar value associated with the environment's situation, indicating how "good" it is (e.g., video game score, forward progress for a robot).

• **Goal:** For the agent to learn the right action for each situation over time to **maximize reward**.

• **Assumption:** The agent can observe the full state of the environment (not partially observable, which would be POMDPs).

**Formal Definition of an MDP:** An MDP is defined by seven components:

1. **Set of States (S):** All possible configurations of the environment.

2. **Set of Actions (A):** All actions the agent can choose from.

3. **Transition Function (P(s'|s, a)):** The probability of ending up in state *s'* at the next time step, given the agent is in state *s* and took action *a*.

4. **Reward Function (R(s, a, s')):** The reward assigned for a transition from *s* to *s'* after taking action *a*.

5. **Start State (s0) or Start State Distribution:** Where the agent begins.

6. **Discount Factor (gamma, γ):** A value between 0 and 1 that discounts future rewards. **Rewards received sooner are valued more highly** than those received later. This is analogous to money earned today being more valuable due to potential interest earnings. A gamma of 0.9 implies a care for a horizon of about 10 steps, while 0.99 implies a horizon of 100 steps.

7. **Horizon (H):** The number of steps the agent will be acting.

The **goal of a reinforcement learning algorithm** is to find a policy that **maximizes the expected discounted reward over time**.

**Examples of Problems that Map to MDPs:**

• **Cleaning Robot:** State = house layout + robot location; Actions = move, pick up; Reward = dirt removed, objects organized; Gamma = long horizon (0.999).

• **Walking Robot:** State = joint angles/velocities, ground configuration; Actions = motor torques; Reward = standing still, forward progress; Gamma = based on desired behavior duration.

• **Pole Balancing:** A classic low-dimensional control task.

• **Games:** Tetris, Backgammon, Atari, Go.

• **Server Management:** State = requests, server load; Actions = assign jobs; Reward = throughput, research breakthroughs.

• **Shortest Path Problems**.

• **Modeling Animals or People:** E.g., self-driving cars modeling other drivers' behaviors as optimizing their own MDPs.

**Grid World Example (for Intuition Building):** A simple 2D grid environment where an agent can move to neighboring squares.

• **State:** Agent's location (e.g., (1,3)).

• **Actions:** Move Up, Down, Left, Right (unless blocked by boundaries or obstacles).

• **Rewards:** +1 for reaching a diamond square, -1 for falling into a fire pit. Other moves yield 0 reward.

• **Dynamics:** Can be deterministic (actions always succeed) or stochastic (e.g., 80% success, 20% veer off to side).

• **Goal:** Find a policy (mapping states to actions) that maximizes expected discounted reward. The discount factor encourages the **shortest path** to rewards, as longer paths incur more discounting.

**3. Exact Solution Methods for MDPs** For small-scale MDPs, exact methods can compute the optimal policy.

**3.1. Value Iteration** Value iteration is a foundational method.

• *(s)):*\* Represents **the maximum expected discounted sum of rewards** an agent can achieve starting from state *s*, assuming it always acts optimally.

• *(s) calculation):*\*

    ◦ **Deterministic Actions, Gamma = 1 (no discounting), H = 100:** If the agent starts at (4,3) (the diamond), V\*(4,3) = 1. If it starts next to it (3,3), it takes one step to get 1, so V\*(3,3) = 1. Even far-off states like (1,1) will have V\*(1,1) = 1 because there's no discounting and enough horizon. If trapped in a fire pit (4,2), V\*(4,2) = -1.

    ◦ **Deterministic Actions, Gamma = 0.9, H = 100:** Now discounting matters. V\*(4,3) = 1. For (3,3), it takes one step, so V\*(3,3) = 0.9 \* 1 = 0.9. For (2,3), two steps, V\*(2,3) = (0.9)^2 \* 1 = 0.81. For (1,1), five steps, V\*(1,1) = (0.9)^5 \* 1 = 0.59.

    ◦ **Noisy Actions (0.8 success, 0.1 left/right), Gamma = 0.9:** The value of a state becomes recursive. To know V\*(3,3), you need to consider the probabilities of transitioning to neighboring states and their respective values. This is where the Bellman Equation comes in.

• **Value Iteration Algorithm (Bellman Update / Backup):**

    1. **Initialization:** Set V0\*(s) = 0 for all states *s* (value when zero time steps are left).

    2. **Iteration:** For k = 1 to H (horizon), update the value function for each state *s* as follows: *(s) = max\_a Σ\_s' P(s'|s,a) [R(s,a,s') + γ \* V\_{k-1}\*(s')]*\* This means for each state *s*, find the action *a* that maximizes the expected immediate reward *R(s,a,s')* plus the discounted value *γ \* V\_{k-1}*(s')\* from the next state *s'*.

    3. **Demonstration:** In the grid world, after 0 iterations, all values are 0. After 1 iteration, only the diamond square gets +1 and the fire pit -1. After successive iterations, values "fan out" from the high-reward states to neighboring states, reflecting the path and discounting.

• **Convergence:**

    ◦ Value iteration is guaranteed to **converge** to the *)*\* for the discounted infinite horizon problem.

    ◦ At convergence, it satisfies the **Bellman Equation**.

    ◦ The resulting *) is stationary*\*, meaning the optimal action for a given state *s* is always the same, regardless of time.

    ◦ **Intuition for Convergence:** The difference between the optimal value for a finite horizon (V\_H\*) and the infinite horizon (V\*) shrinks to zero as H increases. This is because future rewards are discounted by γ^(H+1), and since γ < 1, this term quickly becomes very small.

    ◦ **Contraction Mapping:** The value iteration update is a **gamma-contraction in the max norm**, meaning it pulls any two value functions closer together with each update. This guarantees convergence to a unique fixed point (V\*) regardless of the initial value function. Once the update is small, it implies proximity to convergence.

• **Environment Parameters & Optimal Policy:** An exercise demonstrates how varying the discount factor (γ) and the noise in action dynamics can lead to different optimal policies (e.g., preferring a close, small reward over a distant, large reward, or avoiding cliffs due to noise).

*(s,a))*\*

• **Definition:** Q\*(s,a) is the **expected discounted sum of rewards** if you start in state *s*, take action *a*, and then act optimally thereafter. It represents the value of being in a state *and* having committed to a specific action.

• **Bellman Equation for Q-Values:** *(s,a) = Σ\_s' P(s'|s,a) [R(s,a,s') + γ \* max\_a' Q*(s',a')]\*\* This equation indicates that the optimal Q-value for (s,a) is the expected immediate reward plus the discounted maximum Q-value achievable from the next state *s'* (by choosing the optimal action *a'*).

• **Q-Value Iteration:** Similar to value iteration, it iteratively computes Q-values and also converges to the optimal Q\*(s,a) for all state-action pairs.

• **Benefit:** Once Q-values are computed, the **optimal action in any state can be directly identified by choosing the action *a* that yields the highest Q-value** for that state *s*.

**3.3. Policy Iteration** Policy iteration is another exact method, often converging in fewer *overall* iterations than value iteration, though each iteration can involve more work.

• **Two Steps:** It alternates between **policy evaluation** and **policy improvement**.

    1. **Policy Evaluation:** Given a fixed policy π\_k, compute its value function V\_πk(s). This is done by solving the Bellman equation for that *fixed* policy (no max\_a operator): **V\_πk(s) = Σ\_a π\_k(a|s) Σ\_s' P(s'|s,a) [R(s,a,s') + γ \* V\_πk(s')]** This is essentially value iteration but constrained to follow the actions prescribed by π\_k (or probabilities in a stochastic policy). It also converges.

    2. **Policy Improvement:** Given V\_πk(s), find a new, improved policy π\_{k+1}(s) by performing a one-step lookahead: **π\_{k+1}(s) = argmax\_a Σ\_s' P(s'|s,a) [R(s,a,s') + γ \* V\_πk(s')]** This means for each state *s*, choose the action *a* that maximizes the expected immediate reward plus the discounted value from the next state, assuming you follow the *old* policy π\_k for all subsequent steps.

• **Convergence:** Policy iteration is guaranteed to *) and optimal value function (V*)\*\*. This is because each policy improvement step produces a better policy (or the same if already optimal), and in a finite MDP, there's a finite number of policies, so it must converge. At convergence, the Bellman optimality equation is satisfied.

**4. Maximum Entropy Formulation (Max-Ent RL)** This approach aims to find a distribution over near-optimal solutions, offering robustness and better exploration capabilities.

• **Motivation:** Instead of a single optimal policy, find a distribution of behaviors that are all good or put higher probability on good behaviors. This leads to **more robust policies** (if one path is blocked, others exist) and **more robust learning** (by introducing more variation in data collection, aiding exploration).

• **Entropy:** A measure of uncertainty over a random variable. Mathematically, for a distribution P, H(P) = -Σ\_i P(x\_i) log\_2(P(x\_i)). Higher entropy means more uncertainty (e.g., a uniform distribution has higher entropy than a sharply peaked one).

• **Max-Entropy MDP Objective:** The goal is to maximize: **Expected Sum of Rewards + β \* Entropy of the Policy**.

    ◦ **β (Beta):** A trade-off factor. Higher β means more emphasis on entropy (leading to more stochastic, exploratory policies). If β = 0, it reduces to standard RL.

• **Solving Max-Entropy MDPs (Max-Ent Value Iteration):**

    ◦ Uses concepts from **constrained optimization** (e.g., Lagrangians).

    ◦ For a one-step problem, the *(s,a))*\*. This effectively means actions with higher Q-values (rewards) get higher probabilities, but *all* actions with non-zero reward will have some probability, making the policy stochastic.

    ◦ The optimal value function V\_k(s) is then computed as a **soft-max** of the Q-values, rather than a direct max as in standard value iteration. The "sharpness" of this soft-max is determined by β.

    ◦ **Benefit:** This formulation naturally leads to **stochastic policies**, which can be beneficial for **exploration** during learning.

**5. Limitations of Exact Methods & Future Direction** While exact methods provide a strong foundation, they have a critical limitation:

• They require **looping over all states and actions**.

• This is **impractical for large-scale problems** where the number of states and actions can be astronomically large (e.g., Tetris, Atari, continuous robot control).

The subsequent lectures in the series will build upon these foundations to explore **approximate solution methods** that can handle these larger, more complex MDPs.

Bottom of Form

NotebookLM can be inaccurate; please double check its responses.