Deep Reinforcement Learning

Lokesh

1 Module 1: Multi-Armed Bandits (MAB)

1.1 1.1 The Setting: Reinforcement Learning in One State

The Multi-Armed Bandit is the simplest form of reinforcement learning. Unlike the full RL problem, it is **non-associative**. There is only one situation (state), and the agent must simply find the best action.

The k-Armed Bandit Problem

You are faced with k different options (actions). After each choice, you receive a numerical reward chosen from a stationary probability distribution depending on the action you selected.

- **Objective:** Maximize the expected total reward over some time period (T steps).
- **Value of an Action (q_*):** The true expected reward of action a.

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a]$$

- **Estimated Value (Q_t):** Since we do not know q_*, we calculate an estimate $Q_t(a)$ at time step t.

1.2 1.2 Action-Value Methods

How do we estimate $q_*(a)$? The most natural way is **Sample Averages**.

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken}}{\text{number of times } a \text{ taken}} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

By the **Law of Large Numbers**, as the number of times we try an action goes to infinity, $Q_t(a) \to q_*(a)$.

Mathematical Deep Dive: Incremental Implementation

Storing every single reward to calculate an average is memory intensive. We can update our average **incrementally**. Let Q_n be the average after $n-1$ selections. Let R_n be the n-th reward.

$$Q_{n+1} = \frac{1}{n} \sum_{i=1}^{n} R_i = \frac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i)$$

Since $Q_n = \frac{1}{n-1} \sum_{i=1}^{n-1} R_i$, we can substitute:

$$Q_{n+1} = \frac{1}{n}(R_n + (n-1)Q_n) = Q_n + \frac{1}{n}[R_n - Q_n]$$

The General Update Rule:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}[\text{Target} - \text{OldEstimate}]$$

1.3 1.3 Tracking Non-Stationary Problems

In many real-world cases (e.g., website click-through rates), the true value $q_*(a)$ changes over time. Sample averages treat all rewards (old and new) equally. This is bad if the world changes. To prioritize

recent rewards, we use a constant step-size parameter $\alpha \in (0, 1]$.

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]$$

This results in a **weighted average** where rewards from long ago are weighted by $(1-\alpha)^k$, decaying exponentially.

1.4 1.4 Exploration vs. Exploitation Algorithms

1.4.1 A. ϵ-Greedy Strategies

The fundamental conflict in RL.

- **Exploit:** Choose $A_t^* = \arg\max Q_t(a)$.
- **Explore:** Choose random action.

ϵ-greedy balances this by behaving greedily most of the time $(1 - \epsilon)$ but exploring randomly with probability ϵ. **Drawback:** It explores equally among all actions, even the ones that we are almost certain are bad.

1.4.2 B. Optimistic Initial Values

Instead of setting initial estimates $Q_0(a) = 0$, we set them to a high value (e.g., +5, if rewards are usually +1).

- **Mechanism:** The agent tries an action, gets +1. The average drops from +5 to +3. It is "disappointed".
- **Result:** It switches to other actions (which are still at +5).
- **Benefit:** This encourages high exploration early on without needing a high ϵ.

1.4.3 C. Upper Confidence Bound (UCB)

This measures the **potential** of an action.

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

- $\ln t$: Natural log of total time. Grows slowly. Ensures infinite exploration.
- $N_t(a)$: Times action a was selected. As this grows, uncertainty shrinks.
- c: Confidence parameter.

Intuition: "I will pick the arm that is either known to be good (high Q) or is unknown (low N)."

1.4.4 D. Gradient Bandit Algorithms (Softmax)

Instead of values Q, we learn numerical **preferences** $H_t(a)$. The probability of picking action a is determined by the Softmax distribution:

$$\pi_t(a) = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}}$$

We update preferences using Stochastic Gradient Ascent.

1.5 Solved Exam Problems: Bandits

Exam Problem: Regular 2024 - The Investor Strategy (ϵ-greedy)

Scenario: Investor trading 4 stocks (A, B, C, D) using ϵ-greedy ($\epsilon = 0.1$) for 100 trades.
Returns:

- A: 70% (+1), 30% (0)
- B: 50% (+2), 50% (0)
- C: 10% (+5), 90% (0)
- D: Guaranteed +0.5

Questions: (a) Model as RL. (b) Calculate expected return over 100 trades.

$\Rightarrow$ Professor's Detailed Solution:

(a) RL Modeling:

- **Agent:** The Investor.
- **Environment:** The Stock Market.
- **State:** Single State (MAB).
- **Actions $\mathcal{A}$:** {Invest A, Invest B, Invest C, Invest D}.
- **Reward Function:** $R_t \in \{0, 0.5, 1, 2, 5\}$.

(b) Expected Return Calculation:

Step 1: Calculate True Values (q_*) We compute the expected value $\mathbb{E}[R]$ for each stock:

$$q_*(A) = 0.7(1) + 0.3(0) = \mathbf{0.7}$$

$$q_*(B) = 0.5(2) + 0.5(0) = \mathbf{1.0} \quad (\textit{Optimal Action } a^*)$$

$$q_*(C) = 0.1(5) + 0.9(0) = \mathbf{0.5}$$

$$q_*(D) = 1.0(0.5) = \mathbf{0.5}$$

Step 2: Determine Policy Probabilities ($\epsilon = 0.1$) The policy $\pi(a)$ is defined as:

- Probability of exploring (random action) $= \epsilon = 0.1$. This is distributed among all 4 arms. $P_{explore} = \frac{0.1}{4} = 0.025$.
- Probability of exploiting (greedy action) $= 1 - \epsilon = 0.9$. This applies only to optimal arm B.

Final Probabilities:

- $P(A) = 0.025$
- $P(B) = 0.9 + 0.025 = \mathbf{0.925}$
- $P(C) = 0.025$
- $P(D) = 0.025$

Step 3: Calculate Expected Reward per Step

$$\mathbb{E}[R] = \sum_a \pi(a) \cdot q_*(a)$$

$$= 0.925(1.0) + 0.025(0.7) + 0.025(0.5) + 0.025(0.5)$$

$$= 0.925 + 0.0175 + 0.0125 + 0.0125 = \mathbf{0.9675}$$

Step 4: Total for 100 Trades

$$Total = 100 \times 0.9675 = \mathbf{96.75}$$

Exam Problem: Makeup 2024 - UCB Manual Calculation

History ($t = 30$ total plays):

- Arm 1: 5 plays, avg reward 0.2
- Arm 2: 10 plays, avg reward 0.5
- Arm 3: 15 plays, avg reward 0.4

Task: Calculate UCB for each arm using $c\sqrt{\frac{2\ln t}{N_t(a)}}$ with $c = 1$.

⇒ **Professor's Detailed Solution:**

Concept: UCB selects $A_t = \arg\max[Q_t(a) + \text{Bonus}]$. First, calculate the common numerator term: $\ln(30) \approx 3.401$. $2\ln(30) \approx 6.802$.

Arm 1 (The Underdog):

- Exploitation: $Q = 0.2$
- Exploration: $\sqrt{\frac{6.802}{5}} = \sqrt{1.3604} \approx 1.166$
- **UCB Score:** $0.2 + 1.166 = \mathbf{1.366}$

Arm 2 (The High Performer):

- Exploitation: $Q = 0.5$
- Exploration: $\sqrt{\frac{6.802}{10}} = \sqrt{0.6802} \approx 0.825$
- **UCB Score:** $0.5 + 0.825 = \mathbf{1.325}$

Arm 3 (The Frequent Choice):

- Exploitation: $Q = 0.4$
- Exploration: $\sqrt{\frac{6.802}{15}} = \sqrt{0.4535} \approx 0.673$
- **UCB Score:** $0.4 + 0.673 = \mathbf{1.073}$

Conclusion: The agent selects **Arm 1**. Despite having the lowest average reward, its uncertainty bonus (due to low visit count) makes it the most attractive option to explore.

Exam Problem: Makeup 2024-25 - Decaying ϵ-greedy

Scenario: Website A/B/C/D test. $Q_{values} = \{0.05, 0.12, 0.09, 0.15\}$. True CTRs $R = \{0.08, 0.15, 0.10, 0.18\}$. Current $\epsilon = 0.15$. (a) Effect of decaying ϵ? (b) Probabilities? (c) Expected Reward?

⇒ **Professor's Detailed Solution:**

(a) Decaying ϵ: Using a fixed ϵ means the agent continues to explore sub-optimal arms forever (infinite regret). By decaying ϵ (e.g., $\epsilon_t = 1/\sqrt{t}$), the agent explores heavily at the start to find the best arm, then gradually converges to pure exploitation, minimizing long-term regret.

(b) Probabilities: Greedy Arm = Arm 4 (Q=0.15).

- $P(Arm4) = (1 - \epsilon) + \epsilon/4 = 0.85 + 0.0375 = \mathbf{0.8875}$
- $P(Arm1, 2, 3) = \epsilon/4 = \mathbf{0.0375}$

(c) Expected Reward (using True CTRs):

$$\begin{aligned}\sum P(i)R(i) &= 0.0375(0.08 + 0.15 + 0.10) + 0.8875(0.18) \\ &= 0.0375(0.33) + 0.15975 \\ &= 0.012375 + 0.15975 = \mathbf{0.1721}\end{aligned}$$

Exam Problem: EC3 Regular 2025 - Healthcare Agent

Data ($t = 1$ to 5): A1(+1), A2(-1), A2(+1), A1(+1), A1(-1). **Task:** Find action at $t = 6$ using UCB ($C = 1$).

⇒ **Professor's Detailed Solution:**

Step 1: Statistics at $t = 5$

- **A1:** Count=3. Rewards sum $= 1 + 1 - 1 = 1$. $Q = 1/3 \approx 0.333$.
- **A2:** Count=2. Rewards sum $= -1 + 1 = 0$. $Q = 0$.

Step 2: UCB Scores $\ln(5) \approx 1.609$.

- A1: $0.333 + \sqrt{1.609/3} = 0.333 + 0.732 = \mathbf{1.065}$
- A2: $0.0 + \sqrt{1.609/2} = 0.0 + 0.897 = \mathbf{0.897}$

Result: Choose **A1** (Recommend).

---

2 Module 2: Finite Markov Decision Processes

2.1 2.1 The Agent-Environment Interface

In Bandits, actions had no long-term consequences. In MDPs, actions influence not just immediate rewards, but also **future states**. This formally describes **sequential decision making**.

The MDP Tuple

An MDP is defined by $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$:

1. $\mathcal{S}$: The set of all possible **States**.
2. $\mathcal{A}$: The set of all possible **Actions**.
3. $\mathcal{P}$: The **Transition Dynamics**.

$$p(s'|s,a) \doteq \Pr\{S_{t+1} = s' | S_t = s, A_t = a\}$$

4. $\mathcal{R}$: The **Reward Function**. $r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$.
5. γ: The **Discount Factor** $\in [0,1]$.

2.2 2.2 The Markov Property

"The state captures all relevant information from the history." A state S_t is Markov if:

$$\Pr\{S_{t+1}|S_t, A_t\} = \Pr\{S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, \dots, S_0\}$$

Once the state is known, the history can be thrown away. It is sufficient statistic for the future.

2.3 2.3 Returns and Episodes

The goal of the agent is to maximize the **Cumulative Return** G_t.

- **Episodic Tasks:** The interaction breaks into subsequences (episodes) that end in a terminal state (e.g., Mario dying or winning).

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$

- **Continuing Tasks:** The interaction goes on forever (e.g., a thermostat). To avoid infinite sums, we use **discounting**.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Mathematical Deep Dive: Recursive Definition of Return

This is critical for Bellman equations:

$$G_t = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots)$$

$$G_t = R_{t+1} + \gamma G_{t+1}$$

2.4 2.4 Policies and Value Functions

- **Policy $\pi(a|s)$:** A mapping from states to probabilities of selecting each possible action.
- **State-Value Function $v_\pi(s)$:** How good is it to be in state s following policy π?

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

- **Action-Value Function $q_\pi(s, a)$:** How good is it to take action a in state s following policy π?

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

2.5 2.5 The Bellman Equations

The value functions satisfy recursive relationships. These are the most important equations in RL.

Bellman Expectation Equation

The value of a state is the immediate reward plus the discounted value of the next state, averaged over all possibilities.

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

2.6 2.6 Optimal Value Functions

Solving a reinforcement learning task means finding a policy that achieves a lot of reward over the long run.

- $v_*(s) = \max_\pi v_\pi(s)$
- $q_*(s, a) = \max_\pi q_\pi(s, a)$

Bellman Optimality Equation

Under the optimal policy, the value of a state must equal the expected return for the *best* action from that state.

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

2.7 Solved Exam Problems: MDPs

Exam Problem: Regular 2023 - 2-State Value Iteration

Setup: States A, B. Action "Shift".

- A → A (25%, R=X-2), A → B (75%, R=X-3)
- B → A (60%, R=X-1), B → B (20%, R=X), B → Term (20%, R=X+2)

$X = 6$, $\gamma = 0.4$. Init $V(A) = V(B) = 1$. **Task:** Perform 1 iteration of Value Iteration.

⇒ **Professor's Detailed Solution:**

Step 1: Calculate Rewards ($X = 6$)

- $R_{AA} = 4$, $R_{AB} = 3$
- $R_{BA} = 5$, $R_{BB} = 6$, $R_{BTerm} = 8$

Step 2: Bellman Update Rule Since there is only one action, $\max_a$ is trivial. $V_{new}(s) = \sum p(s'|s)[r + \gamma V_{old}(s')]$

Update State A:

$$V(A) = 0.25(4 + 0.4(1)) + 0.75(3 + 0.4(1))$$
$$= 0.25(4.4) + 0.75(3.4)$$
$$= 1.1 + 2.55 = \mathbf{3.65}$$

Update State B:

$$V(B) = 0.6(5 + 0.4(1)) + 0.2(6 + 0.4(1)) + 0.2(8 + 0.4(0))$$

Note: $V(Term) = 0$.

$$= 0.6(5.4) + 0.2(6.4) + 0.2(8.0)$$
$$= 3.24 + 1.28 + 1.6 = \mathbf{6.12}$$

Exam Problem: EC3 Regular 2025 - Warehouse Robot

States S, T, E. S → T. From T, action "Move E":

- 80% go to E (Reward +5)
- 20% loop to T (Reward -1)

$\gamma = 0.9$. $V(S) = 0, V(T) = 2, V(E) = 5$. (a) Write Bellman Expectation Eq. (b) Perform 2 updates.

⇒ **Professor's Detailed Solution:**

(a) Bellman Equation for T:

$$v(T) = 0.8[5 + 0.9v(E)] + 0.2[-1 + 0.9v(T)]$$

(b) Updates: *Update 1:* Use initial $V(T) = 2$.

$$V_1(T) = 0.8[5 + 4.5] + 0.2[-1 + 1.8]$$

$$= 0.8(9.5) + 0.2(0.8) = 7.6 + 0.16 = \mathbf{7.76}$$

Update 2: Use $V_1(T) = 7.76$.

$$V_2(T) = 0.8[9.5] + 0.2[-1 + 0.9(7.76)]$$

$$= 7.6 + 0.2[-1 + 6.984]$$

$$= 7.6 + 0.2[5.984] = 7.6 + 1.1968 = \mathbf{8.7968}$$

Exam Problem: Regular 2024 - Maze Concepts

Robot navigating maze. Reward -1 per step, +100 goal. (a) How does it learn optimal path? (b) Impact of $\gamma = 0.9$ vs 0.99? (c) How to encourage exploration?

⇒ Professor's Detailed Solution:

(a) Learning: The agent learns via value propagation. The +100 at the goal propagates backwards. The state next to goal gets $100 - 1 = 99$. The one before that gets 98. This creates a "value gradient" or slope. The optimal policy is to climb this slope (greedy ascent) to the goal.

(b) Impact of γ:

- $\gamma = 0.9$: Agent is short-sighted. The "pull" of the +100 reward fades quickly with distance ($100 \times 0.9^{10} \approx 34$). It might prefer avoiding a -10 trap locally than reaching the distant +100.
- $\gamma = 0.99$: Agent is far-sighted. The +100 is visible from far away. It will tolerate long paths if they guarantee the goal.

(c) Encouraging Exploration: 1. **Optimistic Initial Values:** Set all $Q(s, a) = +1000$. The agent will explore everything to check if it's actually that good. 2. **Increase ϵ:** Force more random steps. 3. **Entropy Bonus:** Reward the agent for taking actions with uncertain outcomes.

---

1 Module 3: Monte Carlo Methods

1.1 3.1 Learning from Experience

In Module 2 (MDPs), we assumed we had a perfect model of the environment ($p(s'|s,a)$ and $r(s,a)$). We could just "calculate" the optimal policy using Dynamic Programming (Value Iteration). **The Reality:** In most real-world problems (like riding a bike, playing Blackjack, or the stock market), we don't know the physics equations. We only have **raw experience**.

Monte Carlo (MC) Concept

MC methods solve the reinforcement learning problem by averaging sample returns.

- **No Model Required:** We do not need to know transitions P. We just need sample sequences: $S_0, A_0, R_1, S_1, A_1, R_2 \dots S_T, R_T$.
- **Episodic Requirement:** MC methods are defined only for **episodic tasks**. The episode must terminate before values can be updated (because we need the total return G_t).
- **Core Idea:** To estimate $v_\pi(s)$, we simply play many games. Every time we visit state s, we look at the total reward obtained *from that point until the end.* The average of these returns converges to the true value.

$$V(s) \approx \text{average}(G_t \text{ following visits to } s)$$

1.2 3.2 First-Visit vs. Every-Visit MC

An episode might visit the same state s multiple times (e.g., looping in a maze: A $\rightarrow$ B $\rightarrow$ A $\rightarrow$ Term). How do we calculate the average?

- **First-Visit MC:** We estimate $v_\pi(s)$ as the average of the returns following the **first** time s is visited in the episode.
  - *Properties:* The returns are independent and identically distributed (i.i.d) across episodes. It provides an **unbiased** estimate.
  - *Use Case:* Standard definition, theoretically simpler.
- **Every-Visit MC:** We average the returns following **all** visits to s in the episode.
  - *Properties:* Estimates converge to the same value, but handling the data is slightly different. It is **biased** initially but consistent.
  - *Use Case:* More data efficient (uses every piece of data from the episode).

1.3 3.3 Monte Carlo Control

How do we find the optimal policy without a model? We use **Generalized Policy Iteration (GPI)**. 1. **Evaluation:** Estimate $q_\pi(s,a)$ by playing episodes. Note: We must estimate q (Action-Value), not just v (State-Value), because without a model, knowing $v(s)$ doesn't tell us which action leads to the best next state. 2. **Improvement:** Make the policy greedy with respect to the current q estimates. $\pi(s) = \arg\max_a q(s,a)$.

Mathematical Deep Dive: The Exploration Problem & Exploring Starts

A major problem in MC Control is exploration. If the policy becomes deterministic (greedy) too quickly, it may never visit certain state-action pairs (s, a), and thus never learn their true value. **The "Exploring Starts" Assumption:** We can specify that every episode starts in a state-action pair chosen randomly, covering all possibilities. This guarantees infinite exploration. *Real-world limitation:* You can't always start a self-driving car in the middle of a crash to learn how to recover.

1.4 3.4 On-Policy vs. Off-Policy Learning

What if we can't use Exploring Starts? We need a way to ensure exploration while trying to learn the optimal policy.

- **On-Policy:** The agent learns about the policy it is currently using.
  - *Example:* ϵ-soft policies. The agent generally picks the best action but randomly explores with probability ϵ. The policy being learned is "near-optimal" but still explores.
- **Off-Policy:** The agent learns about one policy (**Target Policy** π) while following a different one (**Behavior Policy** b).
  - *Benefit:* The agent can learn the optimal greedy policy (π) while behaving randomly (b) to ensure exploration.
  - *Mechanism:* **Importance Sampling**. Because the data came from b, the returns are "wrong" for π. We weight returns by the ratio of probabilities of the trajectory occurring under π vs. b.

    $$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

    If π would never take an action that b took, the ratio becomes 0, and we ignore the return.

1.5 Solved Exam Problems: Monte Carlo

Exam Problem: Makeup 2023 - Return Estimation

Scenario: We observe 4 episodes in an environment.

- E1: $x, \text{left}, 16, x, \text{right}, 12, x, \text{left}, 16, T, 100$
- E2: $x, \text{left}, 16, x, \text{left}, 16, x, \text{right}, 12, T, 200$
- E3: $x, \text{right}, 15, x, \text{right}, 11, x, \text{left}, 13, T, 150$
- E4: $x, \text{right}, 12, x, \text{left}, 16, x, \text{left}, 16, x, \text{left}, 16, x, 110$

Task: Estimate $V(x)$ using First-Visit Monte Carlo. (Assume $\gamma = 1$).

⇒ Professor's Detailed Solution:

Methodology: For First-Visit MC, we identify the **first occurrence** of state x in each episode and sum all rewards from that point until the episode terminates.

Episode 1: Sequence: $x \xrightarrow{16} x \xrightarrow{12} x \xrightarrow{16} T(+100)$ First visit to x is at the very start. Return $G_1 = 16 + 12 + 16 + 100 = \mathbf{144}$.

Episode 2: Sequence: $x \xrightarrow{16} x \xrightarrow{16} x \xrightarrow{12} T(+200)$ First visit to x is at start. Return $G_2 = 16 + 16 + 12 + 200 = \mathbf{244}$.

Episode 3: Sequence: $x \xrightarrow{15} x \xrightarrow{11} x \xrightarrow{13} T(+150)$ First visit to x is at start. Return $G_3 = 15 + 11 + 13 + 150 = \mathbf{189}$.

Episode 4: Sequence: $x \xrightarrow{12} x \xrightarrow{16} x \xrightarrow{16} x \xrightarrow{16} x(\text{Terminal } 110)$ First visit to x is at start. Return $G_4 = 12 + 16 + 16 + 16 + 110 = \mathbf{170}$.

Final Calculation:

$$V(x) = \frac{144 + 244 + 189 + 170}{4} = \frac{747}{4} = \mathbf{186.75}$$

Exam Problem: EveryVisit PDF - First Visit Logic

Episode Trace: $A \xrightarrow{+3} A \xrightarrow{+2} B \xrightarrow{-4} A \xrightarrow{+4} B \xrightarrow{-3} T$. **Task:** Calculate First-Visit Returns for states A and B.

⇒ Professor's Detailed Solution:

State A: The episode visits A at indices $t = 0, 1, 3$. First-Visit MC only cares about $t = 0$. Sum of all future rewards:

$$G(A) = 3 + 2 + (-4) + 4 + (-3) = \mathbf{2}$$

State B: The episode visits B at indices $t = 2, 4$. First-Visit MC cares about the first occurrence at $t = 2$. Sum of rewards from $t = 2$ onwards:

$$G(B) = (-4) + 4 + (-3) = \mathbf{-3}$$

Exam Problem: EC2 Regular 2025 - Off-Policy Estimation

Scenario: Behavior Policy b: Chooses {Trending: 0.3, Recent: 0.7}. Target Policy π: Chooses {Trending: 1.0, Recent: 0.0}. Episode: Trending (+3) $\rightarrow$ Recent (+4) $\rightarrow$ Trending (+5) $\rightarrow$ Recent (+7). **Task:** Estimate Value of "Recent" using First-Visit MC.

$\Rightarrow$ Professor's Detailed Solution:

Professor's Insight: This is a trick question involving Off-Policy learning. We are asked to estimate the value of the state "Recent" under Target Policy π. Policy π is deterministic: it **always** chooses "Trending" and **never** chooses "Recent".

Look at the episode: The agent took action "Recent" (or transitioned to "Recent") at step 2. Under the target policy π, the probability of choosing "Recent" is 0.0. Therefore, this specific sequence of events is **impossible** under the target policy.

Mathematically, the Importance Sampling Ratio ρ:

$$\rho = \frac{\pi(\text{Recent})}{b(\text{Recent})} = \frac{0.0}{0.7} = 0$$

Since the weight is 0, this return contributes **nothing** to the estimate. The value cannot be updated using this episode.

---

2 Module 4: Temporal Difference (TD) Learning

2.1 4.1 The Best of Both Worlds

TD Learning is the central novelty of reinforcement learning. It combines the benefits of Monte Carlo (learning from experience) and Dynamic Programming (bootstrapping).

TD(0) Prediction: The "Update Now" Method

- **Monte Carlo:** Must wait until the end of the episode to update $V(S_t)$. It uses the actual return G_t.
$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$
- **Dynamic Programming:** Uses the model to look ahead.
- **TD Learning:** Waits only one step! It uses the observed reward R_{t+1} and the *estimate* of the next state $V(S_{t+1})$ as a proxy for the full return.
$$V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD Target}} - V(S_t)]$$

This is called **Bootstrapping**: updating a guess towards another guess.

2.2 4.2 Advantages of TD vs MC

1. **No Model Required:** Like MC, it learns from raw experience.
2. **Online/Incremental:** Unlike MC, it doesn't need to wait for the episode to end. This is crucial for very long or **continuing tasks**.
3. **Variance:** TD usually has **lower variance** than MC because it depends on only one random action/transition, rather than a full sequence of potentially hundreds of random actions in an episode.

Driving Home Analogy

Monte Carlo: You drive home. You only update your travel time estimate once you actually arrive at your driveway. "Oh, that took 40 mins." **TD Learning:** You drive. You hit a traffic jam 5 mins in. You immediately update your estimate: "This is going to take longer than I thought," without waiting to get home.

2.3 4.3 TD Control: SARSA vs. Q-Learning

When we move to control (finding the best policy), we learn Action-Values $Q(s, a)$.

2.3.1 A. SARSA (On-Policy)

State, **A**ction, **R**eward, **S**tate', **A**ction'.

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

Here, A' is the action the agent *actually took* in the next state (following its current policy, including exploration). **Characteristic:** SARSA learns the value of the "risky" policy. If the agent explores near a cliff, SARSA will learn that the cliff edge is dangerous because exploration might cause a fall.

2.3.2 B. Q-Learning (Off-Policy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

Here, we update using the *best possible* action in the next state, regardless of what the agent actually did. **Characteristic:** Q-Learning learns the optimal value function directly. It assumes "I will act optimally in the future," ignoring the fact that it is currently exploring.

2.4 4.4 Maximization Bias

Q-Learning uses the 'max' operator. If our Q estimates have random noise, taking the max over them creates a positive bias (we pick the artificially high ones). **Mathematical Flaw:** $E[\max(X_1, X_2)] \geq \max(E[X_1], E[X_2])$. **Solution:** **Double Q-Learning**. Use two networks/tables. Use one to determine the best action and the other to estimate its value.

2.5 Solved Exam Problems: TD Learning

Exam Problem: Regular 2024 - Navigation Strategy

Scenario: A robot learns to navigate a maze. **Task:** Give a real-world example contrasting On-policy (SARSA) and Off-policy (Q-learning) in navigation. Explain.

⇒ **Professor's Detailed Solution:**

Example: The Cliff Walking Task Imagine a path that runs along the edge of a cliff.

- **Path A (Optimal):** Walks right on the edge. Shortest distance.
- **Path B (Safe):** Walks one square inland. Slightly longer.

The agent uses ϵ-greedy exploration (10% random moves).

Q-Learning (Off-Policy): It updates using $\max Q(S', a)$. It assumes "In the future, I will choose the optimal move (walk on edge)." It effectively ignores the 10% exploration risk. **Result:** It learns the **Optimal Path (A)** along the edge. **Real-world Consequence:** During training, the robot will frequently fall off the cliff because it is still exploring randomly while walking on the edge.

SARSA (On-Policy): It updates using $Q(S', A')$, where A' is the *actual* next move (which might be a random jump off the cliff). It incorporates the risk of the policy into the value. **Result:** It learns the **Safe Path (B)**. It realizes that walking on the edge is "bad" because its own exploration makes it dangerous.

Exam Problem: Regular 2023 - RL Categorization

Task: Create a table contrasting (i) Model-Based vs Free, (ii) Value vs Policy Based, (iii) On vs Off Policy.

⇒ **Professor's Detailed Solution:**

Category	Type A	Type B
Model Availability	**Model-Based:** Uses a model (P, R) to plan (e.g., Dynamic Programming, AlphaGo search). Efficient but requires knowing physics.	**Model-Free:** Learns directly from raw experience (e.g., Q-Learning). Requires more data but works in unknown worlds.
Learning Target	**Value-Based:** Learns value function $Q(s, a)$ and derives policy by $\arg\max$. Good for discrete actions.	**Policy-Based:** Learns the policy $\pi(a\|s)$ parameters directly. Good for continuous actions and stochastic policies.
Data Source	**On-Policy:** Learns the value of the policy being executed (e.g., SARSA). Safe but data-inefficient.	**Off-Policy:** Learns optimal policy while executing a different exploration policy (e.g., Q-Learning). Data efficient (can reuse old data).

---

1 Module 5: n-step Bootstrapping and Planning

1.1 5.1 Unifying MC and TD: The n-step Return

Until now, we viewed TD(0) and Monte Carlo as separate entities. In reality, they are two endpoints of a single spectrum.

The Spectrum of Prediction

Imagine a backup diagram.

- **1-step TD:** Looks ahead exactly one reward and bootstraps.
- **n-step TD:** Looks ahead n rewards and then bootstraps.
- **∞-step (MC):** Looks ahead until the terminal state.

The "sweet spot" for n is often between 3 and 10. This reduces the **bias** of a single guess (bootstrapping) while avoiding the high **variance** of a full episode (Monte Carlo).

Mathematical Deep Dive: The n-step Return $G_{t:t+n}$

We define the target for an n-step update as:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

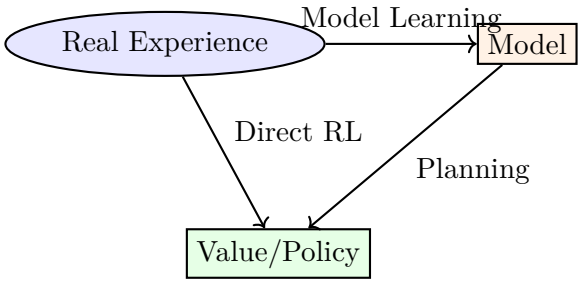
The error we minimize is the difference between our current estimate and this truncated return:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_{t:t+n} - V(S_t) \right]$$

Note that an update for time t can only be performed at time $t+n$, once R_{t+n} and S_{t+n} are observed.

1.2 5.2 Planning: Dyna-Q and Mental Rehearsal

In RL, a **Model** is anything the agent can use to predict how the environment will respond. **Dyna-Q** is a framework that combines **direct RL** (learning from real experience) and **planning** (learning from simulated experience).



Prioritized Sweeping

In a large state space, planning randomly is inefficient. **Prioritized Sweeping** focuses planning on states that are likely to change their values significantly. **The Logic:** When a state's value is updated, we calculate the potential change in the values of its *predecessor* states. If the potential change is above a threshold θ, we add those predecessors to a priority queue.

2 Module 6: Value Function Approximation (DQN)

2.1 6.1 The Transition to Deep RL

Tabular methods fail when the state space is continuous or astronomically large. We need **Generalization**. If we learn that a specific pixel configuration in a video game is dangerous, we should generalize that similar configurations are also dangerous.

We replace the table with a parameterizable function (Neural Network): $Q(s, a; \theta) \approx q_*(s, a)$.

2.2 6.2 The Deadly Triad

Reinforcement Learning is notoriously difficult to combine with Deep Learning. Convergence is only guaranteed if you avoid at least one of the following:

1. **Function Approximation:** Using a non-linear model like a Neural Network.
2. **Bootstrapping:** Updating guesses based on other guesses (TD learning).
3. **Off-policy Training:** Learning from data generated by a different policy.

DQN uses all three, making it fundamentally unstable. The "tricks" of DQN are designed to break this instability.

2.3 6.3 The DQN Architecture and Stability Tricks

Tricking the Triad: Experience Replay & Target Networks

1. Experience Replay ($\mathcal{D}$): In RL, consecutive samples (s_t, a_t, r_t, s_{t+1}) are highly correlated. Stochastic Gradient Descent (SGD) requires independent and identically distributed (i.i.d.) data. *The Fix:* Store transitions in a large buffer. Sample a **random minibatch** for each training step. This "breaks" the temporal correlation.
2. Target Networks (θ^-): If we update $Q(s, a)$ towards a target that also uses the same Q network, the target moves as we learn. This is "chasing your own tail." *The Fix:* Use a separate "Target Network" with weights θ^- to calculate the TD-target. Keep these weights frozen and update them only every C steps ($\theta^- \leftarrow \theta$).

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(\underbrace{R + \gamma \max_{a'} Q(S', a'; \theta^-)}_{\text{Fixed Target}} - Q(S, A; \theta)\right)^2\right]$$

2.4 6.4 Beyond Vanilla DQN

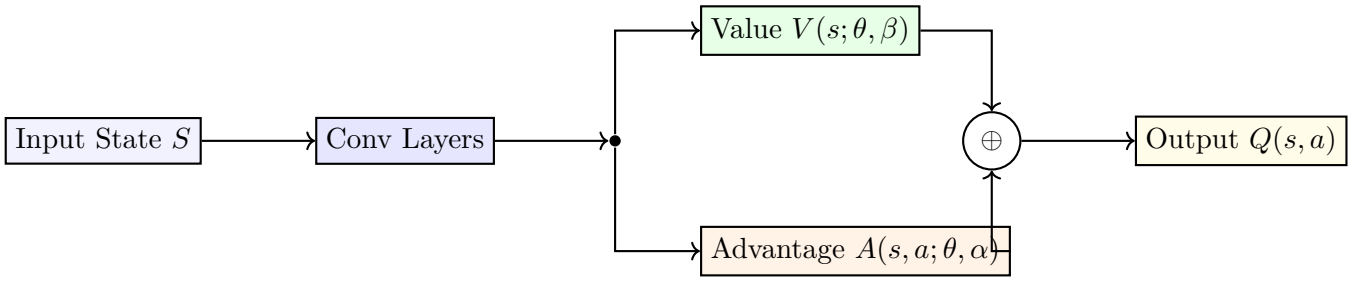
2.4.1 A. Double DQN (DDQN): Fixing Overestimation

Standard Q-learning overestimates values because the max operator is sensitive to noise. *Mathematical Proof:* $\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$. **DDQN Solution:** Use the **Online Network** to choose the best action, but use the **Target Network** to evaluate its value.

$$Y_t^{DDQN} = R_{t+1} + \gamma Q(S_{t+1}, \underbrace{\text{argmax}_a Q(S_{t+1}, a; \theta_t)}_{\text{Selection by Online}}; \theta_t^-)$$

2.4.2 B. Dueling DQN: Advantage vs. Value

In many states, the choice of action doesn't actually matter. Dueling DQN explicitly separates the value of the *state* from the *advantage* of each action.



Mathematical Deep Dive: The Aggregating Layer

Simply adding V and A leads to unidentifiability (you can't uniquely recover V and A from Q). To fix this, we force the advantage of the chosen action to be relative to the mean:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\theta,\alpha) \right)$$

2.5 Solved Exam Problems: Deep RL Focus

Exam Problem: Regular 2023 - DQN Engineering

(c) Why does AlphaGo use supervised learning to learn the initial policy? (d) How does DQN handle the challenges of non-i.i.d data and non-stationary targets?

⇒ **Professor's Detailed Solution:**

(c) Cold Start Problem in Games: RL requires exploration to find rewards. In a game like Go, with 10^{170} states, random exploration will almost never lead to a win, meaning the agent gets zero feedback for millions of steps. **Supervised Learning (Behavior Cloning)** on 30 million human expert moves provides a "policy prior." It teaches the network to "look like a human player" first. This significantly narrows the search space, allowing RL to refine a competent player into a superhuman one.

(d) Technical Stabilization:

1. **Non-i.i.d. Data:** Sequential RL transitions are temporally correlated. **DQN Fix:** **Experience Replay Buffer**. By sampling random batches of history, we mimic i.i.d. conditions, preventing the network from overfitting to the most recent sequence of frames.
2. **Non-Stationary Targets:** The target for the MSE loss depends on the network weights being optimized, creating an unstable feedback loop. **DQN Fix:** **Target Network**. By lagging the target weights (θ^-) behind the online weights (θ), we provide a stationary objective for a fixed interval, allowing the optimization to converge.

Exam Problem: Makeup 2023 - Overestimation math

Explain why standard Q-learning is biased and how Double DQN fixes it using two networks.

⇒ **Professor's Detailed Solution:**

The Bias: Standard Q-learning uses $\max_a Q(s', a)$. If the Q-estimates are noisy (which they always are early in training), the max operator will consistently pick the "luckiest" high value rather than the "truest" high value. This leads to massive overestimation of state values.

The Fix: Double DQN uses the **Online Network** to select the action a^* that maximizes Q, but then asks the **Target Network** "How much is a^* actually worth?".

$$a^* = \text{argmax}_a Q(s', a; \theta) \quad \text{then} \quad Target = R + \gamma Q(s', a^*; \theta^-)$$

Because the noise in θ and θ^- is largely uncorrelated, the Target Network provides an unbiased evaluation of the action selected by the Online Network.

Exam Problem: Regular 2024 - Atari Preprocessing

Why do we stack 4 frames as input to a DQN for Atari games?

⇒ **Professor's Detailed Solution:**

Handling the POMDP (Partially Observable MDP): A single static frame of a game (like Breakout or Pong) does not provide information about the **velocity** or **direction** of moving objects (e.g., the ball). By stacking the 4 most recent frames into a single input tensor, the Convolutional Neural Network can infer temporal features (motion, acceleration). This converts a partially observable state into a fully Markovian state, where the current input contains all information needed to decide the next action.

---

1 Module 7: Policy Gradient Methods

1.1 7.1 Introduction: Why Direct Policy Optimization?

In the previous modules, we focused on **Value-Based Methods** (DQN, SARSA). We learned $Q(s, a)$ and selected actions via ϵ-greedy. Policy Gradient (PG) methods represent the policy $\pi(a|s; \theta)$ directly using a parameterized function (usually a Neural Network) and optimize the weights θ to maximize expected return.

Advantages of Policy Gradients over Value-Based Methods

1. **Continuous Action Spaces:** In robotics, actions are often continuous (e.g., torques). Value-based methods require a max over actions, which is impossible for infinite continuous choices. PG simply outputs the parameters of a distribution (e.g., mean and variance of a Gaussian).
2. **Stochastic Policies:** Value-based methods usually converge to a deterministic greedy policy. However, in partially observable environments or adversarial games (like Rock-Paper-Scissors), the optimal policy is stochastic. PG can learn the exact probabilities.
3. **Better Convergence Properties:** Value-based methods suffer from oscillations because a tiny change in $Q(s, a)$ can cause a discrete, massive flip in the greedy action. PG changes action probabilities smoothly, leading to more stable training.

1.2 7.2 The Policy Gradient Theorem (The Core Math)

We define the performance measure $J(\theta)$ as the expected return.

$$J(\theta) = \mathbb{E}_\pi[G_t]$$

The challenge is that $J(\theta)$ depends on both the actions (which we control) and the state distribution (which depends on the environment's unknown transition physics).

Mathematical Deep Dive: Detailed Derivation of the Policy Gradient Theorem

The gradient of the performance measure is:

$$\nabla_\theta J(\theta) \propto \sum_s d^\pi(s) \sum_a \nabla_\theta \pi(a|s, \theta) Q^\pi(s, a)$$

Using the **Log-Derivative Trick**: Since $\nabla_\theta \ln f(x) = \frac{\nabla_\theta f(x)}{f(x)}$, we can rewrite $\nabla_\theta \pi$ as:

$$\nabla_\theta \pi(a|s, \theta) = \pi(a|s, \theta) \frac{\nabla_\theta \pi(a|s, \theta)}{\pi(a|s, \theta)} = \pi(a|s, \theta) \nabla_\theta \ln \pi(a|s, \theta)$$

Substituting this back:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \ln \pi(a|s, \theta) Q^\pi(s, a)]$$

This is the "Log-Probability" update. It means: "Increase the log-prob of an action, weighted by how good that action was (Q)."

1.3 7.3 REINFORCE: The Monte Carlo Policy Gradient

REINFORCE is the most basic implementation of PG. It uses the actual return from an episode (G_t) as an unbiased sample estimate of $Q^\pi(s, a)$.

REINFORCE Algorithm Steps

1. Initialize policy weights θ randomly.
2. Generate an entire episode using $\pi(\cdot|\cdot;\theta)$: $S_0, A_0, R_1, \ldots, S_T$.
3. For each step t in the episode:
- Calculate the return $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$.
- Update $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi(A_t|S_t, \theta)$.

The High Variance Problem

REINFORCE relies on G_t, which is a single sample of an entire episode's rewards. Because an episode involves many random actions and transitions, G_t can vary wildly from one run to another. This high variance makes the gradient noisy and learning slow.

1.4 7.4 Variance Reduction: Baselines

To reduce variance without introducing bias, we subtract a **Baseline** $b(s)$ from the return.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[(G_t - b(S_t))\nabla_\theta \ln \pi(A_t|S_t, \theta)]$$

A common choice for $b(s)$ is the state-value estimate $\hat{V}(s, w)$. **Logic:** If you get a return of +100, but the state was already expected to give +90, the "surprise" (Advantage) is only +10. We only update based on the surprise.

1.5 7.5 Actor-Critic Methods (A2C / A3C)

Combining Value-Based and Policy-Based methods leads to **Actor-Critic**.

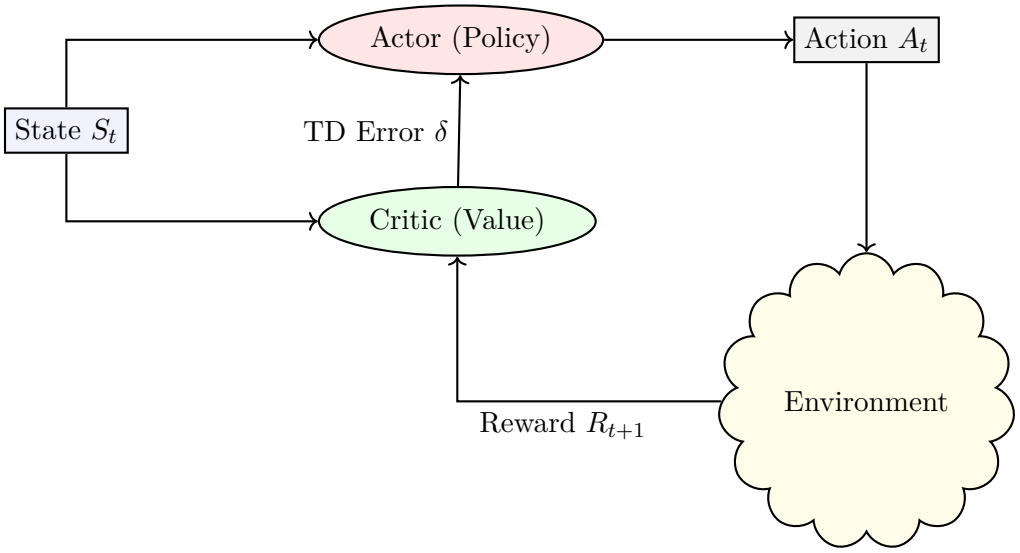
- **The Actor:** The policy $\pi(a|s, \theta)$. It decides which action to take.
- **The Critic:** The value function $\hat{V}(s, w)$. It evaluates the actor's action.

The Advantage Function

We replace the return G_t with the **Advantage** $A(s, a) = Q(s, a) - V(s)$. In a one-step Actor-Critic, we approximate the advantage using the **TD Error** δ:

$$\delta_t = R_{t+1} + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(S_t, w)$$

Actor Update: $\theta \leftarrow \theta + \alpha_\theta \delta_t \nabla_\theta \ln \pi(A_t|S_t, \theta)$
Critic Update: $w \leftarrow w + \alpha_w \delta_t \nabla_w \hat{V}(S_t, w)$



1.6 7.6 Policy Parameterizations

How do we actually define $\pi(a|s, \theta)$?

1. **Discrete Actions:** Softmax Policy.

$$\pi(a|s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum e^{h(s,b,\theta)}}$$

2. **Continuous Actions:** Gaussian Policy. The network outputs two values: $\mu(s, \theta)$ (mean) and $\sigma(s, \theta)$ (standard deviation). The action is sampled from $\mathcal{N}(\mu, \sigma^2)$.

2 Module 8: Advanced Planning and Learning

2.1 8.1 Planning as Lookahead

Planning is the process of using a **Model** to improve a policy. In complex games like Go or Chess, the state space is too large for simple tabular planning. We use **heuristic search**.

2.2 8.2 Monte Carlo Tree Search (MCTS)

MCTS is a rollout-based search algorithm that builds a tree of possible future states. It does not require a full model of transition probabilities—only a "simulator" that can give the next state given an action.

The Four Stages of MCTS

1. **Selection:** Start at root. Use a "Tree Policy" to move down to a leaf node. The most common tree policy is **UCT** (Upper Confidence Bound for Trees):

$$a^* = \arg\max_a \left[\frac{W_i}{N_i} + c\sqrt{\frac{\ln N_{parent}}{N_i}} \right]$$

   This balances high win-rate nodes (W/N) with unexplored nodes (the square root term).

2. **Expansion:** Once a leaf is reached, if it's not terminal, add one or more child nodes.

3. **Simulation (Rollout):** From the new node, play a "fast" random game until the end using a simple "Default Policy."

4. **Backpropagation:** Take the result of the simulation (Win/Loss) and propagate it back up the path to the root, updating W and N for every node.

2.3 8.3 AlphaGo: The Mastermind Architecture

AlphaGo (and its successors AlphaZero) proved that RL could beat the best human players. It combined SL, RL, and MCTS.

The AlphaGo Pipeline

1. **SL Policy Network (P_σ):** Trained via Supervised Learning on 30 million human expert moves. Accuracy $\approx 57\%$.

2. **RL Policy Network (P_ρ):** Initialized from P_σ. Refined through self-play (playing against versions of itself). It becomes significantly stronger than the SL network.

3. **Value Network (V_θ):** Trained to predict the outcome (who wins) of games played by the RL Policy Network. This provides a "position evaluator."

4. **Final MCTS:** During the actual match, MCTS uses the RL Policy to narrow the search (prioritize good moves) and the Value Network to truncate rollouts (evaluate positions without playing to the end).

2.4 8.4 Inverse Reinforcement Learning (IRL)

In standard RL, we have $R \rightarrow \pi^*$. In IRL, we have $\pi_{expert} \rightarrow R$. **The Problem:** Many reward functions can explain the same expert behavior. **Solution:** Maximum Entropy IRL (MaxEnt IRL). It finds the reward function that makes the expert behavior most likely while being the "least committed" (most random) otherwise.

2.5 Solved Exam Problems: Advanced Synthesis

Exam Problem: Regular 2024 - Actor-Critic Variance

Explain, using the concept of 'Advantage', why Actor-Critic reduces variance compared to REINFORCE.

⇒ Professor's Detailed Solution:

REINFORCE uses the full return G_t, which includes the randomness of all actions and transitions from time t to T. This "total randomness" causes high variance. Actor-Critic replaces G_t with the **Advantage** $A(s,a) = Q(s,a) - V(s)$. In its simplest form, this is the TD Error $\delta = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$. **Mechanism:** The TD error only depends on one random transition (S_{t+1}) and one reward (R_{t+1}), relying on the critic's estimate for the rest. By "bootstrapping" via the critic, we effectively average out the future randomness, leading to a much more stable gradient update.

Exam Problem: Makeup 2023 - MCTS Mechanism

In MCTS, explain the difference between the 'Tree Policy' and the 'Default Policy'.

⇒ Professor's Detailed Solution:

- **Tree Policy:** Used during the **Selection** phase to navigate from the root to a leaf. It must balance exploration and exploitation (e.g., UCT). It acts on the part of the state space where we have some statistical data ($N > 0$).
- **Default Policy:** Used during the **Simulation** (Rollout) phase. Once we hit a leaf, we need to quickly estimate its value. We play a game to completion using a fast, often random or simple heuristic policy. This policy does not build a tree; it just provides a sample outcome.

Exam Problem: EC2 2025 - Inverse RL Scenarios

Give a real-world scenario where Inverse RL is more appropriate than standard RL.

⇒ Professor's Detailed Solution:

Autonomous Driving in Urban Traffic. In standard RL, defining a reward function for "good driving" is nearly impossible. If you reward speed, the car crashes. If you reward safety, the car never moves. If you penalize lane departures, it can't avoid obstacles. **IRL Approach:** We record 1,000 hours of expert human driving. IRL infers the reward function that human drivers are implicitly optimizing (a complex balance of safety, smoothness, and speed). We then use that learned reward to train the autonomous agent.

Exam Problem: EveryVisit PDF - AlphaGo Zero

How did AlphaGo Zero differ from the original AlphaGo in terms of training data?

⇒ Professor's Detailed Solution:

The original AlphaGo (Fan/Lee) used **Human Expert Data** for initial Supervised Learning. **AlphaGo Zero** removed this requirement entirely. It started from completely random play (tabula rasa) and used only **Self-Play Reinforcement Learning**. It demonstrated that human data can actually be a "ceiling," and by learning purely from the logic of the game, the agent could discover strategies that humans had never considered in 3,000 years.
