EASYDEEPLEARN

Reinforcement Learning

Interview cheat sheet

214 questions · MDPs, exploration, policy vs value methods, and modern actor-critic algorithms.

#theory #deep-rl #llm #alignment #policy-methods #engineering #exploration #actor-critic #value-methods #applications

Fundamentals & MDPs

Define the RL setting: agent, environment, state, action, reward, policy.

An agent observes a state s from the environment, picks an action a via its policy $\pi(a \mid s)$, receives a reward r and a new state s'. The goal is to maximize the expected cumulative discounted reward (return).

On-policy vs off-policy learning — what's the difference?

On-policy methods learn from data generated by the current policy (e.g., SARSA, REINFORCE, PPO). Off-policy methods learn from data collected by a different (behavior) policy — this enables replay buffers and reusing old data (e.g., Q-learning, DQN, SAC, DDPG).

What does the Bellman equation say?

For a policy pi: $V_\pi(s) = E[r + \gamma \cdot V_\pi(s)]$. For the optimal value: $V \cdot (s) = \max_a$ E[r + gamma · V*(s')].

What role does the discount factor gamma play?

Gamma in [0, 1) weights how much future rewards matter compared to immediate ones. Gamma close to 0 = myopic (only care about immediate reward); gamma close to 1 = far-sighted (care about long-term).

What is the credit assignment problem?

In sequential tasks with delayed rewards, it's hard to know which past action caused a given reward. Long horizons, sparse rewards, and stochasticity make it worse.

Model-based vs model-free RL — when do you prefer each?

Model-based RL learns a model of the environment (transitions and rewards) and uses it to plan (e.g., MuZero, Dreamer). It is more sample-efficient — great when real interactions are expensive (robotics, healthcare).

What is the Markov property and why does it matter?

$P(s_{t+1} \mid s_t, a_t, s_{t-1}, , s_0) = P(s_{t+1} \mid s_t, a_t)$ — the future depends only on the current state, not the history. Enables efficient RL: policy and value functions depend on state alone, not history.

POMDP — how does it differ from MDP?

Partially Observable MDP: agent observes o_t which gives incomplete info about state s_t. Add observation function $O(o \mid s)$.

V(s) vs Q(s, a) — the difference.

V(s) = expected return from state s following policy π. Q(s, a) = expected return from state s taking action a THEN following π.

What is the advantage function?

A(s, a) = Q(s, a) - V(s) → how much better is action a than the average action in state s. Reduces variance in policy-gradient estimates (baseline subtraction preserves the gradient in expectation but shrinks variance).

Different notions of return — MC, TD(0), n-step, GAE.

MC return: $G_t = \Sigma \gamma^k r_{t+k}$ — full episode, unbiased, high variance. TD(0): $r_t + \gamma V(s_{t+1})$ — biased (bootstrap), low variance. n-step: $r_t + \gamma r_{t+1}$ + ... + $\gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n})$ — knob n interpolates.

What is TD error?

$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ — difference between the bootstrapped target and current estimate. Drives all TD-based updates: V ← V + α δ, Q-learning target is δ with max in bootstrap.

Policy iteration vs value iteration.

Policy iteration: alternate (a) policy evaluation $(\text{compute } V\pi \text{ by solving Bellman until convergence})$, (b) policy improvement $(\pi \leftarrow \text{greedy on } V\pi)$. Converges in finite steps.

TD(λ) — how does it work?

Interpolates between TD(0) (λ=0, low variance biased) and MC (λ=1, unbiased high variance). Uses eligibility traces $e_t(s)$ that decay by γλ each step and increment at visited states.

Monte Carlo methods in RL — when to use?

Estimate V_π(s) or Q_π(s, a) via sample-average returns from full episodes. Unbiased (no bootstrap error), but only usable in episodic tasks + high variance.

Tabular vs function-approximation RL — the challenge.

Tabular: one entry per (s, a) — convergence guaranteed but doesn't scale. Function approximation (neural nets): scales to huge state spaces but breaks convergence guarantees.

Why do Bellman operators converge?

Bellman operator T is a γ-contraction in the max norm: $\|\mathrm{TV}_1 - \mathrm{TV}_2\|_\infty \le \gamma \|V_1 - V_2\|_\infty$. By Banach fixed-point theorem, iterating T converges to unique fixed point V* at rate γ^k.

TD vs MC — how they differ in bias-variance.

MC: use full episodic return G_t → unbiased estimator of V_π, but high variance (many random rewards over full trajectory). TD(0): use $r_t + \gamma$ V(s') → biased (bootstrap error from imperfect V estimate) but lower variance (only one reward). n-step and TD(λ) interpolate.

Linear function approximation for value functions — pros and cons.

V(s) ≈ w'φ(s) with hand-crafted features φ. Convergence guarantees under on-policy (LSPI, LSTD) but poor scaling; requires domain-designed features.

When does tabular Q-learning provably converge?

(1) All (s, a) visited infinitely often (adequate exploration), (2) Learning rate α satisfies Robbins-Monro: Σα = ∞, $\Sigma\alpha^2 < \infty$, (3) Bounded rewards. Under these, $Q_k \to$ Q* almost surely.

When does tabular Q-learning fail in practice?

(1) State space too large: even discretized MazeGrid > 1M states makes tabular infeasible. (2) Continuous features: discretization fights curse of dimensionality.

Why is subtracting a baseline OK in policy gradient?

Because $E[b(s)\ \nabla \log\ \pi(a \mid s)]\ =\ 0$ for any function b(s) independent of a. This adds no bias but shrinks variance by centering the reward signal.

The log-derivative trick — why is it central to policy gradients?

For expectation of f under π_θ: ∇_θ E_π[f] = E_π[$f\ \nabla\theta\ \log\ \pi\theta$]. Enables Monte Carlo estimation of gradient using only samples from π and gradient of log-likelihood — no need to differentiate through the reward function or the environment.

What is natural gradient in policy gradient methods?

Follow the direction of steepest ascent in policy space measured by KL divergence (Fisher information metric) instead of Euclidean. Update: θ ← θ + α $F(\theta) - 1$ ∇J where F is Fisher information matrix.

Reparameterization trick in stochastic policies.

For continuous stochastic policy π_θ($a \mid s$) $=\ N(\mu\theta(s),\ \sigma\theta(s)^2)$, sample a = μ + σ ⊙ ε with ε ~ N(0, I) → gradient flows through μ, σ directly. Enables low-variance path-wise gradients rather than score-function REINFORCE-style gradients.

GAE — Generalized Advantage Estimation formula.

$A_t^{\mathrm{GAE}(\gamma,\ \lambda)}\ =\ \Sigma_{l=0}$^∞ $(\gamma\lambda)l\ \delta_{t+l}$ where $\delta_t\ =\ r_t\ +\ \gamma\ V(s_{t+1})\ -\ V(s_t)$. λ = 0 → TD(0) advantage (biased, low var). λ = 1 → MC advantage (unbiased, high var). Typical λ = 0.95 in PPO.

Off-policy policy gradient — how does importance sampling enter?

Data collected under behavior π_b, but want to update target π_θ: use importance weight ρ = π_θ($a \mid s$)/π_b($a \mid s$) → ∇J ≈ E_π_b[ρ ∇log π_θ · A]. Weight variance explodes when π_θ far from π_b.

V-trace off-policy correction formula.

$v_s\ =\ V(s_s)\ +\ \Sigma_t\ \gamma^{t-s}\ (\Pi\ \rho_i)\ \delta_t^V$ with $\delta_t^V\ =\ r_t\ +\ \gamma\ V(s_{t+1})\ -\ V(s_t)$, and $\rho_i\ =\ \min(\rho,\ \pi/\mu)$ clipped importance ratio. Different clipping thresholds for ρ (target) vs c (trace) allow bias-variance trade.

Retrace(λ) — safe off-policy Q updates.

Munos et al. 2016. Q-target: r + γ $\Sigma_l\ (\gamma\lambda)l\ \Pi_{i=1}^{l}\ c_i\ \delta_{t+l}$ with $c_i\ =\ \lambda\ \min(1,\ \pi/\pi_b)$.

Policy gradient as mirror descent — the connection.

Policy gradient can be viewed as mirror descent on the policy simplex with the KL divergence as Bregman divergence. Natural gradient = mirror descent step in log-policy space.

UCB — Upper Confidence Bound for exploration.

In bandits: pick a = argmax $[Q(a)\ +\ c\ \sqrt{(}\ln\ t\ /\ n_a)]$. Confidence bound term inflates rarely-tried actions.

Model bias in MBRL — the challenge.

Learned model has errors that compound over rollout → policy trained in imagined trajectories exploits model exploits. Solutions: (1) short rollouts (MBPO uses only 1-15 steps of imagination between real data), (2) ensemble of models for uncertainty (PETS: use variance to penalize plans), (3) careful data collection (COMBO, MOReL) that regularizes toward known regions, (4) uncertainty-aware planning (avoid uncertain trajectories).

LQR — Linear Quadratic Regulator and where it's still used.

For linear dynamics $x_{t+1}\ =\ A\ x_t\ +\ B\ u_t\ +$ noise and quadratic cost x'Qx + u'Ru, optimal policy is linear in state: u = -K x. Solve via discrete-time Riccati equation.

When does planning beat learning?

(1) Compact known dynamics (games, physics). (2) Long horizons where credit assignment is hard for learning.

Decision Transformer — how does it recast RL?

Chen et al. 2021: treat RL as sequence modeling. Given (return-to-go, state, action) sequence, predict next action autoregressively.

Trajectory Transformer — variant of Decision Transformer.

Janner et al. 2021: model entire trajectory $(s_0,\ a_0,\ r_0,\ s_1,\ a_1,\ r_1,\ )$ as sequence via GPT-style transformer. Planning via beam search over action tokens in learned model.

Nash equilibrium in multi-agent RL — what and when?

Policy profile $(\pi_1,\ ,\ \pi_n)$ where no agent can improve unilaterally. In zero-sum games (Rock-Paper-Scissors, Chess, Go, StarCraft): minmax = maxmin = Nash.

Fictitious play — what is it?

Iterative: each iteration, each player computes best response to the empirical distribution of opponents' past strategies. Converges to Nash in zero-sum games (Robinson 1951).

Options framework — formalization.

Sutton, Precup, Singh 1999. An option = (I, π, β): initiation set I (states where the option can start), intra-option policy π, termination condition β(s) (probability of stopping).

DPO derivation — the key mathematical insight.

The optimal policy under KL-constrained reward max is $\pi \cdot (y \mid x) \propto \pi_{\mathrm{ref}}(y \mid x)\ \exp(r(x,\ y)/\beta)$. Invert: r(x, y) = β $\log(\pi \cdot (y \mid x)/\pi_{\mathrm{ref}}(y \mid x))\ +\ \mathrm{const}$.

Meta-RL — learning to learn.

Train agent across a distribution of related tasks so it adapts to new tasks quickly. Approaches: (1) Recurrent meta-RL (RL^2): LSTM policy learns adaptation from context.

How does transfer learning work in RL?

Warm-start new policy from related-task policy (fine-tune) or pretrained representation. Techniques: (1) Progressive networks (Rusu et al.).

Successor features — what and why?

Barreto et al. 2017. Decompose reward as r = φ(s, a, s') · w.

Reward machines — structured reward specification.

Represent complex non-Markovian rewards as finite-state automata: state tracks task progress (e.g., 'first find A, then find B'). Enables decomposition + credit assignment across sub-goals.

Potential-based reward shaping — why is it safe?

Ng, Harada, Russell 1999. Add r'(s, a, s') = r(s, a, s') + γΦ(s') - Φ(s).

How do you handle partial observability in deep RL?

(1) State-stacking: last k observations as input (DQN frame stack). (2) RNN / LSTM policy: hidden state summarizes history (R2D2).

LQG — Linear Quadratic Gaussian control.

Extension of LQR to noisy partial observations. Optimal controller = Kalman filter (state estimation) + LQR (control).

Generalization in RL — why is it hard?

Deep RL overfits to training environments — Cobbe et al. 2019 showed Atari-trained agents fail on procedurally-generated variants. Root causes: (1) narrow training distribution.

Catastrophic forgetting in continual RL.

When switching between tasks or environments, RL agents forget previously learned skills. Root: SGD on new-task data overwrites old-task weights.

Generalist agents — Gato and DeepMind's approach.

Reed et al. 2022 (Gato): single transformer trained on 600+ tasks (Atari, robotics, chat, image captioning) as tokenized sequences. Uses cross-attention over task tokens.

Quantile regression in QR-DQN — the loss.

Predict N quantiles τ_i of return distribution. Loss on target y for quantile prediction Z_i at level τ_i: ρ_τ($y - Z_i$) where ρ_τ(u) = u(τ - I(u < 0)) (asymmetric absolute error).

Deadly triad — the three ingredients.

(1) Function approximation (neural nets, not tabular). (2) Bootstrapping (TD target uses V, not full MC).

Where is RL heading (2025+)?

Trends: (1) RL for reasoning: verifier-based training becomes standard for math / code / science (o1, R1 continue). (2) Agent RL: WebArena / SWE-bench become primary post-training targets.

How do you choose the discount factor, and what goes wrong at the extremes?

It sets the effective horizon, roughly one over one minus gamma steps, so pick it from how far ahead consequences actually matter in your problem rather than by convention. Too low and the agent is myopic: it ignores delayed reward entirely, which in a game looks like refusing to invest a move for a later gain.

What is the deadly triad and how do practical algorithms cope with it?

The combination of function approximation, bootstrapping, and off-policy learning. Each is fine alone, but together they can make value estimates diverge, because a bootstrapped target is computed from the same approximator being updated, and off-policy data means the states are not distributed according to the policy whose values you are fitting.

Why can't you just run Q-learning on a fixed dataset?

Because the maximization step queries actions the dataset never contains. The target takes a maximum over actions, and for unseen actions the network's value is an extrapolation with no data to correct it, so overestimation errors get selected precisely because they are overestimates.

Value methods (Q-learning, DQN)

What update does Q-learning perform?

Q(s, a) <- Q(s, a) + alpha · ($r + \gamma \cdot \max_a Q(s, a) - Q(s, a)$). This bootstraps toward the maximum future value at s', which is why it's off-policy: the target uses the greedy next action regardless of what the agent actually did.

What are the key tricks that make DQN work?

(1) Experience replay buffer: store transitions and sample randomly to break correlations and improve data efficiency. (2) Target network: use a delayed copy of Q for the TD target, updated slowly, to stabilize training.

Double DQN — what problem does it solve?

Vanilla DQN target: $\max_a Q_{\text{target}}(s, a)$. Same network selects AND evaluates max action → overestimation bias (noise favors overestimated actions).

Dueling DQN — architecture.

Split Q head into two streams: V(s) (state value) and A(s, a) (advantage), combined as $Q(s, a) = V(s) + A(s, a) - \text{mean}_a$ A(s, a) (identifiability constraint). Improves learning when many actions have similar values (V dominates); doesn't waste capacity redundantly encoding V per action.

Prioritized replay — mechanism.

Sample transitions with probability $p_i \propto |\delta_i|$^α (high TD error = more surprising = more informative). Correct sampling bias with importance weights $w_i = (1/N \cdot 1/P_i)$^β applied to the loss.

Rainbow DQN — what six extensions does it combine?

(1) Double DQN, (2) Dueling architecture, (3) Prioritized replay, (4) Multi-step (n-step) returns, (5) Distributional RL (C51: model return distribution not just mean), (6) Noisy nets (parametric exploration replacing epsilon-greedy). Hessel et al. 2018 showed each contributes; combined SOTA on Atari for its size.

Distributional RL (C51, QR-DQN) — the idea.

Model the FULL distribution of returns Z(s, a) instead of just its mean Q(s, a) = E[Z(s, a)]. C51 (Bellemare et al. 2017): discretize into 51 atoms, learn categorical distribution over returns.

Noisy nets — how do they replace ε-greedy?

Add parametric noise to weights of last layer(s): W = μ + σ ⊙ ε with learnable μ, σ. Exploration comes from state-dependent noise, learned per-parameter.

SARSA update — on-policy analog of Q-learning.

Q(s, a) ← Q(s, a) + α [r + γ Q(s', a') - Q(s, a)] where a' is the ACTUAL action taken from s' (sampled from current π). On-policy: target uses the policy's next action, not the greedy one.

How do you size a replay buffer?

Trade-off: bigger buffer → more diverse data, older stale off-policy data. Rule: enough to cover diverse experience but not so large that stale data hurts.

Hard update vs soft (Polyak) update for target networks.

Hard: copy $\theta_{\text{target}} \leftarrow \theta$ every N steps (DQN: every 10k). Simple.

Why does DQN scale poorly to continuous action spaces?

Q-learning requires argmax_a Q(s, a) — over a continuous action space, this is a nested optimization at every step (expensive) or requires discretizing (loses fidelity). Solutions: (1) Actor-critic (DDPG, SAC): explicit π_θ(s) network outputs continuous action; (2) NAF (Normalized Advantage Function): analytic argmax via quadratic-in-a parameterization; (3) sample-based action selection with CEM (QT-Opt).

Fitted Q-Iteration — batch off-policy Q-learning.

Alternate: (1) collect / accumulate dataset D of (s, a, r, s') transitions, (2) fit Q^{k+1} to targets $y_i = r_i + \gamma \max_a Q^k(si, a)$ via supervised regression (any regressor: tree, NN). Repeat.

C51 — how does the projected Bellman update work?

Distribution over N=51 atoms z_i in $[V_{\min}, V_{\max}]$ with probability $p_i(s, a)$. Bellman: sample transition, compute target atom z_i' = $\text{clip}(r + \gamma z_i, V_{\min}, V_{\max})$; this may not land on a valid atom → distribute probability mass proportionally to two nearest atoms (linear interpolation).

IQN — Implicit Quantile Networks.

Instead of predicting discrete atoms (C51) or fixed quantiles (QR-DQN), IQN learns to output ANY quantile τ ∈ [0, 1] given as input to the network. Input concatenates state + embedding of τ.

Why is naive DQN unstable and what specific fixes address which cause?

(1) Correlated sequential data → replay buffer breaks correlation. (2) Moving target Q'-target = Q → target network freezes target for N steps.

Value decomposition (VDN, QMIX) — for multi-agent Q-learning.

In cooperative multi-agent, joint $Q(s, a_1, , a_n)$ grows exponentially. VDN: $Q_{\text{total}} = \Sigma Q_i(s, a_i)$ → decentralized execution + centralized training.

DDPG — Deep Deterministic Policy Gradient.

Off-policy actor-critic for continuous actions. Deterministic policy μ_θ(s) + Q-critic Q_φ(s, a).

TD3 — Twin Delayed DDPG improvements over DDPG.

Fujimoto et al. 2018 fixes: (1) Twin critics: two Q-networks, use min for target → mitigates overestimation. (2) Delayed policy updates: update actor every 2 critic steps → gives critic time to catch up.

SAC — Soft Actor-Critic mechanism and advantages.

Maximum entropy RL: maximize $E[\Sigma \gamma^t (r_t + \alpha H(\pi(\cdot \mid s_t)))]$ where H is policy entropy. Adds entropy bonus → exploration + robustness.

IMPALA — how does it scale actor-critic?

Actor-learner architecture: many parallel actor threads roll out trajectories using slightly-stale policy, single learner updates. Off-policy correction via V-trace (Espeholt et al. 2018): truncated importance sampling with clipping.

Vectorized environments — why and how?

Batch multiple env instances that step in parallel per learner step → wider (batched) but same-length trajectory data. Benefits: (1) better GPU utilization for policy inference (batch forward pass over N envs), (2) more diverse data per update, (3) faster wall-clock time.

Bootstrapped DQN — approximate Thompson sampling for deep RL.

K parallel Q-heads sharing a body, each trained on a different bootstrap of the replay buffer (mask indicates which head learns from which transition). At episode start, sample one head, act greedy w.r.t. it for entire episode → deep exploration (temporally-extended commitment).

Count-based exploration for large state spaces.

Add intrinsic reward $r_{\text{int}} = \beta / \sqrt{N(s)}$ where N(s) = visit count. Naive count fails in continuous / high-dim state → use pseudo-counts from density model (Bellemare et al. 2016) or hashing (Tang et al. 2017).

Random Network Distillation (RND) — mechanism.

Fixed random target network $f_{\text{target}}(s)$ initialized randomly + trainable predictor $f_{\text{pred}}(s)$. Intrinsic reward = $||f_{\text{pred}}(s) - f_{\text{target}}(s)||^2$.

Intrinsic Curiosity Module (ICM) — Pathak et al.

Learn forward model f(s, a) → predicted $\varphi(s_{t+1})$ and inverse model $g(s_t, s_{t+1}) \to a_t$. Encode state via φ trained by inverse loss (predicts a from state pair) → learns features that matter for control.

Go-Explore — how does it solve hard-exploration?

Ecoffet et al. 2019, 2021: (1) archive all visited states (cell-hashed). (2) 'Go' phase: teleport back to promising archived state (or replay actions).

Hindsight Experience Replay (HER) — the trick.

Andrychowicz et al. 2017: for a failed episode aiming at goal g, re-label the trajectory as if the final state achieved was the goal. Failed attempts become successful demonstrations of reaching that alternative goal.

Agent57 — how did it achieve above-human on all Atari?

Combined: (1) meta-controller balancing exploration ↔ exploitation across an episode by choosing among a family of policies with different exploration weights. (2) NGU (Never Give Up) intrinsic motivation: episodic + long-term.

World models (Ha & Schmidhuber, Dreamer) — big idea.

Learn a compact latent dynamics model of the environment: encoder + RNN dynamics + reward predictor. Train policy inside the model ('imagination') rather than the real env.

MuZero — model-based RL without knowing the env dynamics.

Schrittwieser et al. 2020: learn a representation + dynamics + reward + value network end-to-end from experience → MCTS in latent space. No need for environment simulator (unlike AlphaZero).

MBPO — Model-Based Policy Optimization.

Janner et al. 2019: use an ensemble of forward models. Every real step, generate k short imagined rollouts (branch factor: many rollouts, short length ~ 1-15 steps).

How do you improve sample efficiency in deep RL?

(1) Replay buffer + off-policy learning (DQN, SAC). (2) Model-based (Dreamer, MBPO).

Sim-to-real — main techniques.

(1) Domain randomization: randomize physics params (mass, friction, latency) in sim → policy robust to real variation. (2) Domain adaptation: fine-tune on real data.

R2D2 — distributed recurrent DQN.

Kapturowski et al. 2019: recurrent Q-network (LSTM) for partial observability + prioritized replay with sequence-based sampling + distributed actor-learner + burn-in prefix (feed part of sequence with old hidden state before computing loss). Combines R (recurrence) + D (distributed) + D (double / dueling).

Why does DQN stack 4 consecutive frames?

Atari observation is a single frame — insufficient for Markov state (need velocity: ball direction, projectile motion). Stack 4 last frames as input → agent can infer motion.

Reward clipping in DQN — trade-offs.

DQN Atari clips rewards to {-1, 0, +1} → stabilizes Q-learning across games with vastly different reward scales (Ms. Pac-Man vs Skiing). Cost: agent doesn't distinguish 1-point vs 100-point reward → suboptimal for scale-sensitive tasks.

OpenAI Five — Dota 2 milestone.

Five-agent RL (one per hero) trained via self-play on massive scale (128k CPU cores, 256 GPUs). PPO with LSTM policy, shared parameters across heroes with role embedding.

Plasticity loss in deep RL — the problem.

Deep RL agents lose the ability to learn new patterns after long training — 'plasticity loss'. Nikishin et al. 2022: root cause is weight rank collapse + dead ReLU units accumulating.

Policy gradients & actor-critic

What is a policy gradient and why do we use it?

A policy gradient directly optimizes the parameters theta of a stochastic policy $\pi_\theta(a \mid s)$ to maximize expected return: grad $J = E[\text{grad } \log \pi_\theta(a \mid s) \cdot A(s,a)]$. Advantages: handles continuous action spaces, learns stochastic policies (crucial for partial observability), and integrates naturally with neural networks.

What is an actor-critic algorithm?

Actor-critic combines a policy (actor) that picks actions and a value function (critic) that estimates the advantage or value. The critic reduces the variance of the policy-gradient estimate.

Why is PPO so widely used?

PPO is a policy-gradient method with a clipped surrogate objective that prevents the policy from moving too far from the old one at each update. It is on-policy, simple to implement, robust across environments with default hyperparameters, and empirically competitive with more complex methods.

REINFORCE — the vanilla policy-gradient algorithm.

For episode τ with return G(τ), update θ ← θ + α G(τ) ∇_θ log π_θ $(a_t \mid s_t)$ summed over trajectory. Unbiased gradient of expected return via log-derivative trick.

TRPO — Trust Region Policy Optimization.

Constrained optimization: max_θ $E[\pi\theta/\pi_{\text{old}} \cdot A]$ s.t. $E[\text{KL}(\pi_{\text{old}} \mid\mid \pi\theta)]$ ≤ δ.

PPO's clipped surrogate objective — write it and explain.

$L^{\text{CLIP}}(\theta) = E[\min(r_t(\theta)\, A_t,\ \text{clip}(r_t(\theta),\ 1-\varepsilon,\ 1+\varepsilon)\, A_t)]$ where $r_t(\theta) = \pi\_\theta(a \mid s) / \pi_{\text{old}}(a \mid s)$. Clip stops the policy from moving too far from π_{old} on either side. min ensures we take the more conservative estimate.

PPO implementation details that actually matter.

Engstrom et al. 2020 'Implementation Matters': (1) value function clipping, (2) reward scaling by running SD, (3) orthogonal init, (4) Adam with fixed lr ($\beta_2 = 0.999$), (5) global gradient clip to 0.5, (6) GAE with λ = 0.95, (7) normalize advantages per-batch, (8) LR annealing, (9) N epochs per rollout (~4-10). Many published gains are due to these tricks, not the core algorithm.

A2C vs A3C — the difference.

A3C (Asynchronous Advantage Actor-Critic, Mnih 2016): multiple async workers each with own env, updating shared params — lock-free async gradient updates. A2C (synchronous): workers step in parallel, batch gradients synchronously, single update per batch — better GPU utilization, more reproducible.

Why add an entropy bonus to the policy objective?

Encourages stochastic exploration + prevents premature convergence to suboptimal deterministic policy. Loss = -E[log π · A] - β H(π).

PPO vs SAC — which do you pick?

PPO: on-policy, simple to implement + tune, works everywhere reasonably, standard for RLHF of LLMs. SAC: off-policy, sample-efficient (uses replay), maximum entropy → robust, best for continuous control (robotics). Rule: (1) discrete + easy tuning + big rollouts → PPO.

DPO — Direct Preference Optimization instead of RLHF PPO.

Rafailov et al. 2023: skips explicit reward model + PPO. Direct loss on preference pairs: L(θ) = -
$E[\log \sigma(\beta \log \pi\theta(y_w \mid x)/\pi_{\text{ref}}(y_w \mid x) - \beta \log \pi\theta(y_l \mid x)/\pi_{\text{ref}}(y_l \mid x))]$.

GRPO — Group Relative Policy Optimization (DeepSeekMath, R1).

PPO variant that skips the value network. For each prompt, sample G responses, compute rewards, normalize as advantages:
$A_i = (r_i - \text{mean}_r) \,/\, \text{std}_r$.

Categorical vs Gaussian vs beta policies — which and when?

Categorical: discrete actions, softmax over logits. Gaussian: continuous, $N(\mu\theta(s),\ \sigma\theta(s)^2)$. σ can be state-dependent or state-independent parameter.

Value function clipping in PPO — why and how?

$L^{\text{VF}} = \max((V\theta(s_t) - V_{\text{targ}})^2$, (
$\text{clip}(V\theta(s_t),\ V_{\text{old}} - \varepsilon,\ V_{\text{old}} + \varepsilon) - V_{\text{targ}})^2$) → same trust-region-style clip for the value function. Prevents value update from swinging wildly per epoch.

How does batch size / rollout length affect PPO?

Larger rollout length → lower variance advantage estimates (more MC-like), but slower feedback. Batch size ($n_{\text{actors}} \times \text{rollout}_{\text{length}}$): must be big enough for a stable gradient estimate; typical 2048-16384.

Ornstein-Uhlenbeck noise vs Gaussian noise for continuous control.

OU: temporally correlated noise: $\text{dx}_t = \theta(\mu - x_t)\text{dt} + \sigma\, \text{dW}_t$ → mean-reverting Brownian. Used in original DDPG for exploration on physical control tasks — smooth trajectories match physical inertia.

MADDPG — Multi-Agent DDPG.

Lowe et al. 2017. Each agent has its own actor $\pi_i(o_i)$ and critic $Q_i(s,\ a_1,\ ,\ a_n)$ (centralized: sees ALL agents' actions).

MAPPO — Multi-Agent PPO.

Yu et al. 2022. Simply PPO with a centralized value function using joint observations, shared across cooperative agents.

Hierarchical RL — why and how?

Decompose long-horizon tasks into (1) high-level policy over sub-goals or options, (2) low-level policy executing skills. Advantages: temporal abstraction (reason over many steps at high level), sample efficiency, transfer of skills.

DIAYN — Diversity is All You Need for skill discovery.

Eysenbach et al. 2018. Unsupervised skill learning: (1) sample skill z from prior, (2) train $\pi(a \mid s,\ z)$ to maximize mutual information between z and visited states s: max I(s; z).

Goal-conditioned RL — setup.

Policy $\pi(a \mid s,\ g)$ conditioned on desired goal g. Reward r(s, a, g) = f(distance to g) or -1 until g reached.

Interview: 'explain why PPO is on-policy but uses a ratio to importance-sample.'

On-policy in spirit: PPO uses only recent data collected under π_{old} ($\text{close to } \pi\theta$). The importance ratio r = π_θ / π_{old} accounts for the fact that after several gradient steps within the epoch, π_θ has drifted from the data-generating π_{old} — but not far (clipping enforces this).

Curriculum learning in RL — how to design one?

Order training from easy to hard tasks: (1) start with dense reward, gradually sparsify. (2) Start with easy env, add complexity (obstacles, distractors).

What is the clipping in PPO actually protecting you from?

From a policy update so large that the data you collected no longer describes the policy you now have. Policy gradient estimates are only valid near the behaviour policy, so a big step invalidates the very samples that justified it, and the classic symptom is a policy that collapses after a promising run.

When does the sample inefficiency of on-policy methods stop mattering?

When samples are cheap and stability is expensive, which is the case with a fast parallelizable simulator. If you can run thousands of environment steps per second across many workers, PPO's need for fresh data on every update costs wall-clock time you have, and you get back a method that is markedly less brittle and has far fewer hyperparameters that can silently destroy a run.

Exploration strategies

What is the exploration-exploitation tradeoff?

Exploitation uses the best-known action to maximize immediate reward; exploration tries new actions to discover potentially better ones. Too much exploitation = premature convergence; too much exploration = wasted samples.

CEM — Cross-Entropy Method for planning.

Sample-based optimization: (1) sample K action sequences from Gaussian, (2) evaluate returns, (3) select elite (top 10%), (4) refit Gaussian to elites, (5) repeat. Simple, embarrassingly parallel, no gradients.

ε-greedy — how do you schedule ε?

Start ε high (0.5-1.0) for broad exploration, anneal linearly or exponentially to a small floor (0.01-0.1). Atari DQN: linear 1.0 → 0.1 over first 1M steps, then 0.1 → 0.01 over 24M more.

How does posterior sampling drive exploration in deep RL?

Maintain posterior over Q or model parameters. To choose action: sample θ ~ $p(\theta \mid \text{data})$, act greedily w.r.t. sampled θ.

Why is Montezuma's Revenge such a famous benchmark?

Extreme sparse reward: dozens of steps needed before any positive reward. DQN + ε-greedy scores 0.

Parameter noise for exploration — how is it different?

Plappert et al. 2017: add noise directly to policy parameters (θ + σε) instead of action noise. Produces consistent behavior over an episode (θ is fixed until reset) → temporally-correlated exploration.

The 'noisy TV problem' in curiosity-driven exploration.

If agent gets intrinsic reward for high prediction error, it can find sources of pure noise (TV showing static, random-generated stimuli) and stay 'exploring' those forever — infinite curiosity but no learning. Fixes: (1) inverse-model features (ICM) — only rewards prediction errors about controllable aspects.

Emergent tool use in RL — canonical example.

Baker et al. 2019 (OpenAI): hide-and-seek multi-agent RL discovered emergent tool use — hiders build ramps, seekers box surf, hiders lock ramps, seekers exploit physics glitches. Six curriculum stages emerged without engineering.

You want to optimize which of five banners to show. Bandit or full RL?

A contextual bandit, because showing a banner does not meaningfully change the state the next user arrives in, so there is nothing to credit across time. That single simplification removes the hardest part of reinforcement learning and gives you a problem with clean theory, fast convergence, and well-understood algorithms such as Thompson sampling or upper confidence bounds.

How do you explore safely when the environment is real paying users?

Bound the damage rather than trusting the algorithm. Restrict exploration to a small traffic slice, so a bad action affects a known fraction of users, and exclude segments where the cost of a mistake is high.

The reward arrives only at the end of a 500-step episode. What do you do?

Shorten the effective credit assignment problem. Reward shaping is the direct approach, adding intermediate signal, and if you shape it as a potential-based difference the optimal policy provably does not change, which is the only shaping I would introduce without careful evaluation.

Model-based & planning

Monte Carlo Tree Search — how it works in AlphaGo/MuZero.

Iteratively build a tree of possible action sequences. Each iteration: (1) Select — walk from root using UCB1 (or PUCT) balancing exploration + Q.

AlphaZero — key ideas.

(1) MCTS as policy-improvement operator: search improves the raw NN prior. (2) Self-play generates training data: play against your current model.

Dyna — early model-based RL framework.

Sutton 1990: interleave (1) real interaction with env, (2) model updates from real data, (3) planning: multiple simulated Q-learning updates from the learned model. Rebuilds Q from cheap simulated experience.

Model Predictive Control (MPC) in RL — how does it work?

At each step: (1) roll out learned or hand-crafted model for horizon H, (2) optimize action sequence over that horizon (CEM, LQR, iLQR, gradient descent), (3) execute the first action, (4) re-plan next step. Uses: robotics locomotion (PETS, PlaNet), industrial control, quadcopter flight.

Reward design & shaping

What is reward shaping and what are its pitfalls?

Reward shaping adds intermediate rewards to guide the agent, e.g., dense proxies instead of only a sparse success signal. Done right (potential-based shaping) it preserves the optimal policy.

Reward engineering — practical guidelines.

(1) Start with sparse task reward (1 for success, 0 otherwise) — cleanest signal. (2) If too sparse, add potential-based shaping (safe).

How do you catch reward hacking before it reaches production?

Watch for the signature: reward climbing while every measure you actually care about stagnates or degrades. That means you must instrument metrics that are deliberately not part of the reward, since a reward you optimize can no longer serve as its own audit.

Deep RL engineering

Learning-rate schedule for PPO / SAC — what works?

PPO: linear anneal from 3e-4 to 0 over training. Adam optimizer, $\beta_2 = 0.999$.

Population-Based Training (PBT) for RL.

Jaderberg et al. 2017: run N agents in parallel with different hyperparameters. Periodically: worst agents copy weights from top-N + perturb hyperparameters.

Interview: 'your DQN agent isn't learning — how do you debug?'

(1) Verify env: hardcoded oracle policy should score well. (2) Check reward signal: is it too sparse / too noisy?

Policy averaging — what is it and why?

Take exponentially-weighted average of policy weights over training (Polyak averaging on policy). Reduces variance of policy across training + gives smoother behavior.

How do you efficiently train RLHF at scale?

(1) Sample generations offline, cache. (2) Reward model inference batched separately from policy.

Your policy is perfect in simulation and useless on the real robot. How do you close the gap?

Assume the policy has learned to exploit simulator inaccuracies, because that is the usual cause rather than a lack of capacity. Domain randomization is the main tool: vary masses, friction, latencies, sensor noise and lighting during training so the policy must be robust across a distribution of dynamics rather than optimal for one wrong set.

Two RL algorithms report different scores in a paper. Why should you be skeptical?

Because reinforcement learning results have unusually high variance across seeds, and a difference computed from three runs is frequently noise. The distribution of final performance is often bimodal, with some seeds failing entirely, so a mean hides more than it shows and a maximum over seeds is not a result at all.

Offline RL

What is off-policy evaluation and why is it hard?

Off-policy evaluation estimates the value of a target policy using data collected by a different behavior policy — without deploying the new policy. It's crucial in medicine, finance, and any high-stakes system where trying the new policy is risky.

What is offline RL and why does it matter?

Learn policy from fixed dataset of (s, a, r, s') collected by some behavior policy — NO env interaction during training. Motivation: (1) real environments too expensive / risky to explore (healthcare, autonomous, industrial).

Why is distribution shift the core challenge in offline RL?

Policy π_θ may query Q at (s, a) never seen in data → Q's extrapolation is nonsense → policy exploits Q's overestimation of OOD actions → learned policy fails. Naive off-policy methods (SAC on offline data) diverge catastrophically.

BCQ — Batch-Constrained Q-Learning.

Fujimoto et al. 2019. Constrain policy to only take actions similar to those in the data.

CQL — Conservative Q-Learning.

Kumar et al. 2020. Add penalty to standard Q-learning loss: expectation of Q over the policy (pushed down) minus expectation over data (pushed up) → pessimistic on OOD actions.

IQL — Implicit Q-Learning.

Kostrikov et al. 2022. Avoids explicit policy constraint or Q pessimism.

TD3+BC — the simplest offline RL baseline.

Fujimoto & Gu 2021. TD3 (actor-critic) + BC (behavior cloning) regularizer added to policy loss: min E[-Q(s, π(s))] + λ $(\pi(s) - a_{\text{data}})^2$.

Behavior Cloning (BC) — when does it work?

Supervised learning: fit $\pi(a \mid s)$ to (s, a) pairs from expert demos via MLE. Works when: (1) expert data is high quality, (2) test distribution matches expert distribution (no drift), (3) enough coverage.

DAgger — Dataset Aggregation for imitation learning.

Ross et al. 2011. Iterative BC: (1) train π on expert data, (2) roll out π in env, (3) query expert for correct action at every state visited, (4) add (state, expert-action) to dataset, (5) retrain.

Inverse Reinforcement Learning (IRL) — the setup.

Given expert demonstrations, infer the reward function that makes the expert optimal. Ill-posed (many rewards consistent).

GAIL — Generative Adversarial Imitation Learning.

Ho & Ermon 2016. Similar to GAN: generator = policy π, discriminator D distinguishes agent trajectories from expert trajectories.

Reward hacking in RLHF — canonical examples.

Model finds ways to score high on r_φ without being genuinely helpful: (1) sycophancy: agree with user regardless of truth. (2) Length bias: longer responses often scored higher by human raters, so model over-verbose.

Constitutional AI — mechanism.

Anthropic 2022. Two phases: (1) SL from AI feedback: model critiques + revises its own responses per written constitution (helpful, harmless principles); train on (prompt, revised response).

How does the KL penalty interact with reward hacking?

KL to π_{ref} bounds how far model can drift → limits reward hacking of imperfect r_φ. But: (1) large model can find hacks within KL budget ($\text{spurious features that both } r\varphi \text{ likes AND } \pi_{\text{ref}} \text{ probably would generate}$).

Safe RL — how do you constrain during learning?

(1) CMDP (Constrained MDP): maximize reward subject to $E[\Sigma\, c_t] \leq$ threshold. Solved via Lagrangian or projected gradient.

Sycophancy in RLHF — what causes it and how to fix?

Model tells users what they want to hear rather than truth. Cause: human labelers prefer agreeable responses; RM learns to reward agreement patterns.

How does RL contribute to jailbreak defense?

(1) Safety-tuned RM penalizes harmful outputs. (2) Adversarial RLHF: red-team probes for jailbreaks, add examples to preference data.

How is o1-style reasoning safer or riskier?

Safer: (1) longer reasoning gives more opportunities to catch errors + reject harmful requests. (2) Explicit deliberation can invoke safety principles.

Why is RL hard for autonomous driving?

(1) Safety: cannot explore in real traffic. (2) Long-tail rare events matter (fatal edge cases).

RL in healthcare — key challenges.

(1) Sample efficiency (real trials cost billions). (2) Safety absolute — can't try random treatments.

Interview: 'how do you make an RL policy safe to deploy?'

(1) Simulation validation across held-out scenarios + adversarial tests. (2) Formal / conservative bounds where possible (CMDP).

Safety verifier reward — what and how?

Automatic reward from external verifier that judges safety: (1) content filter classifier flags harmful outputs. (2) LLM judge with harm rubric.

Multi-agent RL

Multi-agent RL — what makes it fundamentally different?

(1) Environment now non-stationary from each agent's perspective (other agents' policies change during learning) — breaks Markov assumption. (2) Emergent behaviors (cooperation, competition, mixed-motive).

Centralized Training Decentralized Execution (CTDE).

Cooperative multi-agent paradigm: during training, use global state + all agents' info (centralized critic). At execution, each agent uses only its local observations (decentralized policy).

Self-play — why does it work for games?

Agent plays against copies of itself (past or current) → generates unlimited training data, difficulty auto-scales as agent improves. In zero-sum games: converges to Nash equilibrium (Fictitious Self-Play, PSRO).

League play in AlphaStar — mechanism.

Vinyals et al. 2019. Population of agents in 3 roles: (1) Main agents — general strong players.

How do agents learn to cooperate?

(1) Shared reward → both agents want same outcome (value decomposition, MAPPO). (2) Communication learning (RIAL, DIAL, CommNet — learn what/when to communicate).

Emergent communication in MARL — canonical result.

Agents can learn to communicate over discrete tokens via reward gradient (differentiable via Gumbel-softmax or reinforce). Foerster et al.'s DIAL demonstrated on switch-riddle: agents develop protocol to signal correct answer.

AlphaStar StarCraft II — main technical innovations.

Vinyals et al. 2019. (1) LSTM policy + transformer over units.

RLHF & LLM alignment

What is RLHF and how does it fit into training an LLM?

RLHF (Reinforcement Learning from Human Feedback) fine-tunes a pretrained language model to align with human preferences. Steps: (1) supervised fine-tuning on curated demonstrations; (2) train a reward model on human preference pairs; (3) optimize the LLM policy against the reward model using PPO, with a KL penalty to stay close to the reference policy.

RLHF pipeline — the three stages in detail.

(1) SFT: fine-tune pretrained LM on curated (prompt, ideal response) demos → gives baseline capable model. (2) Reward Modeling: collect preference pairs (prompt, response A vs B, human picks winner); train reward model r_φ(x, y) via Bradley-Terry loss: -log $\sigma(r(x,\, y_w) \,-\, r(x,\, y_l))$.

Bradley-Terry model in reward modeling.

Probability preferred response $y_w \,>\, y_l$ given prompt x: $P(y_w \,>\, y_l \mid x) \,=\, \sigma(r(x,\, y_w) \,-\, r(x,\, y_l))$. Loss: -log $\sigma(r(x,\, y_w) \,-\, r(x,\, y_l))$ over preference dataset.

Why does RLHF include a KL penalty?

PPO fine-tunes against learned reward model r_φ, which is imperfect and prone to reward hacking ($\text{model finds spurious features that } r\varphi \text{ overestimates}$). KL penalty β $\text{KL}(\pi\theta \;||\; \pi_{\text{ref}})$ keeps π_θ close to SFT reference → limits how badly model can exploit reward.

DPO vs PPO for RLHF — practical comparison.

PPO: needs reward model + rollouts + KL + value function; complex + unstable. DPO: single stage on preference pairs, no RM or PPO, much simpler + more stable.

RLAIF — RL from AI Feedback.

Replace human labelers with LLM judge for preference labels. Bai et al. 2022 (Constitutional AI): iteratively use a strong LLM to critique + revise responses per a written constitution.

RL for reasoning (o1, DeepSeek-R1) — the paradigm shift.

Train LLM to generate long chain-of-thought reasoning by RL with automatic reward from verifier (math answer correct, code passes tests). Model learns to think longer, retry, self-correct.

How does RL train chain-of-thought reasoning?

Reward on FINAL answer only (verifier: math correct? code passes tests?). Intermediate tokens get gradient via policy gradient on the final reward, credit-assigned across the response.

Verifier reward vs preference-based reward — when to use each?

Verifier: objective correctness (math, code, structured tasks) — automatic, cheap, unambiguous, scalable. Best when correctness is well-defined.

Process reward vs outcome reward for reasoning.

Outcome reward: only final answer scored. Simple but sparse — credit assignment across long CoT is hard.

Iterative RLHF — why do you keep going?

Single-round RLHF: preference data collected on SFT model → RM trained → policy trained. Problem: new policy visits states RM was never trained on → RM predictions drift.

What can RL do that SFT cannot?

SFT does behavior cloning of ideal responses. RL adds: (1) preference between responses (SFT can't compare).

How do reward models scale with size?

Larger RMs (closer in size to policy) give better preferences: better calibration, less easy-to-exploit gaps. But: too small → doesn't understand nuance; too large → expensive per training step.

How do you fix length bias in RLHF?

(1) Length-normalize reward: divide by response length (log or square-root). (2) Add length penalty $\lambda \cdot$ length to policy loss.

ORPO — Odds Ratio Preference Optimization.

Hong et al. 2024. Combines SFT and preference optimization in one loss without a reference model.

KTO — Kahneman-Tversky Optimization.

Ethayarajh et al. 2024. Uses prospect theory (loss aversion): people care more about avoiding losses than gaining.

What is 'alignment tax' in RLHF?

Fine-tuning for alignment (safety, style) degrades other capabilities. E.g., RLHF for helpfulness sometimes hurts factual accuracy or reasoning.

How does RL train tool-use in LLM agents?

(1) Reward on task success (e.g., verifier checks correct API call result). (2) Multi-step trajectories with intermediate tool calls.

Interview: 'walk me through the tradeoffs of SFT-only vs SFT+RLHF vs SFT+DPO.'

SFT only: fast + simple + reliable + no reward hacking. Ceiling limited by demo quality; can't leverage preferences.

ORPO vs DPO — key difference.

DPO: needs reference policy π_{ref} (extra memory + inference cost during training). ORPO: no reference policy — uses odds ratio between chosen/rejected.

LLM agent benchmarks — what tests capability?

(1) WebArena / VisualWebArena: browse web + execute tasks. (2) ALFWorld: navigate textual household environment.

'Verifier + refiner' RL pattern for reasoning.

(1) Sample many reasoning chains from model. (2) Verifier (external tool or LLM judge) scores each.

In RLHF, why does the reward model become unreliable as training progresses?

Because the policy moves off the distribution the reward model was trained on. Preference data came from earlier, weaker samples, so the reward model is accurate there and increasingly extrapolating as the policy improves, which the policy then exploits: it finds text scoring highly under the reward model that humans do not actually prefer.

Why do many teams choose DPO over PPO for alignment?

Mostly engineering cost. PPO-based RLHF requires four models in memory at once, the policy, the reference, the reward model and the critic, plus a generation loop inside training, which makes the pipeline expensive, slow and full of hyperparameters that can silently ruin a run.

Applications & safety

AlphaGo — what made it different from prior Go engines?

Silver et al. 2016. (1) Deep NN combining value and policy (previously hand-crafted).

Is AlphaFold RL?

No — AlphaFold uses supervised learning on protein structure database (PDB). Loss: predicted vs true atomic coordinates.

OpenAI's Rubik's Cube robot — key techniques.

OpenAI 2019. (1) PPO in simulation with Automatic Domain Randomization (ADR): progressively harder randomization of physics parameters.

Modern quadruped locomotion — RL pipeline.

(1) Massively parallel simulation (IsaacGym / Isaac Sim, MuJoCo MJX, Genesis) — thousands of robots in parallel on one GPU. (2) PPO on domain-randomized environment (friction, mass, latency, terrain).

Industrial RL applications and challenges.

Real applications: data center cooling (DeepMind 40% reduction), chip floorplan (DeepMind Nature 2021), advertising bidding, portfolio optimization, drug discovery (design molecules), catalyst design, tokamak plasma control (DeepMind + EPFL). Challenges: (1) sample efficiency (real interactions expensive).

How is a recommender system a bandit / RL problem?

Contextual bandit: user context = state, item shown = action, click/watch = reward. Long-term reward → full RL for session-level or LTV optimization.

RL in trading / finance — techniques.

(1) Portfolio optimization: policy allocates capital across assets. (2) Market making: bid/ask spread policy.

Interview: 'design exploration for a recommendation system.'

(1) Warm-start with content-based recommendations for cold start. (2) Contextual bandit with Thompson sampling: Beta posteriors on CTR give principled exploration.

Standard deep RL benchmark suites.

(1) Atari 57 (DQN-era, still Rainbow / R2D2 / Agent57). (2) MuJoCo continuous control (Hopper, Walker, Ant, Humanoid) — SAC / PPO staple.

RT-2 — Vision-Language-Action model.

Brohan et al. 2023 (Google). Fine-tune vision-language model (PaLI-X / PaLM-E) to output action tokens for robotics.

A product manager asks for reinforcement learning. When should you talk them out of it?

Whenever the problem is really supervised prediction with a decision rule on top, which covers most business cases. If the action does not change the future state of the world, you do not need sequential decision making: predict the outcome and optimize the decision analytically.

Interview scenarios

Interview: 'design an RL system for X' — what's your framework?

Structure: (1) Define MDP: state (features observable), action (decision), reward (business KPI), horizon. (2) Identify challenges: sparse / delayed rewards, safety, sample efficiency.

Interview: 'when should you NOT use RL?'

(1) Supervised data + clear labels available — SL is faster + safer. (2) Task has one shot (no sequential decisions).

Interview: 'you're given 100 hours of expert data + can simulate — how do you train an agent?'

(1) BC first: quick baseline from 100 hours of demos → benchmark. (2) Warm-start RL from BC weights → faster than from scratch.

Interview: 'DQN vs PPO vs SAC vs DPO — which for what?'

DQN: discrete actions, Q-learnable, replay works → Atari, tabular-like. PPO: general workhorse, on-policy, LLMs (RLHF), games — reliable + tunable.

Interview: 'summarize when RL wins in production.'

RL wins when: (1) task has sequential decisions with delayed reward that SL / bandit can't capture. (2) You can simulate cheaply or have massive logged interaction data (offline RL).