Reinforcement Learning Cheatsheet

Generated from Lecture Notes for CSCI 6904 (Deep Reinforcement Learning) - 2025 Summer

August 10, 2025

Contents

1 Formalization of the RL Problem

1.1 Markov Property

A state S_t is Markov if $p(S_{t+1} \mid S_t) = p(S_{t+1} \mid S_1, \ldots, S_t)$. This means the future is conditionally independent of the past, given the present state.
Or
The Markov property, also known as the "memoryless property", is a characteristic of stochastic processes where the future state of a system depends only on its current state, not on its past history.

1.2 Markov Decision Process (MDP)

An MDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ where:

- $\mathcal{S}$: The set of states.
- $\mathcal{A}$: The set of actions.
- P: The state-transition probability matrix, where $p(s'|s,a) = \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\}$.
- R: The reward function, where $r(s,a) = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a]$.
- $\gamma \in [0,1]$: The discount rate.

Goal: Find the optimal policy $\pi^*(a|s) = \arg\max_\pi \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s]$.

1.3 Goals and Rewards

Rewards define the objective of the problem. A policy's goal is to maximize the expected cumulative discounted rewards over time.

1.4 Discount Rate γ

The discount rate γ controls the preference for immediate versus future rewards. A value of $\gamma = 0$ results in a myopic agent that only considers immediate rewards, while $\gamma = 1$ makes the agent treat all future rewards equally, assuming the task eventually terminates.

1.5 Returns and Episodes

The return G_t is the total discounted reward from time step t. It is defined as:

- For continuing tasks: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.
- For episodic tasks: $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$, where T is the terminal time step.

The return can be expressed recursively as $G_t = R_{t+1} + \gamma G_{t+1}$, with $G_T = 0$.

1.6 Partially Observable MDP (POMDP)

A POMDP is a generalization of an MDP defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, P, R, E, \gamma \rangle$, which includes a set of observations $\mathcal{O}$ and an emission probability E, defined as $p(o_t \mid s_t)$. In a POMDP, the agent's policy is based on observations, $\pi(a|o)$, rather than states.

2 Major Components of an RL Agent

2.1 Policy

An agent's behavior maps states to actions.

- Deterministic: $a = \pi(s)$.
- Stochastic: $\pi(a|s) = p(a|s)$.

2.2 Value Function

Prediction of expected return.

- State-value: $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s]$.
- Action-value (Q-function): $q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$.

Bellman Equation

The Bellman equation relates a state's value to its immediate reward and the discounted value of the next state.

For a policy π (state-value form):

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V_\pi(s')]$$

Meaning:

- Weight actions by policy probability.
- Weight next states by transition probability.
- Add immediate reward + discounted future value.

Optimal Value Function

For optimal $V_*(s)$:

$$V_*(s) = \max_a \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V_*(s')]$$

Replace sum over actions with max for best choice.

2.3 Bellman Expectation Equation

The Bellman expectation equation for the state-value function $V_\pi(s)$ is:

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s]$$

Here, the expectation operator $\mathbb{E}_\pi$ represents an average over actions selected by the policy π and the subsequent next states produced by the environment's dynamics. It can be conceptually understood as answering the question: "On average, if I follow policy π from this state, what will I get?"

2.4 Optimality in the Bellman Equation

The relationship between prediction and control problems is central to Reinforcement Learning:

- **Prediction problem:** Given a policy π, find its value function V_π.
- **Control problem:** Find the optimal policy π_* that maximizes the return.

The Bellman optimality equation is what we solve to find the optimal value function V_*:

$$V_*(s) = \max_a \mathbb{E}\big[R_{t+1} + \gamma V_*(S_{t+1}) \mid S_t = s, A_t = a\big]$$

Once we have the optimal value function V_*, the optimal policy π_* can be found by acting greedily with respect to it:

$$\pi_*(s) = \arg\max_a \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V_*(s')\big]$$

Recap

- Bellman expectation: value = average reward + average future value under π.
- Bellman optimality: value = best reward + best future value possible.

2.5 Model

An agent's representation of its environment.

- Dynamics: $p(s', r|s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}$.
- State transition: $p(s'|s, a)$.
- Reward: $r(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$.

2.6 Categorizing RL Agents

- **Policy-based**: A policy, but no value function.
- **Value-based**: A value function, with an implicit policy.
- **Actor-Critic**: Both a policy and a value function.
- **Model-free**: A policy or value function, but no model.
- **Model-based**: A policy or value function, plus a model.

Category	Core Idea	How It Works	Pros	Cons
Policy-based	Learn the policy directly (e.g., **REINFORCE, PPO**).	Use gradients to adjust policy parameters, often with randomness.	Handles continuous actions well; learns varied strategies.	Can be sample-inefficient and unstable.
Value-based	Learn a value function, pick best actions (e.g., **DQN, Double DQN**).	Update values (like Q-values) and act greedily.	Efficient for discrete tasks; stable targets.	Struggles with continuous actions; risk of overestimation.
Actor-Critic	Combine policy and value function (e.g., **A2C, SAC**).	Actor chooses actions, critic evaluates them.	Merges strengths of both; handles continuous actions well.	Coupled training can be unstable; needs careful tuning.
Model-free	Learn from direct experience only (e.g., **PPO, DQN**).	Learn policy/values via trial and error.	Simple and robust when modeling is hard.	Needs lots of data; no lookahead planning.
Model-based	Learn/use environment model (e.g., **Dyna-Q, Dreamer**).	Simulate future steps to improve.	Efficient; allows long-term planning.	Model errors can mislead; hard to learn accurate models.

3 Introduction to Policy Gradient Methods

3.1 Policy

The parameterized policy is defined as: $\pi(a|s,\theta) = \Pr(A_t = a \mid S_t = s, \theta_t = \theta)$.

- **Discrete Actions**: A softmax policy, where $\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$, and $h(s,a,\theta)$ is a numerical preference for action a.
- **Continuous Actions**: A Gaussian policy, where the action is sampled from a normal distribution, $a \sim \mathcal{N}(\mu(s,\theta), \sigma^2(s,\theta))$. The probability density function is given by $\pi(a|s,\theta) = \frac{1}{\sigma_\theta(s)\sqrt{2\pi}} \exp\left(-\frac{(a-\mu_\theta(s))^2}{2\sigma_\theta^2(s)}\right)$.

3.2 Objective

The objective is to find the parameter θ that maximizes the expected return $J(\theta)$, which is defined as:

$$J(\theta) \doteq \mathbb{E}_{\pi_\theta}[G_t \mid S_t = s_0] = v_{\pi_\theta}(s_0)$$

Policy gradient methods achieve this by using an iterative update rule, often referred to as **Gradient Ascent**:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$$

4 Policy Gradients for One-Step MDPs / Contextual Bandits

One-step MDPs are defined as a process where an agent:

- Starts in a state s sampled from a distribution $d(s)$.
- Takes an action a sampled from the policy $\pi_\theta(a|s)$.

- Receives a reward $r = R_{s,a}$ and the episode terminates.

The objective is to maximize the expected reward, given by:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_s d(s) \sum_a \pi_\theta(a|s) R_{s,a}$$

The policy gradient is then calculated as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot r] \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(A_i|S_i) \cdot R_{S_i,A_i}$$

The policy parameters are updated using a gradient ascent rule:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

This approach differs from Supervised Learning (SL) Maximum Likelihood in that the RL update is weighted by the reward r, whereas the SL update is an unweighted average.

4.1 Algorithm ($\sim$ REINFORCE for one-step)

1. Run the policy: Sample a state $s \sim d(s)$ and an action $a \sim \pi_\theta(a|s)$.
2. Compute the gradient estimate: $\nabla_\theta J(\theta) = \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(A_i|S_i) R_{S_i,A_i}$.
3. Update the policy parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.

5 Policy Gradients for Multi-Step MDPs

5.1 Objective

The objective is to maximize the expected return from a starting state, which is equivalent to the state-value function:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t \mid S_t = s] = v_{\pi_\theta}(s)$$

5.2 Policy Gradient Theorem

The policy gradient theorem provides a way to compute the gradient of the objective function:

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) \cdot q_\pi(s,a)\right]$$

This is often approximated with the Monte Carlo return G_t:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) \cdot G_t\right]$$

This replaces the instantaneous reward with a long-term value $q_\pi(s,a)$ to guide learning.

5.3 REINFORCE Algorithm

1. Sample trajectories $\{\tau_i\}$ from the policy π_θ.
2. For each time step t in a trajectory, compute the return $G_t^i = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k^i$.
3. Estimate the policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta(A_t^i|S_t^i) G_t^i$$

4. Update the policy parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.

Note: This algorithm encourages good trajectories by increasing the probability of actions that lead to high returns.

- **Discrete actions**: Use a softmax policy.
- **Continuous actions**: Use a Gaussian policy.

This method suffers from high variance in gradient estimates.

5.4 REINFORCE with Baseline

A baseline $b(s)$ (e.g., the state-value function $v(s)$) can be subtracted from the return to reduce variance without changing the expected value of the gradient.

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) \cdot (q_\pi(s,a) - b(s)) \right]$$

5.4.1 Algorithm

1. Sample trajectories $\{\tau_i\}$ from π_θ.
2. Compute returns G_t^i.
3. Estimate the policy gradient using a learned state-value function $\hat{v}_w(s)$ as the baseline:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta(A_t^i|S_t^i)(G_t^i - \hat{v}_w(S_t^i))$$

4. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$.
5. Define the value function loss:

$$L(w) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} (G_t^i - \hat{v}_w(S_t^i))^2$$

6. Update the value function parameters: $w \leftarrow w - \alpha_w \nabla_w L(w)$.

6 Off-Policy Policy Gradients

6.1 Concepts

- **On-policy** methods learn a policy from experience collected by that same policy.
- **Off-policy** methods learn a target policy π using data generated by a different behavior policy μ.

Off-policy learning offers key advantages: it can reuse old data, allows the agent to learn an optimal policy while actively exploring with a different behavior policy, and enables learning about multiple policies simultaneously.

6.2 Importance Sampling

Importance sampling is a technique that allows us to estimate the expected value of a function under one distribution, using samples from another. This is the core of off-policy policy gradients.

$$\mathbb{E}_{x \sim p}[f(x)] = \mathbb{E}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right]$$

The term $\frac{p(x)}{q(x)}$ is the importance sampling weight, which corrects for the difference between the two distributions.

6.3 Off-Policy REINFORCE with Importance Sampling

This method adapts REINFORCE to an off-policy setting by using importance sampling. It allows the algorithm to learn from data collected by an older policy, $\pi_{\theta_{old}}$. The gradient is estimated using the importance sampling weight to correct for the change in policy:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \frac{\pi_\theta(A_t^i|S_t^i)}{\pi_{\theta_{old}}(A_t^i|S_t^i)} \nabla_\theta \log \pi_\theta(A_t^i|S_t^i) G_t^i$$

The parameters are then updated:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

After the update, the old policy is replaced by the new one:

$$\theta_{old} \leftarrow \theta$$

This approach is notably used in algorithms like Proximal Policy Optimization (PPO), which is essential for training large language models like those in ChatGPT. PPO clips the importance sampling weight to prevent excessively large updates, which helps stabilize learning.

7 Case Study: Dialog Systems like ChatGPT

7.1 Training Phases

The development of advanced dialog systems like ChatGPT typically follows three key phases:

1. **Pretraining (LM)**: A large language model (LM) is first pretrained on a vast corpus of text to learn language patterns and predict the next token.
2. **Instruction Finetuning (SL)**: The pretrained model is then finetuned on a dataset of high-quality instructions and responses using supervised learning (SL) to better follow directions.
3. **Reinforcement Learning with Human Feedback (RLHF)**: The model is further aligned with human preferences using reinforcement learning.

7.2 Reinforcement Learning with Human Feedback (RLHF)

RLHF leverages policy gradients to refine the model's behavior based on human input. The process involves:

1. **Response Sampling**: The policy π_θ generates a response (action a) to a given prompt (state s).
2. **Human Feedback**: Human labelers provide feedback on the responses, often by ranking them.
3. **Reward Model Training**: A separate reward model, $r_\psi(s, a)$, is trained to predict the human preference score for any response, learning from the collected rankings.
4. **Policy Update**: The policy is updated using a policy gradient algorithm with the reward signal from the reward model. A simplified update rule is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(A_i|S_i) r(S_i, A_i)$$

5. **Off-Policy Considerations**: To improve sample efficiency, off-policy methods with importance sampling are used. A **KL-divergence penalty** is also added to the objective function to prevent the new policy from deviating too drastically from the reference policy, thus maintaining model stability and preventing reward hacking.

8 Actor-Critic Methods

8.1 Recap: Policy Gradient Methods

8.1.1 Monte-Carlo Policy Gradient (REINFORCE)

- **Objective**: The goal is to maximize the expected return from a starting state, represented by the state-value function:
$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t \mid S_t = s_0] = v_{\pi_\theta}(s_0)$$
- **Policy Gradient Theorem**: The gradient is estimated as the expectation of the log-policy gradient scaled by the action-value function:
$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) q_\pi(s,a)]$$
- **Update Rule**: The parameters are updated in the direction of the sampled return G_t, which is a Monte-Carlo estimate of $q_\pi(s,a)$:
$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi_\theta(A_t|S_t) G_t$$

This method estimates the action-value function using a complete trajectory's return, which, while unbiased, can have high variance.

8.1.2 REINFORCE with Baseline

To address the high variance of REINFORCE, a baseline $b(s)$ is introduced.

- **Modified Gradient**: The gradient is now proportional to the advantage function, which is the difference between the action-value and a state-dependent baseline. This reduces variance without changing the expected value of the gradient.
$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)(q_\pi(s,a) - b(s))]$$
- **Update Rule**: The update is based on the advantage estimate:
$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi_\theta(A_t|S_t)(G_t - b(S_t))$$
- A common and effective baseline is a learned value function, $b(S_t) = \hat{v}(S_t, w)$.

8.2 Actor-Critic Methods

8.2.1 Core Concepts

Actor-Critic methods combine a policy network (the **actor**) with a value function network (the **critic**).

- The **actor** is the policy, parameterized by θ, which chooses the action.
- The **critic** is a value function, parameterized by w, which estimates the value of the state or state-action pair.
- Instead of waiting for a full trajectory's return (Monte Carlo), the actor-critic method uses the critic's value estimate to update the actor. This allows for updates at every step, reducing the variance and speeding up learning.

The critic is used to bootstrap, providing a value estimate to guide the actor.

8.2.2 Advantage Actor-Critic

This approach uses the **advantage function** as the critic's signal to the actor. **The advantage function measures how much better an action is compared to the average for that state.**

- **Advantage function**: $A(s,a) = q_\pi(s,a) - v_\pi(s)$.
- **Advantage estimate**: A common way to estimate the advantage is with the Temporal-Difference (TD) error:
$$\hat{A}(S_t, A_t) = R_{t+1} + \gamma \hat{v}_w(S_{t+1}) - \hat{v}_w(S_t)$$
This is an estimate of how much the reward was better (or worse) than expected.
- **Gradient Update**: The policy gradient is then estimated using this advantage signal:
$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(A_t|S_t) \hat{A}(S_t, A_t)$$

8.2.3 (Batch) Advantage Actor-Critic Algorithm

This algorithm trains both the actor and the critic networks simultaneously using a batch of sampled trajectories.

1. **Sample Trajectories**: Collect a set of trajectories $\{\tau_i\}$ using the current policy π_θ.
2. **Update Critic**: The critic network's parameters w are updated to minimize the squared TD-error loss.
$$L(w) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} (R_{t+1}^i + \gamma \hat{v}_w(S_{t+1}^i) - \hat{v}_w(S_t^i))^2$$
The parameters are updated via gradient descent: $w \leftarrow w - \alpha_w \nabla_w L(w)$.
3. **Evaluate Advantage**: For each step in the batch, the advantage estimate is calculated using the updated critic.
$$\hat{A}(S_t^i, A_t^i) = R_{t+1}^i + \gamma \hat{v}_w(S_{t+1}^i) - \hat{v}_w(S_t^i)$$
4. **Update Actor**: The actor network's parameters θ are updated using the batch of advantage estimates.
$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta(A_t^i|S_t^i) \hat{A}(S_t^i, A_t^i)$$
The parameters are updated via gradient ascent: $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$.

8.2.4 Network Designs

- **Two-Network Design**: The actor and critic are two separate neural networks with their own parameters.
- **Shared-Network Design**: A single network is used, with a shared trunk that processes the state input, and two separate heads for outputting the policy (actor) and the value estimate (critic). This can be more parameter-efficient.

9 Dynamic Programming (DP)

9.1 Bellman Equations

- State-value: $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$.
- Action-value: $q_\pi(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')]$.
- Optimal state-value: $v^*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^*(s')]$.
- Optimal action-value: $q^*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q^*(s', a')]$.

9.2 Iterative Policy Evaluation (Prediction)

"If I follow this strategy, how good is each situation I'll encounter?" This process does not alter the policy; it simply evaluates it. The algorithm estimates the "value" (long-term expected reward) of each state under a fixed policy π.

Why is this useful? You need to know if a policy is good before improving it. This is the "prediction" part of RL.

Input: An MDP $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ and a policy π.
Output: v_π.

1. Start with an initial guess. Initialize $v_0(s) = 0$ for all $s \in \mathcal{S}$.
2. Repeat until convergence: $v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$ (Bellman equation).
3. Stop when values stop changing much

9.3 Policy Iteration (Control)

This algorithm finds an optimal policy π^* (the best actions to take in each state) and its corresponding optimal value function V^*. It operates by alternating between evaluating the current policy and improving it, a process known as policy iteration.

Why it is useful: Policy iteration is a control method, not just a prediction one. Its goal is to find the best possible strategy to maximize long-term rewards, making it a foundational algorithm for solving control problems in reinforcement learning.

Input: MDP $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$.
Output: v^*, π^*.

1. Initialize π_0 arbitrarily (e.g., random actions).
2. Policy Evaluation: Compute v_{π_k} using iterative evaluation (section 9.2).
3. Policy Improvement: $\pi_{k+1}(s) = \arg\max_a q_{\pi_k}(s, a) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi_k}(s')]$ (greedy choice based on current v).
4. Repeat until $\pi_{k+1} = \pi_k$.

9.4 Value Iteration (Control)

Similar to Policy Iteration, this algorithm finds an optimal policy π^* and its value function V^*. It combines the evaluation and improvement steps into one by repeatedly applying the Bellman optimality equation to directly compute the optimal value function V^*, from which the optimal policy can be extracted.

Why it is useful Value Iteration is a control method: it not only predicts but also finds the best strategy. It is simpler and often converges faster than Policy Iteration for large problems because it does not require a separate, exhaustive policy evaluation phase – because it combines policy improvement and evaluation into a single update.

- Initialize $v_0(s) = 0$.
- Repeat until convergence: $v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$.
- Once converged, get Optimal policy: $\pi^*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^*(s')]$.

10 Model-Free Prediction

10.1 Monte-Carlo (MC) Learning

- Learn $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$.
- Update: $V(s) \leftarrow V(s) + \alpha(G_t - V(s))$.
- Incremental: $N(s) \leftarrow N(s) + 1$; $V(s) \leftarrow V(s) + \frac{1}{N(s)}(G_t - V(s))$.

10.2 Temporal-Difference (TD(0)) Learning

- Update: $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$.
- TD error: $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$.

10.3 n-Step TD Learning

- n-step return: $G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$.
- Update: $V(S_t) \leftarrow V(S_t) + \alpha(G_{t:t+n} - V(S_t))$.

11 Model-Free Control

We consider a Markov Decision Process (MDP) with a finite state space $\mathcal{S}$, action space $\mathcal{A}$, transition kernel $p(s', r \mid s, a)$, and a discount factor $\gamma \in [0, 1)$. The goal of control is to find an optimal policy $\pi^\star$ that maximizes the expected discounted return. All methods below learn an action-value function $Q(s, a)$ directly from experience without an explicit model of the environment's dynamics.

11.1 On-Policy Monte Carlo (MC) Control

Monte Carlo control improves a policy using complete returns from sampled episodes while following (approximately) the same policy it evaluates.

Policy

We use an ϵ-greedy policy with respect to the current Q-function:

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}(s)|}, & \text{if } a \in \arg\max_b Q(s, b), \\ \dfrac{\epsilon}{|\mathcal{A}(s)|}, & \text{otherwise.} \end{cases}$$

Target

For a time step t within an episode, the return is calculated as:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+1+k},$$

where T is the terminal time step.

Incremental Update

With a constant stepsize $\alpha \in (0, 1]$, the stochastic approximation update is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(G_t - Q(S_t, A_t)\big).$$

(With tabular, first-visit MC you may also use running averages.)

Control Loop (GLIE in practice)

Repeatedly:

1. Generate an episode by following the current ϵ-greedy policy π.
2. For each visited state-action pair (S_t, A_t), compute G_t and update Q as described above.
3. Improve π to be ϵ-greedy with respect to the updated Q.

To ensure convergence (in tabular finite MDPs), use GLIE (greedy in the limit with infinite exploration), e.g., $\epsilon_k \downarrow 0$ but $\sum_k \epsilon_k = \infty$.

Notes

- **Pros:** Unbiased targets (true returns), conceptually simple.
- **Cons:** Requires full episodes; high-variance targets; can be slow for long episodes.

11.2 Sarsa (On-Policy TD Control)

Sarsa learns on-policy using one-step bootstrapping.

Update

At each step $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big).$$

This is a TD(0) update on the action-value function.

Policy for Exploration

The actions A_t and A_{t+1} are both chosen from the same ϵ-greedy policy π with respect to the Q-function (on-policy).

Control Loop

1. Initialize $Q(s, a)$ arbitrarily; choose ϵ, α.
2. For each episode:
   (a) Initialize state S_0, choose action $A_0 \sim \epsilon$-greedy$(Q(S_0, \cdot))$.
   (b) For $t = 0, 1, 2, \ldots$ until terminal:
   - Take action A_t, observe reward R_{t+1} and next state S_{t+1}.
   - Choose next action $A_{t+1} \sim \epsilon$-greedy$(Q(S_{t+1}, \cdot))$.
   - Update $Q(S_t, A_t)$ with the Sarsa rule.

Notes

- The on-policy nature makes the learned Q reflect exploratory actions, which can be safer (e.g., in a cliff-walking environment).
- Bootstrapped targets reduce variance compared to MC but may introduce bias.

11.3 Q-Learning (Off-Policy TD Control)

Q-learning learns the value of the greedy policy while potentially behaving ϵ-greedily for exploration; it is off-policy via a max target.

Update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\Big).$$

Behavior vs. Target Policies

- **Behavior policy:** The policy used for generating experience (e.g., ϵ-greedy w.r.t. Q).
- **Target policy:** The greedy policy with respect to the current Q-function (implicit in the $\max_{a'}$ operator).

Convergence (Tabular)

Under standard assumptions (finite MDP, sufficient exploration, appropriate stepsizes), Q-learning converges to $Q^\star$ with probability 1.

Notes

- It's off-policy and bootstrapped; it can be sensitive to overestimation bias (mitigated by Double Q-learning).
- Often more sample-efficient than MC and is a widely used baseline.

11.4 n-Step Sarsa

n-step Sarsa trades off bias and variance by bootstrapping after n rewards.

Forward n-Step Return

For a transition starting at time t, the truncated return is defined as:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

with the convention that if the episode terminates before $t + n$, then $Q(S_{t+n}, A_{t+n})$ is omitted (or treated as 0).

Update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(G_{t:t+n} - Q(S_t, A_t)\big).$$

Scheduling

At each time t, an update for a prior time step $\tau = t - n + 1$ is performed. This yields a pipeline of delayed updates. As $n \to 1$, we recover Sarsa; as n approaches the episode length, we approach on-policy MC.

Notes

- Larger n: lower bias, higher variance, and more delay.
- Smaller n: higher bias, lower variance, and faster updates.
- Backward-view equivalents use eligibility traces (e.g., Sarsa(λ)) to implement an exponentially weighted average over different values of n.

Practical Considerations (All Methods)

- **Initialization:** Optimistic Q_0 values can encourage exploration.
- **Stepsize:** A constant α is good for nonstationary settings, while a diminishing α_t (e.g., Robbins–Monro) supports convergence proofs.
- **Exploration scheduling:** GLIE strategies (e.g., $\epsilon_t \downarrow 0$) balance exploration and exploitation.
- **Terminal handling:** Set Q-values at terminal states to 0.
- **Continuous actions:** The 'max' operator in Q-learning is difficult; consider actor-critic or deterministic policy gradient methods instead.

Algorithm Boxes (Tabular)

On-Policy MC Control (Every-Visit, Incremental)

1. Initialize $Q(s,a)$ arbitrarily; choose $\epsilon \in (0,1)$, $\alpha \in (0,1]$.
2. Loop for each episode:
   (a) Generate episode $(S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T)$ using an ϵ-greedy policy derived from Q.
   (b) For $t = 0, \ldots, T-1$:

$$G_t \leftarrow \sum_{k=0}^{T-t-1} \gamma^k R_{t+1+k}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(G_t - Q(S_t, A_t)\big).$$

Sarsa (On-Policy TD(0) Control)

1. Initialize $Q(s,a)$ arbitrarily; choose ϵ, α.
2. Loop for each episode:
   (a) Initialize state S_0, choose action $A_0 \sim \epsilon$-greedy$(Q(S_0, \cdot))$.
   (b) For $t = 0, 1, 2, \ldots$ until terminal:
   - Take A_t, observe R_{t+1}, S_{t+1}.
   - Choose $A_{t+1} \sim \epsilon$-greedy$(Q(S_{t+1}, \cdot))$.
   - Update $Q(S_t, A_t)$ using the Sarsa rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big).$$

Q-Learning (Off-Policy TD Control)

1. Initialize $Q(s,a)$ arbitrarily; choose ϵ, α.
2. Loop for each episode:
   (a) Initialize state S_0.
   (b) For $t = 0, 1, 2, \ldots$ until terminal:
   - Choose $A_t \sim \epsilon$-greedy$(Q(S_t, \cdot))$.
   - Take A_t, observe R_{t+1}, S_{t+1}.
   - Update $Q(S_t, A_t)$ using the Q-Learning rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\big).$$

n-Step Sarsa (Forward View)

1. Initialize $Q(s,a)$ arbitrarily; choose $\epsilon, \alpha, n \geq 1$.
2. Loop for each episode:
   (a) Generate a trajectory $(S_0, A_0, R_1, S_1, A_1, \dots)$ using an ϵ-greedy policy derived from Q.
   (b) For $t = 0, 1, 2, \dots$ until all updates applied:
   - Let $\tau \leftarrow t - n + 1$.
   - If $\tau \geq 0$, define the n-step return:

$$G_{\tau:\tau+n} = \sum_{k=0}^{n-1} \gamma^k R_{\tau+1+k} + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$$

   (with truncation at terminal states), and update:

$$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\big(G_{\tau:\tau+n} - Q(S_\tau, A_\tau)\big).$$

Conceptual Differences Summary

- **Target:** MC uses the full return G_t; Sarsa uses a one-step bootstrapped target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$; Q-learning uses a max-operator bootstrapped target $R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$; n-step Sarsa uses an intermediate n-step return $G_{t:t+n}$.
- **On- vs Off-Policy:** MC control and Sarsa are on-policy; Q-learning is off-policy.
- **Bias/Variance:** MC has low bias but high variance; TD methods have more bias but lower variance. The n-step methods provide a spectrum to balance this trade-off by adjusting the step size n.

12 Deep Q-Networks (DQN)

12.1 Deep Q-Learning

- Represent $Q(s, a; w) \approx q^*(s, a)$.
- Loss: $L(w) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i; w))^2$, where $y_i = r_i + \gamma \max_{a'} Q(s_i', a'; w^-)$.
- Update w via stochastic gradient descent (SGD).

12.2 DQN Algorithm (with Experience Replay and Target Network)

1. Initialize replay buffer $\mathcal{B}$; initialize online Q-network $Q(s, a; w)$; initialize target network $Q(s, a; w^-) \leftarrow Q(s, a; w)$.
2. For episode $= 1, \dots$:
   (a) Reset env, receive initial state s_0; set $t \leftarrow 0$.
   (b) While not done:
   i. Select a_t via ϵ-greedy from $Q(\cdot; w)$ given s_t.
   ii. Execute a_t, observe r_{t+1}, next state s_{t+1}, and done flag $d_{t+1} \in \{0, 1\}$.
   iii. Store $(s_t, a_t, r_{t+1}, s_{t+1}, d_{t+1})$ in $\mathcal{B}$.
   iv. Sample a mini-batch $\{(s_i, a_i, r_i, s_i', d_i)\}_{i=1}^N$ from $\mathcal{B}$.
   v. Compute targets: $y_i = r_i + \gamma(1 - d_i) \max_{a'} Q(s_i', a'; w^-)$.
   vi. Update w by minimizing $L(w) = \frac{1}{N} \sum_i \big(y_i - Q(s_i, a_i; w)\big)^2$ (often Huber loss).
   vii. Every K steps: update target, e.g., hard $w^- \leftarrow w$ or soft $w^- \leftarrow \tau w^- + (1 - \tau) w$.
   viii. Optionally anneal ϵ; $t \leftarrow t + 1$; $s_t \leftarrow s_{t+1}$.

12.3 Improvements to DQN

- **Double DQN**: Target $y_i = r_i + \gamma Q(s'_i, \arg\max_{a'} Q(s'_i, a'; w), w^-)$ to reduce overestimation.
- **Prioritized Replay**: Sample transitions with priority $p_i \propto |\delta_i| + \epsilon$; update with importance sampling weights.
- **Multi-step Returns**: $y_i = \sum_{k=1}^{n} \gamma^{k-1} r_{i+k} + \gamma^n \max_{a'} Q(s'_{i+n}, a'; w^-)$.

13 Continuous Control with Q-Learning

13.1 Action Discretization

- Discretize continuous actions into bins (e.g., 10 bins per dimension).

13.2 Stochastic Optimization (CEM)

1. Sample M actions from distribution $p(a)$ (e.g., Gaussian $\mathcal{N}(\mu, \sigma^2)$).
2. Evaluate $Q(s, a_i)$ for each.
3. Select top K elites.
4. Refit $p(a)$ to elites (e.g., update μ, σ).
5. Best action: $\arg\max_a Q(s, a) \approx$ elite mean or max.

13.3 Deep Deterministic Policy Gradient (DDPG)

- Actor: $\mu(s; \theta) \approx \arg\max_a Q(s, a; w)$.
- Critic update: $L(w) = \frac{1}{N} \sum_i (y_i - Q(s_i, \mu(s_i; \theta); w))^2$, where $y_i = r_i + \gamma Q(s'_i; \mu(s'_i; \theta^-); w^-)$.
- Actor update: Maximize $J(\theta) = \frac{1}{N} \sum_i Q(s_i, \mu(s_i; \theta); w)$.
- Exploration: $a_t = \mu(s_t; \theta) + \mathcal{N}_t$.
- Soft target updates: $\theta^- \leftarrow \tau\theta^- + (1 - \tau)\theta$, and similarly for w^-.

14 Offline Reinforcement Learning

14.1 What is Offline RL?

Offline RL (also known as batch RL or fully off-policy RL) is a subfield of reinforcement learning where the goal is to learn an optimal policy π from a fixed, pre-collected dataset $D = \{(s_i, a_i, s'_i, r_i)\}$ without any further interaction with the environment. The dataset can be collected by any behavior policy π_β (e.g., a random policy, an expert human, or a series of past RL runs).

The objective is to find a policy π that maximizes the expected cumulative reward:

$$\max_\pi \sum_{t=0}^{T} \mathbb{E}_{s_t \sim d^\pi, a_t \sim \pi(a|s)}[r(s_t, a_t)]$$

Once learned, the policy is deployed in the real environment. A related task is **Off-Policy Evaluation (OPE)**, which aims to estimate the value of a given policy π from the offline dataset D, i.e., $J(\pi) = \mathbb{E}_\pi[\sum_{t=1}^{T} r(s_t, a_t)]$.

14.2 How is Offline RL Possible?

Offline RL can be effective by:

- **Extracting good behaviors**: Learning to distinguish between high-reward and low-reward behaviors within a mixed-quality dataset.
- **Generalization**: A good action observed in one state can inform the model about good actions in similar, unseen states.
- **Stitching**: The algorithm can recombine segments of different trajectories from the dataset to form a new, superior policy. This is sometimes more powerful than imitation learning, as it can create behaviors that were never seen in their entirety.

A key example is offline QT-Opt for robotic grasping, which achieved a high success rate by learning from a large, fixed dataset of grasping attempts.

14.3 Why is Offline RL Hard?

The primary challenge in offline RL is the **distribution shift** between the learned policy π and the data-collecting behavior policy π_β.

- **Counterfactual queries**: The learned policy may select **out-of-distribution (OOD)** actions that are not present in the dataset. Since the value function is only trained on in-distribution actions, it may produce wildly inaccurate and overly optimistic Q-value estimates for these OOD actions.
- **Generalization issues**: Function approximation, especially with deep neural networks, can amplify these errors, leading to a policy that learns to exploit these erroneous Q-values and performs poorly in the real environment. This is a common failure mode, where naive applications of off-policy algorithms like DQN can lead to a significant overestimation of Q-values on offline data.

14.4 Policy Constraint Methods

One way to mitigate distribution shift is to constrain the learned policy to stay close to the behavior policy. This can be formalized as:

$$\pi(a|s) = \arg\max_{\pi} \mathbb{E}[Q(s,a)] \text{ s.t. } D_{KL}(\pi||\pi_\beta) \leq \epsilon$$

This approach can be difficult because the behavior policy π_β is often unknown. Moreover, if the constraint is too strict, the policy may not be able to improve, and if it's too loose, the distribution shift problem persists.

14.5 Conservative Q-Learning (CQL)

Conservative Q-Learning (CQL) is a method that directly addresses the overestimation problem by modifying the Q-learning objective. Its goal is to **conservatively lower the Q-values for OOD actions** while keeping the Q-values for in-distribution actions high. The CQL loss function is:

$$L_{CQL}(Q) = \alpha \left(\mathbb{E}_{s\sim D, a\sim\mu(a|s)}[Q(s,a)] - \mathbb{E}_{(s,a)\sim D}[Q(s,a)]\right) + \mathbb{E}_{(s,a,s')\sim D}\left[\left(Q(s,a) - \left(r(s,a) + \gamma \mathbb{E}_{a'\sim\pi(a'|s')}[Q(s',a')]\right)\right)^2\right]$$

The first term is the key addition: it pushes down the Q-values on a set of sampled OOD actions (e.g., from a uniform distribution μ) while pushing up the Q-values for actions present in the dataset.

14.5.1 CQL Algorithm Pseudocode

1. Initialize the Q-network $Q(s, a; w)$, policy network $\pi(a|s; \theta)$, and populate a replay buffer with the offline dataset D.
2. For each training step:
3. Sample a mini-batch of transitions from D.
4. Update the Q-network parameters w using a stochastic gradient descent (SGD) step on the $L_{CQL}(Q)$ loss.
5. Update the policy parameters θ:
   - **For discrete actions**: The policy is updated to be greedy with respect to the learned Q-values: $\pi(a|s) = \arg\max_{a'} Q(s, a')$.
   - **For continuous actions**: The policy is updated to maximize the expected Q-value: $\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}_{a \sim \pi(a|s)}[Q(s, a)]$.

14.6 Additional Methods

- **Implicit Q-Learning (IQL)**: A method that avoids explicitly querying OOD actions by using an expectile regression loss.
- **Advantage-Weighted Actor-Critic (AWAC)**: An algorithm that performs policy updates by weighting offline samples based on their estimated advantage.
- **Model-Based Methods**: Algorithms like MOPO (Model-based Offline Policy Optimization) and COMBO (Conservative Model-Based Optimization) learn a world model from the offline data and use it to generate synthetic transitions, while often incorporating a conservative penalty to avoid model-exploitation.
- **Sequence Modeling**: Methods like the Trajectory Transformer treat RL as a sequence modeling problem, learning to predict future returns and actions from past trajectories.

15 Actor-Critic Methods - Part 3

15.1 Batch Advantage Actor-Critic (A2C)

A2C is a stable on-policy method that reduces variance compared to REINFORCE by using a learned value function to estimate the advantage.

- **Advantage function**: The advantage is estimated using the TD-error:

$$\hat{A}(S_t, A_t) = R_{t+1} + \gamma \hat{v}_w(S_{t+1}) - \hat{v}_w(S_t)$$

15.1.1 Algorithm

1. Sample a batch of trajectories $\{\tau_i\}$ using the current policy $\pi_\theta(a|s)$.
2. **Critic Update**: Update the critic network parameters w to minimize the Mean Squared Error (MSE) loss on the TD-error:

$$L(w) = \frac{1}{N} \sum_i \sum_t \left(R^i_{t+1} + \gamma \hat{v}_w(S^i_{t+1}) - \hat{v}_w(S^i_t) \right)^2$$

$$w \leftarrow w - \alpha_w \nabla_w L(w)$$

3. **Advantage Evaluation**: Calculate the advantage estimate for each time step in the batch.

4. **Actor Update**: Update the actor network parameters θ using the advantage-weighted policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \sum_t \nabla_\theta \log \pi_\theta(A_t^i | S_t^i) \hat{A}(S_t^i, A_t^i)$$

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$$

15.2 Trust Region Policy Optimization (TRPO)

TRPO is an on-policy method that ensures policy updates are small and stable by introducing a trust region constraint. It maximizes a surrogate objective (true objective) function subject to a KL-divergence constraint on the policy change.

$$\max_\theta \mathbb{E}_{\pi_{\theta_{old}}} \left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)} A_{\pi_{\theta_{old}}}(S_t, A_t) \right]$$

$$\text{s.t. } \mathbb{E}_{\pi_{old}}[D_{KL}(\pi_{old} || \pi_\theta)] \leq \delta$$

This method is complex to implement due to the second-order optimization required to handle the constraint.

15.3 Proximal Policy Optimization (PPO)

PPO is a popular, simpler alternative to TRPO that uses a clipped surrogate objective to achieve a similar effect of constraining policy updates.

$$\max_\theta \mathbb{E}_{\pi_{\theta_{old}}} \left[\min \left(r_t(\theta) \hat{A}_t, (r_t(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$ is the probability ratio and $\hat{A}_t$ is the advantage estimate. The 'clip' function prevents the policy from making excessively large changes, thereby avoiding catastrophic updates.

15.3.1 PPO Algorithm

The PPO algorithm is similar to Batch A2C but uses the PPO clipped objective for the actor update.

1. Sample trajectories $\{\tau_i\}$ from the current policy $\pi_{\theta_{old}}$.
2. Update the critic network (as in A2C).
3. Evaluate the advantage function (as in A2C).
4. Update the actor network for multiple epochs using the PPO objective on the same batch of data.
5. Set the old policy parameters: $\theta_{old} \leftarrow \theta$.

16 Off-Policy Actor-Critic Methods

16.1 Online Actor-Critic (On-Policy)

The basic actor-critic algorithm updates after each single time step. While simple, it has issues with instability when used with deep neural networks.

1. Take action $A_t \sim \pi_\theta(\cdot|S_t)$, observe $S_t, A_t, R_{t+1}, S_{t+1}$.
2. **Critic Update**: Update the value function $\hat{v}_w$ towards the target $y_t = R_{t+1} + \gamma \hat{v}_w(S_{t+1})$.
3. **Actor Update**: Update the policy π_θ using the TD-error as the advantage signal: $\hat{A}(S_t, A_t) = y_t - \hat{v}_w(S_t)$.

16.2 Off-Policy Actor-Critic

This is a more modern approach that combines the benefits of actor-critic methods with a replay buffer, allowing it to be off-policy. To support off-policy learning, it typically uses a Q-function instead of a V-function.

1. Take action A_t from the current policy π_θ and store the transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in a replay buffer $\mathcal{B}$.
2. Sample a mini-batch of transitions from $\mathcal{B}$.
3. **Critic Update**: Update the critic's Q-network $\hat{q}_w$ towards a target value y_i.

   $$y_i = R_{i+1} + \gamma \hat{q}_{w^-}(S_{i+1}, A_{i+1}^{\pi_\theta})$$

   where $A_{i+1}^{\pi_\theta} \sim \pi_\theta(\cdot|S_{i+1})$ is a sampled action from the current policy.
4. **Actor Update**: Update the actor π_θ to maximize the Q-values.

   $$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(A_i^{\pi_\theta}|S_i) \hat{q}_w(S_i, A_i^{\pi_\theta})$$

5. **Target Network Update**: The target critic network w^- is updated slowly using a soft update: $w^- \leftarrow \tau w^- + (1-\tau)w$.

16.3 Deep Deterministic Policy Gradient (DDPG)

DDPG is an off-policy, actor-critic algorithm for continuous action spaces. It uses a deterministic policy $\mu_\theta(s)$ for acting, with added noise for exploration, $A_t = \mu_\theta(S_t) + \mathcal{N}_t$.

1. Store transitions in a replay buffer $\mathcal{B}$.
2. Sample a mini-batch from $\mathcal{B}$.
3. **Critic Update**: The critic network is updated towards a target y_i that uses a target policy μ_{θ^-} and a target critic network w^- to compute the next state's value.

   $$y_i = R_{i+1} + \gamma \hat{q}_{w^-}(S_{i+1}, \mu_{\theta^-}(S_{i+1}))$$

4. **Actor Update**: The actor is updated by maximizing the expected Q-value, essentially performing gradient ascent on the Q-function.

   $$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \hat{q}_w(S_i, \mu_\theta(S_i))$$

5. **Soft Target Update**: The target networks θ^- and w^- are updated softly.

17 Soft Actor-Critic (SAC)

17.1 Maximum Entropy Reinforcement Learning

SAC is an off-policy algorithm that aims to find a policy that not only maximizes the reward but also has high entropy. This encourages exploration and leads to more robust, stable policies. The objective function is augmented with an entropy term:

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{\rho_\pi}[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))]$$

where $\mathcal{H}(\pi(\cdot|s_t)) = \mathbb{E}_{a_t \sim \pi}[-\log \pi(a_t|s_t)]$ is the policy's entropy and α is the temperature parameter that controls the trade-off between reward and entropy.

17.2 Soft Value Functions

The Bellman equations are modified to include the entropy term, resulting in "soft" value functions.

- **Soft state-value**:
$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi}[Q^\pi(s_t, a_t) - \alpha \log \pi(a_t|s_t)]$$
- **Soft action-value**:
$$Q^\pi(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}}[V^\pi(s_{t+1})]$$

17.3 SAC Algorithm

SAC uses a stochastic policy and two Q-networks to mitigate overestimation bias. It is particularly effective for continuous control tasks.

1. Sample action A_t from the policy π_θ, execute it in the environment, and store the transition in a replay buffer D.
2. Sample a mini-batch of transitions from D.
3. **Critic Update**: Update both Q-networks to minimize the MSE loss against a soft Q-target, which is constructed using the minimum of the two Q-networks to reduce overestimation.
$$J_Q(\phi) = \mathbb{E}_{(s,a,s')\sim D}\left[\left(Q_\phi(s,a) - \left(r + \gamma\left(\min_{j=1,2} Q_{\phi_j^-}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right)\right)\right)^2\right]$$
where $\tilde{a}'$ is a sampled action from the current policy.
4. **Actor Update**: Update the actor to minimize the expected Kullback-Leibler (KL) divergence to the exponentiated Q-function, which simplifies to maximizing the entropy-regularized objective.
$$J_\pi(\theta) = \mathbb{E}_{s\sim D, a\sim\pi_\theta}[\alpha \log \pi_\theta(a|s) - Q_\phi(s,a)]$$
For continuous actions, a **reparameterization trick** is used to make this differentiable.
5. Update the temperature parameter α (optional) and perform soft updates on the target networks.

18 Model-Based Reinforcement Learning

18.1 Overview

Model-based RL is a paradigm where an agent learns a model of the environment's dynamics from its experience. This model is then used for planning to inform and construct better policies or value functions. This contrasts with model-free RL, which learns policies and value functions directly from experience without explicitly learning a model.

18.1.1 Model-Free vs. Model-Based

- **Model-Free RL**: The agent learns a policy $\pi(a|s)$ or a value function $Q(s,a)$ directly from interactions with the environment. Examples include DQN and Policy Gradient methods.
- **Model-Based RL**: The agent learns a dynamics model $p(s', r|s, a)$ that predicts the next state s' and reward r given the current state s and action a. This model is then used to simulate new experiences and improve the policy.

18.2 Advantages and Disadvantages

18.2.1 Advantages

- **High Sample Efficiency**: By simulating experience from the learned model, the agent can learn a good policy with far fewer real-world interactions.
- **Rapid Adaptation**: The model can be quickly updated with new data, allowing the agent to adapt to changes in the environment or new tasks.
- **Efficient Model Learning**: Learning a predictive model is a supervised learning problem, which is often more stable and data-efficient than learning a policy via trial and error.

18.2.2 Disadvantages

- **Two Sources of Error**: Performance is sensitive to both the accuracy of the learned model and the effectiveness of the planning algorithm used with the model. Errors can compound over long planning horizons.

19 Model Learning

Learning a dynamics model is a supervised learning task. Given a dataset of transitions $\mathcal{D} = \{(s_i, a_i, r_{i+1}, s'_{i+1})\}$, the model learns to predict the next state and reward.

- **Reward Model**: A regression model predicts the reward, $r_{i+1} = f_{r\eta}(s_i, a_i)$.
- **State Transition Model**: A regression or density estimation model predicts the next state, $s'_{i+1} \sim f_{s\eta}(s_i, a_i)$.

19.1 Types of Dynamics Models

- **Deterministic Models**: The model outputs a single predicted next state, $s'_{i+1} = f_{s\eta}(s_i, a_i)$. This is suitable for environments with low stochasticity.
- **Stochastic Models**: The model outputs a probability distribution over the next state, from which a sample can be drawn. This is more robust for uncertain environments.

20 Combining Model-Free RL with a Model

A common approach is to use a learned model to generate synthetic data, which is then used to train a separate model-free RL algorithm. This is a powerful way to boost sample efficiency. However, a major challenge is **distribution shift**: the model's accuracy is limited to the states it has seen, and long synthetic rollouts can accumulate prediction errors, leading to a "bad" policy.

20.1 Algorithm (v1 - Short Rollouts)

1. Run the current policy $\pi_\theta(a|s)$ in the real environment to collect a dataset $\mathcal{D}_{env}$.
2. Learn a dynamics model $f_\eta(s, a)$ from $\mathcal{D}_{env}$.
3. For a number of steps, select states s_i from $\mathcal{D}_{env}$ and use the learned model f_η and policy π_θ to generate short, synthetic rollouts. Add these to a model-generated dataset $\mathcal{D}_{model}$.
4. Use an off-policy RL algorithm (e.g., SAC, DQN) to improve the policy π_θ on the combined dataset.
5. Repeat the process by running the improved policy in the real environment.

20.2 Dyna-Q Style Algorithm

This algorithm uses a model to supplement real-world experience, often in a continuous loop.

1. In the real environment, take an action $A_t \sim \pi_\theta$, and store the transition in a replay buffer $\mathcal{B}_{env}$.
2. Use the data in $\mathcal{B}_{env}$ to update the dynamics model f_η.
3. Perform a small number of model-based planning steps:
   - Sample a state s from $\mathcal{B}_{env}$.
   - Sample an action a (e.g., from a uniform distribution or a policy).
   - Use the model to predict the next state and reward: $s', r \sim f_\eta(s, a)$.
   - Add the synthetic transition (s, a, r, s') to a model-generated buffer $\mathcal{B}_{model}$.
4. Update the model-free components (Q/Critic/Actor) using a combination of transitions from $\mathcal{B}_{env}$ and $\mathcal{B}_{model}$.

This approach uses short rollouts from the model to avoid accumulating errors and combines real data to anchor learning, leading to more stable performance.

21 Direct Policy Learning with a Model

This approach directly optimizes the policy by backpropagating gradients through the learned dynamics model. The key idea is to treat the environment model as part of the computational graph.

1. Run the policy π_θ in the real environment to collect a dataset $\mathcal{D}_{env}$.
2. Learn a dynamics model $f_\eta(s, a)$ from $\mathcal{D}_{env}$.
3. To improve the policy, unroll the policy's actions through the model's dynamics for a few steps. The reward from this unrolling is used to compute a loss.
4. Backpropagate the gradients of this loss through the dynamics model and into the policy network to update the policy parameters θ.
5. Repeat the process by running the updated policy in the real environment to collect new data.

This method can be very sample-efficient, but it can suffer from issues like vanishing or exploding gradients when unrolling over many time steps, and its effectiveness is highly dependent on the accuracy of the learned model.

22 Model-Based Reinforcement Learning - Part 2

22.1 Recap: Model-Based vs. Model-Free RL

22.1.1 Core Concepts

- **Model-Free RL**: The agent learns a policy or value function directly from interactions with the environment, without explicitly learning a model of the environment's dynamics.
- **Model-Based RL**: The agent learns a dynamics model $p(s', r|s, a)$ from data and uses this model for planning to construct or improve the policy. This approach integrates learning (of the model) and planning.

22.1.2 Key Trade-offs

- **Advantages**: Model-based methods are known for their high sample efficiency (they require fewer real-world interactions), rapid adaptation to environment changes, and the ability to handle uncertainty by planning with a probabilistic model.
- **Disadvantages**: The main challenge is the presence of two sources of error: model approximation error and error in the value or policy construction from the model's output.

22.2 Background Planning (Model-Free RL with a Model)

This approach uses a learned model to generate synthetic experience, which is then used to train a model-free algorithm. A key challenge is **distribution shif**t, where long simulated rollouts can accumulate errors, leading to a poor policy. This is mitigated by using short rollouts and regularly collecting new data from the real environment.

22.2.1 Algorithm (Short Rollouts)

1. Run the current policy $\pi_0(a|s)$ in the real environment to collect a dataset of transitions $\mathcal{D}_{env}$.
2. Learn a dynamics model $f_\eta(s, a)$ from $\mathcal{D}_{env}$.
3. Select states s_i from $\mathcal{D}_{env}$ and use the learned model f_η to generate short rollouts of synthetic transitions. Add these to a model-generated dataset $\mathcal{D}_{model}$.
4. Improve the policy π_θ using an off-policy RL algorithm (e.g., SAC) on the combined real and synthetic data.
5. Repeat the process by executing the updated policy in the real environment.

22.3 Direct Policy Learning with a Model

This method directly optimizes the policy by backpropagating the gradient of a performance objective through the learned dynamics model.

- The policy and model are treated as a single computational graph.
- The algorithm unrolls the policy's actions through the model to compute a simulated reward, and then backpropagates gradients of this reward to update the policy.

While potentially very sample-efficient, this approach is susceptible to vanishing or exploding gradients and relies heavily on the accuracy of the learned model.

22.4 Decision-Time Planning

In contrast to background planning, which improves the policy over time, decision-time planning uses the model to select the best action for the current state. The model is essentially used as a simulator to find an optimal action sequence. The goal is to find an action sequence that maximizes the sum of future rewards:

$$a_1, \ldots, a_T = \arg\max_{a_1,\ldots,a_T} \sum_{t=1}^{T} \gamma^{t-1} r_\eta(s_t, a_t) \quad \text{s.t.} \quad s_{t+1} = f_\eta(s_t, a_t)$$

22.4.1 Stochastic Optimization

- **Random Shooting Method**: This method samples a large number of random action sequences and evaluates each one using the learned model. The first action of the best-performing sequence is then executed. This is simple but can be inefficient in high-dimensional or long-horizon problems.
- **Cross-Entropy Method (CEM)**: CEM is an iterative optimization algorithm that refines a distribution of action sequences.
    1. Sample a number of action sequences from a distribution (e.g., a Gaussian).
    2. Evaluate the performance of each sequence using the model.
    3. Select the top-performing "elite" sequences.
    4. Fit a new distribution to these elite sequences to generate the next batch of samples.
    5. Repeat until convergence.

22.4.2 Monte Carlo Tree Search (MCTS)

MCTS is a planning algorithm that explores a search tree to find the best action. It is commonly used in game-playing (e.g., AlphaGo).

1. **Selection**: Starting from the root, traverse the tree by choosing actions that balance exploration (visiting less-explored nodes) and exploitation (choosing nodes with high value estimates) until a leaf node s_L is reached. A common selection strategy is the UCT formula:

$$Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$$

2. **Expansion**: If the selected leaf node has not been fully expanded, create a new child node for an unvisited action.
3. **Simulation**: From the new leaf node, run a simulation (a "rollout") using a fast default policy until a terminal state is reached.
4. **Backpropagation**: The result of the simulation is used to update the value estimates and visit counts of all nodes from the leaf back to the root.

22.5 Model-Based RL with Decision-Time Planning

This approach, often called Model Predictive Control (MPC), combines a learned model with decision-time planning.

1. Collect data using an initial policy and learn a dynamics model $f_\eta(s,a)$.
2. For a given state, use the learned model and a planning algorithm (e.g., CEM or MCTS) to find the best sequence of actions.
3. Execute only the first action of this sequence in the real environment.
4. Observe the next state, add the transition to the dataset, and repeat the process for the new state.

This is a robust method because it uses the model for planning, but only commits to the first action, allowing it to replan at every step with new, real-world information.

23 Safe Reinforcement Learning

23.1 Overview of Safety Problems

Safe Reinforcement Learning (Safe RL) is a field dedicated to ensuring that an agent's behavior adheres to constraints and avoids undesirable, potentially harmful, outcomes. Standard RL aims to maximize a single reward function, which may not be sufficient to guarantee safety. Key issues in Safe RL include:

- **Optimal but Unsafe Behavior**: An optimal policy can lead to high rewards but also high-risk actions that should be avoided.
- **Safe Exploration**: The agent must act safely not only at convergence but also during the training process, where it actively explores the environment.
- **Irreversible States**: Preventing the agent from entering states from which a return to a safe state is impossible.

23.2 Constrained Markov Decision Processes (CMDP)

To formalize safety, Safe RL often uses the Constrained Markov Decision Process (CMDP) framework. A CMDP extends a standard MDP by adding one or more cost functions and associated thresholds.

- **Classic MDP**: $\langle S, A, P, R, \gamma \rangle$, where the objective is to maximize $V(s_0) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$.
- **CMDP**: A CMDP adds a cost function $C : S \times A \to \mathbb{R}$ and a set of thresholds b_i. The objective is to maximize the expected return subject to a set of constraints on the expected cumulative cost.

23.2.1 Optimally Safe Policies

An optimally safe policy π_c^* is one that achieves the highest possible return while satisfying all safety constraints.

$$\pi_c^* = \arg\max_{\pi_c \in \Pi_c} V(s_0)$$

where Π_c is the set of all policies that satisfy the constraints. These constraints can be on the expected total cost or on the probability of a catastrophic event.

- **Expected Safety**: The expected cumulative cost must be below a threshold:

$$f_c(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t C(s_t, a_t)\right] \leq b_i$$

- **Almost Surely Safe**: The probability of the cumulative cost exceeding the threshold is zero.

$$P_\pi \left(\sum_{t=0}^{\infty} \gamma^t C(s_t, a_t) \leq b_i\right) = 1$$

23.3 Solving CMDPs: Methods

A variety of methods exist to solve CMDPs, including policy constraint algorithms, reward shaping, and modified action selection strategies.

23.3.1 Constrained Policy Optimization (CPO)

CPO is an extension of the TRPO algorithm that adds a safety constraint. It finds a policy update that maximizes the objective while ensuring the KL divergence from the old policy and the cumulative cost are within specified bounds.

$$\max_\theta \mathbb{E}_{\pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\pi_{\theta_{\text{old}}}}(s, a)\right]$$

$$\text{s.t. } \mathbb{E}_{\pi_{\theta_{\text{old}}}}[D_{KL}(\pi_{\theta_{\text{old}}} || \pi_\theta)] \leq \delta \quad \text{and} \quad \mathbb{E}_{\pi_{\theta_{\text{old}}}}[C(s, a)] \leq b_i$$

23.3.2 Primal-Dual Methods (Lagrangian)

Lagrangian methods convert a constrained optimization problem into an unconstrained one by introducing a Lagrange multiplier λ for each constraint. The objective is to find a saddle point of the Lagrangian function.

$$\min_{\lambda} \max_{\theta} \mathcal{L}(\theta, \lambda) = \mathbb{E}_{\pi}[R(s,a)] - \lambda \left(\mathbb{E}_{\pi}[C(s,a)] - b\right)$$

The method iteratively updates the policy to maximize this objective and the multiplier to enforce the constraint.

1. Initialize policy parameters θ and Lagrange multiplier $\lambda \geq 0$.
2. For each iteration, compute policy and value losses. The actor loss is augmented with the cost term:
$$\mathcal{L}_{\theta} = -\mathbb{E}_{\pi}[\log \pi_{\theta}(a|s) \cdot A] - \lambda \left(\mathbb{E}_{\pi}[C(s,a)] - b\right)$$
3. Update θ via a gradient step on $\mathcal{L}_{\theta}$.
4. Update the Lagrange multiplier: $\lambda \leftarrow \lambda + \alpha \left(\mathbb{E}_{\pi}[C(s,a)] - b\right)$. This increases λ if the cost constraint is violated, penalizing future cost increases.

23.3.3 State Augmentation (SauteRL)

This method reformulates a CMDP as a standard MDP by augmenting the state with a "safety budget." The budget decreases with each cost incurred.

- **Augmented State**: The state becomes $\tilde{S} = S \times Z$, where Z is a variable that tracks the remaining safety budget.
- **Modified Reward**: The agent receives the original reward $R(s,a)$ only if the safety budget is non-negative ($z \geq 0$). Otherwise, the reward is zero.
- **Budget Update**: The budget is updated at each step: $z_{t+1} = z_t - f_c(s_t, a_t)$, with an initial budget $z_0 = b_i$.

A standard RL algorithm (e.g., PPO or SAC) can then be used to solve this augmented MDP. This allows the policy to generalize across different constraints by simply changing the initial budget.

23.3.4 Action Selection (MASE)

The Meta-Algorithm for Safe Exploration (MASE) is a high-level approach that guides exploration to prevent unsafe actions. It assumes a safety margin and an emergency stop action.

- **Uncertainty Quantification**: A model estimates the cost of an action and its uncertainty, $\Gamma(s,a)$.
- **Safe Actions**: A set of "safe" actions is defined as those where the estimated cost plus uncertainty does not exceed the safety threshold, i.e., $A^{+} = \{a \mid \mu(s,a) + \Gamma(s,a) \leq b_i\}$.
- **Exploration**: The agent's policy is restricted to choosing actions from the safe set A^{+}. If no safe actions are available, an emergency action (e.g., stopping) is taken.

This ensures the agent acts safely during training while learning a policy that is also safe at convergence.

23.4 Frontiers in Safe RL

Safe RL is a rapidly evolving field with ongoing research in several key areas:

- **Safe Offline RL**: Learning safe policies from a fixed dataset without further interactions.
- **Safe Model-Based RL**: Using a learned model to plan for safe, long-horizon behaviors.
- **Safe Exploration**: Developing methods to learn and explore safely in the real world.
- **Reward Hacking**: Preventing agents from finding unintended ways to get high rewards that violate human-defined safety principles.

24 Exploration in RL

24.1 Random Exploration

Random exploration is a fundamental strategy for balancing the trade-off between **exploration** (trying new actions to find better policies) and **exploitation** (choosing the best-known action).

- **ϵ-Greedy**: This is a simple but effective strategy where, with a probability of $1 - \epsilon$, the agent selects the action with the highest estimated value (Q-value). With probability ϵ, it chooses a random action. The value of ϵ is often decayed over time to shift the balance from exploration to exploitation as the agent learns more about the environment.
- **In Deep RL**: For continuous action spaces, exploration can be achieved by adding noise (e.g., Gaussian noise) to the agent's deterministic policy output. For discrete actions, a softmax function over the Q-values or an entropy regularization term in the policy's objective can be used to encourage randomness.

24.2 Novelty Seeking Exploration

Novelty-seeking methods provide an **intrinsic reward** to the agent for visiting new or less-explored states. This encourages the agent to actively seek out new information.

- **Count-Based Exploration**: A bonus reward is given for visiting a state. The bonus is inversely proportional to the number of times the state has been visited. For example, the bonus could be $B(N(s,a)) = 1/\sqrt{N(s,a)}$, where $N(s,a)$ is the visit count for a state-action pair.
- **Pseudo-Counts**: For environments with large or continuous state spaces, it's not feasible to keep a simple count. Pseudo-count methods use a density model to approximate how many times a state has been visited. The bonus is based on the novelty of a state as measured by this model.
- **Prediction Errors**: The agent's bonus reward is proportional to the error of a predictive model (e.g., a neural network) trying to predict the next state. High prediction error suggests a novel state, encouraging the agent to explore it further.

24.3 Posterior Sampling Methods

These methods, also known as Bayesian exploration, model the uncertainty in the value function or model of the environment.

- **Upper Confidence Bound (UCB)**: An optimistic exploration strategy that adds a bonus to the Q-value based on the uncertainty of the action's value. The bonus term, $c\sqrt{\ln t / N_t(a)}$, encourages the agent to choose actions with high estimated values and actions that have been tried less often ($N_t(a)$ is low).
- **Bootstrapped DQN**: An ensemble of K Q-networks is trained on bootstrapped samples from the replay buffer. At the start of each episode, one Q-network is randomly selected, and the agent follows its greedy policy. This method effectively samples from a posterior distribution over Q-functions.

25 Transfer Learning in RL

Transfer learning aims to leverage knowledge from a source task to accelerate learning in a new, target task.

25.1 Domain Adaptation

Domain adaptation addresses the challenge of transferring a policy from a source environment (e.g., a simulator) to a target environment (e.g., the real world) where the observations or dynamics may be different.

- **Observation Adaptation**: This involves learning a representation of the state that is invariant to the domain (e.g., ignoring visual differences between a simulated and real robot arm). Adversarial methods are a common approach, where a domain classifier is trained to distinguish between source and target observations, while an encoder is simultaneously trained to fool the classifier, thus producing a domain-invariant representation.
- **Dynamics Adaptation**: When the physics of the two environments differ, the dynamics model can be adapted. This can be done by using a penalty on transitions that are unlikely in the target environment.

25.2 Domain Randomization

Instead of trying to match the source domain to the target, domain randomization varies the source domain's parameters (e.g., friction, lighting, textures) to such a degree that the learned policy becomes robust enough to generalize to the real-world target domain.

25.3 Multi-Task Transfer

This involves learning a single policy that can solve multiple tasks simultaneously or a policy that can quickly adapt to a new task.

- **Contextual Policies**: A policy takes both the state and a task-specific context as input: $\pi(a|s, \text{context})$. The context can be a one-hot vector representing the task ID or a continuous vector embedding of the task.
- **Goal-Conditioned Policies**: The policy is trained to reach any of a set of goals, which are provided as a part of the input. Hindsight Experience Replay (HER) is a technique that can make this more data-efficient by "relabeling" past trajectories with the goal that was actually achieved, even if the intended goal was not met.

26 Frontiers in RL

RL is a rapidly advancing field with many open challenges and new research directions.

26.1 Meta-Learning

Meta-learning, or "learning to learn," aims to create agents that can quickly adapt to new tasks from a distribution of tasks.

- **Model-Agnostic Meta-Learning (MAML)**: An algorithm that learns a good set of initial policy parameters such that a policy can be adapted to a new task with only a few gradient steps.

26.2 Inverse Reinforcement Learning (IRL)

IRL addresses the **reward specification problem** by inferring a reward function from expert demonstrations. Instead of manually designing a reward, the agent learns what the expert's goal is by observing their behavior.

26.3 Hierarchical RL

Hierarchical RL introduces a hierarchy of policies to solve complex, long-horizon problems. A high-level policy selects a "sub-goal" or "option," which is then executed by a lower-level policy. This provides a temporal abstraction that can make learning more efficient.

26.4 Foundation Models for RL

Recent research explores using large pre-trained models (e.g., LLMs) to provide high-level planning, world models, or reward functions. The idea is to pre-train a model on a vast amount of data and use it as a foundation for decision-making.

26.5 Continual RL

Continual RL focuses on agents that operate in vast, non-stationary worlds without a clear start or end, where rewards are sparse, and the environment changes over time. Challenges include irreversibility, rich observations, and catastrophic forgetting.

Sample Question and Answer

Question 1: REINFORCE with Baseline

The policy gradient update with a baseline is given by $\theta_{t+1} = \theta_t + \alpha \nabla_\theta \log \pi_\theta(A_t|S_t)(G_t - b(S_t))$, where G_t is the Monte-Carlo return and $b(S_t)$ is a learned value function $\hat{v}(S_t, w)$.

1. **Why is it mathematically valid to subtract a baseline $b(S_t)$ that only depends on the state? Explain why this does not introduce bias into the policy gradient estimate. (2 sentences)**

   Answer: Subtracting a state-dependent baseline $b(S_t)$ is valid because its expected value, when multiplied by the score function $\nabla_\theta \log \pi_\theta(a|s)$, is zero. This property ensures that the expectation of the overall gradient estimate remains unchanged, meaning the update is still an unbiased estimate of the true policy gradient.

2. **What is the primary practical benefit of using a baseline in policy gradient methods, and how does the term $(G_t - b(S_t))$ relate to the advantage function? (3 sentences)**

   Answer: The primary benefit of using a baseline is to reduce the variance of the policy gradient estimates, which leads to more stable and faster learning. The term $(G_t - b(S_t))$ is a Monte-Carlo estimate of the advantage function $A(S_t, A_t)$. This is because G_t is an estimate of the Q-value $q_\pi(S_t, A_t)$ and the baseline $b(S_t)$ is an estimate of the state-value $v_\pi(S_t)$.

Question 2: Proximal Policy Optimization (PPO)

The PPO algorithm uses a clipped surrogate objective:

$$\max_\theta \mathbb{E}_{\pi_{\theta_{old}}} \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$.

1. **What is the fundamental problem that the clip function is designed to prevent?**

   Answer: The clip function is designed to prevent the policy from making excessively large updates by discouraging the probability ratio $r_t(\theta)$ from moving too far from 1. This avoids catastrophic performance collapses that can occur with unconstrained on-policy updates, effectively creating a "trust region."

2. **How does this objective offer a simpler and more practical alternative to its predecessor, TRPO?**

   Answer: PPO's clipped objective provides a simpler, first-order optimization method to constrain the policy update. This avoids the complex and computationally expensive second-order optimization (like conjugate gradient) required by TRPO to solve its KL-divergence constraint.

Question 3: Soft Actor-Critic (SAC)

The SAC objective function is augmented with an entropy term:

$$\pi^* = \arg\max_{\pi} \sum_t \mathbb{E}_{\rho^\pi}[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))]$$

1. **What is the primary role of the temperature parameter α? Answer:** The temperature parameter α controls the trade-off between maximizing the cumulative reward and maximizing the policy's entropy.

2. **What are two key benefits of maximizing policy entropy in addition to the expected reward, and how is this reflected in the soft Q-function update? Answer:** Maximizing entropy encourages better exploration by preventing premature convergence to a suboptimal policy and leads to more robust, stable policies. This is reflected in the soft Q-function update by adding the entropy of the next state's policy to the target value: $y_i = r_i + \gamma(\min_j Q_j(s', a') - \alpha \log \pi(a'|s'))$. This modification ensures that the value functions account for the future entropy rewards.

Question 4: Offline Reinforcement Learning (CQL)

Offline RL learns from a fixed dataset, which presents a major challenge known as distribution shift.

1. **Briefly explain why out-of-distribution (OOD) actions are problematic for standard off-policy algorithms like Q-learning in the offline setting. Answer:** Standard Q-learning can produce erroneously high Q-values for OOD actions not present in the data because the function approximator has no data to constrain its estimates for those actions. The policy then learns to exploit these errors, leading to poor performance when deployed in the real world.

2. **The Conservative Q-Learning (CQL) algorithm adds a specific regularizer to the Bellman error loss. What is the high-level goal of this regularizer, and what two things does it do to the Q-function? Answer:** The high-level goal of the CQL regularizer is to combat Q-value overestimation for OOD actions. It explicitly pushes down the Q-values for actions that are likely OOD (sampled from a distribution $\mu(a|s)$). Simultaneously, it pushes up the Q-values for actions that are actually present in the dataset.

Question 5: Dynamic Programming vs. Model-Free Control

Consider two fundamental algorithms: Value Iteration (a Dynamic Programming method) and Q-Learning (a Model-Free Control method). Both aim to find the optimal action-value function q^*.

1. **What is the single most important difference in the assumptions these two algorithms make about the environment? (2 sentences) Answer:** Value Iteration assumes a complete model of the environment is known, meaning it requires the full transition probabilities $p(s', r|s, a)$ and reward function $R(s, a)$. Q-Learning, being model-free, makes no such assumption and can learn directly from sampled experiences (s, a, r, s') without knowing the underlying dynamics.

2. **Given that Value Iteration can be more computationally efficient for small, known environments, why is Q-Learning often more practical for solving large-scale or real-world problems? (2 sentences) Answer:** In most real-world problems, the true dynamics of the environment are unknown and far too complex to be explicitly defined or stored. Q-Learning's ability to learn from direct interaction and experience makes it applicable to these complex scenarios where building an explicit model is infeasible.

Question 6: On-Policy vs. Off-Policy Learning

Sarsa is a classic on-policy TD control algorithm, while Q-Learning is its off-policy counterpart.

1. **Explain the difference in their update rules and how it relates to their on-policy vs. off-policy nature. (2 sentences)**

   - **SARSA update**: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
   - **Q-Learning update**: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)]$

   Answer: Sarsa is on-policy because it uses the action A_{t+1} actually taken by the current policy to form its target, learning the value of the policy it is currently following. Q-Learning is off-policy because it uses the greedy action $\max_{a'} Q(S_{t+1}, a')$ to form its target, learning the value of the optimal (greedy) policy, regardless of which exploratory action was actually taken.

2. **What is the main advantage of off-policy learning (like Q-Learning) over on-policy learning (like Sarsa) in terms of data efficiency? Why might an on-policy method sometimes be preferred? (3 sentences) Answer:** The main advantage of off-policy learning is data efficiency, as it can learn the optimal policy from data generated by any behavior policy, including old data stored in a replay buffer. On-policy methods, in contrast, must discard data after each policy update. However, an on-policy method might be preferred for its stability, as it directly learns about the consequences of the policy it's actually executing, which can lead to smoother convergence, especially in the presence of function approximation.

Question 7: Model-Based Reinforcement Learning

Model-based RL approaches learn a model of the environment to aid in policy learning.

1. **What is the primary advantage and a key disadvantage of model-based RL? (2 sentences) Answer:** The primary advantage is sample efficiency, as the learned model can generate many simulated experiences, reducing the need for real-world interaction. A key disadvantage is that the performance is limited by the accuracy of the learned model; errors in the model can be exploited by the policy, leading to poor real-world performance.

2. **How does the "Dyna-Q" style algorithm, which combines model-based and model-free elements, attempt to mitigate this key disadvantage? (2 sentences) Answer:** Dyna-Q mitigates the problem of model error by continuously updating the model with real experience collected from the environment. It combines planning with model-generated data and learning from real data, which helps to correct for inaccuracies in the model and ground the policy in reality.

Question 8: Deep Deterministic Policy Gradient (DDPG)

DDPG is an off-policy algorithm designed for continuous control.

1. **Why is DDPG considered an "Actor-Critic" algorithm? Briefly describe the roles of the actor and the critic. (2 sentences) Answer:** DDPG is an Actor-Critic algorithm because it uses two separate networks: an actor that learns a deterministic policy ($\mu_\theta(s)$) to select actions, and a critic that learns a Q-function ($Q_w(s, a)$) to evaluate those actions. The critic guides the actor's learning by providing a gradient signal, telling the actor how to adjust its policy to select actions that lead to higher Q-values.

2. **Since the actor's policy is deterministic, how does DDPG ensure sufficient exploration of the state-action space during training? (1 sentence) Answer:** DDPG ensures exploration by adding noise (typically from a stochastic process like Ornstein-Uhlenbeck or simple Gaussian noise) to the actions selected by the deterministic actor during the training phase.