RL in Self-Driving Cars

Shibam Sundar Mahakud
Roll No: 24B2232

June 14, 2025

In reinforcement learning (RL), an agent interacts with an environment over discrete time steps. At each time step t:

- The agent is in a state $s_t \in \mathcal{S}$
- It takes an action $a_t \in \mathcal{A}(s_t)$
- It receives a reward $r_t \in \mathbb{R}$
- The environment transitions to a new state s_{t+1}

The goal is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the expected cumulative reward.

1 Q-Learning

1.1 Definition

Q-Learning is a model-free, off-policy RL algorithm used to learn optimal actions in a Markov Decision Process (MDP). The agent learns a policy that maximizes the cumulative discounted reward without knowing environment dynamics.

1.2 Markov Decision Process (MDP)

Q-Learning assumes the environment is a MDP:

$$\text{MDP} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$$

- **State space** $\mathcal{S}$: Set of all possible states
- **Action space** $\mathcal{A}$: Set of all possible actions
- **Transition function** $P(s'|s, a)$: Probability of moving to state s' from s after action a
- **Reward function** $R(s, a, s')$: Immediate reward for transition $(s, a) \to s'$
- **Discount factor** $\gamma \in [0, 1)$: Values future rewards less than immediate ones

1.3 Objective: Maximizing Return

The **return** at time t is:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

The agent's goal is to maximize the expected return $\mathbb{E}_\pi[G_t]$ under policy π.

1.4 Value Functions

- **State-Value Function**:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s\right]$$

- **Action-Value Function**:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a\right]$$

1.5 Bellman Equations

- **State Value Bellman Equation**:

$$V^{\pi}(s) = \sum_{a} \pi(a|s)\left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s')\right]$$

- **Action Value Bellman Equation**:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') Q^{\pi}(s',a')$$

1.6 Optimal Action-Value Function

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

The **Bellman Optimality Equation** is:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

The optimal policy is:

$$\pi^*(s) = \arg\max_{a} Q^*(s,a)$$

1.7 Q-Learning Update Rule

Q-Learning is an **off-policy, model-free** algorithm that learns $Q^*(s,a)$ by interacting with the environment. The update rule is:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right]$$

where:

- α: learning rate
- γ: discount factor

The term inside the bracket is the **temporal difference (TD) error**:

$$\delta = r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)$$

1.8 Exploration vs Exploitation

Q-Learning uses an **ε-greedy policy** for exploration:

- With probability ε: choose a random action
- With probability $1-\varepsilon$: choose $\arg\max_a Q(s,a)$

1.9 Convergence of Q-Learning

Q-Learning converges to Q^* with probability 1 if:

- All state-action pairs are visited infinitely often
- The learning rate $\alpha_t(s, a)$ satisfies:

$$\sum_{t=1}^{\infty} \alpha_t(s, a) = \infty, \quad \sum_{t=1}^{\infty} \alpha_t^2(s, a) < \infty$$

1.10 Q-Learning Algorithm

1. Initialize $Q(s, a)$ arbitrarily
2. For each episode:
   - Initialize state s
   - Repeat until terminal:
     (a) Choose action a using ε-greedy policy from Q
     (b) Take action a, observe r, s'
     (c) Update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

     (d) $s \leftarrow s'$

1.11 Limitations

- Works only with discrete state-action spaces
- Memory intensive for large state/action spaces
- Slow convergence in high-dimensional environments

2 Monte Carlo Control

2.1 Definition

Monte Carlo (MC) Control methods are used in reinforcement learning (RL) to estimate optimal policies based on experience. Monte Carlo methods do not require a model of the environment. Instead, they use sample episodes generated by interaction with the environment.

2.2 Markov Decision Process (MDP)

MCC assumes the environment is a MDP:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$$

where:

- **State space** $\mathcal{S}$: Set of all possible states
- **Action space** $\mathcal{A}$: Set of all possible actions
- **Transition function** $P(s'|s, a)$: Probability of moving to state s' from s after action a
- **Reward function** $R(s, a, s')$: Immediate reward for transition $(s, a) \to s'$
- **Discount factor** $\gamma \in [0, 1)$: Values future rewards less than immediate ones

2.3 Policy

A policy $\pi(a|s)$ is the probability of taking action a in state s. For deterministic policies, $\pi(s)$ directly maps to an action.

2.4 Return

The **return** G_t is the total discounted reward from time t:

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

2.5 Value Functions

- **State-value function** under policy π:

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

- **Action-value function** under policy π:

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

2.6 Monte Carlo Methods

Monte Carlo methods learn value functions by averaging returns from sample episodes. These methods require episodes to terminate.

2.6.1 First-Visit and Every-Visit Monte Carlo

Let $\mathcal{D}$ be the set of episodes generated from interacting with the environment.

- **First-visit MC**: For each state-action pair (s, a), average the returns of the first occurrence in each episode.
- **Every-visit MC**: For each state-action pair (s, a), average the returns every time the pair is encountered.

For both, we compute the estimate:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s,a)} G_i(s, a)$$

where $N(s, a)$ is the number of times (s, a) was visited and $G_i(s, a)$ is the return following the i^{th} visit.

2.7 Monte Carlo Control

The goal is to find the optimal policy π^* such that:

$$\pi^*(s) = \arg\max_a q_*(s, a)$$

Monte Carlo Control uses policy iteration with MC-based evaluation.

2.7.1 General Procedure

1. Initialize $Q(s, a)$ arbitrarily and π (e.g., random or ϵ-greedy).
2. Generate episodes using the current policy π.
3. For each (s, a) in the episode, compute return G_t.
4. Update the action-value estimate:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[G_t - Q(s, a)]$$

   (Alternatively, maintain count-based averages)

5. Improve the policy using the updated action-values:

$$\pi(s) \leftarrow \arg\max_a Q(s, a)$$

2.8 ϵ-Greedy Policy Improvement

To ensure sufficient exploration, we often use ϵ-greedy policies:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_a Q(s, a) \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases}$$

2.8.1 Why Exploration Matters

Without exploration, MC control may converge to a suboptimal policy. ϵ-greedy ensures every action is taken with non-zero probability, avoiding local optima.

2.9 GLIE and Convergence

GLIE: *Greedy in the Limit with Infinite Exploration.* To guarantee convergence to the optimal policy:

- All state-action pairs must be visited infinitely often (exploration).
- The policy becomes greedy as time $\to \infty$.

A common GLIE strategy is to decay ϵ over time:

$$\epsilon_t = \frac{1}{t}$$

2.9.1 Theorem (Convergence of MC Control)

Under the GLIE conditions, the Monte Carlo Control algorithm converges with probability 1 to the optimal action-value function $q_*(s, a)$ and optimal policy π^*.

2.10 Algorithm: Monte Carlo Control with Exploring Starts (ES)

1. Initialize $Q(s, a)$ arbitrarily.
2. Initialize a policy π.
3. For each episode:
   - Start from a random state-action pair (exploring start).
   - Generate the full episode.
   - For each (s, a) pair in the episode:

– Compute the return G following the first occurrence.
– Update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[G - Q(s,a)\right]$$

– Update policy:

$$\pi(s) \leftarrow \arg\max_a Q(s,a)$$

3 PPO (Proximal Policy Optimization)

3.1 Definition

Proximal policy optimization (PPO) is an on-policy, policy gradient reinforcement learning method for environments with a discrete or continuous action space. It directly estimates a stochastic policy and uses a value function critic to estimate the value of the policy.

3.2 Policy Gradient Methods

In policy-based methods, we parameterize the policy with parameters θ, i.e., $\pi_\theta(a|s)$. The objective is to maximize $J(\theta)$ using gradient ascent:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot \hat{A}_t\right]$$

where $\hat{A}_t$ is an estimator of the advantage function:

$$A_t = Q(s_t, a_t) - V(s_t)$$

3.3 Problems with Vanilla Policy Gradient

Although the vanilla policy gradient works, it suffers from high variance and instability. Large policy updates can degrade performance. Trust Region Policy Optimization (TRPO) was introduced to address this, but it's complex to implement due to second-order derivatives.

3.4 PPO Objective Function

Let:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

Then, the PPO clipped objective is:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

where ϵ is a small hyperparameter (e.g., 0.2). This objective prevents the new policy from straying too far from the old one.

If $r_t(\theta) > 1+\epsilon$ or $r_t(\theta) < 1-\epsilon$, the clipped term takes over, reducing the incentive to push the policy further. This keeps updates conservative but still effective.

3.5 Generalized Advantage Estimation (GAE)

To reduce variance in $\hat{A}_t$, PPO often uses GAE:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

where λ is a parameter that controls the bias-variance tradeoff.

3.6 Final PPO Algorithm Summary

1. Initialize policy π_θ and value function V_ϕ.
2. Collect trajectories using current policy.
3. Compute advantage estimates $\hat{A}_t$ using GAE.
4. Optimize the clipped objective $L^{\text{CLIP}}(\theta)$ using stochastic gradient ascent.
5. Update value function by minimizing:
$$L^{\text{VF}}(\phi) = \mathbb{E}_t \left[(V_\phi(s_t) - R_t)^2 \right]$$
6. Repeat.

3.7 Complete PPO Loss

The full objective optimized by PPO combines:

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t \left[L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\phi) + c_2 S[\pi_\theta](s_t) \right]$$

Where:

- L^{VF}: Value function loss.
- $S[\pi_\theta](s_t)$: Entropy bonus for exploration.
- c_1, c_2: Coefficients balancing the terms.

3.8 Advantages of PPO

- Easy to implement.
- Stable updates due to clipping.
- Compatible with neural networks and large-scale problems.

4 DDPG (Deep Deterministic Policy Gradient)

4.1 Definition

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy actor-critic algorithm for continuous action spaces. It combines the strengths of DQN and deterministic policy gradient methods.

4.2 Markov Decision Process (MDP)

An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$ where:

- $\mathcal{S}$: set of states
- $\mathcal{A}$: set of actions
- $P(s'|s, a)$: transition probability
- $r(s, a)$: reward function
- $\gamma \in [0, 1)$: discount factor

The goal is to learn a policy π that maximizes the expected return:

$$J(\pi) = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right]$$

4.3 Value Functions

- **State-value function**:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,\middle|\, s_0 = s\right]$$

- **Action-value function**:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \,\middle|\, s_0 = s, a_0 = a\right]$$

4.4 Policy Gradient Methods

4.4.1 Stochastic Policy Gradient

For stochastic policies $\pi(a|s)$, the objective is:

$$J(\theta) = \mathbb{E}_{s\sim d^{\pi}, a\sim\pi_{\theta}}\left[r(s, a)\right]$$

The policy gradient theorem:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{s\sim d^{\pi}, a\sim\pi_{\theta}}\left[\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a)\right]$$

4.4.2 Deterministic Policy Gradient

For deterministic policies $\mu_{\theta}(s)$:

$$J(\theta) = \mathbb{E}_{s\sim d^{\mu}}\left[Q^{\mu}(s, \mu_{\theta}(s))\right]$$

Deterministic policy gradient:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_{\theta}\mu_{\theta}(s) \nabla_a Q^{\mu}(s, a)\big|_{a=\mu_{\theta}(s)}\right]$$

4.5 Deep Deterministic Policy Gradient (DDPG)

DDPG combines:

- Deterministic Policy Gradient
- Deep Q-Learning
- Actor-Critic Architecture

4.5.1 Networks

- **Actor network** $\mu(s|\theta^{\mu})$: outputs continuous action
- **Critic network** $Q(s, a|\theta^{Q})$: approximates Q-value
- Target networks: μ' and Q' for stability

4.5.2 Experience Replay

A buffer $\mathcal{D}$ stores transitions (s_t, a_t, r_t, s_{t+1}). Minibatches are sampled randomly to break correlations.

4.5.3 Critic Loss Function

Minimize the Bellman error:

$$L(\theta^{Q}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q(s, a|\theta^{Q}) - y\right)^2\right]$$

Where the target is:

$$y = r + \gamma Q'(s', \mu'(s'|\theta^{\mu'})|\theta^{Q'})$$

4.5.4 Actor Update

The actor is updated via the deterministic policy gradient:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_{s\sim\mathcal{D}} \left[\nabla_a Q(s, a|\theta^Q)\big|_{a=\mu(s)} \nabla_{\theta^\mu} \mu(s|\theta^\mu) \right]$$

4.5.5 Target Networks

Target networks are updated slowly:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

4.6 DDPG Algorithm

Algorithm 1 Deep Deterministic Policy Gradient

Initialize actor $\mu(s|\theta^\mu)$ and critic $Q(s, a|\theta^Q)$ with random weights
Initialize target networks $\theta^{\mu'} \leftarrow \theta^\mu$, $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer $\mathcal{D}$
for episode = 1 to M **do**
    Initialize state s
    for t = 1 to T **do**
        Select action with exploration noise: $a = \mu(s|\theta^\mu) + \mathcal{N}_t$
        Execute action, observe reward r and next state s'
        Store transition (s, a, r, s') in buffer $\mathcal{D}$
        Sample minibatch of N transitions from $\mathcal{D}$
        Compute target:

$$y_i = r_i + \gamma Q'(s'_i, \mu'(s'_i|\theta^{\mu'})|\theta^{Q'})$$

        Update critic by minimizing loss:

$$L = \frac{1}{N}\sum_i (Q(s_i, a_i|\theta^Q) - y_i)^2$$

        Update actor using the gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{a=\mu(s)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)$$

        Update target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

        $s \leftarrow s'$
    end for
end for

4.7 Exploration with Noise

Since the policy is deterministic, exploration is added by injecting noise $\mathcal{N}_t$, typically from an Ornstein-Uhlenbeck process for temporally correlated noise.

$$\mathcal{N}_{t+1} = \theta(\mu - \mathcal{N}_t) + \sigma\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, 1)$$

4.8 Advantages and Limitations

Advantages

- Works in high-dimensional, continuous action spaces
- Sample efficient due to replay buffer
- Off-policy and can reuse experiences

Limitations

- Sensitive to hyperparameters
- Prone to overestimation bias
- Requires careful exploration strategies

5 SAC (Soft Actor Critic)

5.1 Definition

Soft Actor-Critic (SAC) is an advanced off-policy reinforcement learning algorithm designed for continuous action spaces. It builds upon the maximum entropy framework to improve exploration and stability. This document provides a detailed mathematical foundation for SAC, starting from the basics of reinforcement learning.

5.2 MDP

In reinforcement learning (RL), an agent interacts with an environment modeled as a Markov Decision Process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$, where:

- $\mathcal{S}$: state space
- $\mathcal{A}$: action space
- $P(s'|s, a)$: transition probability
- $r(s, a)$: reward function
- $\gamma \in [0, 1)$: discount factor

The goal is to learn a policy $\pi(a|s)$ that maximizes the expected cumulative reward:

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

5.3 Maximum Entropy Reinforcement Learning

SAC uses the maximum entropy objective:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right) \right]$$

where:

- $\mathcal{H}(\pi(\cdot|s_t)) = -\mathbb{E}_{a_t \sim \pi}[\log \pi(a_t|s_t)]$ is the entropy of the policy.
- $\alpha > 0$ is the temperature parameter, which balances reward maximization and entropy.

The entropy term encourages stochasticity in the policy, promoting exploration.

5.4 Soft Q-Function and Soft Value Function

The **soft Q-function** under policy π is defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t,a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right) \middle| s_0 = s, a_0 = a \right]$$

The **soft value function** is:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi} \left[Q^\pi(s,a) - \alpha \log \pi(a|s) \right]$$

The Bellman backup for the soft Q-function is:

$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s'\sim P} \left[V^\pi(s') \right]$$

5.5 Policy Improvement: Soft Actor Update

Given a soft Q-function, the optimal policy π^* minimizes the KL divergence:

$$\pi^*(\cdot|s) = \arg\min_{\pi'} D_{\mathrm{KL}} \left(\pi'(\cdot|s) \middle\| \frac{\exp\left(\frac{1}{\alpha} Q^\pi(s,\cdot)\right)}{Z^\pi(s)} \right)$$

where $Z^\pi(s)$ is the partition function for normalization. This leads to the following loss function for training the policy network:

$$J_\pi = \mathbb{E}_{s_t\sim\mathcal{D}, a_t\sim\pi} \left[\alpha \log \pi(a_t|s_t) - Q(s_t,a_t) \right]$$

5.6 Q-Function Update

The soft Q-function is updated by minimizing the Bellman residual:

$$J_Q = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}} \left[\left(Q(s_t,a_t) - \hat{Q}(s_t,a_t) \right)^2 \right]$$

where the target is:

$$\hat{Q}(s_t,a_t) = r_t + \gamma \mathbb{E}_{a_{t+1}\sim\pi} \left[Q_{\text{target}}(s_{t+1},a_{t+1}) - \alpha \log \pi(a_{t+1}|s_{t+1}) \right]$$

5.7 Value Function

In earlier versions of SAC, a separate value function was used:

$$J_V = \mathbb{E}_{s_t\sim\mathcal{D}} \left[\frac{1}{2} \left(V(s_t) - \mathbb{E}_{a_t\sim\pi} \left[Q(s_t,a_t) - \alpha \log \pi(a_t|s_t) \right] \right)^2 \right]$$

In modern SAC, this is omitted for simplicity and replaced by the clipped double Q approach.

5.8 Temperature Parameter Tuning

The temperature parameter α is often adjusted automatically by minimizing:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi} \left[-\alpha \log \pi(a_t|s_t) - \alpha \bar{\mathcal{H}} \right]$$

where $\bar{\mathcal{H}}$ is the target entropy.

5.9 Complete SAC Algorithm Summary

- Sample a batch of transitions from replay buffer $\mathcal{D}$.
- Update Q-functions by minimizing J_Q.
- Update policy network using J_π.
- Optionally update temperature α.
- Update target Q-networks using Polyak averaging:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$