Reinforcement Learning: A Historical and Mathematical Overview (1950-2024)

Miquel Noguer i Alonso
Artificial Intelligence Finance Institute

November 22, 2024

Abstract

This paper provides an exhaustive overview of the evolution of Reinforcement Learning (RL) from its inception in the 1950s to the cutting-edge developments up to 2024. Emphasis is placed on the mathematical foundations and theoretical advancements that have shaped the field. We delve deeply into the principles of dynamic programming, temporal difference learning, and the integration of deep learning techniques. By tracing the chronological progression of RL, we aim to elucidate how foundational concepts have evolved into sophisticated algorithms that address complex, real-world problems. This comprehensive examination is intended for readers with a solid mathematical background seeking an in-depth understanding of RL's development.

Contents

1 Introduction

Reinforcement Learning (RL) is a subfield of artificial intelligence concerned with how agents ought to take actions in an environment to maximize some notion of cumulative reward. Unlike supervised learning, where the agent is provided with correct input-output pairs, RL involves learning from the consequences of actions in a trial-and-error fashion. This process is akin to how humans and animals learn from interactions with their environment.

The history of RL is rich and multifaceted, drawing from disciplines such as control theory, psychology, neuroscience, and computer science. This paper traces the chronological development of RL, highlighting key milestones and the mathematical underpinnings that have driven progress in the field.

1.1 Motivation and Scope

Understanding the historical context of RL provides valuable insights into why certain algorithms were developed and how they address specific challenges. By examining the mathematical formulations in detail, we aim to offer a comprehensive resource that bridges theoretical concepts with practical implementations.

1.2 Structure of the Paper

The paper is organized chronologically to reflect the evolution of RL:

- **Section 2** covers the foundational period (1950-1980), focusing on dynamic programming and Markov Decision Processes.
- **Section 3** discusses the emergence of model-free methods like Q-Learning and Temporal Difference Learning (1980-2000).
- **Section 4** explores the integration of deep learning with RL (2000-2020), leading to deep reinforcement learning algorithms.
- **Section 5** examines advanced techniques and applications (2020-2024), including policy gradient methods, actor-critic algorithms, and Decision Transformers.
- **Section 6** outlines future directions and ongoing research challenges.

2 Theoretical Foundations (1950-1980)

The period from 1950 to 1980 laid the groundwork for RL, with significant contributions from control theory and the formalization of decision-making processes.

2.1 Markov Decision Processes (MDPs)

2.1.1 Historical Context

The concept of Markov Decision Processes (MDPs) was introduced by Bellman [1957] as a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. MDPs provide a formalization that is essential for understanding and developing RL algorithms.

2.1.2 Definition and Components

Definition 1 (Markov Decision Process). *An MDP is defined as a tuple* (S, A, P, R, γ)*, where:*

- S*: A finite set of states representing all possible configurations of the environment.*
- A*: A finite set of actions available to the agent.*
- $P(s'|s,a)$*: The transition probability function, specifying the probability of transitioning from state* s *to state* s' *given action* a*.*
- $R(s,a,s')$*: The reward function, giving the immediate reward received after transitioning from* s *to* s' *via action* a*.*
- $\gamma \in [0,1]$*: The discount factor, representing the importance of future rewards compared to immediate rewards.*

Markov Property An essential feature of MDPs is the Markov property, which states that the future is independent of the past given the present state and action. Mathematically, this is expressed as:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1}|s_t, a_t)$$

This property simplifies the analysis and computation of optimal policies, as it allows us to consider only the current state and action when making decisions.

2.1.3 Objective of the Agent

The agent's goal in an MDP is to find a policy $\pi : S \to A$ that maximizes the expected cumulative discounted reward, also known as the return. The return G_t at time t is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Policy A policy $\pi(a|s)$ is a mapping from states to probabilities of selecting each possible action. Policies can be deterministic, where $\pi(a|s)$ selects the action with certainty, or stochastic, where actions are selected according to a probability distribution.

2.2 Dynamic Programming and the Bellman Equations

2.2.1 Introduction to Dynamic Programming

Dynamic programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. In the context of RL, DP provides a suite of algorithms for computing optimal policies given a perfect model of the environment as an MDP.

2.2.2 State-Value Function

The state-value function $V^\pi(s)$ under a policy π gives the expected return starting from state s:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \middle| s_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right]$$

This function quantifies the long-term value of a state when following policy π.

2.2.3 Bellman Expectation Equation for $V^\pi(s)$

The Bellman expectation equation provides a recursive relationship for $V^\pi(s)$:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[r_{t+1} + \gamma V^\pi(s_{t+1}) \middle| s_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^\pi(s')\right] \end{aligned} \tag{1}$$

Derivation The derivation of the Bellman equation relies on the law of total expectation and the Markov property:

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} \middle| s_t = s\right] \\
&= \sum_a \pi(a|s) \left(\sum_{s'} P(s'|s,a) \left(R(s,a,s') + \gamma \mathbb{E}_\pi \left[G_{t+1} \middle| s_{t+1} = s' \right] \right) \right) \\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left(R(s,a,s') + \gamma V^\pi(s') \right)
\end{aligned}
$$

Interpretation The equation states that the value of a state under policy π is equal to the expected immediate reward plus the expected discounted value of the next state, averaged over all possible actions and next states.

2.2.4 Action-Value Function

The action-value function $Q^\pi(s,a)$ provides the expected return starting from state s, taking action a, and thereafter following policy π:

$$
Q^\pi(s,a) = \mathbb{E}_\pi\left[G_t \middle| s_t = s, a_t = a\right]
$$

2.2.5 Bellman Expectation Equation for $Q^\pi(s,a)$

Similarly, the Bellman equation for $Q^\pi(s,a)$ is:

$$
\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}\left[r_{t+1} + \gamma \mathbb{E}_\pi\left[G_{t+1} \middle| s_{t+1}\right] \middle| s_t = s, a_t = a\right] \\
&= \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^\pi(s') \right] \\
&= \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s',a') \right] \qquad (2)
\end{aligned}
$$

Significance The action-value function is crucial for selecting optimal actions, as it quantifies the expected return of taking a specific action in a given state.

2.2.6 Bellman Optimality Equations

For optimal policies, the Bellman equations take on a maximization form.

Optimal State-Value Function

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

Optimal Action-Value Function

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

Derivation These equations arise from the principle of optimality, which states that any subsequence of optimal actions must itself be optimal. The optimal value functions satisfy these equations, providing a recursive means to compute them.

2.3 Solution Methods: Value Iteration and Policy Iteration

2.3.1 Value Iteration

Value iteration is an algorithm that iteratively applies the Bellman optimality equation to converge to the optimal value function $V^*(s)$.

Algorithm Steps

1. Initialize $V(s)$ arbitrarily for all $s \in S$.
2. Repeat until convergence:
    - For each state $s \in S$:

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

Convergence Under standard assumptions (e.g., finite state and action spaces), value iteration converges to $V^*(s)$ as the number of iterations approaches infinity.

2.3.2 Policy Iteration

Policy iteration involves two main steps: policy evaluation and policy improvement.

Policy Evaluation For a fixed policy π, compute the state-value function $V^\pi(s)$ by solving the system of linear equations defined by the Bellman expectation equation.

Policy Improvement Update the policy by acting greedily with respect to $V^\pi(s)$:

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

Algorithm Steps

1. Initialize policy π arbitrarily.
2. Repeat until policy stabilizes:
   (a) Policy Evaluation: Compute $V^\pi(s)$.
   (b) Policy Improvement: Update π to π_{new}.

Convergence Policy iteration converges to the optimal policy π^* in a finite number of iterations for finite MDPs.

2.4 Limitations of Dynamic Programming

While dynamic programming methods are powerful, they have several limitations:

- **Curse of Dimensionality**: The computational complexity grows exponentially with the number of states.
- **Model Requirement**: DP requires full knowledge of the environment's dynamics (P and R), which is often not available in practical applications.
- **Storage**: Storing value functions and policies for large state spaces can be impractical.

2.5 Example: Grid World Application

2.5.1 Scenario Description

Consider a grid world environment where an agent must navigate from a starting position to a goal state. Each cell in the grid represents a state, and the agent can move in four directions: up, down, left, and right.

2.5.2 MDP Formulation

- **States (S)**: All grid cells.
- **Actions (A)**: $\{\text{Up}, \text{Down}, \text{Left}, \text{Right}\}$.

- **Transition Probabilities** (P): Deterministic or stochastic movements.
- **Rewards** (R): Negative reward (cost) for each move; large positive reward for reaching the goal.
- **Discount Factor** (γ): Typically set less than 1 to ensure convergence.

2.5.3 Applying Value Iteration

By applying value iteration, we iteratively update the value function for each state until convergence. This process yields the optimal value function $V^*(s)$ and the corresponding optimal policy.

2.5.4 Visualization and Interpretation

The resulting value function can be visualized as a heatmap over the grid, indicating the desirability of each state. The optimal policy guides the agent towards the goal while minimizing costs.

2.5.5 Insights

This example illustrates how DP methods can be applied to solve MDPs in simple environments. It also highlights the challenges that arise when scaling to larger or more complex environments.

3 Model-Free Methods: Q-Learning and Temporal Difference Learning (1980-2000)

The period from 1980 to 2000 saw significant advancements in RL through the development of model-free methods, which allow agents to learn optimal policies without explicit knowledge of the environment's dynamics.

3.1 Temporal Difference (TD) Learning

3.1.1 Introduction

Temporal Difference (TD) learning, introduced by Sutton [1988], combines ideas from Monte Carlo methods and dynamic programming. It enables agents to learn directly from raw experience without a model of the environment.

3.1.2 TD Prediction

TD prediction focuses on learning the value function $V^\pi(s)$ for a given policy π.

TD(0) Update Rule The simplest TD algorithm, TD(0), updates the value estimate based on the observed reward and the estimated value of the next state:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

Interpretation The term $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the TD error, representing the difference between the predicted value and the observed return. The value function is updated to minimize this error.

3.1.3 Convergence Properties

TD(0) converges to $V^\pi(s)$ under certain conditions, such as using a suitable learning rate schedule and visiting all states sufficiently often.

3.2 Q-Learning

3.2.1 Introduction

Q-Learning, developed by Watkins [1989], is a model-free, off-policy algorithm that aims to learn the optimal action-value function $Q^*(s, a)$.

3.2.2 Q-Learning Update Rule

The core update rule for Q-Learning is:

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

Explanation This update adjusts the estimate of $Q(s_t, a_t)$ towards the observed reward plus the discounted maximum estimated value of the next state. The use of $\max_{a'} Q(s_{t+1}, a')$ reflects the assumption that the agent will act optimally in the future.

3.2.3 Properties of Q-Learning

- **Off-Policy**: Q-Learning learns the optimal policy regardless of the agent's current policy.

- **Convergence**: Under certain conditions (e.g., appropriate exploration and learning rates), Q-Learning converges to $Q^*(s, a)$.

3.3 SARSA Algorithm

3.3.1 Introduction

SARSA (State-Action-Reward-State-Action) is an on-policy algorithm that updates the action-value function based on the action actually taken by the policy.

3.3.2 Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

Explanation Unlike Q-Learning, SARSA updates the value of $Q(s_t, a_t)$ using the action a_{t+1} selected by the current policy. This makes it sensitive to the agent's exploration strategy.

3.3.3 Comparison with Q-Learning

- **On-Policy vs. Off-Policy**: SARSA is on-policy, while Q-Learning is off-policy.
- **Exploration Impact**: SARSA accounts for the agent's exploration behavior, potentially leading to safer policies in certain environments.

3.4 Exploration-Exploitation Trade-off

3.4.1 Importance of Exploration

Effective learning requires a balance between exploiting known information to maximize reward and exploring new actions to discover their potential.

3.4.2 ϵ-Greedy Policy

A common exploration strategy is the ϵ-greedy policy:

- With probability ϵ, select a random action (exploration).
- With probability $1-\epsilon$, select the action with the highest estimated value (exploitation).

3.4.3 Adaptive Exploration Strategies

More sophisticated strategies include:

- **Decay of ϵ**: Gradually reduce ϵ over time to shift from exploration to exploitation.
- **Optimistic Initialization**: Initialize Q-values optimistically to encourage exploration.
- **Upper Confidence Bound (UCB)**: Use statistical confidence intervals to guide exploration.

3.5 Function Approximation

3.5.1 Need for Function Approximation

In environments with large or continuous state spaces, storing and updating Q-values for every state-action pair is infeasible. Function approximation methods generalize learning across similar states and actions.

3.5.2 Linear Function Approximation

Represents the value function as a linear combination of features:

$$Q(s, a; \theta) = \theta^\top \phi(s, a)$$

Gradient Descent Updates Parameters θ are updated using gradient descent to minimize the squared TD error.

3.5.3 Nonlinear Function Approximation

Neural networks can approximate complex, nonlinear functions, enabling agents to handle high-dimensional inputs such as images.

3.6 Example: Cliff Walking Problem

3.6.1 Environment Description

An agent must navigate along the edge of a cliff to reach a goal state. Stepping off the cliff results in a large negative reward.

3.6.2 Comparing SARSA and Q-Learning

- **SARSA**: Learns a policy that avoids the cliff by accounting for the agent's exploration, leading to safer paths.
- **Q-Learning**: Learns the optimal policy that skirts the edge of the cliff to minimize steps but may risk falling off during learning due to the exploratory actions.

3.6.3 Insights

This example illustrates how the choice between on-policy and off-policy methods can affect the learned policy, especially in environments where risky actions have severe consequences.

4 Deep Reinforcement Learning (2000-2020)

The integration of deep learning with RL algorithms led to significant breakthroughs, enabling agents to tackle complex tasks with high-dimensional inputs.

4.1 Function Approximation with Neural Networks

4.1.1 Historical Background

Early attempts to use neural networks in RL faced challenges with stability and convergence. The resurgence of deep learning in the 2010s, fueled by advances in training techniques and computational resources, revitalized interest in combining neural networks with RL.

4.1.2 Advantages of Neural Networks

Neural networks can approximate complex, nonlinear functions and handle high-dimensional sensory inputs, such as images or raw sensor data.

4.2 Deep Q-Networks (DQN)

4.2.1 Introduction

Introduced by Mnih et al. [2015], DQN was a landmark achievement that demonstrated the potential of deep RL by achieving human-level performance on Atari games using raw pixel inputs.

4.2.2 Architecture

DQN employs a convolutional neural network (CNN) to process image inputs and output Q-values for each possible action.

4.2.3 Stabilizing Techniques

To address the instability issues inherent in combining Q-Learning with deep neural networks, DQN introduces two key techniques:

Experience Replay

- Stores past experiences $(s_t, a_t, r_{t+1}, s_{t+1})$ in a replay buffer.
- Samples mini-batches of experiences uniformly at random for training.
- Breaks the correlations between sequential data and stabilizes learning.

Target Networks

- Maintains a separate target network $Q(s, a; \theta^-)$ with parameters θ^-.
- Updates the target network parameters periodically by copying from the main network.
- Provides a stable target for the TD error, reducing oscillations in the learning process.

4.2.4 Training Process

1. Initialize the main network $Q(s, a; \theta)$ and the target network $Q(s, a; \theta^-)$.
2. Collect experiences by interacting with the environment using an ϵ-greedy policy.
3. Store experiences in the replay buffer.
4. Sample mini-batches from the replay buffer.
5. Compute the target Q-values using the target network.
6. Update the main network parameters by minimizing the loss function.
7. Periodically update the target network parameters.

4.2.5 Loss Function

The loss function minimized during training is:

$$L_i(\theta_i) = \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1})}\left[(y_i - Q(s_t, a_t; \theta_i))^2\right]$$

where the target y_i is computed using the target network:

$$y_i = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-)$$

4.2.6 Impact and Significance

DQN demonstrated that deep neural networks could successfully approximate value functions in high-dimensional spaces, paving the way for further research in deep RL.

4.3 Improvements and Variants of DQN

4.3.1 Double DQN

Problem Addressed Standard DQN tends to overestimate action values due to the maximization step in the target calculation.

Solution Double DQN, proposed by van Hasselt et al. [2016], decouples action selection and evaluation:

$$y_i^{\text{Double}} = r_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta_i); \theta^-)$$

4.3.2 Dueling DQN

Motivation In some states, the choice of action has little impact on the value function. Separating the representation of state-value and advantage functions can improve learning efficiency.

Architecture Dueling DQN splits the network into two streams:

- **Value Stream**: Estimates the state-value function $V(s; \theta, \beta)$.
- **Advantage Stream**: Estimates the advantage function $A(s, a; \theta, \alpha)$.

The Q-values are combined as:

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

4.3.3 Prioritized Experience Replay

Problem Addressed Uniform sampling from the replay buffer may be inefficient, as some experiences are more valuable for learning.

Solution Prioritized Experience Replay [Schaul et al., 2016] samples experiences based on the magnitude of their TD error, focusing learning on more significant updates.

Implementation The probability of sampling an experience i is:

$$P(i) = \frac{|\delta_i|^\omega}{\sum_k |\delta_k|^\omega}$$

where δ_i is the TD error and ω determines the degree of prioritization.

4.4 Policy Gradient Methods

4.4.1 Introduction

Policy gradient methods optimize the policy directly by computing gradients of the expected return with respect to policy parameters.

4.4.2 Policy Parameterization

Policies are parameterized as $\pi(a|s;\theta)$, where θ are the parameters of a neural network.

4.4.3 Policy Gradient Theorem

The gradient of the expected return $J(\theta)$ is given by:

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

Derivation Overview Starting from $J(\theta) = \mathbb{E}_{\pi_\theta}[G_t]$, the gradient is obtained using the log-derivative trick and the policy's probability density function.

4.4.4 REINFORCE Algorithm

Algorithm Steps

1. Collect trajectories τ by following the policy π_θ.

2. Compute returns G_t for each time step.

3. Update the policy parameters:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

Variance Reduction Subtracting a baseline $b(s_t)$ reduces variance without introducing bias:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \left(G_t - b(s_t)\right)$$

4.5 Actor-Critic Methods

4.5.1 Conceptual Framework

Actor-critic methods combine value-based and policy-based approaches:

- **Actor**: The policy $\pi(a|s;\theta)$ that selects actions.
- **Critic**: The value function $V(s;w)$ that evaluates the current policy.

4.5.2 Advantage Function

The advantage function $A(s,a) = Q(s,a) - V(s)$ measures the benefit of taking action a in state s over the average action.

4.5.3 Policy Update

The actor updates the policy parameters in the direction suggested by the critic:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

4.5.4 Value Function Update

The critic updates the value function parameters by minimizing the TD error:

$$w \leftarrow w - \beta \nabla_w \left(r_{t+1} + \gamma V(s_{t+1}; w) - V(s_t; w)\right)^2$$

4.6 Asynchronous Advantage Actor-Critic (A3C)

4.6.1 Motivation

A3C addresses the limitations of experience replay in continuous or partially observable environments and improves training efficiency.

4.6.2 Parallel Training

Multiple worker agents interact with their own copies of the environment in parallel, updating shared global parameters asynchronously.

4.6.3 Benefits

- **Stability**: Avoids the need for experience replay by leveraging parallelism.
- **Exploration**: Different workers explore different parts of the state space.
- **Efficiency**: Utilizes multiple CPU cores effectively.

5 Advanced Techniques and Applications (2020-2024)

Recent years have witnessed significant advancements in RL algorithms, addressing challenges related to sample efficiency, stability, and scalability.

5.1 Proximal Policy Optimization (PPO)

5.1.1 Introduction

PPO [Schulman et al., 2017] simplifies the trust region optimization approach, making it more practical for large-scale problems.

5.1.2 Clipped Surrogate Objective

PPO uses a clipped surrogate objective to prevent large policy updates:

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

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

5.1.3 Advantages

- **Stability**: Prevents destructive policy updates.
- **Simplicity**: Easier to implement compared to other trust region methods.
- **Performance**: Achieves state-of-the-art results in various benchmarks.

5.2 Soft Actor-Critic (SAC)

5.2.1 Objective

SAC [Haarnoja et al., 2018] incorporates an entropy term into the objective to encourage exploration:

$$J(\pi) = \sum_t \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right]$$

5.2.2 Entropy Regularization

The entropy term $\mathcal{H}(\pi(\cdot|s_t))$ promotes stochastic policies, preventing premature convergence to suboptimal deterministic policies.

5.2.3 Benefits

- **Sample Efficiency**: Off-policy nature allows reusing past experiences.
- **Stability**: Improved convergence properties.
- **Performance**: Effective in continuous control tasks.

5.3 Deep Deterministic Policy Gradient (DDPG)

5.3.1 Introduction

DDPG [Lillicrap et al., 2015] extends DPG to use deep neural networks, enabling RL in continuous action spaces.

5.3.2 Key Components

- **Actor Network**: Outputs deterministic actions.
- **Critic Network**: Estimates the Q-value of state-action pairs.

- **Target Networks**: Slow-moving copies of the actor and critic networks for stable updates.

5.3.3 Exploration Strategy

Uses additive noise (e.g., Ornstein-Uhlenbeck process) to the actions during training for exploration.

5.4 Distributional RL

5.4.1 Concept

Distributional RL models the distribution of possible returns, rather than just the expected return.

5.4.2 Quantile Regression DQN (QR-DQN)

QR-DQN [Dabney et al., 2018] approximates the return distribution using quantile regression, capturing the uncertainty and variability in returns.

5.4.3 Benefits

- **Improved Performance**: Capturing the full distribution can lead to better policies.
- **Risk-Sensitive Policies**: Allows for consideration of risk in decision-making.

5.5 Decision Transformers

5.5.1 Reframing RL as Sequence Modeling

Decision Transformers [Chen et al., 2021] treat RL as a sequence modeling problem, using transformer architectures to predict actions based on past trajectories.

5.5.2 Architecture

Processes sequences of return-to-go, states, and actions:

$$(R_t, s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1}, \dots)$$

5.5.3 Advantages

- **Leverages Pretrained Models**: Can utilize pretrained transformer models.
- **Simplifies Training**: Avoids the need for explicit value function estimation.
- **Flexibility**: Can condition on desired returns to control agent behavior.

6 Conclusion

From its early theoretical foundations in the mid-20th century to the sophisticated algorithms of today, reinforcement learning has undergone remarkable evolution. The development of model-free methods, the integration with deep learning, and the recent innovations in policy optimization and sequence modeling have significantly expanded the capabilities of RL agents.

As we look to the future, addressing challenges related to sample efficiency, safety, and scalability will be essential. Continued interdisciplinary collaboration and the integration of RL with other AI domains hold the promise of unlocking new applications and advancing the field further.

A Mathematical Proofs and Derivations

A.1 Proof of the Policy Gradient Theorem

Theorem 1 (Policy Gradient Theorem). *For any differentiable policy $\pi_\theta(a|s)$, the gradient of the expected return $J(\theta)$ with respect to policy parameters θ is:*

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

Proof. Starting from the expected return:

$$J(\theta) = \sum_s d^\pi(s) \sum_a \pi_\theta(a|s) r(s,a)$$

where $d^\pi(s)$ is the stationary distribution of states under policy π.

Taking the gradient:

$$\nabla_\theta J(\theta) = \sum_s \nabla_\theta d^\pi(s) \sum_a \pi_\theta(a|s) r(s,a) + \sum_s d^\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) r(s,a)$$

Assuming $\nabla_\theta d^\pi(s)$ is negligible, we focus on the second term:

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

Using the identity $\nabla_\theta \pi_\theta(a|s) = \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)$:

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

$\square$

A.2 Derivation of the Bellman Optimality Equation

Starting from the definition of the optimal value function:

$$V^*(s) = \max_\pi V^\pi(s)$$

Using the Bellman expectation equation:

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

Since $V^*(s') \geq V^\pi(s')$ for all policies π, we have:

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

References

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957. ISBN 978-0-691-07951-6.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Parsa Lee, Aditya Grover, and Sham Kumar. Decision transformer: Reinforcement learning via sequence modeling. In *Advances in Neural Information Processing Systems*, volume 34, pages 15084–15097, 2021. URL `https://proceedings.neurips.cc/paper/2021/hash/7f489f640400adbfba07ba77a50ba99a-Abstract.html`.

Will Dabney, Georg Ostrovski, David Silver, and Remi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018. doi: 10.1609/aaai.v32i1.11791.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning*, ICML'18, pages 1861–1870. PMLR, 2018.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015. URL `https://arxiv.org/abs/1509.02971`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *4th International Conference on Learning Representations*, 2016. URL `https://arxiv.org/abs/1511.05952`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL `https://arxiv.org/abs/1707.06347`.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988. doi: 10.1023/A:1022633531479.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100. AAAI Press, 2016. doi: 10.5555/3016100.3016191.

Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, Cambridge, UK, 1989. Ph.D. thesis.