Reinforcement Learning

IIT ROPAR - Minor in AI

11th April, 2025

Contents

1 Learning to Drive: A Reinforcement Learning Journey

Imagine Sarah, a 16-year-old who just received her learner's permit and is excited to learn how to drive. Sarah's process of learning to drive mirrors the fundamental principles of reinforcement learning in a way that anyone can understand.

On her first day behind the wheel, Sarah has theoretical knowledge from driver's education but little practical experience. Her father sits beside her as they drive through an empty parking lot. Initially, her movements are awkward—she presses the gas pedal too hard, causing uncomfortable jerks, and turns the steering wheel either too much or too little.

Each time Sarah makes a mistake, her father provides immediate feedback: "Ease up on the gas," or "Turn more gradually." When she successfully executes a maneuver, he offers positive reinforcement: "Great job maintaining that smooth stop!" This feedback loop—action, consequence, adjustment—is the essence of reinforcement learning.

As weeks pass, Sarah practices regularly. She notices that certain actions consistently lead to better outcomes. Gradually releasing the brake pedal results in smoother starts. Looking far ahead helps her maintain a steady lane position. She's developing what reinforcement learning calls a "policy"—a strategy for selecting actions based on her current situation.

Sarah faces the classic exploration-exploitation dilemma. Should she stick with the techniques that seem to work (exploitation) or try new approaches that might be better (exploration)? When practicing in the parking lot, she experiments more freely, trying different accelerator pressure or turning techniques. On busy roads, she relies on her proven methods.

At first, Sarah requires immediate feedback for each action. But as she improves, she begins to understand how her current actions affect future situations—slowing early for a turn allows for a smoother sequence of maneuvers. This ability to connect present actions to delayed outcomes is akin to how reinforcement learning algorithms handle the challenge of delayed rewards.

Sarah also learns to process multiple inputs simultaneously—monitoring her speed, watching for pedestrians, maintaining lane position, and anticipating traffic signals. Her brain creates a mental model of how driving works, allowing her to predict outcomes of her actions in different situations—similar to how model-based reinforcement learning algorithms operate.

After six months of practice, Sarah passes her driving test. Her learning doesn't stop there—she continues to refine her driving policy through experience, adapting to new situations like highway driving, adverse weather conditions, or unfamiliar vehicles. This lifelong learning and adaptation exemplifies how reinforcement learning continuously improves performance over time.

Sarah's journey from novice to competent driver illustrates the core principles of reinforcement learning: learning through trial and error, balancing exploration and exploitation, connecting actions to both immediate and delayed outcomes, and continuously adapting to maximize long-term success. As we delve into the technical aspects of reinforcement learning, keep Sarah's driving experience in mind—the mathematical concepts may be complex, but the fundamental process mirrors how humans naturally learn many skills.

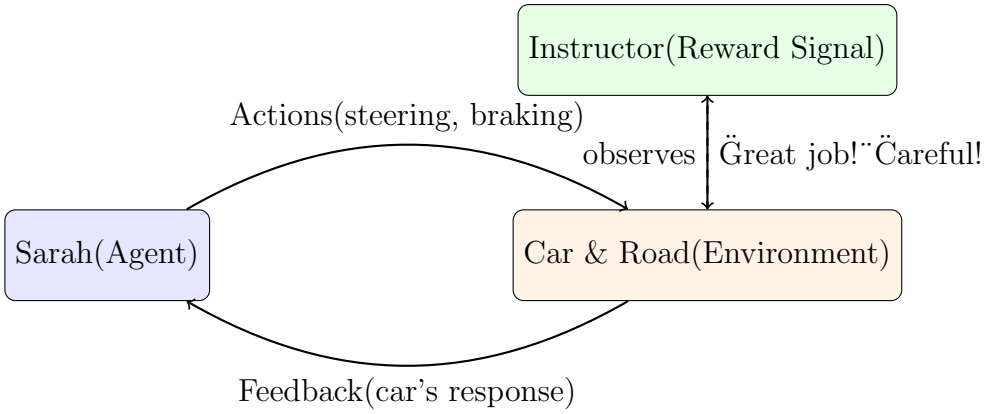

Figure 1: The driving reinforcement learning feedback loop

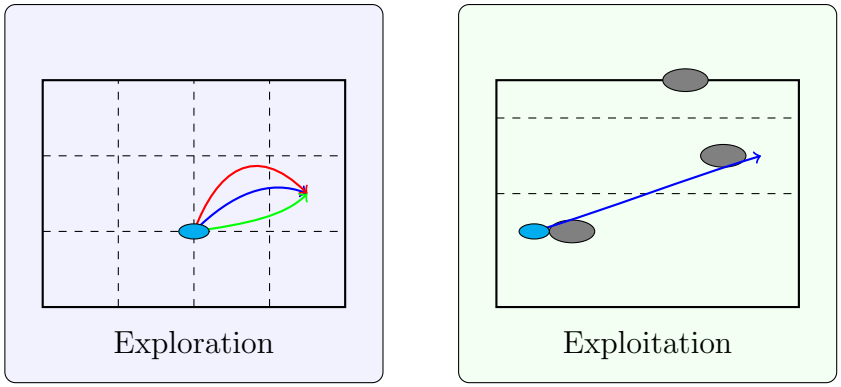

Figure 2: Exploration in a safe environment vs. exploitation on busy roads

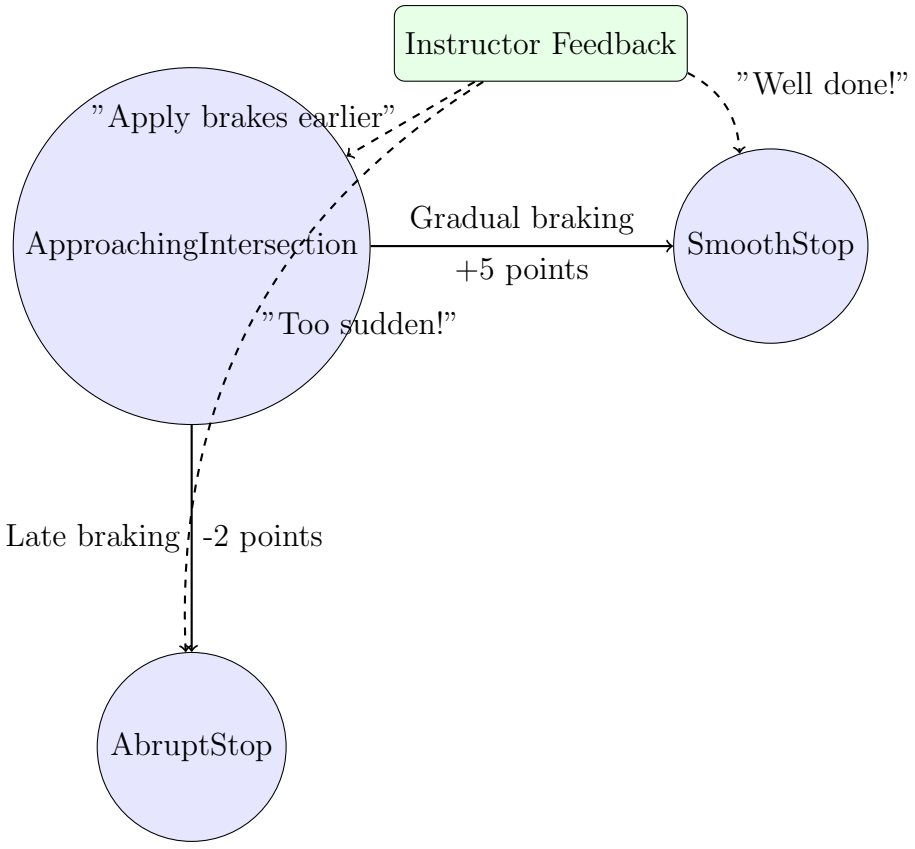

Figure 3: State-action-reward example from Sarah's driving lessons

2 Introduction to Reinforcement Learning

2.1 Definition and Basic Concepts

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment. Unlike supervised learning (which requires labeled data) or unsupervised learning (which discovers patterns in unlabeled data), RL learns from experiences and rewards.

Definition: Reinforcement Learning is a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives while interacting with a complex, uncertain environment.

The key characteristic that distinguishes RL from other machine learning approaches is its trial-and-error nature combined with the delayed reward mechanism. The agent must discover which actions yield the highest rewards by trying them out.

2.2 Comparison with Other Learning Paradigms

Supervised Learning	**Unsupervised Learning**	**Reinforcement Learning**
Learns from labeled examples provided by an external supervisor	Learns patterns from unlabeled data	Learns from interaction with environment
Direct feedback for each example	No feedback	Delayed feedback (reward signal)
Goal: Predict correct outputs	Goal: Find hidden structure	Goal: Maximize total reward
Examples: Classification, regression	Examples: Clustering, dimensionality reduction	Examples: Game playing, robotics, resource management

2.3 Real-world Applications

Reinforcement Learning has shown remarkable success in various domains:

- **Games:**
  - AlphaGo/AlphaZero (Go, Chess, Shogi)
  - OpenAI Five (Dota 2)
  - AlphaStar (StarCraft II)
- **Robotics:**
  - Robot manipulation and locomotion
  - Dexterous manipulation
  - Soft robotics control
- **Autonomous Driving:**

  - Lane keeping
  - Navigation
  - Decision making at intersections

- **Finance:**
  - Algorithmic trading
  - Portfolio management
  - Risk management

- **Other Applications:**
  - Energy management (smart grids)
  - Healthcare (treatment optimization)
  - Recommendation systems
  - Natural language processing tasks

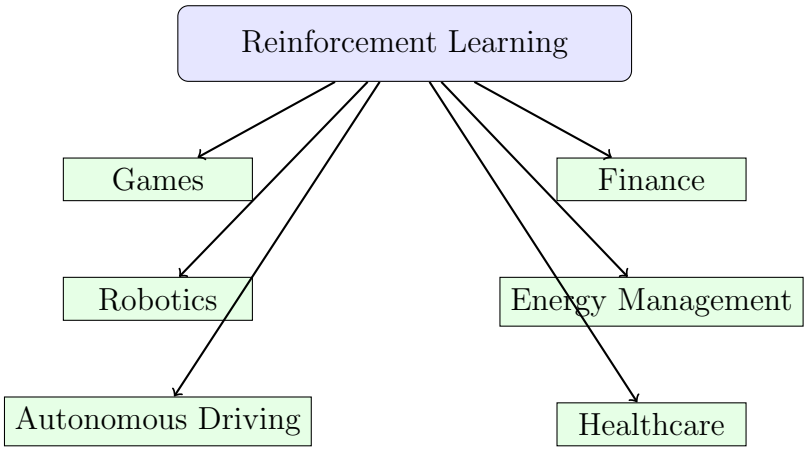

Figure 4: Applications of Reinforcement Learning

3 Key Components of Reinforcement Learning

The reinforcement learning framework consists of several core components that interact in a continuous cycle.

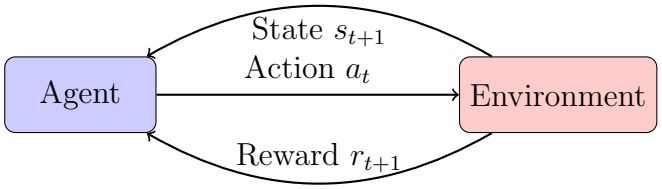

Figure 5: The Agent-Environment Interaction in Reinforcement Learning

3.1 Agent

The agent is the learner and decision-maker that interacts with the environment.

The agent has these key characteristics:

- Makes decisions by selecting actions
- Receives feedback in form of rewards and new states
- Contains the policy (strategy for selecting actions)
- May maintain value functions, models, or other components
- Aims to maximize cumulative reward over time

3.2 Environment

The environment is everything outside the agent that the agent interacts with.

Environment characteristics:

- Responds to agent's actions
- Presents new situations (states) to the agent
- Provides rewards
- Can be deterministic or stochastic
- Can be fully or partially observable
- Can be episodic or continuing

3.3 State/Observation

The state (or observation) represents the current situation of the environment.

Important aspects of states:

- Complete state: Contains all information about the environment (may not be available to the agent)
- Observation: Information available to the agent (may be partial)
- State features: Individual components that make up the state representation
- State space: The set of all possible states

Mathematically, we often denote the state at time t as $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space.

3.4 Action

Actions are the decisions made by the agent that influence the environment.

Key points about actions:

- Can be discrete (finite set of choices) or continuous (real-valued)
- The set of available actions may depend on the current state
- Action space: The set of all possible actions
- Actions cause state transitions and generate rewards

We typically denote an action at time t as $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of actions available in state s_t.

3.5 Reward

The reward is a scalar signal that indicates how good the agent's action was.

Reward characteristics:

- Immediate feedback on the agent's action
- Typically represented as a scalar value
- Can be positive (desirable outcomes), negative (penalties), or zero
- The agent's goal is to maximize cumulative reward, not just immediate reward
- Reward design is crucial for successful RL applications

The reward at time t is often denoted as $r_t = r(s_{t-1}, a_{t-1}, s_t)$, indicating it depends on the previous state, the action taken, and the resulting state.

3.6 Policy

The policy is the agent's strategy for selecting actions.

Policy properties:

- Maps states to actions or probability distributions over actions
- Can be deterministic: $a = \pi(s)$
- Can be stochastic: $\pi(a|s) = P(A_t = a|S_t = s)$
- The ultimate goal of RL is to find an optimal policy π^*
- Can be represented by various structures (tables, neural networks, etc.)

4 Exploration vs. Exploitation Dilemma

One of the fundamental challenges in reinforcement learning is balancing exploration (trying new actions to discover their effects) versus exploitation (using known good actions to maximize reward).

4.1 The Dilemma

Exploitation: Taking the action that is expected to yield the highest reward based on current knowledge.
Exploration: Taking actions to gain more information about the environment, potentially discovering better strategies.

This dilemma arises because:

- Too much exploitation might miss better long-term strategies
- Too much exploration might waste resources trying suboptimal actions
- The optimal balance changes over time as learning progresses

4.2 Common Approaches to Balance Exploration and Exploitation

1. **ϵ-greedy:** With probability $1-\epsilon$, take the greedy action (exploitation); with probability ϵ, take a random action (exploration).
2. **Decaying ϵ-greedy:** Start with a high ϵ value and gradually decrease it over time.
3. **Softmax exploration:** Select actions according to a probability distribution based on their estimated values.

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{1}$$

   where τ is a temperature parameter controlling exploration.

4. **Upper Confidence Bound (UCB):** Select actions based on their estimated value plus an exploration bonus.

$$a_t = \arg\max_a \left[Q(s_t, a) + c\sqrt{\frac{\ln t}{N(s_t, a)}} \right] \tag{2}$$

   where $N(s_t, a)$ counts how often action a has been taken in state s_t.

5. **Thompson Sampling:** Maintain a probability distribution over possible values and sample from this distribution.
6. **Intrinsic Motivation/Curiosity:** Generate internal rewards for exploring novel states.

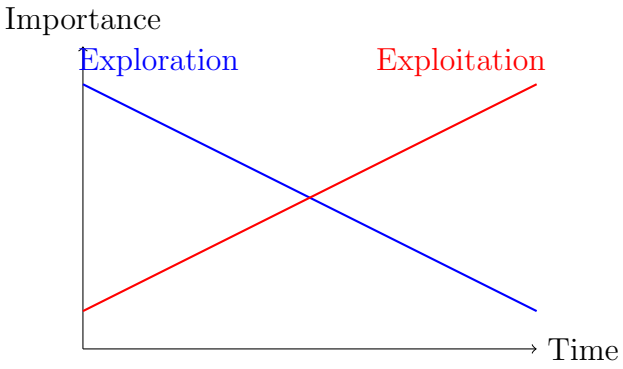

Figure 6: Typical balance between exploration and exploitation over time

5 Types of Reinforcement Learning Algorithms

Reinforcement learning algorithms can be categorized along several dimensions. Understanding these categories helps in selecting appropriate algorithms for specific problems.

5.1 Model-based vs. Model-free

Model-based RL:

- Builds an explicit model of the environment
- Model predicts state transitions and rewards: $P(s'|s,a)$ and $r(s,a,s')$
- Can use the model for planning (simulating possible futures)
- Examples: Dyna-Q, MBVE (Model-Based Value Estimation)
- Advantages: Sample efficiency, planning capability
- Disadvantages: Model errors can affect performance

Model-free RL:

- Learns directly from experience without building an explicit model
- Estimates value functions or policies directly
- Examples: Q-learning, SARSA, Policy Gradient methods
- Advantages: Simplicity, robustness to model errors
- Disadvantages: Often requires more samples

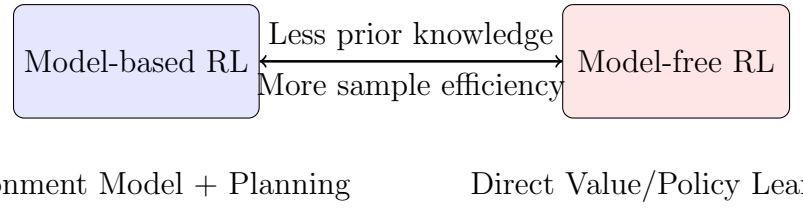

Figure 7: Model-based vs. Model-free Reinforcement Learning

5.2 Value-based vs. Policy-based

Value-based RL:

- Learns value functions (state values V(s) or state-action values Q(s,a))
- Derives policy from value function (e.g., greedy with respect to Q)
- Examples: Q-learning, DQN, SARSA
- Advantages: Often more stable learning
- Disadvantages: Limited to discrete or discretized action spaces

Policy-based RL:

- Directly learns the policy function $\pi(a|s)$
- Optimizes policy parameters to maximize expected return
- Examples: REINFORCE, PPO, TRPO
- Advantages: Works well with continuous actions, can learn stochastic policies
- Disadvantages: Often high variance in learning

Actor-Critic:

- Combines value-based and policy-based approaches
- Actor (policy) determines actions
- Critic (value function) evaluates actions
- Examples: A2C, A3C, SAC, TD3
- Advantages: Combines benefits of both approaches

5.3 On-policy vs. Off-policy

On-policy RL:

- Learns about the policy being currently followed
- Must generate new data when policy changes
- Examples: SARSA, PPO, TRPO
- Advantages: Often more stable learning
- Disadvantages: Less sample-efficient

Off-policy RL:

- Can learn about optimal policy while following a different (behavior) policy
- Can reuse past experience (experience replay)
- Examples: Q-learning, DQN, SAC
- Advantages: More sample-efficient, can learn from demonstrations
- Disadvantages: Can be less stable, may require importance sampling

5.4 Deterministic vs. Stochastic Policies

Deterministic policies:

- Map each state to a single action: $a = \pi(s)$
- Examples: DPG, DDPG, TD3
- Advantages: Simpler optimization in continuous action spaces
- Disadvantages: Cannot handle multimodal optimal policies

Stochastic policies:

- Map each state to a probability distribution over actions: $\pi(a|s)$
- Examples: REINFORCE, A2C, PPO, SAC
- Advantages: Natural exploration, can handle multimodal optimal policies
- Disadvantages: More complex optimization

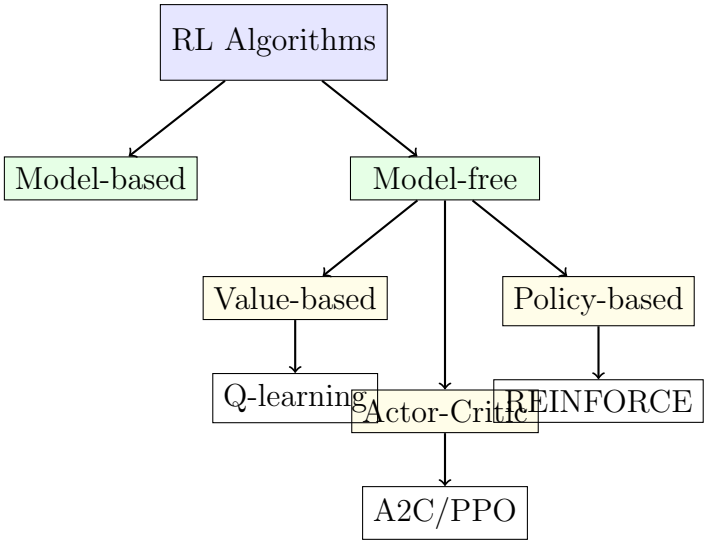

Figure 8: Taxonomy of Reinforcement Learning Algorithms

6 OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It provides a standard API to communicate between learning algorithms and environments, and a standard set of environments.

6.1 Introduction to the Library

OpenAI Gym was designed with these principles:

- **Simplicity:** Easy to use, understand, and extend
- **Reproducibility:** Standardized environments for fair comparison
- **Diversity:** Wide range of environments with varying complexity
- **Compatibility:** Works with any numerical computation library (NumPy, TensorFlow, PyTorch, etc.)

Basic installation:

```
pip install gym
```

6.2 Environment Setup

Creating and initializing a Gym environment:

```
import gym

# Create environment
env = gym.make('CartPole-v1')

# Reset environment to initial state
observation = env.reset()

# Display environment (optional)
env.render()
```

6.3 Action and Observation Spaces

Gym environments define their observation and action spaces using the Space classes.

Common space types:

- **Discrete:** A finite set of values $\{0, 1, \ldots, n-1\}$
- **Box:** An n-dimensional continuous space with bounds
- **Dict:** A dictionary of simpler spaces
- **Tuple:** A tuple of simpler spaces
- **MultiBinary:** A space of binary vectors
- **MultiDiscrete:** A vector of discrete spaces

Accessing space information:

```
print("Action space:", env.action_space)
print("Observation space:", env.observation_space)

# For Box spaces, get dimensions and bounds
if isinstance(env.observation_space, gym.spaces.Box):
    print("Observation dimensions:", env.observation_space.shape)
    print("Observation bounds:", env.observation_space.low, env.observation_space.hig
```

6.4 Interacting with Environments

The core interaction loop in Gym:

Key components:

- **reset():** Initializes the environment and returns initial observation
- **step(action):** Takes the specified action and returns:
  - *observation:* The new state
  - *reward:* The reward for the action
  - *done:* Whether the episode has ended

Algorithm 1 Basic OpenAI Gym Interaction Loop

$observation \leftarrow env.reset()$
$done \leftarrow False$
while not $done$ **do**
    $action \leftarrow agent.choose\_action(observation)$
    $observation, reward, done, info \leftarrow env.step(action)$
    $agent.learn(observation, reward, done)$
    $env.render()$ ▷ Optional visualization
end while
$env.close()$

  - *info:* Additional information (debug info, metrics, etc.)

- **render():** Visualizes the current state
- **close():** Closes the environment and frees resources

6.5 Wrappers

Gym wrappers allow modifying the behavior of environments:

Types of wrappers:

- **ObservationWrapper:** Modifies observations
- **RewardWrapper:** Modifies rewards
- **ActionWrapper:** Modifies actions
- **Wrapper:** Generic wrapper that can modify all aspects

Example of using wrappers:

```
# Normalize observations to range [0,1]
from gym.wrappers import NormalizeObservation
env = gym.make('CartPole-v1')
env = NormalizeObservation(env)

# Time limit wrapper
from gym.wrappers import TimeLimit
env = TimeLimit(env, max_episode_steps=1000)
```

7 Practical Examples

7.1 CartPole Environment

Environment Description:

- A pole is attached to a cart moving along a frictionless track
- The goal is to balance the pole by applying forces to the cart
- The episode ends when the pole falls too far from vertical or the cart moves too far from center

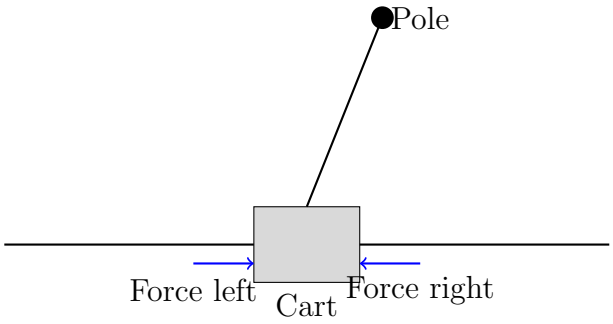

Figure 9: CartPole Environment

State Space (Observation):

- Cart Position: $[-4.8, 4.8]$
- Cart Velocity: $[-\infty, \infty]$
- Pole Angle: $[-24, 24]$
- Pole Angular Velocity: $[-\infty, \infty]$

Action Space:

- 0: Push cart to the left
- 1: Push cart to the right

Reward:

- $+1$ for every timestep the pole remains upright

Episode Termination:

- Pole angle exceeds ± 12 degrees
- Cart position exceeds ± 2.4 units from center
- Episode length exceeds 500 timesteps

Example Q-learning Implementation:

```
import gym
import numpy as np

# Create environment
env = gym.make('CartPole-v1')

# Discretize the continuous state space
def discretize_state(state):
    # Define bins for each state dimension
    cart_pos_bins = np.linspace(-2.4, 2.4, 10)
    cart_vel_bins = np.linspace(-4, 4, 10)
    pole_ang_bins = np.linspace(-0.2094, 0.2094, 10)  # ~12 degrees
    pole_vel_bins = np.linspace(-4, 4, 10)

    # Digitize state to get bin indices
    discretized = [
        np.digitize(state[0], cart_pos_bins),
        np.digitize(state[1], cart_vel_bins),
        np.digitize(state[2], pole_ang_bins),
        np.digitize(state[3], pole_vel_bins)
    ]
    return tuple(discretized)

# Q-learning parameters
alpha = 0.1  # Learning rate
gamma = 0.99  # Discount factor
epsilon = 1.0  # Exploration rate
epsilon_min = 0.01
epsilon_decay = 0.995

# Initialize Q-table
q_table = {}

# Training
num_episodes = 1000
for episode in range(num_episodes):
    state = env.reset()
    state = discretize_state(state)
    done = False
    total_reward = 0

    while not done:
        # Epsilon-greedy action selection
        if state not in q_table:
            q_table[state] = [0, 0]  # Initialize Q-values for new state

        if np.random.random() < epsilon:
            action = env.action_space.sample()  # Explore
```

```
        else:
            action = np.argmax(q_table[state])  # Exploit

        # Take action
        next_state, reward, done, _ = env.step(action)
        next_state = discretize_state(next_state)
        total_reward += reward

        # Q-learning update
        if next_state not in q_table:
            q_table[next_state] = [0, 0]

        best_next_action = np.argmax(q_table[next_state])
        td_target = reward + gamma * q_table[next_state][best_next

best_next_action = np.argmax(q_table[next_state])
        td_target = reward + gamma * q_table[next_state][best_next_action]
        td_error = td_target - q_table[state][action]
        q_table[state][action] += alpha * td_error

        # Move to next state
        state = next_state

    # Decay epsilon
    epsilon = max(epsilon_min, epsilon * epsilon_decay)

    # Print episode results
    if (episode + 1) % 100 == 0:
        print(f"Episode: {episode+1}, Total Reward: {total_reward}, Epsilon: {epsilon
```

7.2 Mountain Car Environment

Environment Description:

- A car is positioned between two mountains
- The goal is to drive the car up the right mountain
- The car's engine is not strong enough to climb the mountain directly
- The agent must learn to build momentum by moving back and forth

State Space (Observation):

- Position: $[-1.2, 0.6]$
- Velocity: $[-0.07, 0.07]$

Action Space:

- 0: Push left

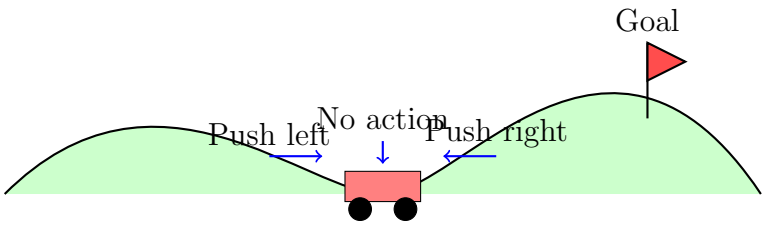

Figure 10: Mountain Car Environment

- 1: No push
- 2: Push right

Reward:

- -1 for each time step until goal reached
- This encourages the agent to reach the goal as quickly as possible

Episode Termination:

- Car position reaches the goal position of 0.5
- Maximum of 200 time steps reached

Example SARSA Implementation:

```
import gym
import numpy as np
import matplotlib.pyplot as plt

# Create environment
env = gym.make('MountainCar-v0')

# Discretize the continuous state space
def discretize_state(state):
    pos_bins = np.linspace(-1.2, 0.6, 20)
    vel_bins = np.linspace(-0.07, 0.07, 20)

    discretized = [
        np.digitize(state[0], pos_bins),
        np.digitize(state[1], vel_bins)
    ]
    return tuple(discretized)

# SARSA parameters
alpha = 0.2
gamma = 0.99
epsilon = 1.0
epsilon_min = 0.01
```

```
epsilon_decay = 0.9995

# Initialize Q-table
q_table = {}

# Training
num_episodes = 2000
rewards_history = []

for episode in range(num_episodes):
    state = env.reset()
    state = discretize_state(state)

    # Initialize action using epsilon-greedy
    if state not in q_table:
        q_table[state] = [0, 0, 0]

    if np.random.random() < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(q_table[state])

    done = False
    total_reward = 0

    while not done:
        # Take action
        next_state, reward, done, _ = env.step(action)
        next_state = discretize_state(next_state)
        total_reward += reward

        # Choose next action using epsilon-greedy
        if next_state not in q_table:
            q_table[next_state] = [0, 0, 0]

        if np.random.random() < epsilon:
            next_action = env.action_space.sample()
        else:
            next_action = np.argmax(q_table[next_state])

        # SARSA update (on-policy)
        q_table[state][action] += alpha * (
            reward + gamma * q_table[next_state][next_action] - q_table[state][action]
        )

        # Move to next state and action
        state = next_state
        action = next_action
```

```
    # Decay epsilon
    epsilon = max(epsilon_min, epsilon * epsilon_decay)

    # Store rewards for plotting
    rewards_history.append(total_reward)

    # Print episode results
    if (episode + 1) % 100 == 0:
        avg_reward = np.mean(rewards_history[-100:])
        print(f"Episode: {episode+1}, Avg Reward (last 100): {avg_reward:.2f}, Epsilo

# Plot training progress
plt.figure(figsize=(10, 5))
plt.plot(rewards_history)
plt.title('Rewards per Episode')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.grid(True)
plt.savefig('mountain_car_training.png')
plt.close()
```

7.3 Atari Games

OpenAI Gym also provides environments for classic Atari 2600 games through the Atari Learning Environment (ALE).

Available Games:

- Pong, Breakout, Space Invaders, Enduro, etc.
- Different versions exist with frame skipping, reward clipping, etc.

State Space:

- RGB image of the game screen (210x160x3)
- Often preprocessed (grayscale, resizing, frame stacking)

Action Space:

- Discrete set of possible controller actions
- Typically contains 4-18 actions depending on the game

Deep Q-Network (DQN) Architecture for Atari:

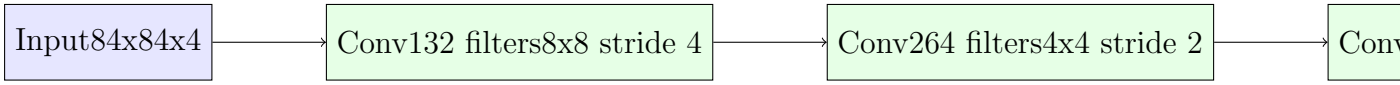

Figure 11: DQN Architecture for Atari Games

Key Innovations in DQN:

- **Experience Replay:** Stores experience tuples (s, a, r, s') in a replay buffer and samples random batches for learning
- **Target Network:** Separate network for generating targets in Q-learning, updated periodically
- **Frame Stacking:** Uses multiple consecutive frames as input to capture motion
- **Reward Clipping:** Clips rewards to $\{-1, 0, +1\}$ to handle different reward scales across games

8 Concepts in Reinforcement Learning

8.1 Episodes

An episode is a complete sequence of interaction from an initial state to a terminal state.

Key points about episodes:

- Has a defined starting state and ending condition
- Finite in length
- Provides a natural way to segment experience
- Used to compute returns and update policies/values

Episodes are particularly important in *episodic tasks*, where there's a natural notion of a final state (like winning a game or reaching a goal). In contrast, *continuing tasks* have no natural endpoint and continue indefinitely.

8.2 State Spaces

The state space is the set of all possible states the environment can be in.

Types of state spaces:

- **Discrete state spaces:** Finite number of states (e.g., chess positions)
- **Continuous state spaces:** Infinite number of states (e.g., robot joint angles)
- **Low-dimensional:** Few state variables (e.g., CartPole with 4 dimensions)
- **High-dimensional:** Many state variables (e.g., image observations)
- **Fully observable:** Agent sees complete state
- **Partially observable:** Agent sees only part of the state

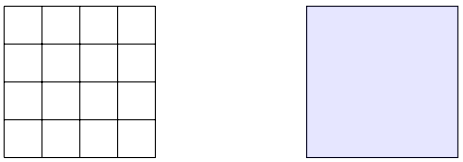

Figure 12: Discrete vs. Continuous State Spaces

8.3 Reward Design

Designing appropriate reward functions is crucial for successful reinforcement learning.

Principles of good reward design:

- **Alignment:** Rewards should align with the actual goals
- **Sparsity vs. Density:**
  - Sparse: Rewards only at completion (easier to specify but harder to learn)
  - Dense: Frequent intermediate rewards (easier to learn but may cause suboptimal behavior)
- **Shaped Rewards:** Adding intermediate rewards to guide learning
- **Avoiding Reward Hacking:** Preventing agents from exploiting reward functions

Examples of reward functions:

- **Simple goal achievement:** $r = 1$ for reaching goal, $r = 0$ otherwise
- **Time penalty:** $r = -0.1$ per step + $r = 1$ for goal
- **Distance-based shaping:** $r = -$distance to goal
- **Progress-based:** $r =$ current progress $-$ previous progress

8.4 Markov Property

The Markov property is a fundamental concept in reinforcement learning.

Definition: A state has the Markov property if the future depends only on the current state and not on the history of how the agent arrived at that state. Mathematically, for states s and s', actions a, and rewards r:

$$P(s', r|s, a, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s', r|s, a) \tag{3}$$

Implications of the Markov property:

- State contains all relevant information for decision making
- Past history can be discarded once state is known
- Forms the basis for Markov Decision Processes (MDPs)
- Simplifies learning and decision making

When the Markov property doesn't hold (partially observable environments), agents often need to use history or build internal models.

9 Challenges in Reinforcement Learning

9.1 Delayed Rewards

One of the fundamental challenges in reinforcement learning is that rewards may be delayed, making it difficult to identify which actions led to success.

Credit assignment problem: Determining which actions in a sequence contributed to the eventual reward.
Approaches to address delayed rewards:

- **Temporal Difference Learning:** Updates estimates based on subsequent estimates
- **Eligibility Traces:** Keeps track of recently visited states/actions
- **Reward Shaping:** Adding intermediate rewards
- **Hierarchical RL:** Breaking tasks into subtasks with shorter reward horizons

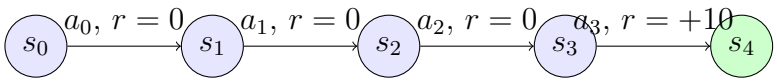

Figure 13: Delayed Reward Example: Only the final transition yields a reward

9.2 Continuous vs. Discrete Action Spaces

Discrete action spaces:

- Finite set of possible actions
- Can be handled by value-based methods (e.g., Q-learning, DQN)
- Easier to explore exhaustively
- Examples: Game moves, direction choices

Continuous action spaces:

- Infinite set of possible actions (real-valued)
- Requires policy-based or actor-critic methods (e.g., DDPG, SAC, PPO)
- Exploration is more complex
- Examples: Joint torques, steering angles

Approaches for handling continuous action spaces:

- **Discretization:** Divide continuous space into bins
- **Policy gradient methods:** Learn a parametrized policy directly

- **Actor-critic with deterministic policies:** DDPG, TD3
- **Actor-critic with stochastic policies:** SAC, PPO

9.3 Partial Observability

In many real-world scenarios, the agent doesn't have access to the complete state of the environment.

Partially Observable Markov Decision Process (POMDP):

- Agent receives observations that don't fully capture the state
- Example: Robot with limited sensors, poker with hidden cards
- The true state is hidden, making optimal decisions harder

Approaches for handling partial observability:

- **Recurrent policies:** Using RNNs (LSTM, GRU) to maintain internal state
- **History-based methods:** Including past observations in the state
- **Belief state methods:** Maintaining a probability distribution over possible states
- **State estimation techniques:** Kalman filters, particle filters

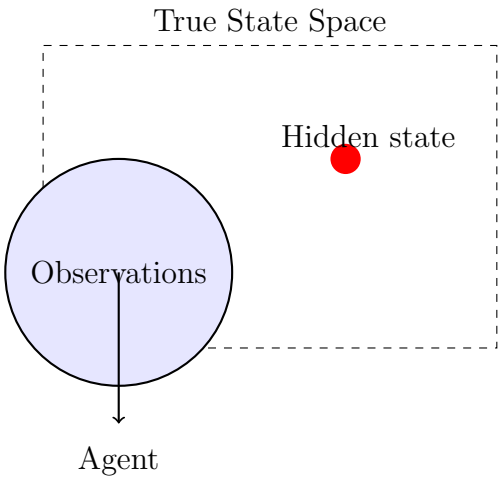

Figure 14: Partial Observability: Agent cannot directly observe all aspects of the true state

10 Conclusion

Reinforcement learning is a powerful machine learning paradigm for solving sequential decision-making problems. This document covered:

- The fundamentals of reinforcement learning
- Key components (agent, environment, state, action, reward, policy)
- Types of RL algorithms
- OpenAI Gym for practical implementations
- Advanced concepts and challenges

The field continues to advance rapidly with new algorithms, applications, and integration with other machine learning techniques. As computational power increases and algorithms improve, reinforcement learning will likely play an increasingly important role in developing intelligent systems that can make optimal decisions in complex, uncertain environments.