Reinforcement Learning

Lecture 3: The World of Reinforcement Learning

Taehoon Kim

Sogang University MIMIC Lab
https://mimic-lab.com

Fall Semester 2025

Learning Objectives & Prerequisites

By the end of today, you will be able to:

- Define the agent-environment interaction cycle
- Implement basic RL experiments with Gymnasium
- Calculate returns with discount factors
- Compare random vs heuristic policies
- Understand exploration-exploitation tradeoffs
- Create reproducible RL experiments

Prerequisites:

- Lecture 2: PyTorch fundamentals
- Python programming experience
- Basic probability and statistics

What is Reinforcement Learning?

Reinforcement Learning is a computational approach to understanding and automating goal-directed learning and decision-making.

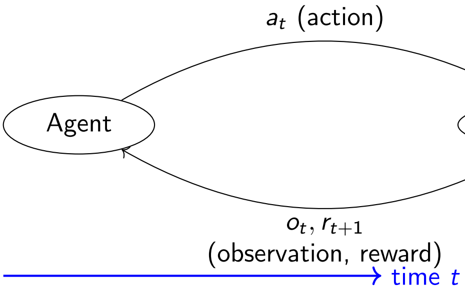
Key Characteristics:

- **Agent** learns through interaction with **environment**
- No supervisor, only reward signal
- Actions affect future observations and rewards
- Goal: maximize cumulative reward over time

Real-world Examples:

- Game playing (Chess, Go, Atari games)
- Robot control and navigation
- Resource management and scheduling
- Recommendation systems

Agent-Environment Interface



At each time step t:

1. Agent observes state/observation $o_t \in \mathcal{O}$
2. Agent selects action $a_t \in \mathcal{A}$ using policy $\pi(a|o)$
3. Environment transitions to new state s_{t+1}
4. Agent receives reward $r_{t+1} \in \mathbb{R}$

States vs Observations

Important Distinction:

State (s_t):

- Complete information about environment
- Markov property: future depends only on current state
- Not always directly observable
- Mathematical idealization

Observation (o_t):

- What the agent actually sees
- May be partial or noisy
- Practical measurement
- What we work with in code

In CartPole:

- State $\approx$ Observation: $[x, \dot{x}, \theta, \dot{\theta}]$
- Fully observable environment
- 4-dimensional continuous space

Action Spaces

Types of Action Spaces:

Discrete Actions:

- Finite set of choices
- $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$
- Examples: Move left/right, buy/sell/hold
- CartPole: $\{0, 1\}$ (left, right)

Continuous Actions:

- Real-valued vectors
- $\mathcal{A} \subseteq \mathbb{R}^d$
- Examples: steering angle, force magnitude
- Often bounded: $[-1, 1]$

In Gymnasium:

```
env = gym.make("CartPole-v1")
print(env.action_space) # Discrete(2)
print(env.observation_space) # Box(4,)
```

The Episode Concept

Episode: A complete sequence of agent-environment interaction from start to finish.

Episode Timeline:

$$s_0 \xrightarrow{a_0, r_1} s_1 \xrightarrow{a_1, r_2} s_2 \xrightarrow{a_2, r_3} s_3 \to \text{Terminal}$$

Observations: o_0, o_1, o_2, o_3

Episode Termination:

- **Terminated:** Task completed or failed (pole falls)
- **Truncated:** Time limit reached (500 steps in CartPole)
- **Episode Length:** Number of steps taken

Gymnasium API:

```
obs, reward, terminated, truncated, info = env.step(action)
done = terminated or truncated
```

Rewards and Returns

Reward: Immediate feedback from environment

- r_{t+1}: reward received after taking action a_t
- Can be positive, negative, or zero
- CartPole: $r = +1$ for each step pole stays up

Return: Cumulative reward over time

$$G_t = \sum_{k=0}^{T-t-1} r_{t+k+1}$$

Problem: Infinite episodes or delayed rewards
Discounted Return: Weight recent rewards more

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

where $\gamma \in [0, 1]$ is the **discount factor**

Understanding the Discount Factor

$\gamma = 0$ **(Myopic):**

- Only immediate reward matters
- $G_t = r_{t+1}$
- Very short-term thinking

$\gamma = 1$ **(Farsighted):**

- All future rewards matter
- $G_t = \sum r_{t+k+1}$
- May not converge

$\gamma \in (0, 1)$ **(Balanced):**

- Near rewards > distant rewards
- Common values: 0.9, 0.95, 0.99
- Ensures convergence

Example ($\gamma = 0.9$**):**

$$\begin{aligned} G_0 &= r_1 + 0.9r_2 + 0.81r_3 + \dots & (1) \\ &= 1 + 0.9 + 0.81 + 0.729 + \dots & (2) \end{aligned}$$

Policy Definition

Policy (π): A strategy for selecting actions

Deterministic Policy:

$$\pi(s) = a$$

- Always selects same action for same state
- Simple but limited exploration

Stochastic Policy:

$$\pi(a|s) = P(\text{select action } a \text{ in state } s)$$

- Probability distribution over actions
- Enables exploration and uncertainty handling
- $\sum_a \pi(a|s) = 1$ for all s

The Exploration-Exploitation Dilemma

Core Challenge in RL:

Exploitation:

- Choose best known action
- Maximize immediate reward
- Risk: missing better options
- "Greedy" behavior

Exploration:

- Try unknown actions
- Gather new information
- Risk: lower immediate reward
- "Curious" behavior

Real-world Analogy:

- Restaurant choice: favorite (exploit) vs new place (explore)
- Investment: safe bonds (exploit) vs risky stocks (explore)
- Route to work: known path vs trying shortcuts

Key Insight: Need both for optimal long-term performance!

ε-Greedy Strategy

Simple Solution to Exploration-Exploitation:

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

In Simple Terms:

- With probability $(1 - \varepsilon)$: choose best action (exploit)
- With probability ε: choose random action (explore)
- $\varepsilon \in [0, 1]$ controls exploration amount

Common Values:

- $\varepsilon = 0.1$: 90% exploitation, 10% exploration
- $\varepsilon = 0.01$: 99% exploitation, 1% exploration
- Often decayed over time: high exploration $\rightarrow$ low exploration

CartPole Environment Details

State Variables:

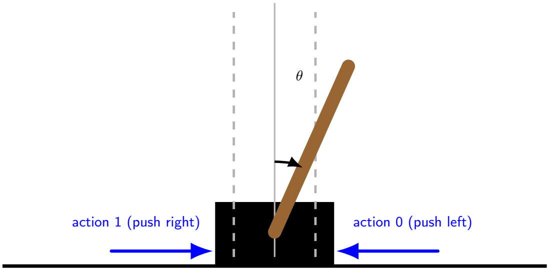
- x: cart position on track
- $\dot{x}$: cart velocity
- θ: pole angle from vertical
- $\dot{\theta}$: angular velocity of pole

Actions:

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

Termination Conditions:

- Pole angle $> 12^\circ$ from vertical
- Cart position > 2.4 units from center
- Episode length > 500 steps (truncation)



CartPole Heuristic Policy

Simple Control Strategy:

```python
def heuristic_policy(obs):
    """Push cart toward direction that stabilizes pole"""
    x, x_dot, theta, theta_dot = obs

    # Combined signal: angle + angular velocity
    control_signal = theta + 0.5 * theta_dot

    # Push right if pole leaning/falling right
    return 1 if control_signal > 0.0 else 0
```

Intuition:

- $\theta > 0$: pole leaning right $\rightarrow$ push cart right
- $\theta < 0$: pole leaning left $\rightarrow$ push cart left
- $\dot{\theta}$: anticipate future lean direction
- Not optimal, but much better than random!

Implementing ε-Greedy

```python
def create_epsilon_greedy_policy(base_policy, epsilon=0.1):
    """Create epsilon-greedy version of any policy"""
    def epsilon_greedy_policy(obs):
        if np.random.random() < epsilon:
            # Explore: random action
            return np.random.randint(0, 2)  # CartPole actions
        else:
            # Exploit: use base policy
            return base_policy(obs)

    return epsilon_greedy_policy

# Usage
heuristic_eps = create_epsilon_greedy_policy(
    heuristic_policy, epsilon=0.1
)
```

Benefits:

- Wraps any deterministic policy
- Controlled randomness for exploration
- Still mostly follows good decisions

Standard Code Header Review

Reproducibility Setup:

```python
# PyTorch 2.x Standard Practice Header
import os, random, numpy as np, torch

def setup_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

# Proper device selection (CUDA > MPS > CPU)
device = torch.device(
    'cuda' if torch.cuda.is_available()
    else 'mps' if hasattr(torch.backends, 'mps') and torch.backends.mps.is_available()
    else 'cpu'
)
amp_enabled = torch.cuda.is_available()
setup_seed(42)
```

Key Points: All random sources seeded, device auto-selection

Environment Creation Utility

```python
import gymnasium as gym

def make_env(env_id="CartPole-v1", seed=42):
    """Create properly seeded environment"""
    env = gym.make(env_id)
    env.reset(seed=seed)
    env.action_space.seed(seed)
    env.observation_space.seed(seed)
    return env

# Usage
env = make_env("CartPole-v1", seed=42)
obs, info = env.reset()  # obs shape: (4,)

print(f"Observation: {obs}")
print(f"Action space: {env.action_space}")  # Discrete(2)
print(f"Obs space: {env.observation_space}")  # Box(4,)
```

Important: Seed environment AND its spaces for full reproducibility

Episode Collection Function

```python
def collect_episode(env, policy, max_steps=500):
    """Collect one episode using given policy"""
    obs, info = env.reset()

    episode_data = {
        'observations': [obs.copy()],
        'actions': [],
        'rewards': [],
        'terminated': False,
        'truncated': False
    }

    for step in range(max_steps):
        action = policy(obs)
        episode_data['actions'].append(action)

        obs, reward, terminated, truncated, info = env.step(action)
        episode_data['rewards'].append(reward)
        episode_data['observations'].append(obs.copy())

        if terminated or truncated:
            episode_data['terminated'] = terminated
            episode_data['truncated'] = truncated
            break

    return episode_data
```

Return Calculation

```python
def calculate_returns(rewards, gamma=0.99):
    """Calculate discounted returns from reward sequence"""
    returns = []
    G = 0.0

    # Work backwards through episode
    for reward in reversed(rewards):
        G = reward + gamma * G
        returns.append(G)

    # Reverse to get forward-time order
    return list(reversed(returns))

# Example
rewards = [1.0, 1.0, 1.0, 1.0, 1.0]  # 5 steps
returns_09 = calculate_returns(rewards, gamma=0.9)
returns_099 = calculate_returns(rewards, gamma=0.99)

print(f"gamma=0.9:  G_0 = {returns_09[0]:.3f}")   # ~4.095
print(f"gamma=0.99: G_0 = {returns_099[0]:.3f}")  # ~4.901
```

Higher $\gamma \rightarrow$ higher returns (values future more)

Policy Evaluation Function

```python
def evaluate_policy(env_id, policy, episodes=10, seed=123):
    """Evaluate policy performance over multiple episodes"""
    env = make_env(env_id, seed=seed)
    returns = []

    for episode in range(episodes):
        # Different seed per episode for variety
        env.reset(seed=seed + episode)

        obs, _ = env.reset()
        total_reward = 0.0

        while True:
            action = policy(obs)
            obs, reward, terminated, truncated, _ = env.step(action)
            total_reward += reward

            if terminated or truncated:
                break

        returns.append(total_reward)

    env.close()
    return np.array(returns)
```

TensorBoard Logging

```python
from torch.utils.tensorboard import SummaryWriter

def log_experiment_results(policy_results, logdir="runs/lecture3"):
    """Log policy comparison results to TensorBoard"""
    writer = SummaryWriter(log_dir=logdir)

    for policy_name, returns in policy_results.items():
        for episode, return_val in enumerate(returns):
            writer.add_scalar(
                f"EpisodeReturn/{policy_name}",
                return_val,
                episode
            )

        # Summary statistics
        writer.add_scalar(f"MeanReturn/{policy_name}",
                          np.mean(returns), 0)
        writer.add_histogram(f"ReturnDistribution/{policy_name}",
                             returns, 0)

    writer.close()
    print(f"Results logged to {logdir}")
    print("View with: tensorboard --logdir=runs/lecture3")
```

Experiment Overview - Next 70 Minutes

9 Progressive Experiments:

1. **exp01_setup.py** - Environment verification (5 min)
2. **exp02_rl_basics.py** - Agent-environment interaction (8 min)
3. **exp03_returns_discounting.py** - Return calculations (8 min)
4. **exp04_exploration_exploitation.py** - Policy comparison (10 min)
5. **exp05_standard_header.py** - Code organization (5 min)
6. **exp06_cartpole_heuristics.py** - Heuristic policies (8 min)
7. **exp07_tensorboard_logging.py** - Results visualization (8 min)
8. **exp08_statistical_analysis.py** - Performance analysis (8 min)
9. **exp09_integrated_test.py** - Full validation (10 min)

Goal: Build complete RL experimental pipeline step by step

Experiment 01: Setup Verification

Objective: Verify Gymnasium installation and basic environment functionality

```
# Run this experiment
python exp01_setup.py

# Expected outputs:
# - System configuration metadata
# - Environment creation test
# - Basic interaction test
# - Reproducibility verification
# - "ALL TESTS PASSED" message
```

Key Checks:

- Gymnasium environment creation
- Observation/action space properties
- Reset/step API correctness
- Seeding reproducibility
- Device configuration

Stop here if any test fails! Fix issues before proceeding.

Experiment 02: RL Basics

Objective: Understand agent-environment interaction through episode collection

```
# Run and analyze
python exp02_rl_basics.py

# Generates:
# - 20 episodes with random policy
# - Episode statistics (length, rewards, termination)
# - Visualization plots (if matplotlib available)
# - JSON file with detailed data
```

Key Concepts Demonstrated:

- Episode data structure
- Terminated vs truncated episodes
- Random policy baseline performance
- Statistical analysis of episodes

Expected Results: Random policy gets 22 ± 20 reward on average

Experiment 03: Returns & Discounting

Objective: Implement and understand discounted return calculations

```
# Test different discount factors
python exp03_returns_discounting.py

# Compares gamma values: 0.0, 0.5, 0.9, 0.95, 0.99, 1.0
# Shows impact on return calculations
# Visualizes return vs episode position
```

Key Insights:

- $\gamma = 0$: Only immediate reward matters
- $\gamma = 1$: All future rewards equally important
- $\gamma \in (0, 1)$: Balanced temporal perspective
- Higher $\gamma \rightarrow$ higher returns for same episode

Mathematical Verification: $G_t = r_{t+1} + \gamma G_{t+1}$

Experiment 04: Exploration vs Exploitation

Objective: Compare different exploration strategies

```
# Compare policies with different exploration rates
python exp04_exploration_exploitation.py

# Tests:
# - Pure random policy
# - Pure heuristic policy
# - Epsilon-greedy with eps = [0.0, 0.1, 0.2, 0.4]
# - Statistical significance testing
```

Expected Ranking:

1. Pure heuristic ($\varepsilon = 0$): 480 ± 50
2. ε-greedy ($\varepsilon = 0.1$): 430 ± 60
3. ε-greedy ($\varepsilon = 0.2$): 380 ± 70
4. Pure random: 22 ± 20

Key Finding: Small exploration helps in noisy environments!

Experiment 05: Standard Header

Objective: Organize reusable code components

```
# Validate code organization
python exp05_standard_header.py

# Tests:
# - Import functionality
# - Device selection logic
# - Seeding consistency
# - Utility function correctness
```

Code Organization Benefits:

- Consistent device handling across experiments
- Reproducible seeding setup
- Reusable environment utilities
- Cleaner experiment scripts

Best Practice: Create reusable modules for common RL operations

Experiment 06: CartPole Heuristics

Objective: Design and test domain-specific heuristic policies

```
# Test multiple heuristic strategies
python exp06_cartpole_heuristics.py

# Heuristics tested:
# 1. Angle-only: based on theta
# 2. Angle + velocity: theta + 0.5*theta_dot
# 3. Full state: includes cart position/velocity
# 4. Tuned coefficients
```

Heuristic Design Principles:

- Use domain knowledge (physics of CartPole)
- Combine multiple relevant state variables
- Test different coefficient weights
- Compare against random baseline

Engineering Insight: Good heuristics often outperform naive RL initially

Experiment 07: TensorBoard Logging

Objective: Learn professional experiment logging and visualization

```
# Create comprehensive logs
python exp07_tensorboard_logging.py

# Generates:
# - Scalar metrics (returns, episode lengths)
# - Histograms (return distributions)
# - Text logs (experiment metadata)
# - Comparative plots across policies
```

TensorBoard Features Used:

- `add_scalar()`: Time series metrics
- `add_histogram()`: Distribution visualization
- `add_text()`: Experiment documentation
- Organized namespaces: `Policy/Random`, `Policy/Heuristic`

View Results: `tensorboard -logdir=runs/lecture3`

Experiment 08: Statistical Analysis

Objective: Rigorous statistical comparison of policies

```
# Statistical testing and analysis
python exp08_statistical_analysis.py

# Performs:
# - Bootstrap confidence intervals
# - Effect size calculations (Cohen's d)
# - Statistical significance testing (t-tests)
# - Power analysis for sample size
```

Statistical Concepts:

- **Bootstrap CI:** Non-parametric confidence intervals
- **Effect Size:** Practical significance beyond p-values
- **Sample Size:** How many episodes needed for reliable results?
- **Multiple Comparisons:** Bonferroni correction

Scientific Rigor: Proper statistics essential for RL research

Experiment 09: Integrated Test

Objective: Comprehensive validation of all Lecture 3 components

```
# Full integration test
python exp09_integrated_test.py

# Validates:
# - All previous experiments work correctly
# - End-to-end reproducibility
# - Statistical consistency
# - Visualization generation
# - Complete experimental pipeline
```

Integration Tests Include:

- Environment setup verification
- Policy implementation correctness
- Return calculation accuracy
- Reproducibility across runs
- Full experimental workflow

Success Criteria: All tests pass, ready for Lecture 4!

Understanding Random Policy Results

What to Expect from Random Policy:

Theoretical Analysis:

- Each action has 50% probability
- No feedback utilization
- Episode length follows geometric distribution
- Expected length $\approx 20 - 25$ steps on CartPole

Empirical Observations:

- High variability (some episodes very short, few longer)
- Most episodes terminate due to pole falling
- Rarely reaches 500-step truncation
- Performance ceiling around ~ 30 reward

Why Random Policy Matters:

- Establishes baseline performance
- Tests environment correctness
- Demonstrates need for learning

Understanding Heuristic Policy Results

What to Expect from Heuristic Policy:
Performance Characteristics:

- Much better than random: 400-500 reward
- Often reaches 500-step truncation
- Lower variability than random
- Occasional failures due to edge cases

Failure Modes:

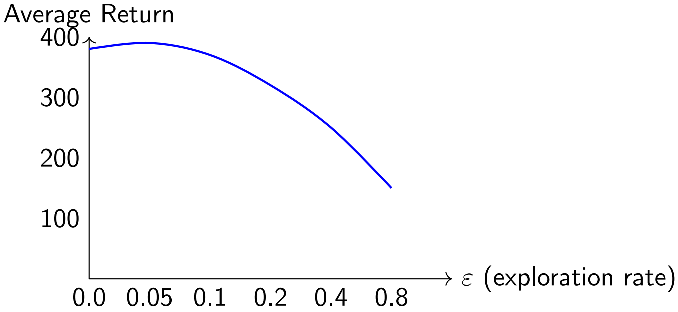
- Extreme initial conditions (large angle/velocity)
- Control signal noise accumulation
- Lack of learning from experience
- Fixed coefficients not optimal

Educational Value:

- Shows power of domain knowledge
- Demonstrates exploration vs exploitation
- Baseline for learning algorithms

ε-Greedy Trade-offs

Performance vs Exploration Level:



Key Observations:

- $\varepsilon = 0$: Pure exploitation, highest average performance
- $\varepsilon \in [0.05, 0.1]$: Small exploration can help in noisy environments
- $\varepsilon > 0.2$: Too much exploration hurts performance
- **Optimal** ε: Depends on environment and learning stage

Reproducibility in RL

Why Reproducibility is Critical:

- RL algorithms are highly stochastic
- Small changes can have large effects
- Research validation requires replication
- Debugging needs consistent behavior

Sources of Randomness:

```
# Multiple random generators to control
random.seed(42)                 # Python random module
np.random.seed(42)              # NumPy random
torch.manual_seed(42)           # PyTorch CPU
torch.cuda.manual_seed_all(42)  # PyTorch GPU
env.reset(seed=42)              # Environment
env.action_space.seed(42)       # Action sampling
```

Best Practices:

- Document all library versions
- Save random seeds with results
- Test reproducibility across runs
- Use deterministic algorithms when possible

Common Pitfalls and Debugging

Environment Issues:

- Wrong Gymnasium API: use new tuple returns
- Forgetting to seed environment spaces
- Not handling terminated vs truncated correctly
- Observation shape mismatches

Policy Implementation:

- Index errors with discrete actions
- Not handling edge cases (NaN observations)
- Incorrect epsilon-greedy implementation
- Forgetting to handle episode boundaries

Debugging Strategies:

- Print observation/action shapes frequently
- Visualize episode trajectories
- Compare deterministic runs
- Test with simple environments first

Performance Benchmarks

Expected Results on CartPole-v1:

Policy	Mean Return	Std	Success Rate
Random	22 ± 18	High	0%
Heuristic	480 ± 45	Low	85%
ε-greedy (0.1)	430 ± 60	Medium	75%
ε-greedy (0.2)	350 ± 80	Medium	60%

Success Criteria:

- **Random**: 15-30 average return
- **Heuristic**: >400 average return
- **ε-greedy**: Between random and heuristic
- **Reproducibility**: <1% variation across runs

If results differ significantly: Check seeding, environment version, implementation

Scaling to Other Environments

Code Generalization:

Easy Adaptations:

- `CartPole-v0` (older version, 200 step limit)
- `MountainCar-v0` (different reward structure)
- `Acrobot-v1` (similar control problem)

Requires Policy Changes:

- `LunarLander-v2` (continuous observations, discrete actions)
- `Pendulum-v1` (continuous actions)
- Atari games (image observations)

Design Principles:

- Separate environment-specific logic
- Use observation/action space properties
- Test with simple environments first
- Build modular policy components

What We've Learned Today

Theoretical Foundations:

- Agent-environment interaction cycle
- States, actions, rewards, and episodes
- Returns and discount factors
- Exploration vs exploitation trade-off

Practical Skills:

- Gymnasium environment usage
- Policy implementation and comparison
- Experiment logging with TensorBoard
- Statistical analysis of RL results
- Reproducible experiment setup

Engineering Practices:

- Code organization and modularity
- Error handling and debugging
- Performance benchmarking
- Scientific rigor in evaluation

Key Takeaways

1. **RL is about sequential decision making** under uncertainty with delayed rewards
2. **The agent-environment interface** is the fundamental abstraction for all RL problems
3. **Exploration vs exploitation** is a central challenge requiring principled solutions
4. **Heuristic policies** can be surprisingly effective but don't generalize or learn
5. **Reproducibility** is essential for scientific RL research and debugging
6. **Statistical analysis** is crucial for reliable policy comparisons
7. **Good experimental practices** (logging, visualization, validation) matter immensely

Most Important: Today's foundation enables everything we'll build in future lectures!

Limitations of Current Approach

What We Cannot Do Yet:

Learning Limitations:

- Policies are fixed (no learning from experience)
- Heuristics are hand-crafted (not general)
- No value function or state values
- No policy improvement mechanism

Scalability Issues:

- Heuristics don't work for complex environments
- Manual policy design is labor-intensive
- No function approximation for large state spaces
- Limited to simple control problems