

Reinforcement Learning

Lecture 3: The World of Reinforcement Learning

Taehoon Kim

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

Fall Semester 2025

Today's Agenda - 3 Hours

- **Segment 1:** Course Introduction & RL Overview [0:00-0:25] - 25 min
- **Segment 2:** Agent-Environment Interface [0:25-0:50] - 25 min
- **Segment 3:** Episodes, Returns & Discounting [0:50-1:15] - 25 min
- **MICRO-BREAK 1** [1:15-1:20] - 5 min
- **Segment 4:** Exploration vs Exploitation [1:20-1:45] - 25 min
- **Segment 5:** Policy Implementation Theory [1:45-2:10] - 25 min
- **MICRO-BREAK 2** [2:10-2:15] - 5 min
- **Segment 6:** Hands-on CartPole Experiments [2:15-3:25] - 70 min
- **Segment 7:** Wrap-up & Next Steps [3:25-3:35] - 10 min

Theory/Practice Split: 30% Theory (54 min) | 70% Practice (126 min)

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.

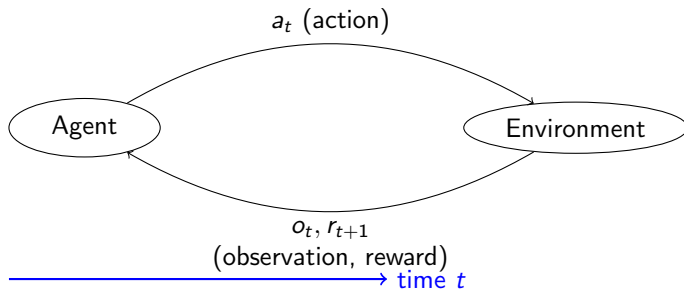
Key Characteristics:

- **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
- $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$
- 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:

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

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:

```
1 obs, reward, terminated, truncated, info = env.step(action)
2 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 equal
- $G_t = \sum r_{t+k+1}$
- May not converge

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

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

Example ($\gamma = 0.9$):

$$G_0 = r_1 + 0.9r_2 + 0.81r_3 + \dots \quad (1)$$

$$= 1 + 0.9 + 0.81 + 0.729 + \dots \quad (2)$$

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!

Simple Solution to Exploration-Exploitation:

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

In Simple Terms:

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

Common Values:

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

CartPole Environment Details

[Diagram Generation Prompt: CartPole environment showing cart on track with pole, indicating forces, angles, and coordinates with labels for state variables]

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:

```
1 def heuristic_policy(obs):  
2     """Push cart toward direction that stabilizes pole"""  
3     x, x_dot, theta, theta_dot = obs  
4  
5     # Combined signal: angle + angular velocity  
6     control_signal = theta + 0.5 * theta_dot  
7  
8     # Push right if pole leaning/falling right  
9     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

```
1 def create_epsilon_greedy_policy(base_policy, epsilon=0.1):
2     """Create epsilon-greedy version of any policy"""
3     def epsilon_greedy_policy(obs):
4         if np.random.random() < epsilon:
5             # Explore: random action
6             return np.random.randint(0, 2) # CartPole actions
7         else:
8             # Exploit: use base policy
9             return base_policy(obs)
10
11     return epsilon_greedy_policy
12
13 # Usage
14 heuristic_eps = create_epsilon_greedy_policy(
15     heuristic_policy, epsilon=0.1
16 )
```

Benefits:

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

Standard Code Header Review

Reproducibility Setup:

```
1  # PyTorch 2.x Standard Practice Header
2  import os, random, numpy as np, torch
3
4  def setup_seed(seed=42):
5      random.seed(seed)
6      np.random.seed(seed)
7      torch.manual_seed(seed)
8      if torch.cuda.is_available():
9          torch.cuda.manual_seed_all(seed)
10
11  # Proper device selection (CUDA > MPS > CPU)
12  device = torch.device(
13      'cuda' if torch.cuda.is_available()
14      else 'mps' if hasattr(torch.backends, 'mps') and torch.backends.mps.is_available()
15      else 'cpu'
16  )
17  amp_enabled = torch.cuda.is_available()
18  setup_seed(42)
```

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

Environment Creation Utility

```
1 import gymnasium as gym
2
3 def make_env(env_id="CartPole-v1", seed=42):
4     """Create properly seeded environment"""
5     env = gym.make(env_id)
6     env.reset(seed=seed)
7     env.action_space.seed(seed)
8     env.observation_space.seed(seed)
9     return env
10
11 # Usage
12 env = make_env("CartPole-v1", seed=42)
13 obs, info = env.reset() # obs shape: (4,)
14
15 print(f"Observation: {obs}")
16 print(f"Action space: {env.action_space}") # Discrete(2)
17 print(f"Obs space: {env.observation_space}") # Box(4,)
```

Important: Seed environment AND its spaces for full reproducibility

Episode Collection Function

```
1 def collect_episode(env, policy, max_steps=500):
2     """Collect one episode using given policy"""
3     obs, info = env.reset()
4
5     episode_data = {
6         'observations': [obs.copy()],
7         'actions': [],
8         'rewards': [],
9         'terminated': False,
10        'truncated': False
11    }
12
13    for step in range(max_steps):
14        action = policy(obs)
15        episode_data['actions'].append(action)
16
17        obs, reward, terminated, truncated, info = env.step(action)
18        episode_data['rewards'].append(reward)
19        episode_data['observations'].append(obs.copy())
20
21        if terminated or truncated:
22            episode_data['terminated'] = terminated
23            episode_data['truncated'] = truncated
24            break
25
26    return episode_data
```

Return Calculation

```
1 def calculate_returns(rewards, gamma=0.99):
2     """Calculate discounted returns from reward sequence"""
3     returns = []
4     G = 0.0
5
6     # Work backwards through episode
7     for reward in reversed(rewards):
8         G = reward + gamma * G
9         returns.append(G)
10
11    # Reverse to get forward-time order
12    return list(reversed(returns))
13
14 # Example
15 rewards = [1.0, 1.0, 1.0, 1.0, 1.0] # 5 steps
16 returns_09 = calculate_returns(rewards, gamma=0.9)
17 returns_099 = calculate_returns(rewards, gamma=0.99)
18
19 print(f"gamma=0.9: G_0 = {returns_09[0]:.3f}") # ~4.095
20 print(f"gamma=0.99: G_0 = {returns_099[0]:.3f}") # ~4.901
```

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

Policy Evaluation Function

```
1 def evaluate_policy(env_id, policy, episodes=10, seed=123):
2     """Evaluate policy performance over multiple episodes"""
3     env = make_env(env_id, seed=seed)
4     returns = []
5
6     for episode in range(episodes):
7         # Different seed per episode for variety
8         env.reset(seed=seed + episode)
9
10        obs, _ = env.reset()
11        total_reward = 0.0
12
13        while True:
14            action = policy(obs)
15            obs, reward, terminated, truncated, _ = env.step(action)
16            total_reward += reward
17
18            if terminated or truncated:
19                break
20
21        returns.append(total_reward)
22
23    env.close()
24    return np.array(returns)
```

TensorBoard Logging

```
1 from torch.utils.tensorboard import SummaryWriter
2
3 def log_experiment_results(policy_results, logdir="runs/lecture3"):
4     """Log policy comparison results to TensorBoard"""
5     writer = SummaryWriter(log_dir=logdir)
6
7     for policy_name, returns in policy_results.items():
8         for episode, return_val in enumerate(returns):
9             writer.add_scalar(
10                 f"EpisodeReturn/{policy_name}",
11                 return_val,
12                 episode
13             )
14
15     # Summary statistics
16     writer.add_scalar(f"MeanReturn/{policy_name}",
17                      np.mean(returns), 0)
18     writer.add_histogram(f"ReturnDistribution/{policy_name}",
19                          returns, 0)
20
21     writer.close()
22     print(f"Results logged to {logdir}")
23     print("View with: tensorboard --logdir=runs/lecture3")
```

MICRO-BREAK 1

5 minutes

Stretch, hydrate, prepare for hands-on experiments

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

```
1 # Run this experiment
2 python exp01_setup.py
3
4 # Expected outputs:
5 # - System configuration metadata
6 # - Environment creation test
7 # - Basic interaction test
8 # - Reproducibility verification
9 # - "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

```
1 # Run and analyze
2 python exp02_rl_basics.py
3
4 # Generates:
5 # - 20 episodes with random policy
6 # - Episode statistics (length, rewards, termination)
7 # - Visualization plots (if matplotlib available)
8 # - 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

```
1 # Test different discount factors
2 python exp03_returns_discounting.py
3
4 # Compares gamma values: 0.0, 0.5, 0.9, 0.95, 0.99, 1.0
5 # Shows impact on return calculations
6 # 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

```
1 # Compare policies with different exploration rates
2 python exp04_exploration_exploitation.py
3
4 # Tests:
5 # - Pure random policy
6 # - Pure heuristic policy
7 # - Epsilon-greedy with eps = [0.0, 0.1, 0.2, 0.4]
8 # - 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

```
1 # Validate code organization
2 python exp05_standard_header.py
3
4 # Tests:
5 # - Import functionality
6 # - Device selection logic
7 # - Seeding consistency
8 # - 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

```
1 # Test multiple heuristic strategies
2 python exp06_cartpole_heuristics.py
3
4 # Heuristics tested:
5 # 1. Angle-only: based on theta
6 # 2. Angle + velocity: theta + 0.5*theta_dot
7 # 3. Full state: includes cart position/velocity
8 # 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

MICRO-BREAK 2

5 minutes

Quick break before final experiments

Experiment 07: TensorBoard Logging

Objective: Learn professional experiment logging and visualization

```
1 # Create comprehensive logs
2 python exp07_tensorboard_logging.py
3
4 # Generates:
5 # - Scalar metrics (returns, episode lengths)
6 # - Histograms (return distributions)
7 # - Text logs (experiment metadata)
8 # - 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

```
1 # Statistical testing and analysis
2 python exp08_statistical_analysis.py
3
4 # Performs:
5 # - Bootstrap confidence intervals
6 # - Effect size calculations (Cohen's d)
7 # - Statistical significance testing (t-tests)
8 # - 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

```
1 # Full integration test
2 python exp09_integrated_test.py
3
4 # Validates:
5 # - All previous experiments work correctly
6 # - End-to-end reproducibility
7 # - Statistical consistency
8 # - Visualization generation
9 # - 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:

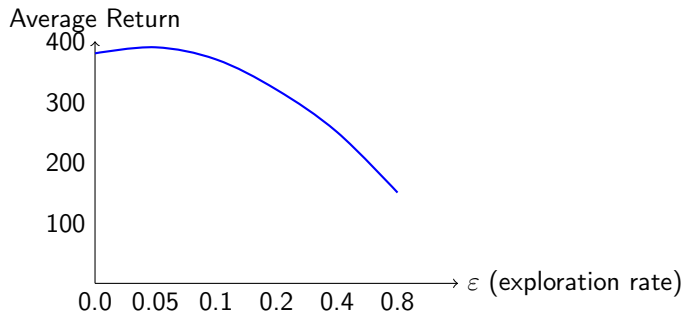
- Extreme initial conditions (large angle/velocity)
- Control signal noise accumulation
- Lack of learning from experience
- Fixed coefficients not optimal

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:

- $\epsilon = 0$: Pure exploitation, highest average performance
- $\epsilon \in [0.05, 0.1]$: Small exploration can help in noisy environments
- $\epsilon > 0.2$: 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:

```
1 # Multiple random generators to control
2 random.seed(42)           # Python random module
3 np.random.seed(42)        # NumPy random
4 torch.manual_seed(42)     # PyTorch CPU
5 torch.cuda.manual_seed_all(42) # PyTorch GPU
6 env.reset(seed=42)        # Environment
7 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

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

Next Lectures Will Address:

- Mathematical foundations (MDPs, Bellman equations)
- Value-based learning (Q-learning)
- Function approximation (neural networks)
- Policy gradient methods

Homework and Next Steps

Take-Home Lab Exercises: (Due before Lecture 4)

- 1 Implement vectorized heuristic policy for faster rollouts
- 2 Design alternative CartPole heuristic using cart velocity
- 3 Create epsilon-decay schedule and analyze effect
- 4 Test policies on MountainCar-v0 environment
- 5 Statistical power analysis: episodes needed for reliable comparison
- 6 Bonus: Implement simple linear policy with REINFORCE (preview)

Preparation for Lecture 4:

- Review probability theory and linear algebra
- Read about Markov Decision Processes (MDPs)
- Familiarize with Bellman equations
- Complete all Lecture 3 experiments successfully

Additional Resources

Books:

- Sutton & Barto: "Reinforcement Learning: An Introduction" (Chapters 1-3)
- Bertsekas: "Dynamic Programming and Optimal Control" (Volume 2)

Online Resources:

- Gymnasium documentation: <https://gymnasium.farama.org/>
- TensorBoard tutorials: <https://pytorch.org/tutorials/>
- OpenAI Spinning Up: <https://spinningup.openai.com/>

Research Papers:

- Mnih et al. (2015): "Human-level control through deep RL"
- Lillicrap et al. (2015): "Continuous control with deep RL"

Practice Environments:

- Classic control: CartPole, MountainCar, Pendulum
- Atari games (for later lectures)
- MuJoCo continuous control (advanced)

Preview of Lecture 4: Mathematical Foundations

Next Week - Markov Decision Processes (MDPs):

Theoretical Topics:

- Formal MDP definition: (S, A, P, R, γ)
- Markov property and state transitions
- Value functions: $V^\pi(s)$ and $Q^\pi(s, a)$
- Bellman equations and dynamic programming
- Policy evaluation and policy iteration

Practical Implementation:

- Tabular value function representation
- Policy evaluation algorithm
- Value iteration implementation
- GridWorld environment examples
- Exact solution methods

Bridge to Today: Transform intuitive policies into principled value-based learning

Final Checklist - Before You Leave

Verify You Can:

- ✓ **Run all experiments successfully** (exp01 through exp09)
- ✓ **Explain agent-environment interaction cycle**
- ✓ **Calculate discounted returns by hand**
- ✓ **Implement epsilon-greedy policies**
- ✓ **Use TensorBoard for experiment logging**
- ✓ **Reproduce experimental results with seeding**

If Any Item Fails:

- Review corresponding experiment
- Check installation and dependencies
- Ask questions in office hours
- Complete before attempting homework

Success Criteria: Ready to proceed to mathematical foundations!

Key Takeaways

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

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

Course Progress Update

Completed So Far:

- ✓ **Lecture 1:** AI-ML-RL hierarchy, environment setup
- ✓ **Lecture 2:** PyTorch fundamentals, neural networks
- ✓ **Lecture 3:** RL fundamentals, agent-environment interaction

Coming Next (13-week roadmap):

- **Lecture 4:** Mathematical foundations (MDPs, Bellman equations)
- **Lecture 5:** Q-learning and tabular methods
- **Lecture 6:** Deep Q-Networks (DQN)
- **Lecture 7:** DQN project and implementation
- **Lectures 8-11:** Policy gradients, Actor-Critic, PPO, advanced topics
- **Lectures 12-13:** Final projects and presentations

We're building the foundation for modern deep RL step by step!