Reinforcement Learning

Lecture 1: Course Overview and Environment Setup

Taehoon Kim

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

Fall Semester 2025

Learning Objectives

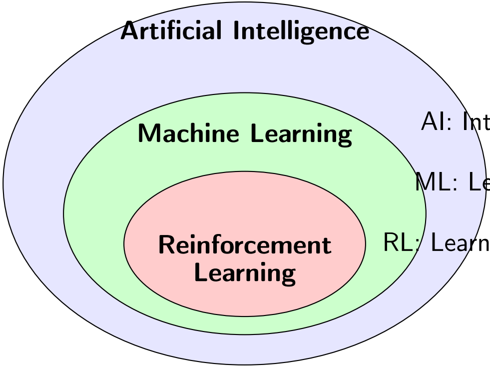
By the end of this lecture, you will:

- Understand the relationship among AI, ML, and RL
- Master the MDP formalism and core RL notation
- Set up a reproducible PyTorch 2.x environment
- Implement the standard code header for the course
- Complete 9 hands-on experiments
- Pass the integrated smoke test

Prerequisites

- Python programming experience
- Basic linear algebra and calculus
- Familiarity with neural networks (helpful)

The AI-ML-RL Hierarchy



AI: Intelligent behavior

ML: Learning from data

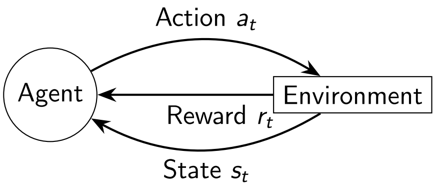
RL: Learning from interaction

What is Reinforcement Learning?

Definition

RL learns optimal behavior through **trial and error** interaction with an environment

- Agent takes actions
- Environment provides rewards
- Goal: maximize cumulative reward
- No explicit supervision



RL vs Supervised Learning

Aspect	Supervised	Reinforcement
Feedback	Immediate labels	Delayed rewards
Data	i.i.d. samples	Sequential, correlated
Exploration	Not needed	Essential
Goal	Minimize error	Maximize return
Training	Offline, batch	Online, interactive

Key Insight

RL faces the **exploration-exploitation dilemma**: Should the agent try new actions (explore) or stick with known good actions (exploit)?

13-Week Course Structure

Foundations (Weeks 1-4)

- Week 1: Environment Setup
- Week 2: Deep Learning Essentials
- Week 3: RL Fundamentals
- Week 4: Mathematical Foundations

Value-Based (Weeks 5-7)

- Week 5: Q-Learning
- Week 6: Deep Q-Networks
- Week 7: DQN Project

Policy-Based (Weeks 8-10)

- Week 8: Policy Gradients
- Week 9: Actor-Critic Methods
- Week 10: PPO

Advanced (Weeks 11-13)

- Week 11: Current Trends
- Week 12: Project Development
- Week 13: Final Presentations

Required Tools and Resources

Software

- Python 3.10-3.12
- PyTorch 2.x
- Gymnasium
- TensorBoard
- Jupyter/Colab
- Git

Hardware

- CPU: Any modern processor
- GPU: Optional but recommended
- RAM: 8GB minimum
- Storage: 20GB free space

Cloud Alternative

Google Colab provides free GPU access - all experiments will run there

Understanding the MDP Framework

Topics

- Markov Decision Processes (MDPs)
- States, Actions, and Rewards
- Policies and Value Functions
- Bellman Equations
- Optimality Conditions

Markov Decision Process (MDP)

An MDP is defined as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$:

- $\mathcal{S}$: State space
- $\mathcal{A}$: Action space
- $\mathcal{P}(s'|s, a)$: Transition probability
- $r(s, a)$: Reward function
- $\gamma \in [0, 1)$: Discount factor

Markov Property

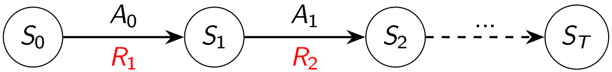
The future depends only on the current state, not the history:

$$P(S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, \ldots) = P(S_{t+1}|S_t, A_t)$$

Episode Structure

An episode is a sequence of interactions:

$$(S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T, S_T)$$



Terminal state S_T ends the episode

Return and Discounting

The **return** G_t is the cumulative discounted reward:

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

Why Discounting?

- Mathematical convenience (convergence)
- Uncertainty about the future
- Preference for immediate rewards

Example

If $\gamma = 0.9$ and rewards are $[1, 2, 3, ...]$:

$$G_0 = 1 + 0.9 \cdot 2 + 0.81 \cdot 3 + ... = 10$$

Policy

A **policy** π defines the agent's behavior:

$$\pi(a|s) = P(A_t = a|S_t = s)$$

Deterministic Policy

$$a = \pi(s)$$

One action per state

Stochastic Policy

$$\pi(a|s) \in [0, 1]$$

Probability distribution over actions

Goal

Find the optimal policy π^* that maximizes expected return:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \right]$$

Value Functions

State Value Function

Expected return starting from state s following policy π:

$$v^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

Action Value Function (Q-function)

Expected return starting from state s, taking action a, then following π:

$$q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Relationship:

The policy $\pi(a|s)$ is a probability distribution over actions given state s. Therefore, the state value is the expected action value under this distribution:

$$v^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \cdot q^\pi(s, a)$$

Bellman Equations

Value functions satisfy recursive relationships:

Bellman Expectation Equation

$$v^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} \mathcal{P}(s'|s,a)[r(s,a) + \gamma v^{\pi}(s')]$$

Bellman Optimality Equation

$$v^{*}(s) = \max_{a} \sum_{s'} \mathcal{P}(s'|s,a)[r(s,a) + \gamma v^{*}(s')]$$

These equations are the foundation for RL algorithms!

Bellman Equations: Detailed Explanation

Bellman Expectation Equation

- Under a given policy π, the value of a state s is
- the weighted sum of possible actions, where weights are given by $\pi(a|s)$.
- Each action leads probabilistically to a next state s' according to $\mathcal{P}(s'|s, a)$, yielding an immediate reward $r(s, a)$.
- Therefore, $v^{\pi}(s)$ is the expected immediate reward plus the discounted value of the next state.

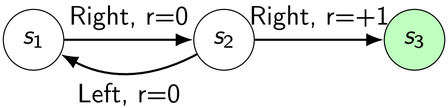
Bellman Optimality Equation

- For the optimal policy, the value of state s is
- the maximum expected return achievable over all possible actions.
- Instead of averaging with $\pi(a|s)$, we take $\max_a$.
- Solving this recursive relation gives the optimal value function $v^*(s)$ and the optimal policy.

Key idea: The Bellman equations express the value of a state as a recursive relationship between immediate reward and future value.

Bellman: Expectation vs Optimality (GridWorld)

Expectation (policy evaluation)

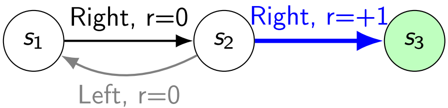


$\pi(\cdot|s_2)$: $\pi(\text{Right}|s_2) = 0.7,\ \pi(\text{Left}|s_2) = 0.3$

$\gamma = 0.9, \quad v^\pi(s_3) = 0$ (terminal state)

$$v^\pi(s_2) = 0.7\,[1 + \gamma v^\pi(s_3)] + 0.3\,[0 + \gamma v^\pi(s_1)]$$
$$v^\pi(s_1) = 0 + \gamma v^\pi(s_2)$$

Optimality (control)



$\gamma = 0.9, \quad v^*(s_3) = 0$

$$v^*(s_2) = \max\{\,1 + \gamma v^*(s_3),\ \gamma v^*(s_1)\,\}$$
$$v^*(s_1) = \gamma v^*(s_2)$$

Greedy policy at s_2: choose Right

Expectation averages action values using $\pi(a|s)$, while Optimality takes the maximum over actions.

Optimal Policy and Value Functions

- Optimal value function: $v^*(s) = \max_\pi v^\pi(s)$
- Optimal Q-function: $q^*(s, a) = \max_\pi q^\pi(s, a)$
- Optimal policy: $\pi^*(a|s) = \arg\max_a q^*(s, a)$

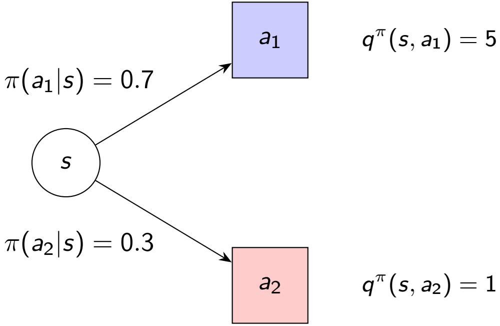
Theorem (Policy Improvement)

For any policy π, the greedy policy with respect to v^π is at least as good as π

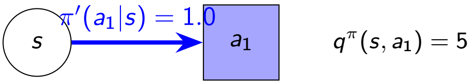
This leads to **policy iteration** and **value iteration** algorithms

Policy Improvement Intuition

Current policy π (mixed actions)



Greedy policy π' (choose best action)



$$v^{\pi'}(s) = \max_a q^{\pi}(s, a) = 5$$

$$v^{\pi}(s) = 0.7 \times 5 + 0.3 \times 1 = 3.8$$

Greedy policy always achieves at least as high value as the original policy.

Types of RL Problems

Dimension	Types	Examples
State space	Discrete/Continuous	Grid/Robot control
Action space	Discrete/Continuous	Chess/Driving
Observation	Full/Partial	Go/Poker
Model	Model-based/free	Planning/Q-learning
Policy	On-policy/Off-policy	SARSA/Q-learning

This Course Focus

- Start with discrete spaces (tabular methods)
- Move to continuous (function approximation)
- Both model-free and model-based approaches

Mathematical Prerequisites

Linear Algebra

- Vector operations
- Matrix multiplication
- Eigenvalues (optional)

Calculus

- Derivatives
- Chain rule
- Gradients

Probability

- Expectations
- Conditional probability
- Distributions

Optimization

- Gradient descent
- Convexity (optional)
- Convergence

Environment Setup Overview

Building Your RL Development Environment

Components to Install

- Python environment (Anaconda/Miniconda)
- PyTorch 2.x with CUDA support
- Essential libraries
- Reproducibility tools
- Version control (Git)

Python Environment Setup

```
# Create conda environment
conda create -n rl2025 python=3.10
conda activate rl2025

# Install PyTorch (with CUDA 11.8)
conda install pytorch torchvision torchaudio \
    pytorch-cuda=11.8 -c pytorch -c nvidia

# Install essential packages
pip install numpy matplotlib pandas tqdm
pip install tensorboard jupyterlab
# gymnasium will be installed in Lecture 3
```

Important

Python 3.10-3.12 required for compatibility

Device Detection Logic

```python
# 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'
)

print(f"Using device: {device}")

# Check CUDA details if available
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
    print(f"CUDA: {torch.version.cuda}")
```

Ensuring Reproducibility

```python
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)

    # Deterministic algorithms
    torch.use_deterministic_algorithms(True)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True

# Always call at start of experiments
setup_seed(42)
```

Critical for reproducing results!

Recommended Project Structure

```
rl2025/
|-- envs/                # Environment configs
|   |-- environment.yml
|   +-- requirements.txt
|-- experiments/         # Experiment scripts
|   +-- exp01_setup.py
|-- runs/               # Logs and checkpoints
|   |-- checkpoints/
|   +-- tensorboard/
|-- notebooks/          # Jupyter notebooks
+-- README.md
```

Keep code, data, and results organized!

Automatic Mixed Precision (AMP)

What is AMP?

Training with mixed float16/float32 precision for:

- 2-3x speedup on modern GPUs
- 50% memory reduction
- Maintained accuracy

Benefits

- Larger batch sizes
- Faster training
- More complex models

Requirements

- CUDA-capable GPU
- PyTorch 1.6+
- Volta architecture or newer

AMP Implementation

```python
from torch.cuda.amp import autocast, GradScaler

# Initialize scaler
scaler = GradScaler()

# Training step with AMP
optimizer.zero_grad()

# Forward pass with autocast
with autocast():
    output = model(input)  # FP16 computation
    loss = criterion(output, target)

# Backward pass with scaling
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

PyTorch 2.x Compilation

```python
# Compile model for optimized execution
model = torch.compile(model, mode='default')

# Different compilation modes:
# 'default': Balanced optimization
# 'reduce-overhead': Minimize kernel launches
# 'max-autotune': Maximum performance

# Fallback for older PyTorch
def compile_if_available(module):
    if hasattr(torch, 'compile'):
        return torch.compile(module)
    return module
```

Up to 2x speedup with torch.compile!

TensorBoard Integration

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

# Initialize writer
writer = SummaryWriter('runs/experiment_1')

# Log scalars
writer.add_scalar('loss/train', loss, step)

# Log histograms
writer.add_histogram('weights', model.fc.weight, step)

# Log model graph
writer.add_graph(model, sample_input)

# Close when done
writer.close()
```

View with: `tensorboard -logdir runs`

Checkpoint Management

```python
# Save checkpoint
checkpoint = {
    'epoch': epoch,
    'model': model.state_dict(),
    'optimizer': optimizer.state_dict(),
    'loss': loss,
    'rng_states': {
        'torch': torch.get_rng_state(),
        'cuda': torch.cuda.get_rng_state_all()
    }
}
torch.save(checkpoint, 'checkpoint.pt')

# Load checkpoint
checkpoint = torch.load('checkpoint.pt')
model.load_state_dict(checkpoint['model'])
optimizer.load_state_dict(checkpoint['optimizer'])
```

Version Control for RL

```bash
# Initialize repository
git init
git config user.name "Your Name"
git config user.email "email@example.com"

# Create .gitignore
echo "runs/" >> .gitignore
echo "__pycache__/" >> .gitignore
echo "*.pt" >> .gitignore

# Track experiment
git add experiment.py
git commit -m "Experiment: DQN baseline
  Config: lr=0.001, batch=32
  Result: 195.3 avg reward"
```

Google Colab Setup

```python
# Colab bootstrap cell
import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    # Install packages
    !pip install -q torch tensorboard

# Mount Google Drive
if IN_COLAB:
    from google.colab import drive
    drive.mount('/content/drive')

# Check GPU
!nvidia-smi  # Should show Tesla T4 or better
```

Free GPU access for experiments!

Unified Starting Point for All Experiments

Components

- Reproducibility (seeds)
- Device management
- AMP support
- Logging utilities
- Checkpoint handling
- Common RL functions

All experiments will import from this header!

Testing Your Setup

```
# Run integrated test
python exp09_integrated_test.py

# Expected output:
# =====================================
# Test 1: Environment Setup     [PASS]
# Test 2: Reproducibility      [PASS]
# Test 3: Model Training        [PASS]
# Test 4: DQN Components        [PASS]
# Test 5: Checkpointing         [PASS]
# Test 6: Logging              [PASS]
# =====================================
# All tests passed!
```

9 Progressive Experiments

1. Environment verification (exp01)
2. PyTorch basics (exp02)
3. Reproducibility (exp03)
4. AMP benchmarks (exp04)
5. Standard header (exp05)
6. Logging setup (exp06)
7. Checkpointing (exp07)
8. Git integration (exp08)
9. Integration test (exp09)

Experiment 1: Environment Verification

Goal: Verify Python and package installations
Tasks:

- Check Python version (3.10-3.12)
- Verify PyTorch installation
- List installed packages
- Create environment files
- Save system information

Run: `python exp01_setup.py`

Exp1: Key Code

```python
def check_python_version():
    version = sys.version_info
    if not (3, 10) <= (version.major, version.minor) <= (3, 12):
        print("Warning: Python 3.10-3.12 recommended")
        return False
    return True

def check_package_installations():
    required = ['torch', 'numpy', 'matplotlib']
    for package in required:
        try:
            __import__(package)
            print(f"[OK] {package}")
        except ImportError:
            print(f"[MISSING] {package}")
```

Experiment 2: PyTorch Basics

Goal: Master PyTorch fundamentals and device management

Tasks:

- Device detection (CUDA $>$ MPS $>$ CPU)
- Tensor operations
- Automatic differentiation
- Performance benchmarking

Key Learning: Proper device selection is critical

Exp2: Device Selection

```python
def get_device():
    if torch.cuda.is_available():
        device = torch.device('cuda')
        print(f"Using CUDA: {torch.cuda.get_device_name(0)}")
    elif hasattr(torch.backends, 'mps') and \
         torch.backends.mps.is_available():
        device = torch.device('mps')
        print("Using MPS (Apple Silicon)")
    else:
        device = torch.device('cpu')
        print("Using CPU")
    return device
```

Experiment 3: Reproducibility

Goal: Ensure experiments are reproducible
Tasks:

- Set seeds for all RNGs
- Test reproducibility
- Handle DataLoader workers
- Save RNG states

Critical: Same seed $\rightarrow$ Same results

Exp3: Complete Seeding

```python
def setup_seed(seed=42, deterministic=True):
    # Python RNG
    random.seed(seed)
    # NumPy RNG
    np.random.seed(seed)
    # PyTorch RNG
    torch.manual_seed(seed)
    # CUDA RNG
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    # Deterministic mode
    if deterministic:
        torch.use_deterministic_algorithms(True)
```

Experiment 4: AMP and Compilation

Goal: Benchmark performance optimizations
Configurations tested:

- Baseline (FP32, no compile)
- AMP only (FP16/BF16)
- Compile only (torch.compile)
- AMP + Compile

Expected speedup: 2-4x on GPU

Exp4: Benchmark Results

Configuration	**Time (ms/step)**	**Speedup**
Baseline (FP32)	100	1.0x
AMP only	60	1.7x
Compile only	55	1.8x
AMP + Compile	35	2.9x

Combining AMP with compilation gives best performance!

Experiment 5: Standard Code Header

Goal: Implement reusable components
Components:

- Seeding functions
- Device management
- AMP context manager
- DQN training step
- Policy evaluation
- Model compilation

This becomes your toolkit for the course!

Exp5: DQN Training Step

```python
def dqn_td_step(q_net, target_q_net, batch,
                gamma=0.99, optimizer=None):
    states, actions, rewards, next_states, dones = batch

    # Current Q-values
    q_values = q_net(states).gather(1, actions.unsqueeze(1))

    # Target Q-values
    with torch.no_grad():
        next_q = target_q_net(next_states).max(1)[0]
        targets = rewards + gamma * (1 - dones) * next_q

    loss = F.smooth_l1_loss(q_values.squeeze(), targets)

    if optimizer:
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    return loss.item()
```

Experiment 6: Logging and TensorBoard

Goal: Set up experiment tracking
Features:

- Automatic system info logging
- Scalar and histogram tracking
- Hyperparameter logging
- Model graph visualization

View results: `tensorboard -logdir runs`

Experiment 7: Checkpointing

Goal: Save and restore training state

What to save:

- Model weights
- Optimizer state
- Learning rate scheduler
- Training step/epoch
- RNG states
- Loss history

Enable training continuation after interruption!

Experiment 8: Git Integration

Goal: Version control for experiments
Best practices:

- Commit before experiments
- Include config hash in commits
- Track results with Git LFS
- Use meaningful commit messages
- Tag successful experiments

Experiment 9: Integration Test

Goal: Validate complete setup

Tests performed:

1. Environment check
2. Reproducibility verification
3. Model training
4. DQN components
5. Checkpoint save/load
6. Logging functionality

Must pass all tests before proceeding!

Key Takeaways

1. **RL is different:** Sequential decisions, delayed rewards
2. **MDP framework:** Foundation for all RL algorithms
3. **Reproducibility matters:** Always set seeds
4. **Device awareness:** CUDA > MPS > CPU
5. **Use optimizations:** AMP and compilation
6. **Track everything:** Logs, checkpoints, versions