AI Enabled Control Engineering

Lecture 11: Simulation Training of DQN, PPO, and TD3 for the Rotary Inverted Pendulum

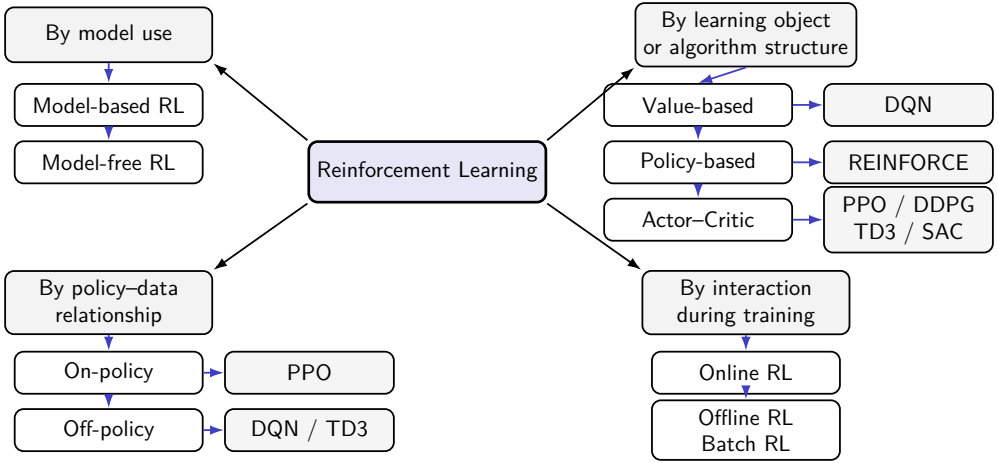
Professor Xun Huang

TA: Zhixiang Ju Haozhe Wang

Aeronautics and Astronautics
College of Engineering
Peking University

huangxun@pku.edu.cn
https://xunger99.github.io/xunger/

Common classification of reinforcement learning algorithms



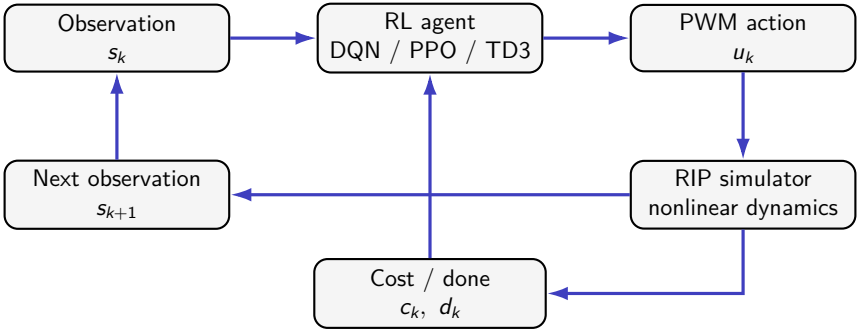
Why simulation training first?

- Real RIP training may be unsafe and time-consuming.
- Simulation allows many episodes, random initial states, and fast hyperparameter search.
- In simulation, time is virtual: training update does not violate a real $5\,\mathrm{ms}$ control period.
- A good simulation policy can be used for pretraining, algorithm selection, and deployment testing.

But simulation is not the real system

Model mismatch, motor dead zone, friction, sensor noise, and communication delay may still degrade real-system performance.

Reinforcement learning problem



One environment step

At step k, the agent observes s_k, outputs action u_k, and the simulator returns

$$s_{k+1}, \qquad c_k, \qquad d_k.$$

Q-learning idea: a table of state–action costs

For discrete states and discrete actions, we may store one value for every pair:

$$Q(s, a).$$

State	$u^{(1)}$	$u^{(2)}$	$u^{(3)}$	$u^{(4)}$
s_1	$Q(s_1, u^{(1)})$	$Q(s_1, u^{(2)})$	$Q(s_1, u^{(3)})$	$Q(s_1, u^{(4)})$
s_2	$Q(s_2, u^{(1)})$	$Q(s_2, u^{(2)})$	$Q(s_2, u^{(3)})$	$Q(s_2, u^{(4)})$
s_3	$Q(s_3, u^{(1)})$	$Q(s_3, u^{(2)})$	$Q(s_3, u^{(3)})$	$Q(s_3, u^{(4)})$

Meaning

$Q(s, a)$ means: after taking action a at state s, how large the long-term cost will be if we act optimally afterwards. For cost minimization,

$$a^*(s) = \arg\min_a Q(s, a).$$

Q-learning update from one transition

At step k, the environment gives one transition sample:

$$(s_k, a_k, c_k, s_{k+1}, d_k).$$

The Bellman target is

$$y_k = c_k + \Gamma(1 - d_k) \min_{a'} Q(s_{k+1}, a').$$

Q-learning updates only the visited table entry:

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) + \eta \left[y_k - Q(s_k, a_k) \right].$$

TD error

$$\delta_k = y_k - Q(s_k, a_k).$$

Why tabular Q-learning is not enough

Tabular Q-learning is intuitive, but it requires a finite table.

For the rotary inverted pendulum, the physical state is continuous:

$$x = \begin{bmatrix} \theta & \dot{\theta} & \alpha & \dot{\alpha} \end{bmatrix}^\top.$$

Even if the action set is finite, the number of possible states is still extremely large.

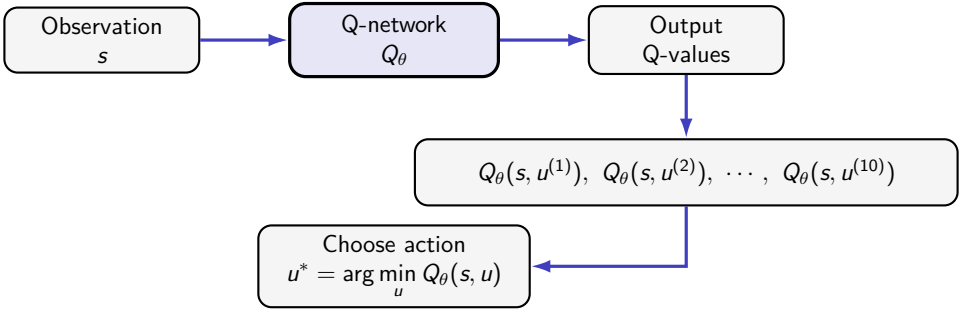
Problem

We cannot store one table entry for every possible state–action pair. Therefore, we need a function approximator.

DQN replaces the Q-table by a neural network:

$$Q_\theta(s, a) \approx Q^*(s, a).$$

DQN: replacing the Q-table by a neural network



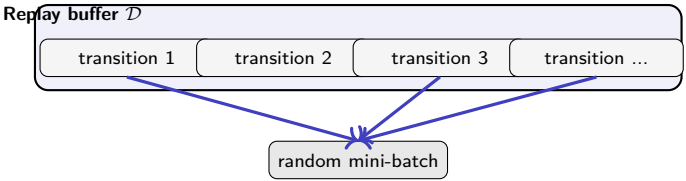
For DQN in this class

The network receives the observation s and outputs one Q-value for each discrete action. The action with the smallest predicted cost is selected during exploitation.

DQN key component 1: experience replay

During interaction, store transitions into a replay buffer:

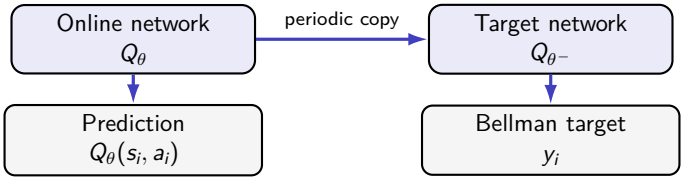
$$\mathcal{D} = \{(s_k, a_k, c_k, s_{k+1}, d_k)\}\,.$$



- Random mini-batches break temporal correlation.
- Old swing-up and balancing experiences can be reused many times.

DQN key component 2: online network and target network

DQN uses two networks:



The target is computed by the delayed network:

$$y_i = c_i + \Gamma(1 - d_i) \min_{a'} Q_{\theta^-}(s_i', a').$$

Effect

The target network makes the Bellman target more stable during training.

DQN key component 3: ϵ-greedy exploration

DQN selects actions by an ϵ-greedy behavior policy:

$$a_k = \begin{cases} \text{random action from } \mathcal{U}_d, & \text{with probability } \epsilon, \\ \arg\min_a Q_\theta(s_k, a), & \text{with probability } 1-\epsilon. \end{cases}$$

- Large ϵ: more random exploration at the beginning.
- Small ϵ: more use of the learned Q-network.
- In practice, ϵ gradually decays during training.

For RIP simulation

Exploration allows the agent to try different PWM actions and discover useful swing-up and balancing behaviors.

DQN loss function

Given a mini-batch

$$\{(s_i, a_i, c_i, s_i', d_i)\}_{i=1}^{B},$$

construct the Bellman target

$$y_i = c_i + \Gamma(1 - d_i) \min_{a'} Q_{\theta^-}(s_i', a').$$

The mini-batch loss is

$$\mathcal{L}_B(\theta) = \frac{1}{B} \sum_{i=1}^{B} \frac{1}{2} \left[y_i - Q_\theta(s_i, a_i) \right]^2.$$

Update the online network:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_B(\theta).$$

TD error

$$\delta_i = y_i - Q_\theta(s_i, a_i).$$

DQN trains the network by reducing this sample Bellman residual.

DQN training workflow: compact table view

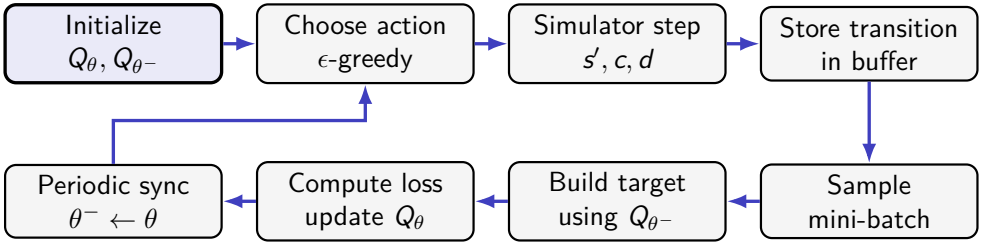
Interaction and data collection

Step	Operation
1	Initialize Q_θ, Q_{θ^-}, replay buffer $\mathcal{D}$, and RIP environment.
2	Reset simulator and obtain initial observation s_0.
3	Select action by ϵ-greedy policy.
4	Map action index a_k to discrete PWM command u_k.
5	Run one nonlinear simulator step: $(s_{k+1}, c_k, d_k) = \texttt{env.step}(u_k)$.
6	Store transition: $\mathcal{D} \leftarrow \mathcal{D} \cup (s_k, a_k, c_k, s_{k+1}, d_k)$.

Network training and saving

Step	Operation
7	After warm-up, sample a mini-batch from replay buffer $\mathcal{D}$.
8	Build Bellman target using the target network: $y_i = c_i + \Gamma(1 - d_i) \min_{a'} Q_{\theta^-}(s_i', a')$.
9	Minimize Bellman residual and update online network Q_θ.
10	Periodically synchronize target network: $\theta^- \leftarrow \theta$.
11	Evaluate current policy and save reward curves, logs, checkpoints, and config snapshots.

DQN training workflow



Summary

DQN keeps the Q-learning Bellman update, but replaces the Q-table with a neural network trained from replay-buffer samples.

DQN for RIP: discrete PWM action space

In the provided DQN setup, the action space is discrete:

$$\mathcal{U}_d = \{-150, -120, -90, -60, -30, 30, 60, 90, 120, 150\}.$$

The neural network outputs one Q value for each PWM action:

$$Q_\theta(s, u^{(1)}), \ldots, Q_\theta(s, u^{(10)}).$$

The selected control command is

$$u_k = u^{(i^*)}, \qquad i^* = \arg\min_i Q_\theta(s_k, u^{(i)}).$$

Cost-minimization convention

If the code uses reward maximization internally, the implementation may use arg max. Always check whether the variable is reward or cost.

Brief introduction to TD3

TD3 is an off-policy Actor–Critic algorithm for continuous actions.

- Actor network outputs a continuous action:

$$u = \mu_\theta(s).$$

- Critic networks estimate action values:

$$Q_{\phi_1}(s, u), \qquad Q_{\phi_2}(s, u).$$

- TD3 uses twin critics, target policy smoothing, and delayed actor updates.

TD3 can output continuous PWM values, but it may be more sensitive to model mismatch, actuator dead zones, and sim-to-real errors.

Brief introduction to PPO

PPO is an on-policy Actor–Critic algorithm.

- The actor outputs an action distribution.
- The critic estimates the state value $V(s)$.
- PPO updates the policy using recent trajectories collected by the current policy.

The clipped objective uses the probability ratio

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

and restricts overly large policy updates.
PPO can be stable in simulation, but on-policy training is usually less sample-efficient than replay-buffer based methods.

DQN, PPO, and TD3 at a glance

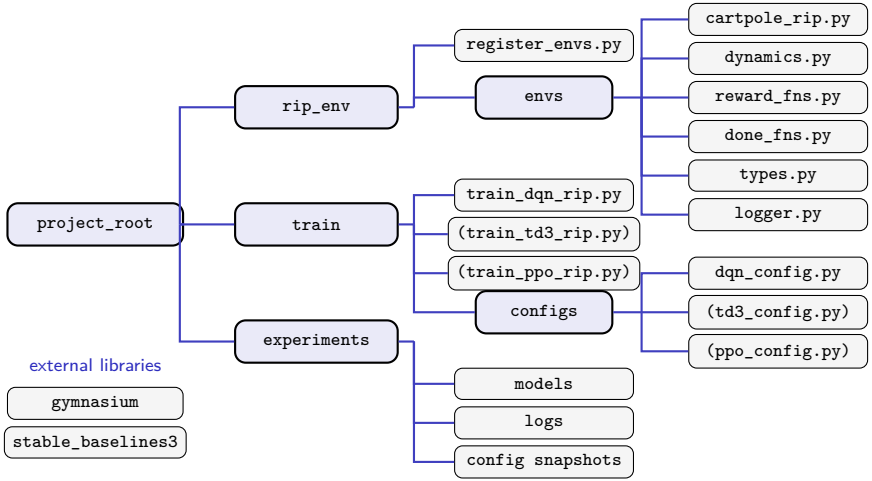
Algorithm	Main type	Action space	Data usage
DQN	value-based	discrete	off-policy replay
PPO	Actor–Critic	discrete/continuous	on-policy trajectories
TD3	Actor–Critic	continuous	off-policy replay

- DQN: simplest connection to Q-learning and Bellman targets.
- PPO: stable policy update through clipped policy optimization.
- TD3: continuous control with twin critics and delayed policy updates.

Part II: RIP simulation training environment

- The RIP simulator is written as a Gymnasium-style environment.
- The environment implements nonlinear dynamics, numerical integration, reward, termination, and logging.
- The training script calls Stable-Baselines3 style algorithms.
- DQN, PPO, and TD3 can share the same physical environment interface.

Project file tree



Environment interface

The environment follows the standard reinforcement learning interface:

$$s_0 = \texttt{env.reset()}.$$

At each step:

$$s_{k+1}, r_k, d_k, \texttt{info} = \texttt{env.step}(a_k).$$

Inside one environment step:

- convert the selected action to PWM;
- integrate the nonlinear RIP dynamics for one sampling interval;
- compute observation, reward, and termination flag;
- optionally write logs.

Important environment parameters

Item	Typical setting in the provided config
Sampling period	$\Delta t = 0.005\,\mathrm{s}$
Maximum episode length	2000 steps
DQN action space size	10 discrete PWM actions
DQN action values	student-defined discrete PWM set
Default velocity estimate	low-pass-filtered angle difference

Action-space tuning

For DQN, the action space is limited to 10 discrete PWM actions. Students may choose their own discrete action values, such as small, medium, or aggressive PWM levels.

Velocity estimation

The provided code uses low-pass-filtered angle differences by default. Students may replace it with the observer designed in Lecture 9, but the report must clearly state which velocity-estimation method is used.

DQN hyperparameters in the provided config

Parameter	Current value in provided config
`total_timesteps`	5.0×10^6
`learning_rate`	1.0×10^{-6}
`buffer_size`	30000
`learning_starts`	2000
`batch_size`	512
`gamma`	0.95
`target_update_interval`	2000
`net_arch`	(64, 64)
`exploration_initial_eps`	1.0
`exploration_final_eps`	0.001
`exploration_decay`	400000

How to run DQN training

1. Open a terminal in the project root.
2. Make sure the Python environment has required packages installed.
3. Run the DQN training entry point:

```
python train/train_dqn_rip.py
```

The script will:

- register the custom RIP environment;
- build the environment, reward function, and termination function;
- create the DQN model;
- run training and periodically save checkpoints;
- save the final model and evaluation logs.

Training output files

After running the training script, check the experiment folder:

`experiments/dqn_rip_YYYYMMDD_HHMMSS/`

Typical contents:

- `run_config.json`: training hyperparameter snapshot;
- `env_config_snapshot.json`: environment configuration snapshot;
- `models/`: checkpoint and final model files;
- `best_model/`: best evaluation model;
- `eval_logs/`: evaluation statistics;
- `env_logs/`: episode or step logs if enabled.

What students should tune

- Learning rate: too large may be unstable; too small may learn slowly.
- Exploration schedule: controls early exploration and later exploitation.
- Reward weights: decide whether the policy prioritizes swing-up, balance, or small PWM.
- Action set: coarse actions are robust but less smooth; fine actions require more learning.
- Termination conditions: affect safety and the learning signal.
- Network size and batch size: affect training speed and stability.

Example: where to modify hyperparameters

```python
# train/configs/dqn_config.py
@dataclass
class DQNRunConfig:
    total_timesteps: int = 5_000_000
    learning_rate: float = 1e-6
    buffer_size: int = 30_000
    learning_starts: int = 2_000
    batch_size: int = 512
    gamma: float = 0.95
    target_update_interval: int = 2000
    exploration_initial_eps: float = 1.0
    exploration_final_eps: float = 0.001
    exploration_decay: float = 400000.0
    net_arch: tuple = (64, 64)
```

Student task

Choose one baseline config, then modify a small number of hyperparameters and compare the training reward and evaluation behavior.

Suggested DQN experiment groups

Experiment	Modify	Question to answer
A	baseline	Does training converge?
B	learning rate	Faster or more unstable?
C	exploration decay	More exploration or premature exploitation?
D	reward weights	Better swing-up or better balance?
E	action set	Robustness vs. smoothness

Report figure

Plot reward curves and at least one simulated control trajectory for each selected configuration.

How to evaluate a trained policy

- Training reward curve: whether learning progresses.
- Evaluation reward: whether the learned policy works without random exploration.
- Control trajectory: $\theta(t), \alpha(t), \dot{\theta}(t), \dot{\alpha}(t)$.
- PWM distribution: whether the policy frequently saturates the actuator.
- Episode length: whether the policy keeps the system within termination limits.

Important

A high reward alone is not enough. Inspect the physical trajectories and PWM commands.

Class 11 report requirements

- DQN part: algorithm workflow, hyperparameters, training curve, simulated control curve, and discussion. **60%**
- PPO part: short algorithm explanation, hyperparameters, training and control result. **20%**
- TD3 part: short algorithm explanation, hyperparameters, training and control result. **20%**
- Compare the steady-state control performance of DQN, PPO, and TD3.
  - choose a stable time interval after the transient response;
  - compare $\alpha(t)$, $\theta(t)$, and PWM curves;
  - compute mean and standard deviation of $|\alpha|$;
  - discuss which algorithm gives smaller angle error and smoother control input.

Summary of Lecture 11

- We classified reinforcement learning algorithms from several viewpoints.
- We explained DQN through Bellman target, replay buffer, target network, and ϵ-greedy exploration.
- We briefly introduced PPO and TD3 as Actor–Critic methods.
- We introduced the Gymnasium-style RIP simulation environment.
- We discussed how to run the training code and tune hyperparameters systematically.

Take-home message

Simulation training is the first screening stage for learning-based control. It helps us test algorithm ideas safely before considering real hardware deployment.