

Reinforce Learning

Play with Stable Baseline 3 (SB3)

소프트웨어 끈대 강의

노기섭 교수

(kafa46@hongik.ac.kr)

Introduction to Stable Baseline 3

Stable Baseline 3 (a.k.a sb3)?

- ✓ RL library built on PyTorch
- ✓ Reliable implementations of RL algorithms
- ✓ Designed for:
Easy use, Reproducibility, Research experiments



Key Features

- Ready-to-use algorithms:
 - PPO, DQN, A2C, SAC, TD3, ...
 - Simple and clean API
 - Works with Gymnasium environments
 - Supports custom environments
- No need to implement algorithms from scratch
 - Focus on experiment design rather than debugging
 - Ideal for rapid prototyping and research

Main References



Official Docs:

<https://stable-baselines3.readthedocs.io/en/master/>

Codes:

<https://github.com/DLR-RM/stable-baselines3>

Paper:

<https://jmlr.org/papers/volume22/20-1364/20-1364.pdf>

We will focus on three practical RL
problems

- Basics - Interface with 3rd parties
- Robot Grasping
- Traffic Signal Control

What is Gym?

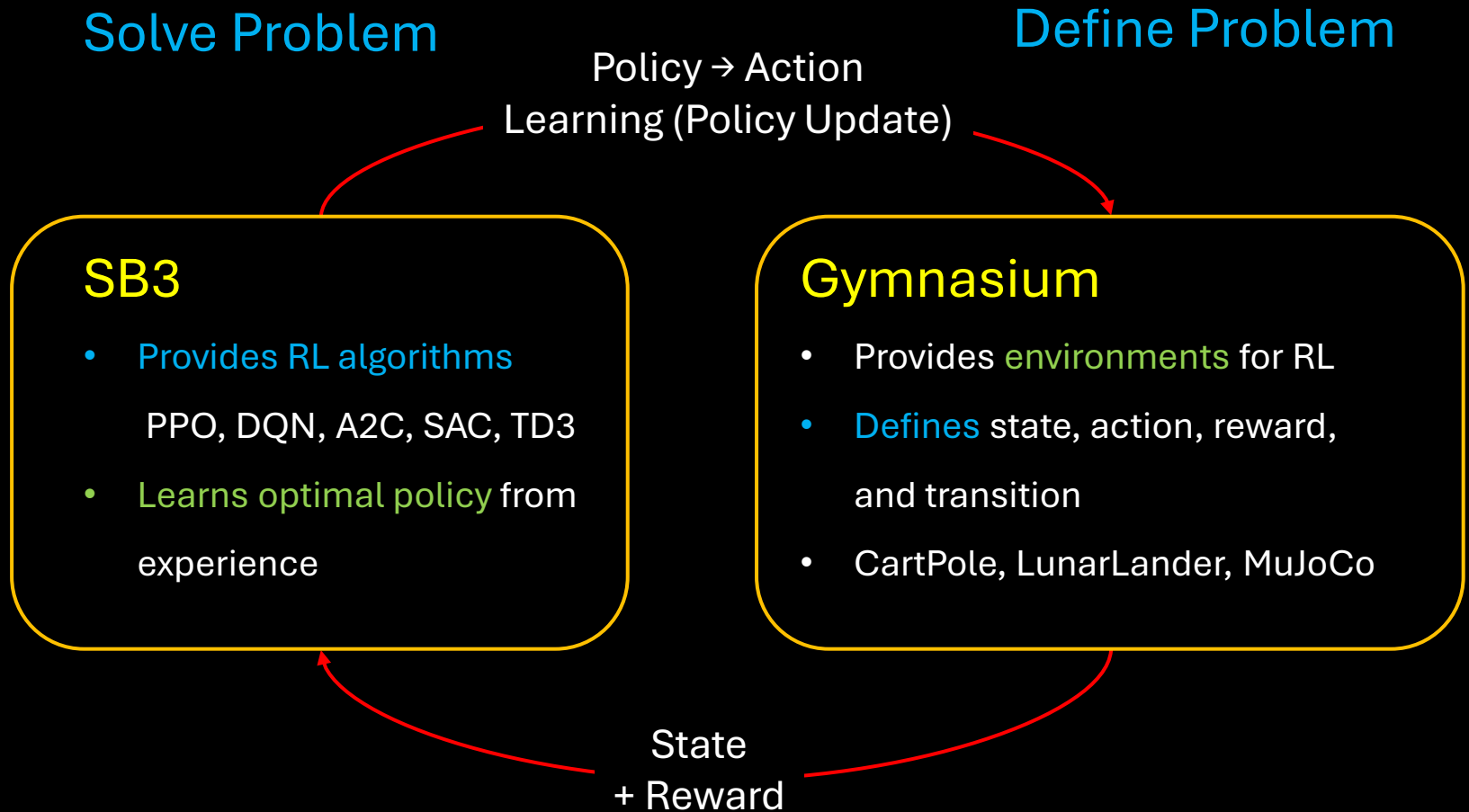
A standard interface for RL environments

→ Not a single fixed package, but an ecosystem

Gym Ecosystem Comparison

Category	Original OpenAI Gym	Gymnasium	Domain-Specific Environments
Maintainer	OpenAI	Farama Foundation	Various (DeepMind, community, etc.)
Status	Deprecated No longer maintained	Actively maintained	Actively developed (depends on package)
SB3 Compatibility	Limited / legacy support	Fully supported (recommended)	Supported if Gym-compatible
Environment Types	Basic environments (CartPole, Atari, etc.)	Same + updated versions	Robotics, games, physics, real-world simulations
Examples	CartPole-v1, MountainCar	LunarLander-v2, etc.	MuJoCo, Atari, PyBullet, custom env

Relation between SB3 & Gymnasium



01. Basics - Interface with 3rd parties

Implementation Guide: Train Structure

```
'''01_train.py'''  
import gymnasium as gym  
from stable_baselines3 import PPO  
  
# 1. Create environment  
env = gym.make("CartPole-v1")  
  
# 2. Select algorithm  
model = PPO("MlpPolicy", env, verbose=0, device="cpu")  
  
# 3. Train model  
model.learn(total_timesteps=100_000)  
  
# 4. Save model  
model.save("./trained_models/ppo-cartpole-10_000.model")
```


Implementation Guide: Evaluation Structure

```
'''02_evaluation.py'''
import gymnasium as gym
from gymnasium.wrappers import RecordVideo
from stable_baselines3 import PPO
from tqdm import tqdm

base_env = gym.make("CartPole-v1", render_mode="rgb_array")

# 에피소드 저장 객체 생성
env = RecordVideo(
    base_env,
    video_folder="videos",
    episode_trigger=lambda ep: True, # 모든 에피소드 저장
    name_prefix="ppo_cartpole_eval" # ./videos/ppo_cartpole_eval-episode-*.mp4
                                     # 동일한 폴더/파일이 있는 경우 덮어씀
)
model = PPO.load("./trained_models/ppo-cartpole-10_000.model", device="cpu")

# 에피소드 플레이
obs, _ = env.reset()
for _ in tqdm(range(2_000)):
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, terminated, truncated, _ = env.step(action)
    if terminated or truncated:
        obs, _ = env.reset()
env.close()
```

Custom Enviroment in Gynasium

Before Building a Custom Environment

Understanding the Gymnasium interface

Why?

- SB3 does **not** directly solve a problem by itself
- **Learns through interaction** with an environment
- **Must provide** the environment in a standard Gymnasium format

Key Points:

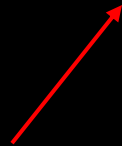
- What class should we **inherit** from?
- Which **methods** must be **overridden**?
- Which methods are **optional**?
- What is the basic code **structure**?
- How do **different problem domains** define different environments?

A Custom Environment Inherits gym.Env

Basic Idea:

- **Gymnasium** provides a **standard interface** for RL environments
- Custom environment follows this interface
- **SB3** can then **interact with it without additional changes**

```
class SimpleGraspEnv(gym.Env):  
    ...
```



By inheriting **gym.Env**,
we promise that our environment
follows the **Gymnasium API**.

Required Interfaces

Four components:

→ Define how agent sees & interacts with environment!

`some_env.action_space`

- Defines which actions the agent can take

`some_env.observation_space`

- Defines the format of the state returned to the agent

`some_env.reset()`

- Initializes the environment and returns the first observation

`some_env.step(action)`

- Applies an action and returns the next observation, reward, and termination signals

Optional Interfaces

Some interfaces are not required:

→ but are often very useful.

`some_env.render()`

- Visualizes the environment for debugging or demonstration

`some_env.close()`

- Releases windows, files, or simulation resources

`some_env.seed()`

- Makes experiments reproducible

Basic Code Structure

```
import gymnasium as gym
from gymnasium import spaces
import numpy as np

class MyEnv(gym.Env):

    def __init__(self):
        super().__init__()
        self.action_space = ...
        self.observation_space = ...

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        ...
        return obs, {}

    def step(self, action):
        ...
        return obs, reward, terminated, truncated, info
```

Terminal Commands

목적: 실습 디렉터리로 이동

```
cd /path/to/your/project/directory/01_basics
```

목적: 학습 스크립트 실행

기능: PPO로 CartPole-v1를 10,000 step 학습하고 모델 저장

예상 결과: progressbar 출력 후 trained_models/ppo-cartpole-10_000.model 생성

```
python 01_train.py
```

목적: 학습 결과 모델 파일 생성 여부 확인

예상 결과: trained_models/ppo-cartpole-10_000.model 파일이 보임

```
ls -lh trained_models
```

목적: 학습된 모델 성능 시각 확인

기능: 평가를 2,000 step 수행하면서 에피소드 영상을 videos 폴더에 저장

예상 결과: videos/ppo_cartpole_eval-episode-*.mp4 파일 생성

```
python 02_evaluation.py
```

목적: 평가 영상 생성 확인

기능: 저장된 mp4 파일 목록 조회

예상 결과: videos 폴더에 평가 영상 파일이 보임

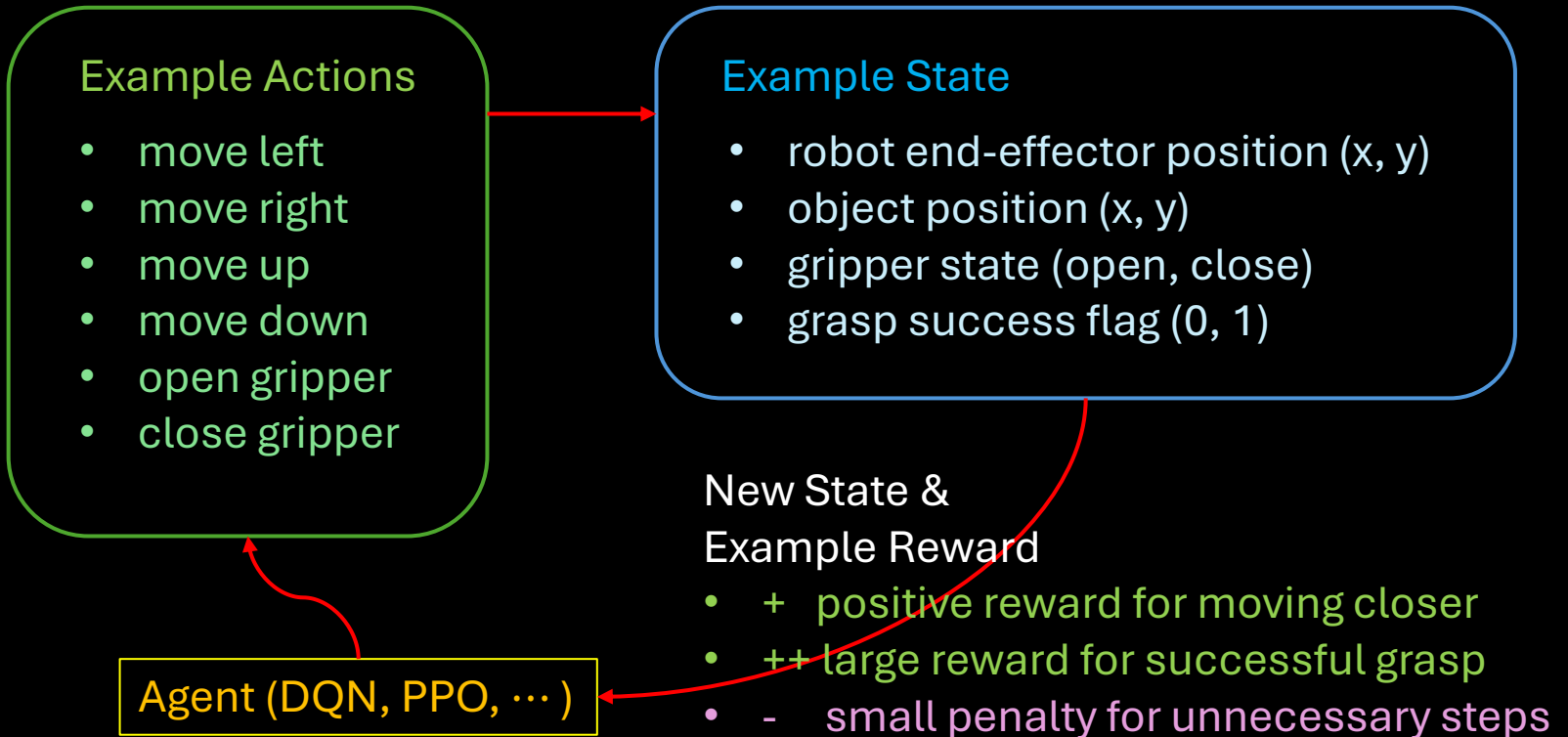
```
ls -lh videos
```


02. Robot Grasping

Phase 1. 2D Grasping with Custom Environment

Objective

- Build a custom **Gymnasium** environment from scratch
- Train an **SB3** agent to **move** toward an object and **grasp** it



Robot Grasping

Phase 1. Custom Environment

02_robot_grasping/grasp_env_2d.py

Initialize Environment

```
'''./envs/grasp_env_2d.py'''
import gymnasium as gym
from gymnasium import spaces
import numpy as np

class SimpleGraspEnv(gym.Env):
    def __init__(self):
        super().__init__()
        self.arm_x = 0.0
        self.arm_y = 0.0
        self.obj_x = 0.0
        self.obj_y = 0.0
        self.gripper = 0
        self.grasped = 0
        self.step_size = 0.05
        self.workspace_min = 0.0
        self.workspace_max = 1.0
        # Max steps per an episode
        self.max_steps: int = 100
        self.step_count: int = 0

        # define action space
        self.action_space = spaces.Discrete(6)

        # define observation space
        self.observation_space = spaces.Box(
            low=0.0,
            high=1.0,
            shape=(6,),
            dtype=np.float32
        )
```

Define **reset** Function

```
def reset(
    self,
    seed: int | None = None,
    # options: Gymnasium API compatibility argument
    options: dict[str, Any] | None = None,
) -> tuple[np.ndarray, dict[str, Any]]:
    super().reset(seed=seed)
    # Sample a new initial state distribution for each episode
    # Use self.np_random so reset(seed=...) controls reproducibility
    if options is not None:
        ...
    else:
        self.arm_x = float(self.np_random.random())
        self.arm_y = float(self.np_random.random())
        self.obj_x = float(self.np_random.random())
        self.obj_y = float(self.np_random.random())
        # initialize gripper state
        self.gripper = 0
        self.grasped = 0
    # initialize per-episode states
    self.gripper = 0
    self.grasped = 0
    self.step_count = 0

    # Gymnasium reset signature requires (observation, info)
    return (self._get_obs(), {})
```

```
def _get_obs(self) -> np.ndarray:
    return np.array([
        self.arm_x, self.arm_y,
        self.obj_x, self.obj_y,
        self.gripper,
        self.grasped
    ], dtype=np.float32)
```

Implement **step** Function

```
def step(
    self, action: int,
) -> tuple[np.ndarray, float, bool, bool, dict[str, Any]]:

    a = Action(action)
    self.step_count += 1

    if a == Action.LEFT:    self.arm_x -= self.step_size
    elif a == Action.RIGHT: self.arm_x += self.step_size
    elif a == Action.UP:    self.arm_y += self.step_size
    elif a == Action.DOWN:  self.arm_y -= self.step_size
    elif a == Action.GRIP_OPEN: self.gripper = 0
    elif a == Action.GRIP_CLOSE: self.gripper = 1

    # Keep coordinates inside observation range [0, 1]
    self.arm_x = float(np.clip(self.arm_x, 0.0, 1.0))
    self.arm_y = float(np.clip(self.arm_y, 0.0, 1.0))

    # Gymnasium step signature requires:
    # (observation, reward, terminated, truncated, info)
    obs = self._get_obs()
    reward = 0.0
    terminated = False
    truncated = self.step_count >= self.max_steps

    info = {"action_name": a.name}
    return obs, reward, terminated, truncated, info
```

```
class Action(IntEnum):
    LEFT = 0
    RIGHT = 1
    UP = 2
    DOWN = 3
    GRIP_OPEN = 4
    GRIP_CLOSE = 5
```

Upgrading **step** function via Helpers (1/2)

```
def step(
    self,
    action: int,
) -> tuple[np.ndarray, float, bool, bool, dict[str, Any]]:

    # :
    # same as previous
    # :

    obs = self._get_obs()
    reward = self._compute_reward()
    terminated = self._check_terminated()
    truncated = self.step_count >= self.max_steps
    info = {
        "action_name": a.name,
        "is_success": self._is_success(),
        "is_failure": self._is_failure(),
    }
    return obs, reward, terminated, truncated, info
```

Upgrading **step** function via Helpers (2/2)

```
def _get_obs(self) -> np.ndarray:
    return np.array([self.arm_x, self.arm_y, self.obj_x, self.obj_y,
                    self.gripper, self.grasped], dtype=np.float32)

def _is_success(self) -> bool:
    """Return True when gripper is closed and close enough to the object."""
    if self.gripper != 1: return False
    dist = np.hypot(self.arm_x - self.obj_x, self.arm_y - self.obj_y)
    return bool(dist <= self.success_threshold)

def _is_failure(self) -> bool:
    """Return True when the arm leaves the valid workspace."""
    out_x = not (self.workspace_min <= self.arm_x <= self.workspace_max)
    out_y = not (self.workspace_min <= self.arm_y <= self.workspace_max)
    return bool(out_x or out_y)

def _check_terminated(self) -> bool:
    """Return Gymnasium terminated status (success or failure)."""
    return self._is_success() or self._is_failure()

def _distance_to_object(self) -> float:
    """Return Euclidean distance between arm and object."""
    return float(np.hypot(self.arm_x - self.obj_x, self.arm_y - self.obj_y))

def _compute_reward(self) -> float:
    """Compute step reward from success/failure and distance shaping."""
    if self._is_success(): return self.success_reward
    if self._is_failure(): return self.failure_penalty
    return -self.step_penalty - (self.distance_scale * self._distance_to_object())
```


Implement `render()` function (1/2)

Add signatures for `render()`

```
class SimpleGraspEnv(gym.Env[np.ndarray, int]):  
    # need when Render Mode implementation  
    metadata = {  
        "render_modes": ["human", "rgb_array"],  
        "render_fps": 10  
    }  
  
    def __init__(  
        self,  
        render_mode: str | None = None  
    ) -> None:  
        super().__init__()  
        self.render_mode = render_mode
```

Implement `render()` function (2/2)

```
def render(self) -> np.ndarray | None:
    """Render environment in minimal human/rgb_array modes."""
    if self.render_mode is None:
        return None

    if self.render_mode == "human":
        print(
            f"step={self.step_count} "
            f"arm=({self.arm_x:.2f}, {self.arm_y:.2f}) "
            f"obj=({self.obj_x:.2f}, {self.obj_y:.2f}) "
            f"gripper={self.gripper}"
        )
        return None

    if self.render_mode == "rgb_array":
        h, w = 256, 256
        frame = np.zeros((h, w, 3), dtype=np.uint8)
        ax, ay = int(self.arm_x * (w - 1)), int(self.arm_y * (h - 1))
        ox, oy = int(self.obj_x * (w - 1)), int(self.obj_y * (h - 1))
        # arm: green pixel, object: red pixel
        frame[ay, ax] = np.array([0, 255, 0], dtype=np.uint8)
        frame[oy, ox] = np.array([255, 0, 0], dtype=np.uint8)
        return frame

    raise ValueError(f"Unsupported render_mode: {self.render_mode}")
```

Implement `close()` function

```
def close(self):  
    return super().close()
```

If your environment has no extra resources,
`close()` can simply be pass

Calling `super().close()`: you delegate to
the base `gym.Env` implementation

In practice:

- With no windows/files/sockets, both are **almost no-op**.
- Keeping `close()` still documents that the **env supports the standard interface**.

Robot Grasping

Phase 2. SB3 Agent + Training

02_robot_grasping/train.py

SB3 Agent (PPO) + Train

```
def main() -> None:
    args = parse_args()
    total_timesteps = args.timesteps
    seed = args.seed
    verbose = args.verbose

    # Directory for trained model artifacts
    model_dir = Path(__file__).resolve().parent / "trained_models"
    model_dir.mkdir(parents=True, exist_ok=True)
    run_id = datetime.now().strftime("%Y%m%d_%H%M%S")
    obs_dir = Path(__file__).resolve().parent / "obs_logs" / run_id

    # 1) Create custom environment and fix reset seed
    env = SimpleGraspEnv()
    env.reset(seed=seed)

    # 2) Build SB3 agent
    model = PPO(policy="MlpPolicy", env=env, verbose=verbose, device="cpu", seed=seed)

    # Save changing observations during training
    obs_callback = ObsSaveCallback(save_dir=obs_dir, save_freq_steps=10_000, file_prefix="grasp_obs",)

    # 3) Train
    model.learn(total_timesteps=total_timesteps, progress_bar=True, callback=obs_callback)

    # 4) Save and cleanup
    model_path = model_dir / f"ppo-grasp-{total_timesteps}.model"
    model.save(str(model_path))
    env.close()

    print(f"saved: {model_path}")
    print(f"obs_logs: {obs_dir}")
```

For Callback implementation,
See next slide

Extend `BaseCallback` to Save Train Logs

```
class ObsSaveCallback(BaseCallback):
    """Save changing observations (`new_obs`)
    to .npy chunks."""

    def __init__(
        self,
        save_dir: str | Path,
        save_freq_steps: int = 10_000,
        file_prefix: str = "obs",
        verbose: int = 0,
    ) -> None:
        super().__init__(verbose)
        self.save_dir = Path(save_dir)
        self.save_freq_steps = int(save_freq_steps)
        self.file_prefix = file_prefix
        self._buffer: list[np.ndarray] = []
        self._file_index = 0

    def _on_training_start(self) -> None:
        # override hook
        self.save_dir.mkdir(
            parents=True, exist_ok=True
        )
```

```
    def _on_step(self) -> bool:
        # override hook: per step
        new_obs = np.asarray(self.locals["new_obs"])
        self._buffer.append(new_obs.copy())
        if self.num_timesteps % self.save_freq_steps == 0:
            self._flush()
        return True

    def _on_training_end(self) -> None:
        # override hook: training end
        self._flush()

    def _flush(self) -> None:
        # helper (new method)
        if not self._buffer:
            return
        arr = np.asarray(self._buffer, dtype=np.float32)
        path = self.save_dir /
            f"{self.file_prefix}_{self._file_index:04d}.npy"
        np.save(path, arr)
        self._file_index += 1
        self._buffer.clear()
```

Build Helper Function to Construct Video (.mp4)

```
"""03_robot_grasping/obs_to_video.py"""
```

```
def normalize_obs_shape(obs: np.ndarray) -> np.ndarray:
    """Normalize to (T, obs_dim)."""
    if obs.ndim == 2: return obs
    if obs.ndim == 3: return obs[:, 0, :] # (T, n_env, obs_dim) -> select first env
    raise ValueError(f"Unsupported obs shape: {obs.shape}")

def obs_to_frame(o: np.ndarray, size: int) -> np.ndarray:
    """Render one observation to RGB frame."""
    frame = np.zeros((size, size, 3), dtype=np.uint8)
    arm_x, arm_y, obj_x, obj_y, gripper, _grasped = o.tolist()
    ax = int(np.clip(arm_x, 0.0, 1.0) * (size - 1))
    ay = int(np.clip(arm_y, 0.0, 1.0) * (size - 1))
    ox = int(np.clip(obj_x, 0.0, 1.0) * (size - 1))
    oy = int(np.clip(obj_y, 0.0, 1.0) * (size - 1))
    frame[max(0, oy - 3):min(size, oy + 4), max(0, ox - 3):min(size, ox + 4)] = np.array(
        [255, 70, 70], dtype=np.uint8
    )
    frame[max(0, ay - 3):min(size, ay + 4), max(0, ax - 3):min(size, ax + 4)] = np.array(
        [80, 255, 80], dtype=np.uint8
    )
    if int(gripper) == 1:
        frame[8:20, 8:20] = np.array([255, 255, 255], dtype=np.uint8)
    return frame

def load_frames_from_npy(path: Path, size: int) -> list[np.ndarray]:
    obs = np.load(path)
    obs = normalize_obs_shape(np.asarray(obs, dtype=np.float32))
    return [obs_to_frame(o, size=size) for o in obs]
```

Complete Codes are
Implemented in
obs_to_video.py

03. Traffic Signal Control

Why Reinforcement Learning for Traffic Control?

Traditional Traffic Signal Control

- Fixed-time traffic signals
- Simple rule-based control using sensors
- Limitation: Poor adaptability to dynamic and unpredictable traffic flows

Reinforcement Learning (RL) Approach

- Learns by interacting with the traffic environment
- Adapts signal timing in real time
- Uses current intersection states
(queue length, traffic flow, congestion)

Optimization Goals

- Minimize Total Waiting Time
- Maximize Throughput
- Reduce CO₂ Emissions
(less stop-and-go behavior)

Traffic Signal Control with SUMO

What is SUMO (**S**imulation of **U**rban **M**obility)?

An open-source, highly portable, microscopic road traffic simulation package designed to handle large networks

sumo-rl:

- A library that provides a Gymnasium-compatible interface for SUMO.
- Key Benefit: Allows seamless integration with Stable Baseline 3 (sb3) algorithms.

<https://github.com/LucasAlegre/sumo-rl>

Install Required Packages

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
pip install stable-baselines3 \
    gymnasium[classic-control] \
    "torch>=2.0"
```

- stable-baselines3: DQN 강화학습 알고리즘 프레임워크
- gymnasium: 강화학습 환경(Environment) 표준 인터페이스 제공
- numpy: 상태 관측값(Observation) 및 보상 등 수치 데이터 배열 연산
- traci: Python 환경에서 SUMO 시뮬레이터를 제어하고 통신하기 위한 API
- sumolib: SUMO 네트워크 파일 파싱 및 실행 파일 경로 탐색 유틸리티

Install **sumo**, **netconvert** binary as system command

In Ubuntu/Debian

install binary

```
sudo apt update
```

```
sudo apt install sumo sumo-tools sumo-doc
```

check installation

```
which sumo
```

```
which netconvert
```

```
sumo --version
```

```
netconvert --version
```

add to environment file

```
echo 'export SUMO_HOME=/usr/share/sumo' >> ~/.bashrc
```

```
echo 'export PYTHONPATH="$SUMO_HOME/tools:$PYTHONPATH"' >> ~/.bashrc
```

```
source ~/.bashrc
```

Project Structure Design

03_traffic_signal_control/

└─ env_sumo_single.py	# SUMO 단일 교차로 Gymnasium 환경 정의(상태/행동/보상/phase 전환)
└─ evaluate.py	# 학습 정책(DQN)과 고정 신호(Fixed-time) 평가, 지표 집계 및 CSV 저장
└─ record_video.py	# 학습된 정책 롤아웃 실행 후 프레임 수집, mp4 영상 생성
└─ train_dqn.py	# SB3 DQN 학습 실행, 체크포인트(.zip) 저장
└─ models/	# 학습된 모델 파일 저장 디렉터리
└─ results/	# 평가 결과 CSV, 로그(TensorBoard 등) 저장 디렉터리
└─ videos/	# 정책 실행 영상(mp4) 저장 디렉터리
└─ sumo_data/	# SUMO 네트워크/신호/교통수요 입력 파일 모음
└─ connections.con.xml	# 진입/진출 엣지 연결 및 신호 linkIndex 정의
└─ edges.edg.xml	# 도로 엣지(방향/차선/속도) 정의
└─ nodes.nod.xml	# 교차로/외곽 노드 위치 및 타입 정의
└─ routes.rou.xml	# 차량 타입, 경로, 교통량(flow) 정의
└─ single.sumocfg	# SUMO 실행 설정(네트워크/라우트/시뮬레이션 시간)
└─ tls.tll.xml	# 신호등 phase 상태(G/y/r) 및 기본 신호 로직 정의

03. Traffic Signal Control (Create Traffic Environment)

Problem Setup (Single-Intersection Traffic Signal Control)

Define the RL task:

controlling **one traffic light** at a single intersection in SUMO

Set the optimization goal:

Reduce congestion-related costs while **keeping traffic flowing**

Clarify agent responsibility:

at each decision step,

choose **exactly one incoming direction** to receive green

Measurable outcomes used later in evaluation:

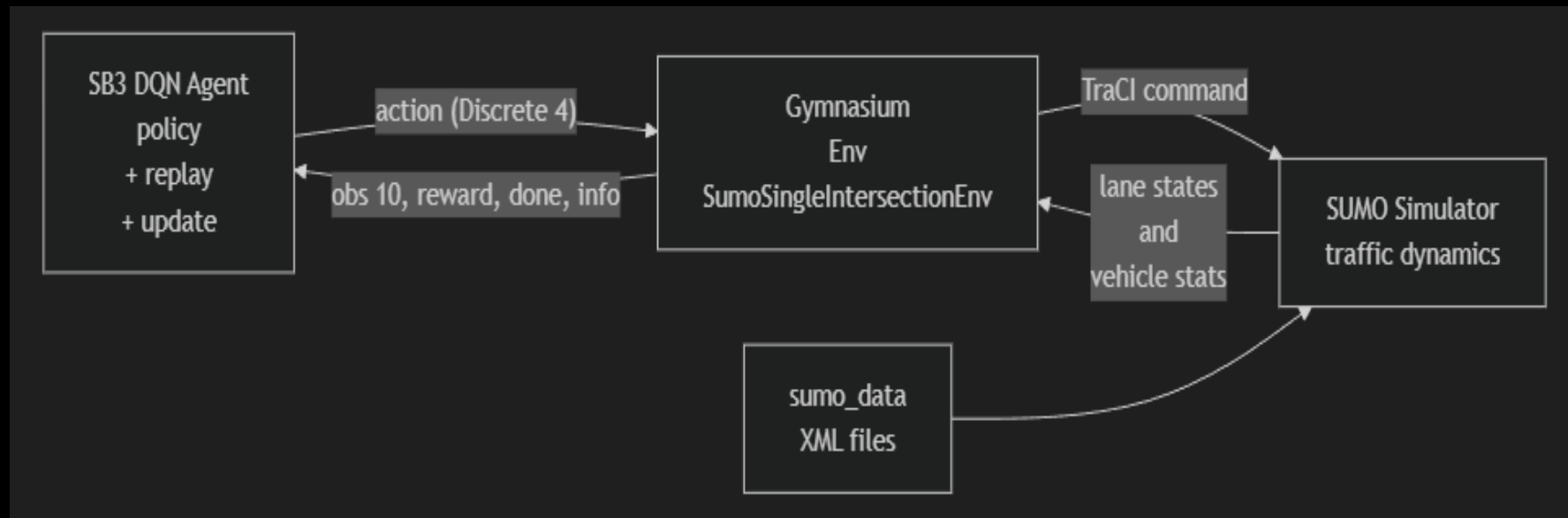
Average waiting time, Average travel time,

Total queue length, Throughput

Point students to implementation entry:

03_traffic_signal_control/**env_sumo_single.py**

SB3-Gymnasium-SUMO: Interface Control Document



- SB3 provides the learning algorithm (DQN) and interacts with the environment through the Gymnasium interface.
- The Gymnasium environment (`env_sumo_single.py`) acts as an adapter that directly controls and queries SUMO.
- SUMO computes the actual traffic simulation state, and the environment converts it into observations/rewards and returns them to the agent.

Action, Observation, Reward

- **Action (Discrete(4))**
 - 0: North-only green (through + left-turn)
 - 1: South-only green (through + left-turn)
 - 2: East-only green (through + left-turn)
 - 3: West-only green (through + left-turn)
- **Observation (shape=(10,))**
 - Queue length for each direction (4)
 - Mean speed for each direction (4)
 - Current phase (normalized) (1)
 - Phase elapsed time (normalized) (1)
- **Reward**
 - `reward = - total_queue_length`
- **Constraints**
 - Insert a yellow phase when switching phases
 - Apply minimum green time

Class Skeleton Design (Gymnasium Env Class Structure)

`class SumoSingleIntersectionEnv(gym.Env)` inherits from `gym.Env`

This class connects SUMO traffic simulation to the Gymnasium RL interface

- SUMO simulates traffic
- Gymnasium defines the RL API
- This class connects both

Core Gymnasium Methods

<code>reset()</code>	<code># start a new episode</code>
<code>step(action)</code>	<code># apply action and return next transition</code>
<code>render()</code>	<code># create visualization frame</code>
<code>close()</code>	<code># close SUMO/TraCI connection</code>

Custom Helpers

<code>_start_sumo()</code>	<code># launch SUMO</code>
<code>_simulate_seconds()</code>	<code># advance simulation</code>
<code>_compute_observation()</code>	<code># build RL state</code>
<code>_episode_metrics()</code>	<code># calculate evaluation metrics</code>

Initialization (Configuration, Spaces, and Internal State)

Key simulation parameters:

delta_time # simulation seconds per RL action
yellow_time # yellow transition time when phase changes
min_green # minimum green duration
max_steps # episode length limit

Agent' selection (options):

0: North-only green
1: South-only green
2: East-only green
3: West-only green

Observation space (vector):

4 queue lengths
4 average speeds
current phase
elapsed phase time

Action space:

spaces.Discrete(4)

Internal state:

current_phase
elapsed_phase_time
sim_step

Reset Flow (Episode Start and Reproducibility)

reset() starts a new episode

Launch SUMO and open a TraCI connection

```
self.current_phase = 0  
self.elapsed_phase_time = 0  
self.sim_step = 0
```

Reset signal state and simulation step count

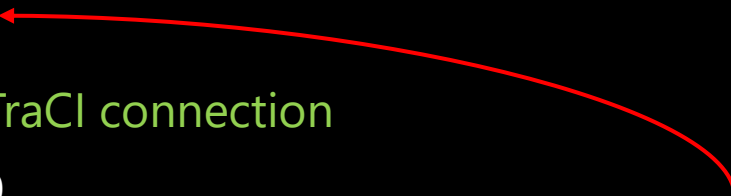
```
self._depart_time.clear()  
self._latest_waiting.clear()  
self._completed_waiting.clear()  
self._completed_travel.clear()
```

Clear episode metric buffers

```
self.conn.trafficlight.setPhase(self.tls_id, self.current_phase)
```

Set initial phase to North-only

```
obs = self._compute_observation()  
return obs, info
```

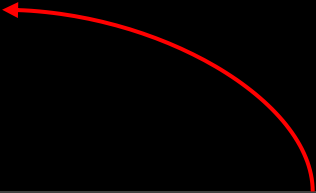


Creates a clean episode, then each training rollout starts from a controlled initial state

Step Logic (Action, Reward, and Termination)

`step(action)` is the **main RL interaction point**

- agent action
 - signal phase update
 - SUMO simulation step
 - observation + reward



`step()` converts an agent action into a traffic signal decision, then returns the result as the next RL transition

Phase switching rule:

```
can_switch = (  
    action != current_phase  
    and elapsed_phase_time >= min_green  
)
```

If switching is allowed:

insert yellow phase
then apply selected green phase

Reward:

$\text{reward} = -\text{total_queue_length}$

Helpers & Metrics (Simulation Signals to RL Data)

Helper methods:

convert SUMO simulation data into RL inputs and evaluation metrics

- `_start_sumo()` # launch SUMO and connect TraCI
- `_find_yellow_phase_index()` # find yellow transition phase
- `_simulate_seconds()` # advance simulation and collect traffic data
- `_compute_observation()` # build observation vector
- `_episode_metrics()` # summarize episode metrics

Main data flow: SUMO simulation

Data for Agent:

- lane queue length,
- lane average speed,
- current signal phase,
- elapsed phase time

Data for Evaluation:

- vehicle departure time,
- vehicle arrival time,
- vehicle waiting time,
- number of arrived vehicles

Key metrics:

- `avg_waiting_time`
- `avg_travel_time`
- `total_queue_length`
- `throughput`

03. Traffic Signal Control

(Training Agent in Our Environment)

Train DQN (1/4) - Training Pipeline

Main flow: `create environment`
→ `wrap with Monitor`
→ `create DQN model`
→ `train with model.learn()`
→ `save trained model`

Code structure (code level):

```
env = SumoSingleIntersectionEnv(...)
```

```
env = Monitor(env)
```

```
model = DQN("MlpPolicy", env, ...)
```

```
model.learn(total_timesteps=...)
```

```
model.save(...)
```

- SB3 does not need to know SUMO details
- It only interacts with the Gymnasium Env interface

Train DQN (2/4) - Environment Wrapping

Create the SUMO traffic signal environment

```
env = SumoSingleIntersectionEnv(  
    use_gui=False,  
    delta_time=args.delta_time,  
    min_green=args.min_green,  
    yellow_time=args.yellow_time,  
    max_steps=args.max_steps,  
)
```

Monitor records episode-level training information

- episode reward
- episode length
- training progress logs

Wrap the environment with Monitor

```
from stable_baselines3.common.monitor import Monitor  
  
# Monitor 래퍼: episode return/length 등 학습 로그 기록  
env = Monitor(env)
```

Custom SUMO environment becomes an SB3-compatible training environment

Train DQN (3/4) - DQN Configuration

Why DQN fits?

- action space is Discrete(4)
- DQN is designed for discrete action problems

```
model = DQN(  
    "MlpPolicy",          # neural net policy for vector observation  
    env,  
    learning_rate=1e-3,    # optimizer 학습률  
    buffer_size=50_000,    # 리플레이 버퍼 최대 크기  
    learning_starts=1_000, # 버퍼가 일정량 쌓인 후 학습 시작  
    batch_size=64,         # 한 번 업데이트에 사용하는 샘플 수  
    gamma=0.99,           # 미래 보상 할인율  
    train_freq=4,          # 환경 스텝 기준 학습 주기  
    target_update_interval=500, # 타겟 Q 네트워크 동기화 주기  
    exploration_fraction=0.3, # epsilon-greedy 탐험 비율 스케줄  
    exploration_final_eps=0.05,  
    verbose=1,  
    seed=args.seed,  
    tensorboard_log="./results/tb_dqn", # TensorBoard 로그 저장 경로  
)
```

Train DQN (4/4) - Learning and Model Saving

Start training with SB3

```
model.learn(  
    total_timesteps=args.timesteps,  
    progress_bar=True,  
)
```



During training:

DQN selects an action
→ environment runs step(action)
→ reward and next observation are returned
→ replay buffer stores experience
→ Q-network is updated



Save trained model

```
out_path = Path(args.model_out)  
out_path.parent.mkdir(parents=True, exist_ok=True)  
model.save(str(out_path))
```

[models/dqn_sumo_single.zip](#)

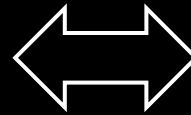
Reused for evaluation
and video recording

03. Traffic Signal Control (Evaluate Policy)

DQN vs Fixed-Time Baseline

DQN Policy

- load trained DQN model
- run DQN episodes
- run Fixed-Time baseline episodes
- collect traffic metrics
- save results to CSV



Fixed-time (Baseline) policy

North

→ South

→ East

→ West... repeat with fixed interval

→ save results to CSV

Metrics:

avg_waiting_time

avg_travel_time

total_queue_length

throughput

results/eval_metrics.csv

Evaluation shows whether the learned policy performs better than a simple fixed-time traffic signal

03. Traffic Signal Control (Record Video)

Record Video - Policy Rollout Visualization

load trained DQN model

→ create SUMO environment → run policy rollout

→ collect render frames → save frames as mp4

Policy action:

```
action, _ = model.predict(obs, deterministic=True)
```

Frame collection:

```
frames.append(env.render())
```

Video output:

```
videos/dqn_policy_rollout.mp4
```

03. Traffic Signal Control (Terminal Command)

Full Pipeline - End-to-End Execution

```
# Move to the traffic signal control example directory  
cd /path/to/your/project/play_with_sb3/03_traffic_signal_control
```

```
# Train a DQN agent and save the trained model  
python train_dqn.py \  
  --timesteps 3000 \  
  --model-out models/dqn_sumo_single_steps-3000
```

```
# Evaluate the trained DQN policy against the fixed-time baseline  
python evaluate.py \  
  --model models/dqn_sumo_single_steps-3000.zip \  
  --episodes 5
```

```
# Run the trained policy and save the rollout as an mp4 video  
python record_video.py \  
  --model models/dqn_sumo_single_steps-3000.zip \  
  --output videos/dqn_policy_rollout.mp4
```



수고하셨습니다..^^..