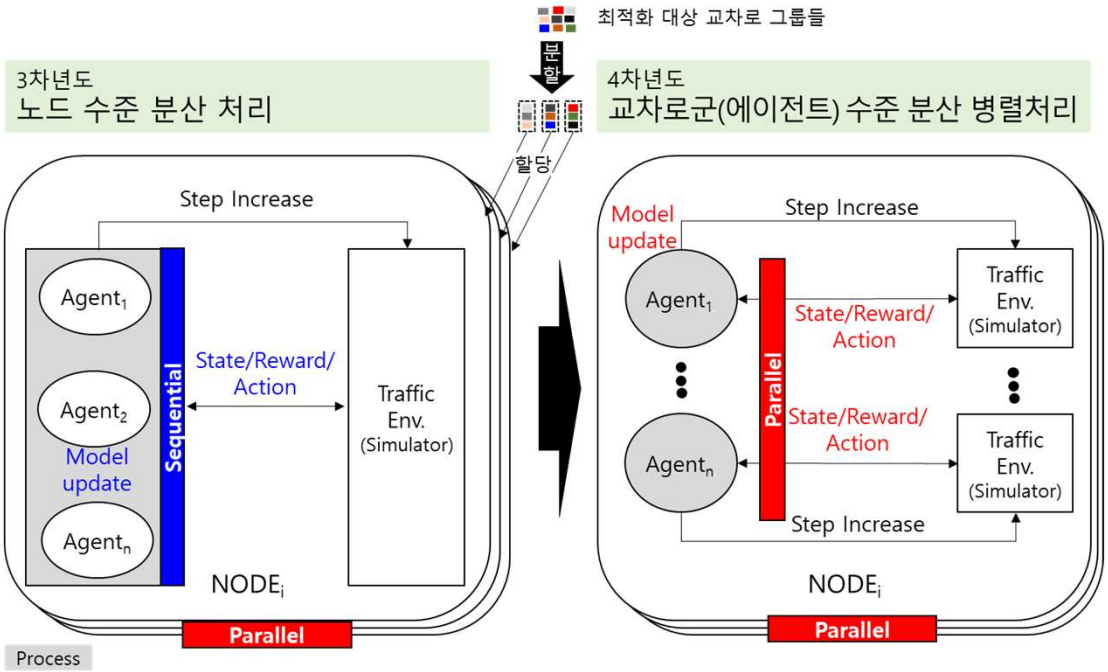
신호 최적화 분산처리 확장 개발을 위한 프로세스 구조 설계(안)

2023. 02.

연구 목표 및 내용 – 분산처리 확장 개발

- 3차년도 진행 내용
  - 4개 교차로군(SA 1, SA 6, SA 17, SA 61)으로 구성된 대전시 서구/유성구 53개 교차로 대상
  - 신호 최적화 학습의 노드 수준 분산 처리
  - 비분산처리 대비 학습 소요 시간 단축 : 해당 지역에 대해서는 1/3로 단축

- 4차년도 계획 : 학습 속도 · 확장성 개선
  - 목표 : 신호 최적화 교차로 200개 이상 지원
  - 분산처리 고도화 : 교차로군(에이전트) 수준의 분산 병렬처리 (에이전트별 프로세스)



신호 최적화 분산처리 확장 개발 연구 내용

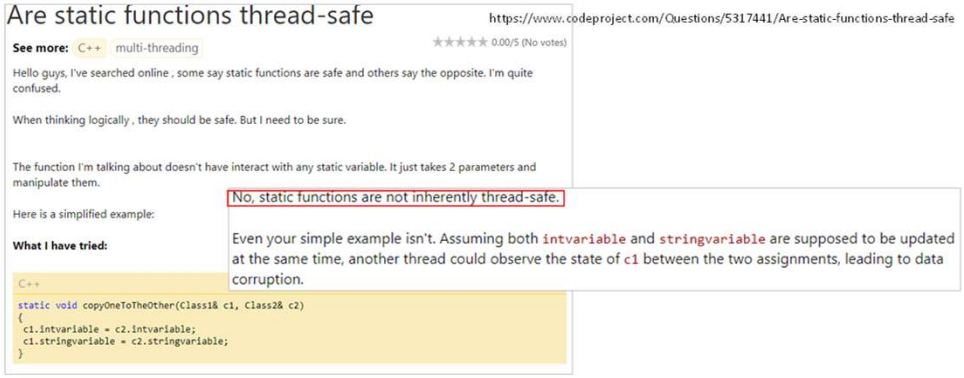
- 목표 : 신호 최적화 신호 교차로 개수 200개 이상
  - 학습시간 단축
    - 학습시간 : 수행시간/epoch. 즉, 1-epoch 수행 소요시간
      - ✓ 시뮬레이션 시간 + 에이전트의 환경 제어 시간 + 에이전트 모델 갱신 시간
  - 고려 제외
    - 교차로 통과시간 개선 ← 다른 소그룹에서 진행 중
    - 시뮬레이션 병렬처리
    - 엣지, 시각화 연동

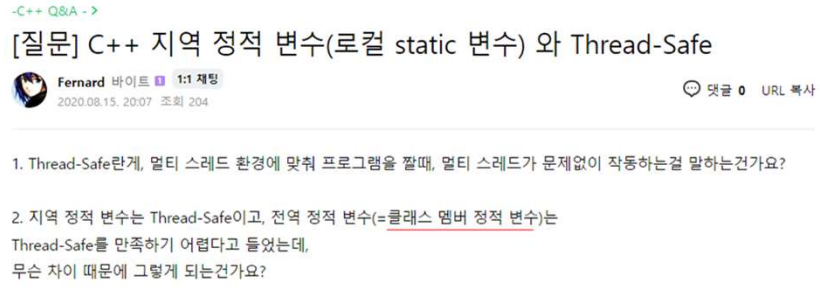
- **노드 수준 분산 처리**에서 **에이전트 수준 분산 병렬처리**로
  - 3차년도: 여러 노드 활용한 노드 수준 병렬처리
    - (학습) 단일 프로세스 내에서 교차로 그룹별로 하나의 에이전트 객체 생성(할당)하여 학습
    - (학습) 하나의 노드에 여러 교차로 그룹이 할당된 경우 교차로 그룹별 순차적으로 학습
      - ✓ 교통 환경 공유; 순차적으로 상태 정보 수집, 행동 추론/적용, 보상 정보 수집, 모델 갱신
    - (검증) 대상 교차로 그룹 전체에 대해 에이전트들의 신호 제어를 순차적으로 적용하며 검증
  - 4차년도(안) : 여러 노드/프로세스 활용한 에이전트(교차로 그룹) 수준 병렬처리
    - (학습) 교차로 그룹별로 하나의 에이전트 프로세스 생성(할당)하여 학습
    - (학습) 동일 노드에 할당된 여러 교차로 그룹에 대해 교차로 그룹별 분산 병렬로 학습
    - (학습) 하나의 에이전트가 동시에 다수의 교통 환경 연동하게 하여 경험 확대(optional)
    - (검증) 대상 교차로 그룹 전체에 대해 에이전트들의 신호 제어를 순차적으로 적용하며 검증
      - ✓ 하나의 교통 환경에 훈련된 모델들의 추론 결과를 적용해야 하므로 순차적일 수 밖에 없다
    - 고려사항 : MT-Safe ? static methods/variables, GIL(Global Interpreter Lock), libsalt,…

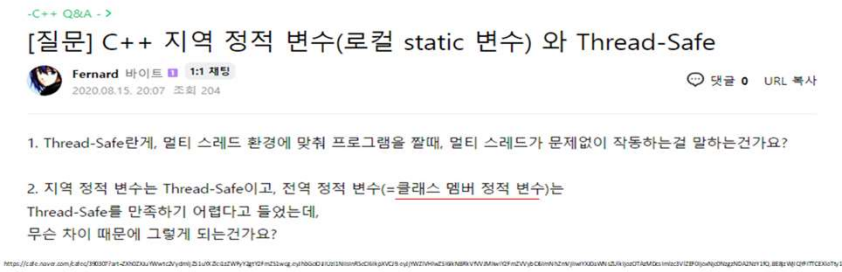
쓰레드를 사용할 수 없을 것 같다

MT-Safe

- MT-Safe in CPP
  - 전역 변수 사용 : unsafe
  - 정적 멤버 변수/메쏘드의 사용
    - 쓰레드 환경에서 공유되므로 MT-Safe 하지 않을 수 있다
      - ✓ 동기화 방법을 사용해야 한다.







쓰레드를 사용할 수 없을 것 같다

Static member methods in SALT simulator

```
libsalt/TrafficSignal.h:29:        static std::vector<std::string> getTLSConnectedLinkID(std::string _tlsID);
libsalt/TrafficSignal.h:30:        static std::vector<std::string> getTLSScheduleIDsByNodeID(std::string _tlsID)
libsalt/TrafficSignal.h:31:        static std::string getCurrentTLSScheduleIDByNodeID(std::string _tlsID);
libsalt/TrafficSignal.h:32:        static TLSSchedule* getCurrentTLSScheduleByNodeID(std::string _tlsID);
libsalt/TrafficSignal.h:33:        static std::vector<std::pair<int,std::string>> getTLSPhasesByNodeID(std::stri
libsalt/TrafficSignal.h:34:        static std::pair<int,std::string> getTLSPhaseByNodeID(std::string _tlsID, std
libsalt/TrafficSignal.h:35:        static std::string getCurrentTLSPhaseStateByNodeID(std::string _tlsID);      lane);
libsalt/TrafficSignal.h:36:        static int getCurrentTLSPhaseIndexByNodeID(std::string _tlsID);
libsalt/TrafficSignal.h:37:        static int getLastTLSPhaseSwitchingTimeByNodeID(std::string _tlsID);         e);
libsalt/TrafficSignal.h:40:        static int setTLSByNodeID(std::string _tlsID, TLSLogic* _logic);
libsalt/Cell.h:47:        static float getAverageVehicleSpeed(const std::string &linkID, int section, int lane);
libsalt/Cell.h:48:        static float getAverageWaitingTime(const std::string &linkID, int section, int lane);
libsalt/Link.h:42:        static float getAverageVehicleSpeed(const std::string& linkID);
libsalt/Link.h:43:        static float getAverageVehicleWaitingTime(const std::string& linkID,
```

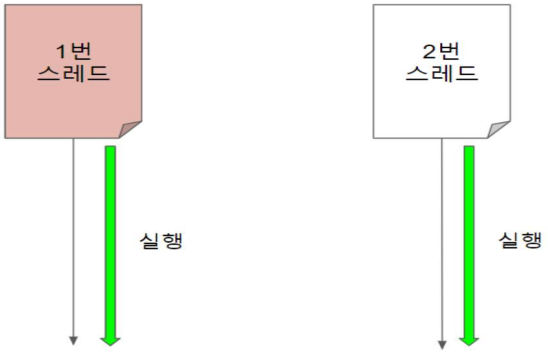
```
libsalt.py:2812:        getNumVehPassed = staticmethod(_libsalt.cell_getNumVehPassed)
libsalt.py:2816:        getSumTravelTime = staticmethod(_libsalt.cell_getSumTravelTime)
libsalt.py:2820:        getSumTravelLength = staticmethod(_libsalt.cell_getSumTravelLength)
libsalt.py:2824:        getAverageWaitingQLength = staticmethod(_libsalt.cell_getAverageWaitingQLength)
libsalt.py:2828:        getAverageDensity = staticmethod(_libsalt.cell_getAverageDensity)
libsalt.py:2832:        getAverageNumVehicles = staticmethod(_libsalt.cell_getAverageNumVehicles)
libsalt.py:2836:        getAverageSpeed = staticmethod(_libsalt.cell_getAverageSpeed)
libsalt.py:2840:        getAverageVehicleSpeed = staticmethod(_libsalt.cell_getAverageVehicleSpeed)
```

쓰레드를 사용할 수 없을 것 같다

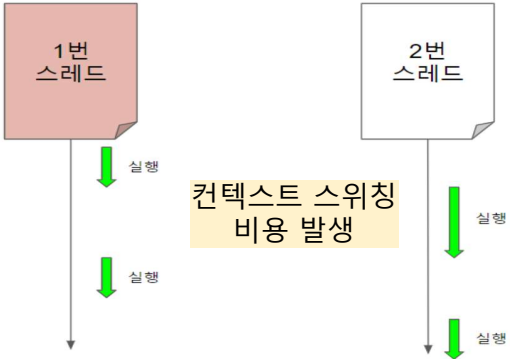
Python – GIL (Global Interpreter Lock)

- 여러 쓰레드가 동시에 Python 바이트 코드를 실행하지 못하게 막는 일종의 뮤텍스
- 파이썬으로 작성된 프로세스는 한 시점에 하나의 쓰레드에만 모든 자원을 할당하고 다른 쓰레드는 접근할 수 없게 막아버리는데, 이 역할을 GIL이 수행

일반적인 멀티쓰레딩



파이썬에서의 멀티쓰레딩



- 멀티 쓰레딩의 성능 libsalt 함수 호출은 CPU 연산일까 I/O 연산일까? CPU 연산
  - CPU 연산이 큰 비중을 차지하는 경우에는 멀티 쓰레딩의 성능이 싱글 쓰레드보다 떨어짐
  - I/O 작업이 큰 비중을 차지하거나 Sleep으로 일정 시간 대기해야 하는 경우에 좋음
- 왜 GIL 을 사용하는가?
  - 메모리 관리시 Reference Count 값을 정확하게 관리하여 적절하게 GC가 처리하도록 하기 위함
- **GIL 방해 받지 않고 병렬 연산 처리 위해 멀티프로세싱을 사용해야**
  - 컨텍스트 스위칭 비용이 크나 독자적인 메모리를 가지기에 GIL의 영향 X

Ref. https://blog.naver.com/ikarte666/222644186685

신호 최적화 분산처리 확장 개발 세부 내용

● 신호 최적화

– (실행 데몬 쪽)프로세스 구조 개선 : Agent별 Process화를 통한 병렬처리

- 실행 데몬 프로세스에서 교차로 그룹별 Agent Process 생성
  - ✓ Agent별 별도의 교통 환경 생성
  - ✓ Agent별로 독립적으로 학습

- 실행 데몬 프로세스 역할
  - ✓ (필요시) 학습 실행 환경 구축 : ~~시나리오 복사~~, 학습 결과(중간) 파일 생성 경로 설정, …
  - ✓ 제어 데몬과 메시지 교환
    - ❖ (재) 학습 실행, 학습 종료
  - ✓ Agent Process 생성 & 제어(학습 진행, 학습 상황 체크)
    - ❖ Agent 프로세스 생성은 분산 처리를 위한 쉘 스크립트에서는 불가
      - 담당해야 할 최적화 대상이 할당된 이후에만 가능

학습 실행 환경 구축 대신….

시뮬레이터에
시뮬레이션 출력 파일
생성 루트 경로를 전달

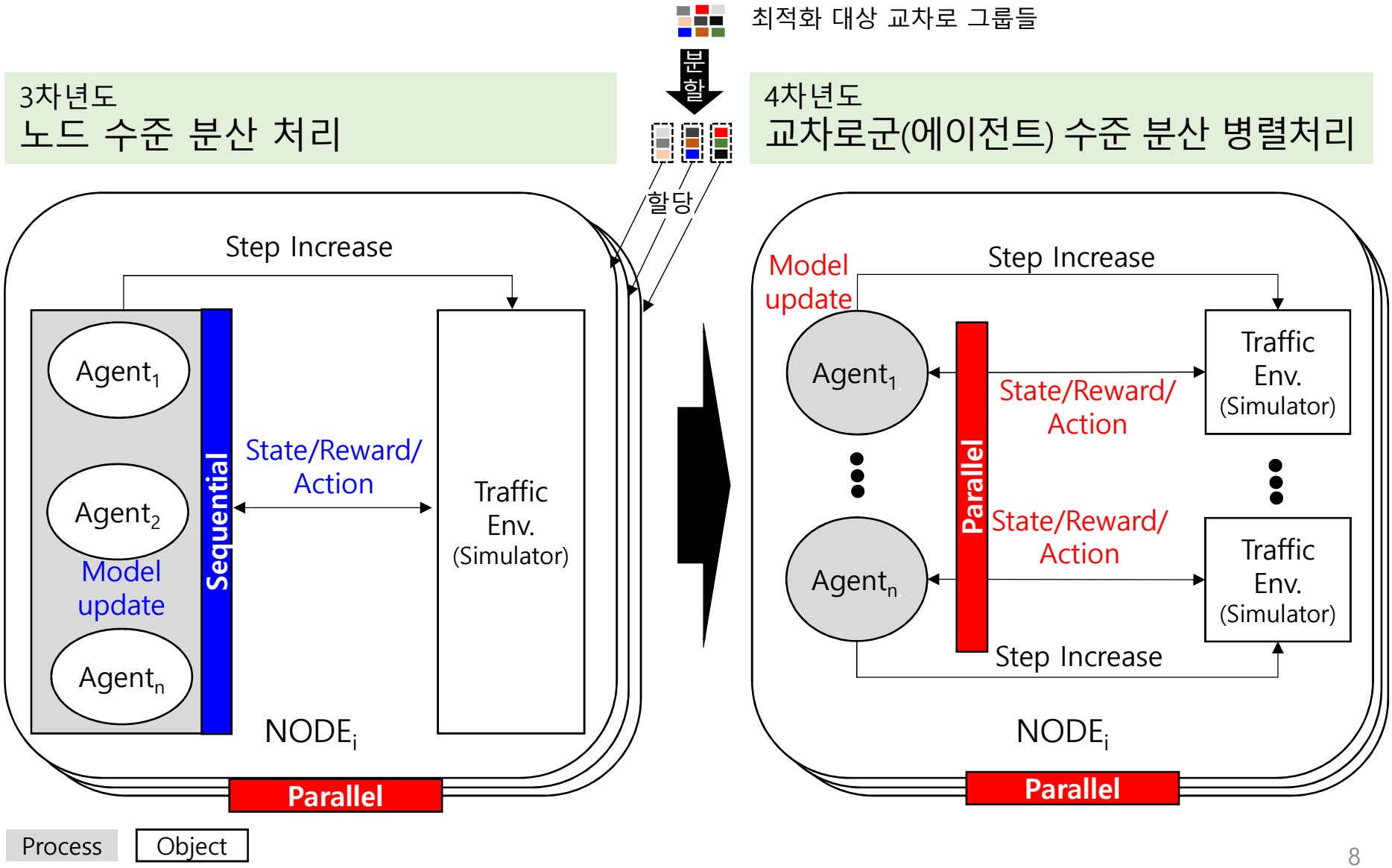
동일 코드/시나리오로 학습 가능

- Agent Process 역할
  - ✓ 학습 진행 : 학습 환경 로딩, 학습 환경과 교류(State-Action-Reward), Step 증가
  - ✓ 학습된 모델 주기적으로 저장
  - ✓ 최적 모델 선정하여 공유 저장소에 복사
  - ✓ 실행 데몬 프로세스에 학습 종료(반복 횟수 만족) 알림

– 하나의 Agent Process가 여러 교통 환경을 통한 동시 학습 진행 : 경험 확대

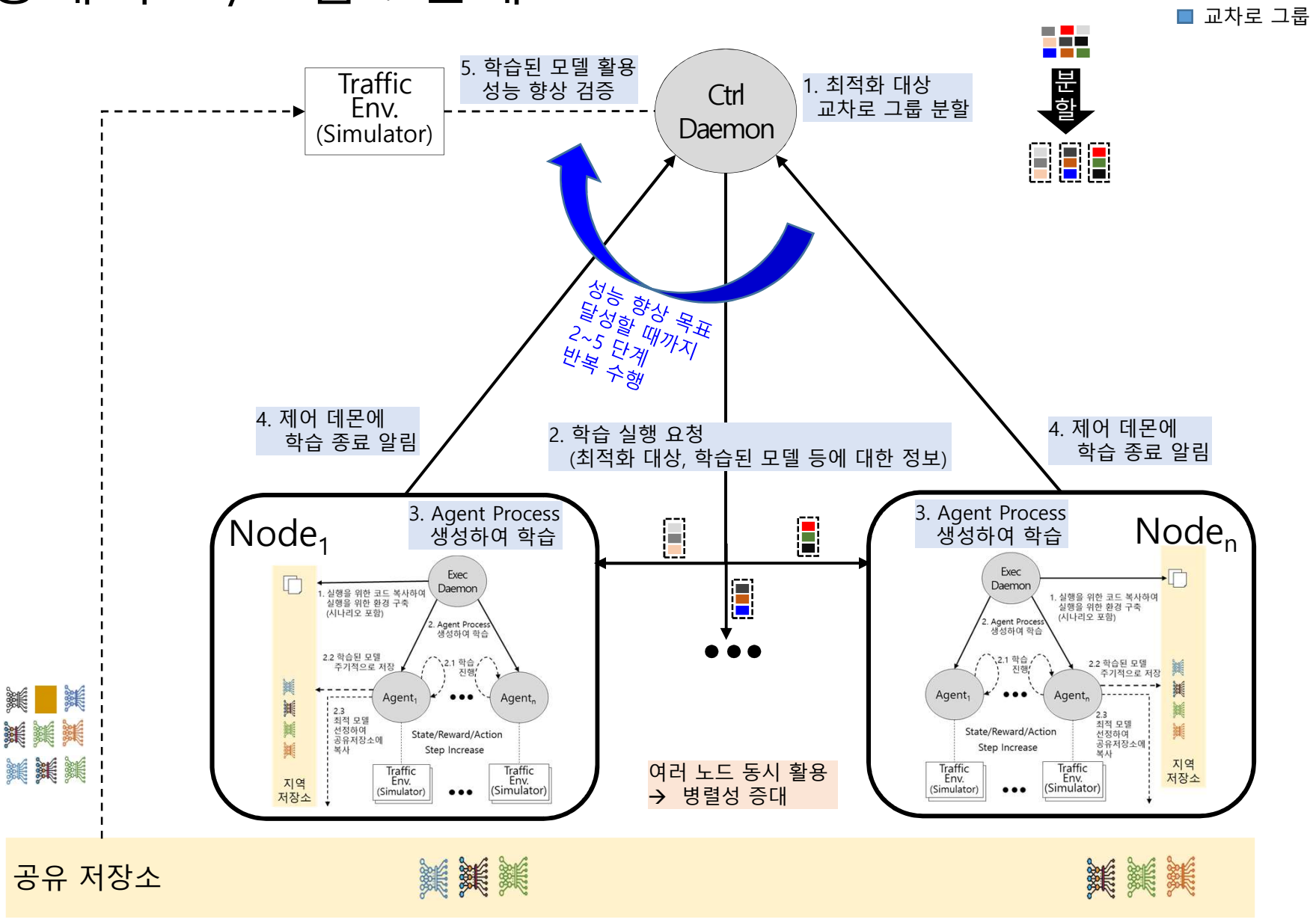
- 하나의 Agent의 제어를 받는 복수의 교통 환경 생성
- Agent는 짧은 시간 내에 다양한 경험 가능

구조 : 학습시 여러 프로세스 이용(process per agent)



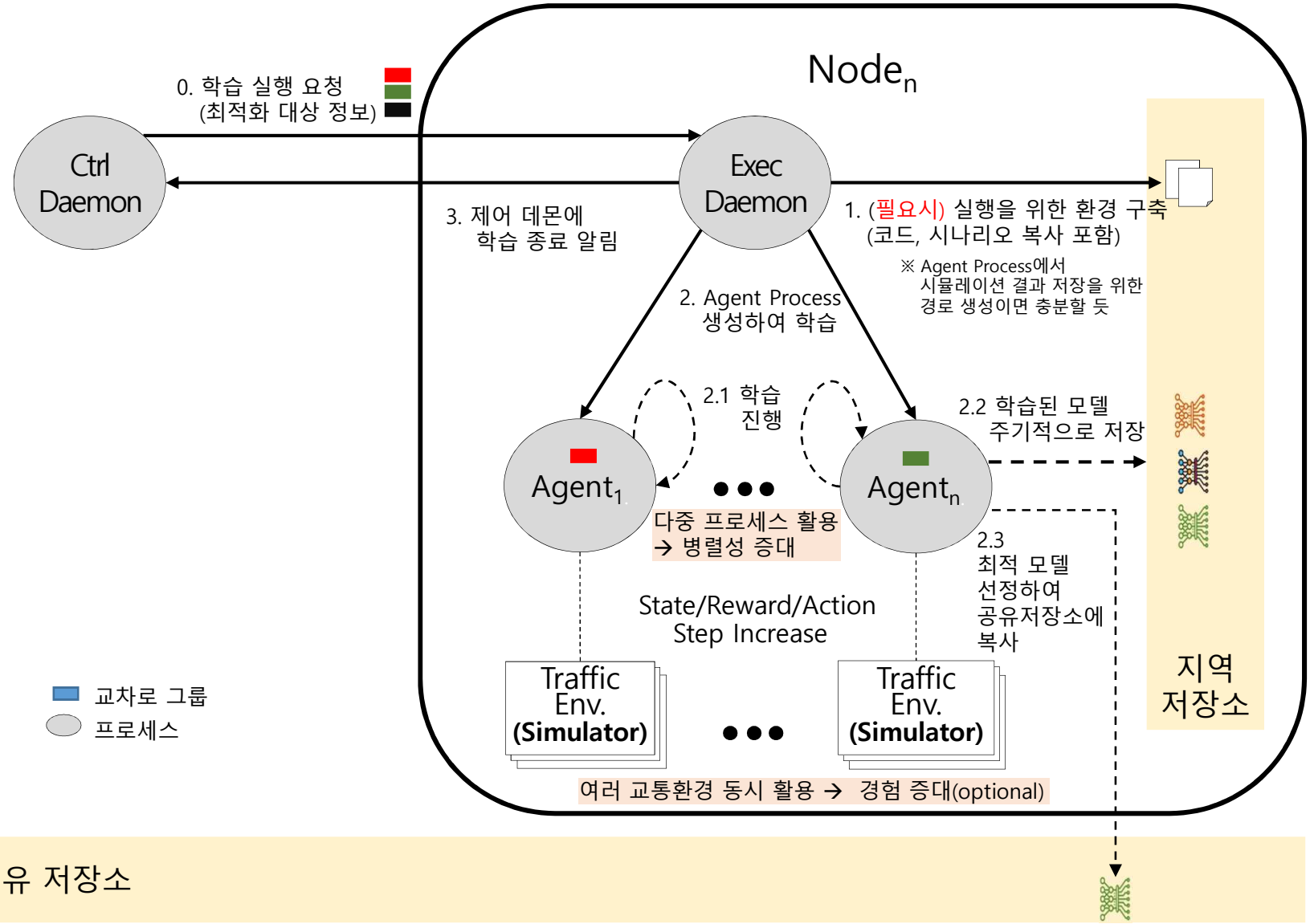
큰 그림에서는 3차년도에 개발한 분산처리에서의 흐름과 동일하다.

상세 구조/흐름 : 전체

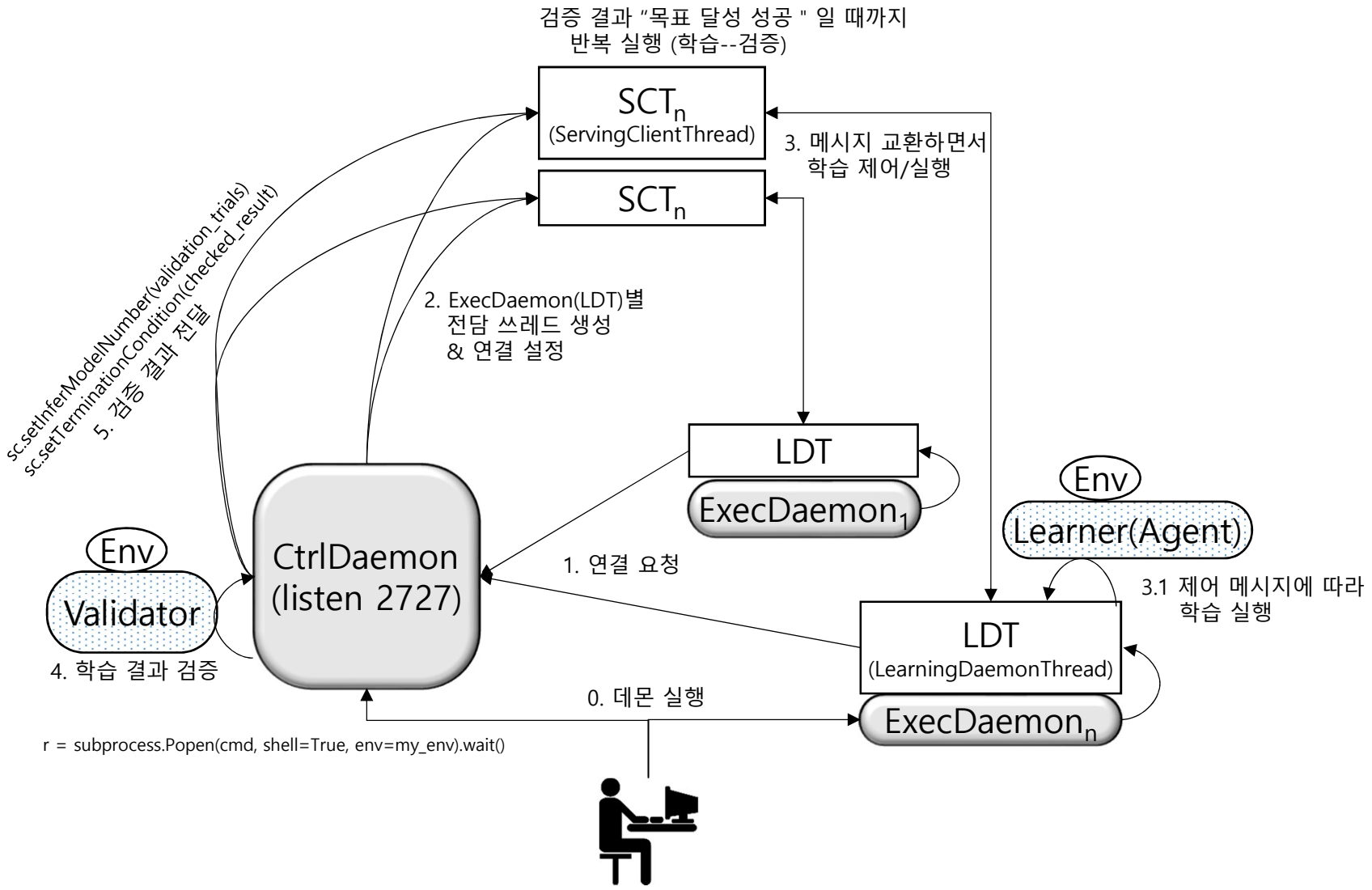


다중 프로세스 활용 → 병렬성 증대
여러 교통환경 동시 활용 → 경험 증대(optional)

상세 구조/흐름 : 실행 노드



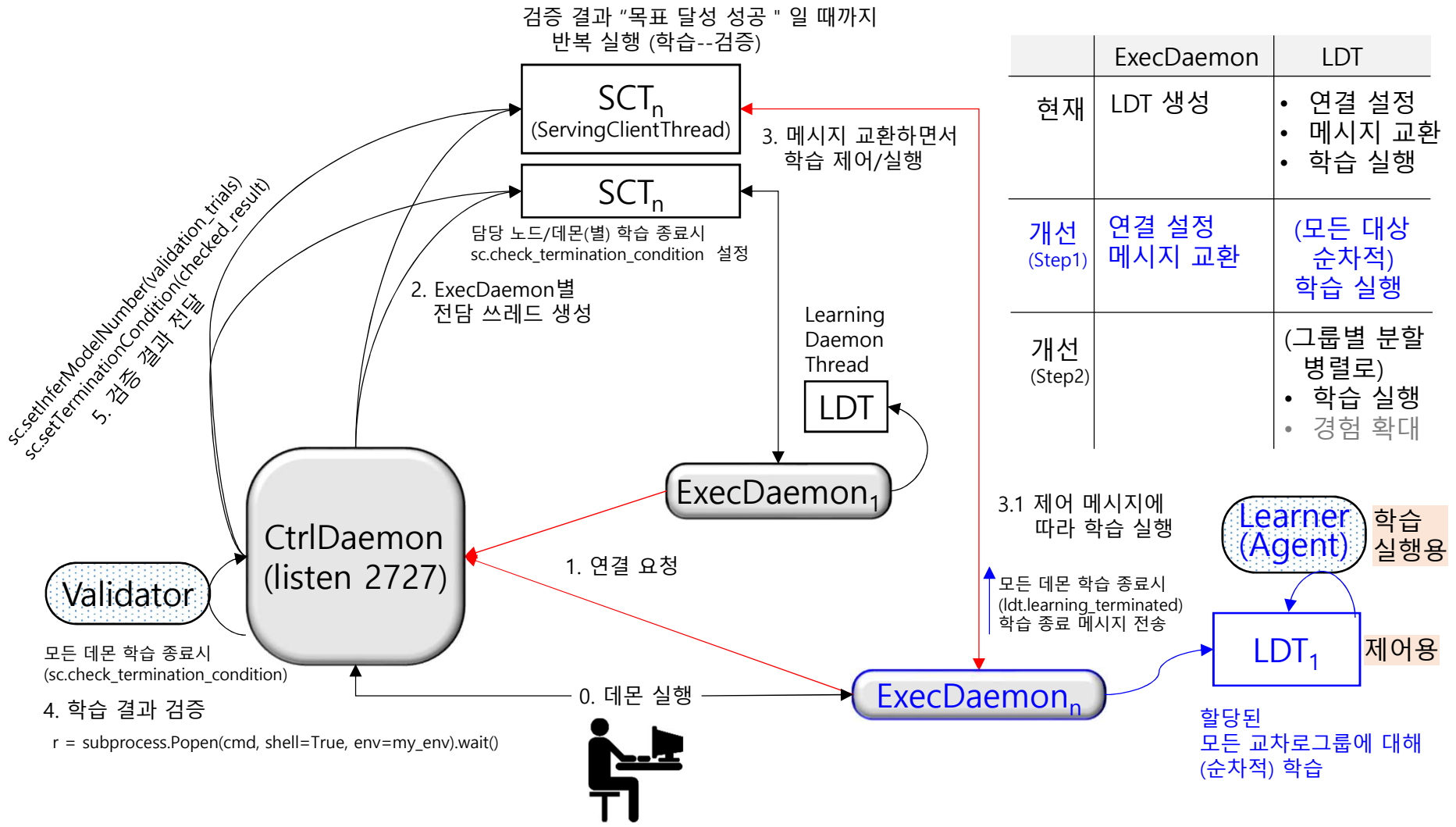
프로세스 구조 : 현재(Nov. 2022)



Daemon Process | Process | Thread | Object

Process 는 라운드마다 생성(실행 후 종료)
Daemon Process는 목표 달성 완료 후 종료
Thread 는 목표 달성 완료 후 종료

프로세스 구조 : 확장 1단계 – 실행을 위한 제어와 실행 분리

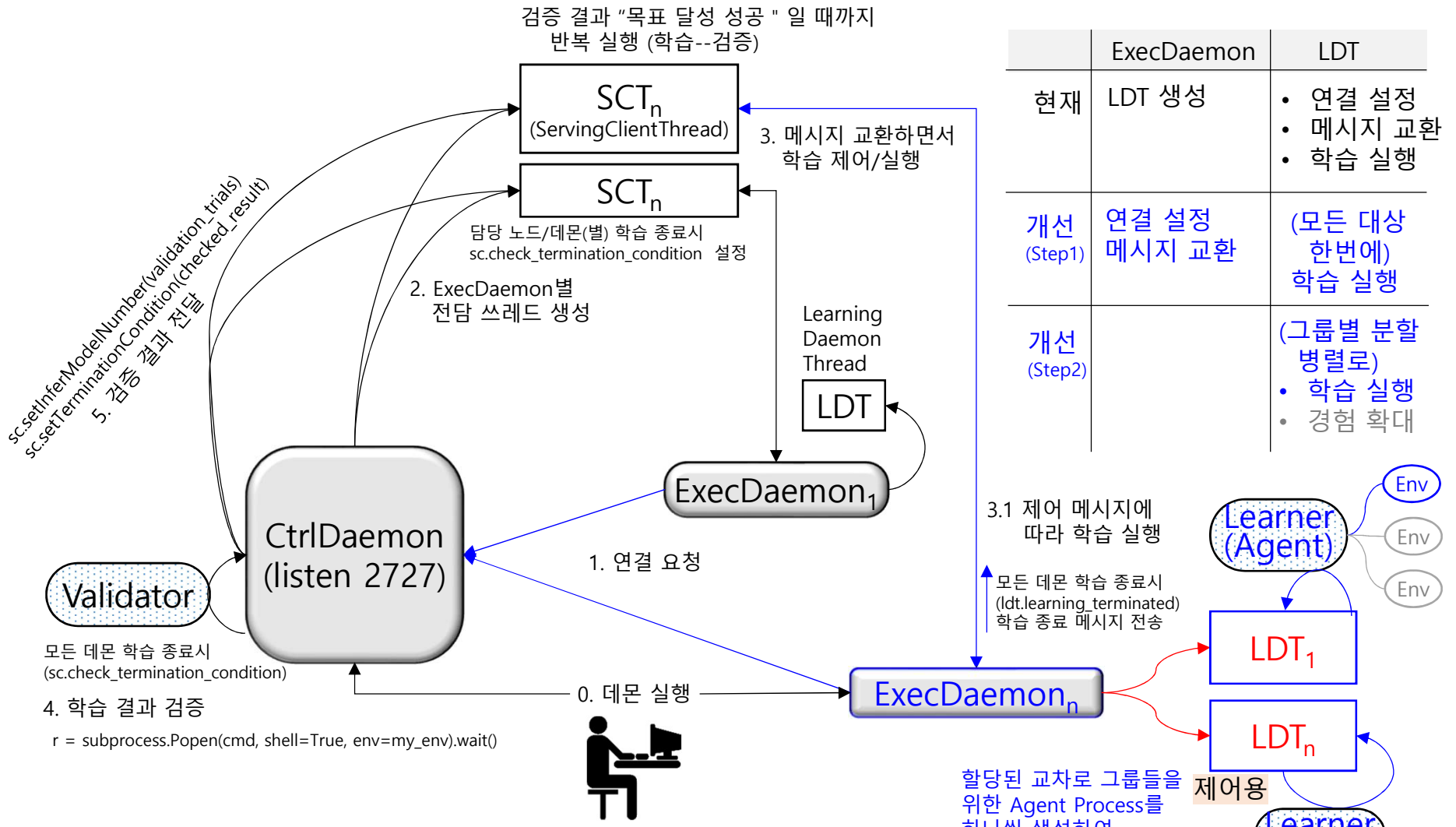


	ExecDaemon	LDT
현재	LDT 생성	• 연결 설정 • 메시지 교환 • 학습 실행
개선 (Step1)	연결 설정 메시지 교환	(모든 대상 순차적) 학습 실행
개선 (Step2)		(그룹별 분할 병렬로) • 학습 실행 • 경험 확대

Daemon Process | Process | Thread | Object

Process 는 라운드마다 생성(실행 후 종료)
Daemon Process는 목표 달성 완료 후 종료
Thread 는 목표 달성 완료 후 종료

프로세스 구조 : 확장 2단계 – 학습 실행 에이전트별 병렬화



	ExecDaemon	LDT
현재	LDT 생성	• 연결 설정 • 메시지 교환 • 학습 실행
개선 (Step1)	연결 설정 메시지 교환	(모든 대상 한번에) 학습 실행
개선 (Step2)		(그룹별 분할 병렬로) • 학습 실행 • 경험 확대

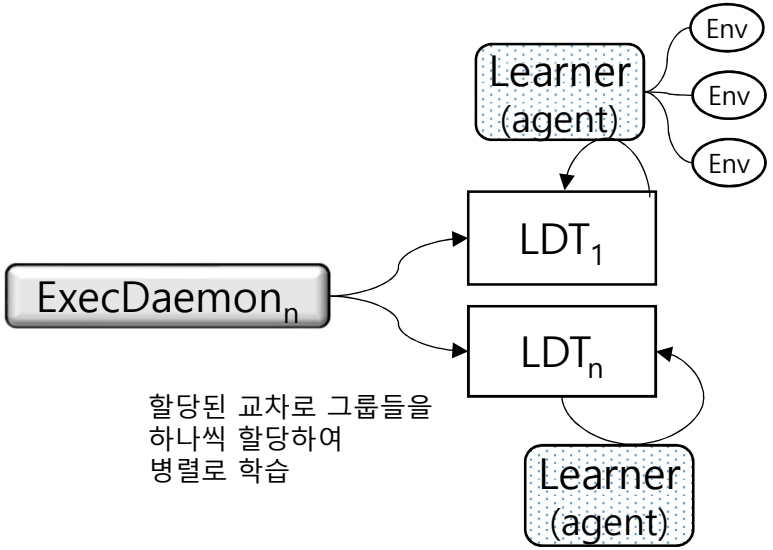
Daemon Process | Process | Thread | Object

Process 는 라운드마다 생성(실행 후 종료)
Daemon Process는 목표 달성 완료 후 종료
Thread 는 목표 달성 완료 후 종료

다른 아이디어

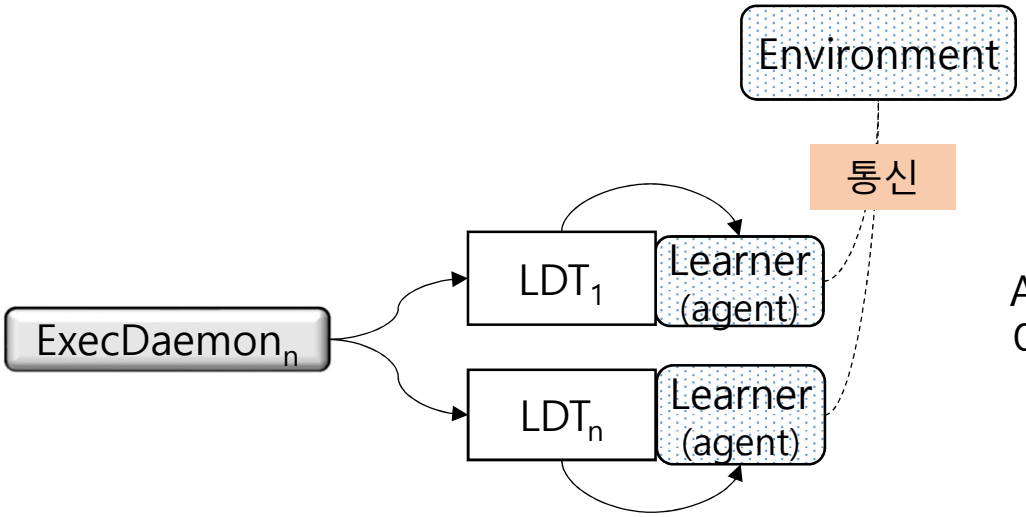
(안 2) 하나의 환경을 여러 학습 에이전트가 공유하며 학습

안 1



Agent 간 영향이 없다
학습은 하나만
나머지는 추론

안 2



가능한 구조인가?
- Env를 별도의 프로세스화
- Agent에서 Env와 연결
- Agent가 Env에 메시지 전달
- Run.py 수정 필요
- 복잡할 듯...

Agent 간 영향을 준다
여러 Agent가 동시 학습

가능한 구조인가?

Daemon Process | Process | Thread | Object

Process 는 라운드마다 생성(실행 후 종료)
Daemon Process는 목표 완료 후 종료

다수 Agent Process가 동일 시나리오 파일을 이용하도록 하는 것에 대한 고민...

현 상태 : 학습 관련 입력/출력(생성)되는 파일

- (암묵적) 파일 경로
  - 시뮬레이터
    - 입력
      - ✓ $IO_HOME/data/envs/salt/${SCENARIO_NAME} : 시나리오를 구성하는 도로, 신호, 수요
    - 출력(생성)
      - ✓ $IO_HOME/output : 시뮬레이터 통계 정보 파일 저장하기위한 최상위 경로
      - ✓ $IO_HOME/output/simulate : 시뮬레이션 모드
      - ✓ $IO_HOME/output/train : 학습 모드
      - ✓ $IO_HOME/output/test : 평가 모드
  - 최적화
    - 생성
      - ✓ $IO_HOME/model : 학습된 모델
        - ❖ $IO_HOME/model/{args.method}
      - ✓ $IO_HOME/logs : 학습 중 로그(텐서보드)
      - ✓ $IO_HOME/data/envs/salt/data : 시험에 사용된 시나리오 복사(백업)

다수 Agent Process가 동일 시나리오 파일을 이용하도록 하는 것에 대한 고민...

현 상태 : 학습 관련 입력/출력(생성)되는 파일

- 최적화 결과 생성 파일 위치 설정
  - 프로그램 실행시 io_home 인자 이용

- SALT 시뮬레이터의 입력 파일 위치 : libsalt.start()의 인자
- SALT 시뮬레이터의 결과 파일 생성 설정
  - 프로그램을 실행시킨 디렉토리의 하부에 output 파일이 생성
  - id를 바꾸어 실행하면 실행마다 다른 파일명으로 생성되게 할 수 있다.

```
{  "scenario": {
     "id": "aaaa"        ← 파일 이름
     "time": {
         "begin": 25200,
         "end": 32400
      },
      ....
      "output": {
         "fileDir": "output/simulate/"   ← 상대 경로
   ,      "period": 30,
         "level": "link",
         "save": 1
      }
}
```

SALT 시뮬레이터의 시나리오 파일

다수 Agent Process가 동일 시나리오 파일을 이용하도록 하는 것에 대한 고민...

MP : multi-processes
ME : multi-environments

입력 파일과 결과 파일 생성 위치 관련 고민(1/3)

- 관점
  - (MP) 여러 프로세스를 동적으로 생성하여 동시에 다수의 에이전트 학습 실행
  - (ME) 하나의 에이전트가 동시 여러 환경 활용하여 학습

- 시뮬레이터
  - 입력 파일
    - 동일한 입력 파일을 이용해도 된다.
    - 필요하다면 심볼릭 링크 이용하면 복사를 위한 시간/공간 절약 가능
  - 출력 파일
    - 이름/경로가 같다면 덮어쓰게 된다
    - 이름을 달리하거나 쓰여지는 위치를 달리해야 한다

- 최적화
  - 경로가 같다면 덮어쓰게 된다.
    - 학습된 모델, 로그

다수 Agent Process가 동일 시나리오 파일을 이용하도록 하는 것에 대한 고민...

MP : multi-processes
ME : multi-environments

입력 파일과 결과 파일 생성 위치 관련 고민(2/3)

- 관점
  - (MP) 여러 프로세스를 동적으로 생성하여 동시에 다수의 에이전트 학습 실행
  - (ME) 하나의 에이전트가 동시 여러 환경 활용하여 학습

- 시뮬레이터
  - 입력 파일
    - 동일한 입력 파일을 이용해도 된다.
    - 필요하다면 심볼릭 링크 이용하면 복사를 위한 시간/공간 절약
  - 출력 파일
    - 이름/경로가 같다면 덮어쓰게 된다
    - 이름을 달리하거나 쓰여지는 위치를 달리해야 한다

(MP) 정상 동작 가능
(ME) 정상 동작 가능

libsalt.start()의 인자 추가
- 결과 파일 생성 루트 상대 경로
(${OUTPUT_HOME}_{idx})

Idx는 ME에 대한 인덱스

- 최적화
  - 경로가 같다면 덮어쓰게 된다.
    - 학습된 모델, 로그

run.py의 인자 추가
- 동시 활용 환경 수
(NUM_CONCURRENT_ENV)

- 결과 파일 생성 루트 상대 경로
(OUTPUT_HOME, default는 "."(현재 위치))

${IO_HOME}/${OUTPUT_HOME}/model
${IO_HOME}/${OUTPUT_HOME}/ log
${IO_HOME}/${OUTPUT_HOME}/scenario.bkup

최적화 학습을 위해 인자로 받은 값을 이용 (OUTPUT_HOME)

시나리오 백업 경로 변경

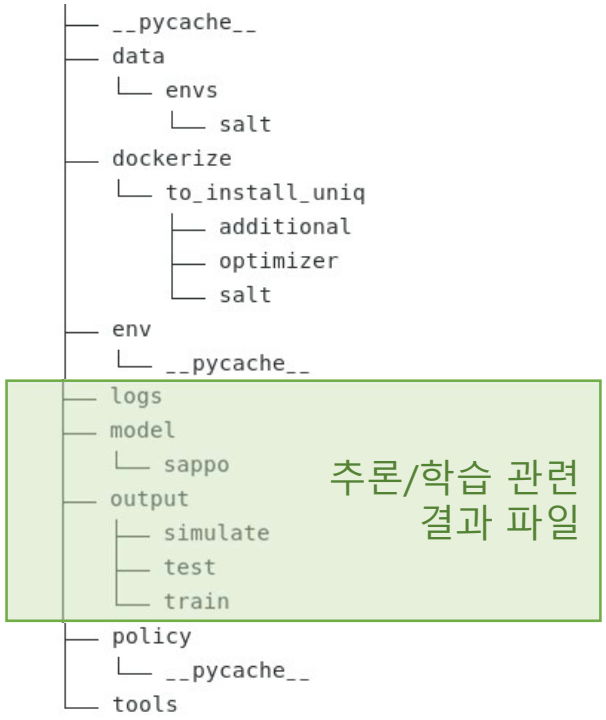
※ 분산 학습시 OUTPUT_HOME은
generateCommand()에서 설정

ExecDaemon에서 학습시에는 target SA 이름으로 설정
CtrlDaemon에서 평가시에는 default 값으로 설정

다수 Agent Process가 동일 시나리오 파일을 이용하도록 하는 것에 대한 고민...

입력 파일과 결과 파일 생성 위치 관련 고민(3/3)

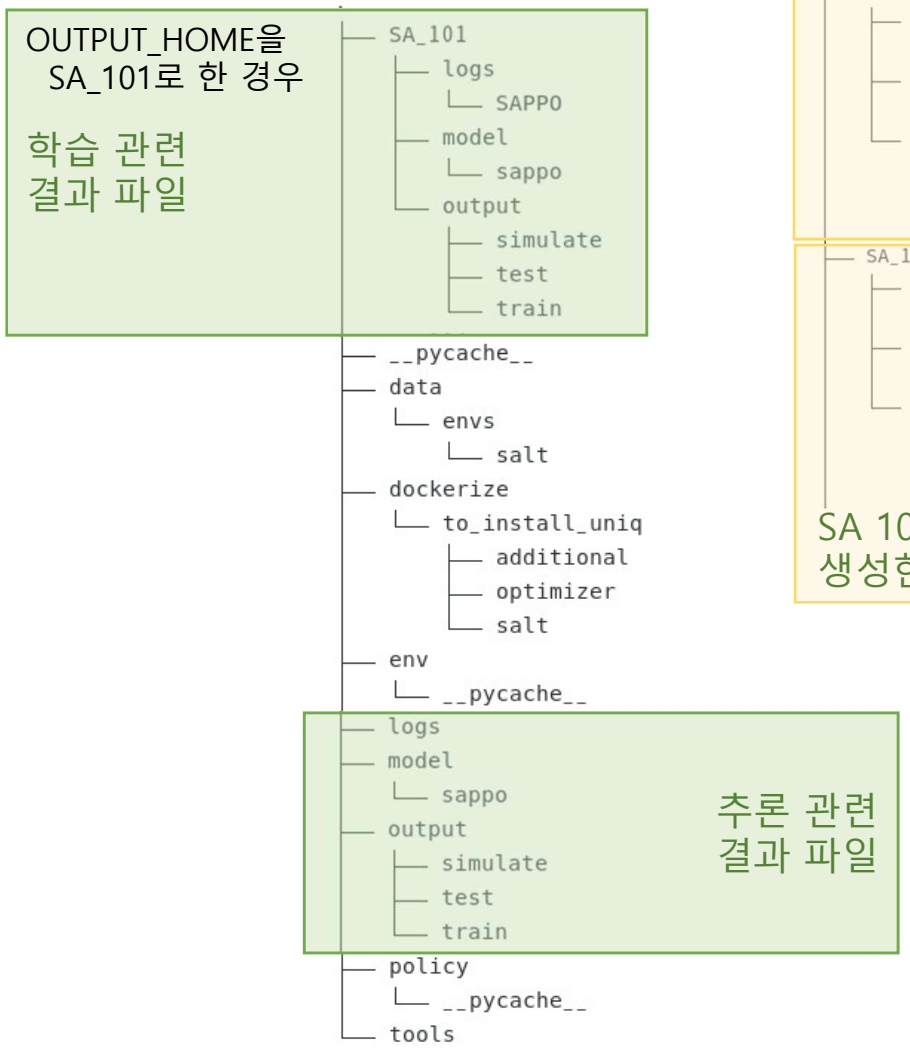
기존 디렉토리 구조



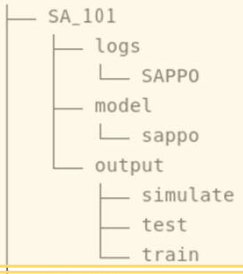
개선된 디렉토리 구조

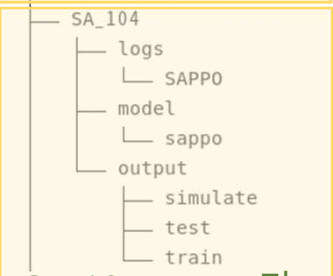
OUTPUT_HOME을
SA_101로 한 경우

학습 관련
결과 파일



SA 101 Agent가
생성한 결과 파일





SA 104 Agent가
생성한 결과 파일

PoC 코드 활용 간이 시험 결과 & 단축 효과 추정

- 에이전트 수준 병렬화를 통한 수행 시간 단축 효과 추정(?)
  - 도안 지역 시나리오 (4 SA, 23개 신호등, 오전 첨두 2시간), 6층 실험실 2개 노드
  - 라운드(200 epoch)별 평균 소요 시간 약 25% 단축
    - 노드 수준 병렬화 : 에이전트 수준 병렬화 = 16575초 : 12309초
      - ✓ 노드 수준 병렬화 : 각 노드에서 하나의 에이전트 프로세스를 생성하여 노드에 할당된 2개의 SA을 순차학습
      - ✓ 에이전트 수준 병렬화 : 각 노드에서 할당된 SA의 개수인 2개의 에이전트 프로세스 생성하여 병렬로 학습
    - SA 모델 갱신의 병렬화를 통해 약 71분(4265초) 단축

- 시뮬레이션 진행을 빠르게 하는 방법에 대해 고민 필요
  - 대상 교차로 수 증가에 따라 시뮬레이션 시간이 길어진다
  - 전체 학습 시간에서 시뮬레이션 시간이 차지하는 비율이 크다
    - 도안 지역 시나리오(4 SA, 23개 신호등, 오전 첨두 2시간)
      1. 순수 시뮬레이션 : 40초
      2. (1) + 데이터 수집(simulate, 4 SA) : 50초
      3. (2) + 모델 제어(train, 1 SA) : 43초
      4. (2) + 모델 제어(test, 4 SA) : 53초
      5. 학습시 모델 갱신(1 SA) : 15~20초

200개 교차로 순수 시뮬레이션 시간은?

    - 추론 시간 : 고정신호 제어(2)와 추론에 의한 제어(5) 비교하면 3초
      - ✓ 데이터 수집, 제어를 위한 비용(시간)
    - 모델 갱신 시간 : 15~20초

Backup slides