rllib 소개

2021. 07. 15.

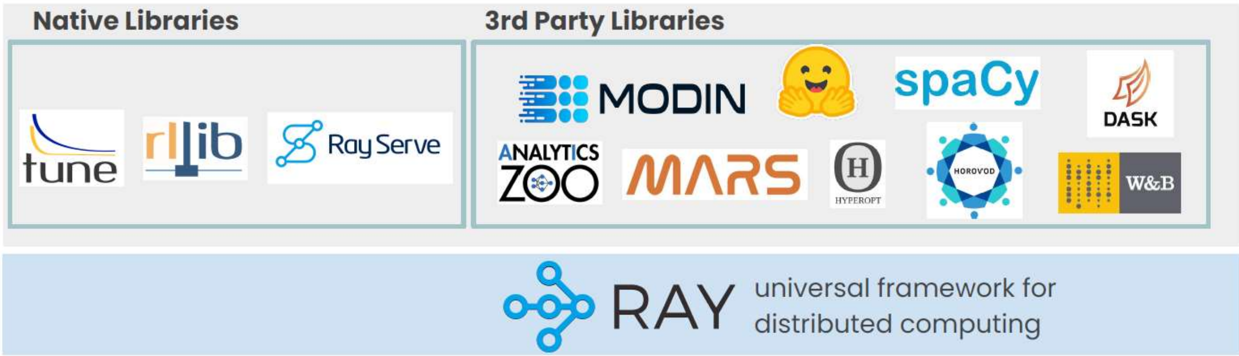
https://docs.ray.io/en/master/rllib.html#

내용

- Overview
  - Ray
  - Rllib key concepts
- RLlib Training API
  - Training and Evaluation
  - Policy customizing
- Customizing
  - callback, exploration, environment, preprocessing, model, action distribution, …
- Available Algorithms

- Internal concepts (policy customizing details)

- 분산 병렬 애플리케이션을 쉽게 구축하기 위한 프레임워크
  - Ian Stoica, RISE Lab @ UC Berkeley
    - Apache Spark
  - Spin-off Anyscale(https://www.anyscale.com/)
  - Apache 2.0 License
  - 머신 러닝 프레임워크들과 강력하게 통합된 Ray Ecosystem
    - Ray Core : 분산/병렬 컴퓨팅을 위한 범용 API
    - Tune : 하이퍼파라미터 최적화 라이브러리
    - Rlib : High-Level 강화학습 라이브러리
    - RaySGD : 여러 Major Deep Learning Framework에 쉽게 확장 가능한 분산 딥러닝 라이브러리
    - Ray Serve : 모델 서빙 라이브러리(서빙 인프라, 모니터링, ..)



https://docs.ray.io/en/master/index.html

Ray를 이용한 Task 병렬화 예

ray.init()
ray.put()
ray.wait()
ray.shutdown()

```
data = [5, 7, 12, 3, 7, 126, 2, ...]
```

순차 실행 : Serial Python

```
def mul(x):
  return x * 10

result = [mul(x) for x in data]
```

- 반복 구조

멀티 프로세싱 : Multiprocessing

```
def mul(x):
  return x * 10

with multiprocessing.Pool(NUM_CPU) as p:
  result = p.map(mul, data)
```

- 기존 코드 구조 변경
- 반복 구조--> map을 이용하는 구조

ray 병렬 실행 : Ray

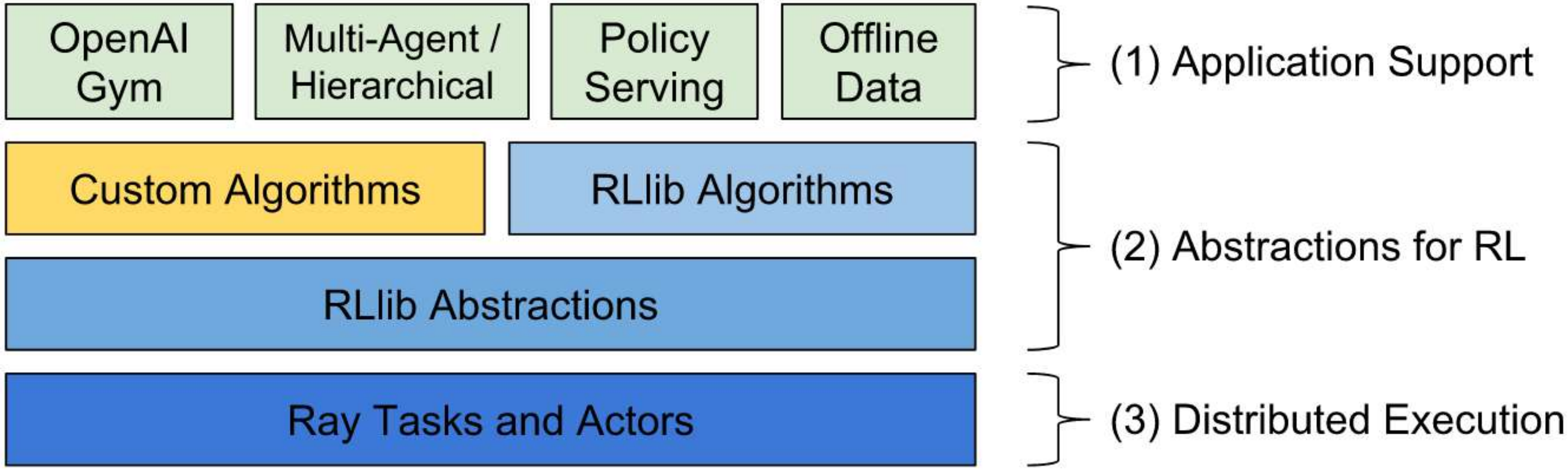
```
@ray.remote
def mul(x):
  return x * 10

result = ray.get([mul.remote(x) for x in data])
```

import ray
ray.init()

- 처리할 함수에 decorator 추가 : @ray.remore
- 함수 호출시 .remote() 메소드 호출
- get() 메소드로 결과 패치

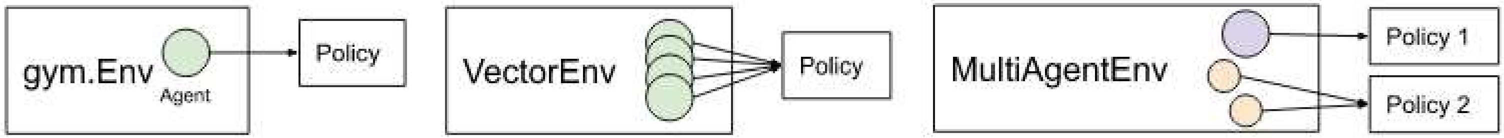
RLlib

- an open-source **library for reinforcement learning** that offers both high scalability and a unified API for a variety of applications
- RLlib natively supports TensorFlow, TensorFlow Eager, and PyTorch, but most of its internals are framework agnostic.



3 Key Concepts : Policies, Sample Batches, Training

- Policies
  - Python classes that define how an agent acts in an environment



  - Rllib has build_tf_policy/build_torch_policy() helper func that you define a trainable policy

```
def policy_gradient_loss(policy, model, dist_class, train_batch):
    logits, _ = model.from_batch(train_batch)
    action_dist = dist_class(logits, model)
    return -tf.reduce_mean(
        action_dist.logp(train_batch["actions"]) * train_batch["rewards"])

# <class 'ray.rllib.policy.tf_policy_template.MyTFPolicy'>
MyTFPolicy = build_tf_policy(
    name="MyTFPolicy",
    loss_fn=policy_gradient_loss)
```

상세 내용은
후반부 슬라이드 참고

3 Key Concepts : Policies, Sample Batches, Training

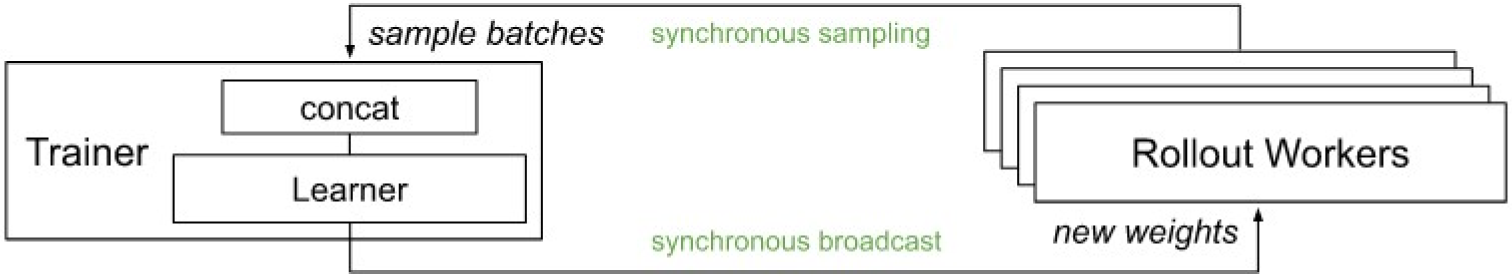
- Sample Batches : from rllib.policy.sample_batch import SampleBatch
  - Rllib 에서 데이터가 교환되는 형식
  - a dictionary with string keys and array-like values

```
{ 'action_logp': np.ndarray((200,), dtype=float32, min=-0.701, max=-0.685, mean=-0.694),
  'actions': np.ndarray((200,), dtype=int64, min=0.0, max=1.0, mean=0.495),
  'dones': np.ndarray((200,), dtype=bool, min=0.0, max=1.0, mean=0.055),
  'infos': np.ndarray((200,), dtype=object, head={}),
  'new_obs': np.ndarray((200, 4), dtype=float32, min=-2.46, max=2.259, mean=0.018),
  'obs': np.ndarray((200, 4), dtype=float32, min=-2.46, max=2.259, mean=0.016),
  'rewards': np.ndarray((200,), dtype=float32, min=1.0, max=1.0, mean=1.0),
  't': np.ndarray((200,), dtype=int64, min=0.0, max=34.0, mean=9.14)}
```

Summarized sample batch의 모음

  - Multiagent 환경에서는 각 Policy 별로 수집된다

- Training/Trainer
  - 학습이나 추론을 위해 분산 워크플로우를 제어(coordinate)하고 Policy 최적화



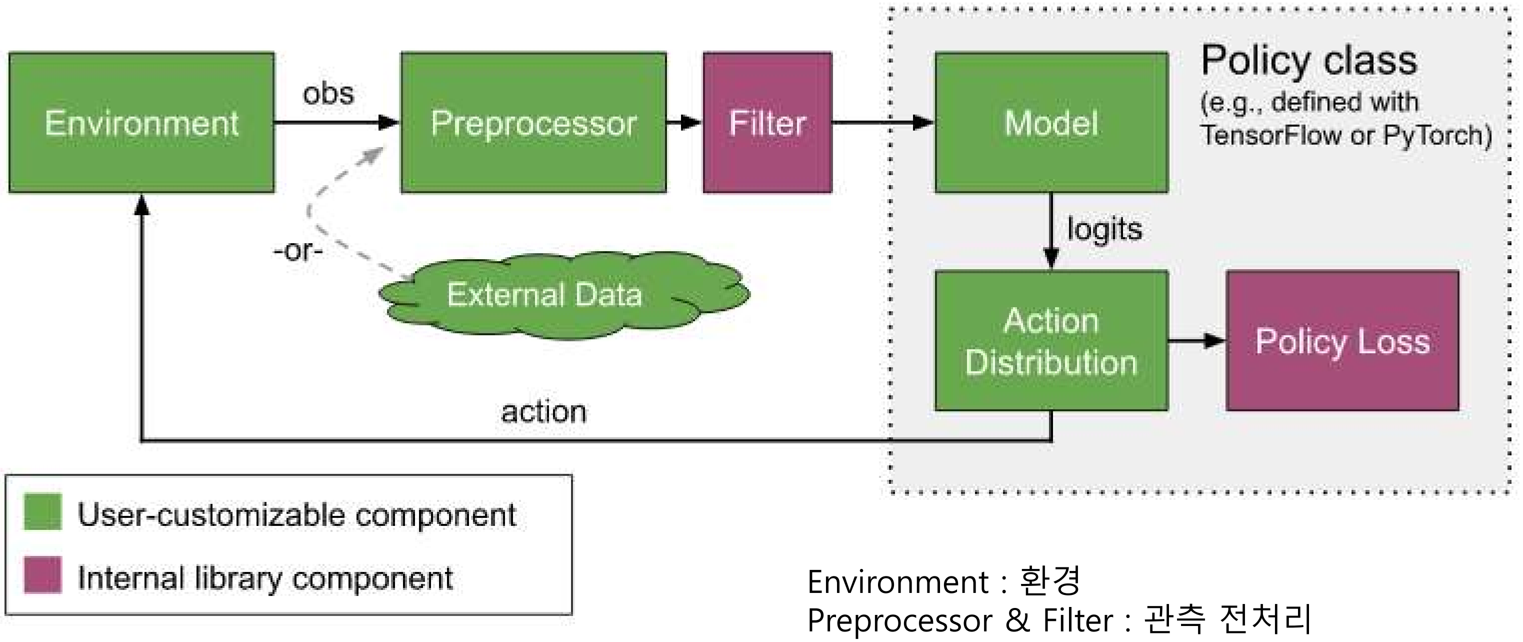
Synchronous Sampling (e.g., A2C, PG, PPO)

Rollout means running an episode with the trained model

Customization

- provides ways to customize almost all aspects of training
  - including neural network models, action distributions, policy definitions: the environment, and the sample collection process

Conceptual Overview of data flow between components in RLlib

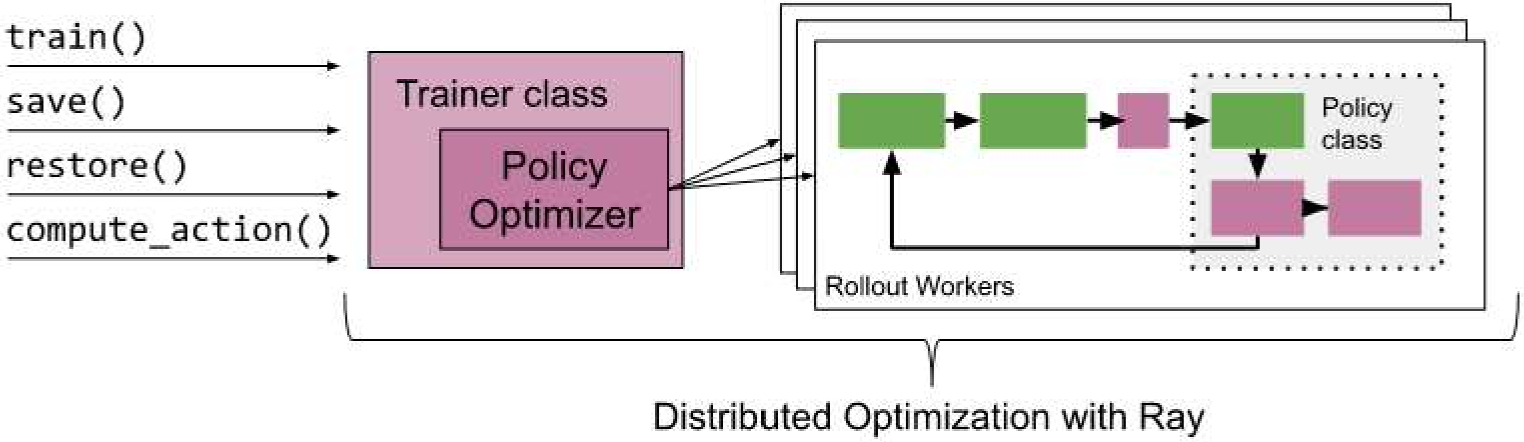


Environment : 환경
Preprocessor & Filter : 관측 전처리
Model : 뉴럴넷
Action Distribution : 모델의 출력을 해석하여 다음 동작 결정

뒤에서 Customizing 방법에 대해 하나하나 설명

RLlib Training APIs

- Trainer class
  - Policy Optimizer를 가지고 있으며, 외부 환경과 상호 작용을 한다
  - Policy에 대해 훈련, Checkpoint, 모델 파라미터 복구, 다음 action 계산을 한다.
  - multi-agent 환경에서는 여러 Policy를 한번에 querying/Optimization 해준다.



Training 단순 DQN trainer 를 이용한 train

[%] rllib train --run DQN --env CartPole-v0 # --config '{"framework": "tf2", "eager_tracing": True}' for eager execution

※ available options include SAC, PPO, PG, A2C, A3C, IMPALA, ES, DDPG, DQN, MARWIL, APEX, and APEX_DDPG

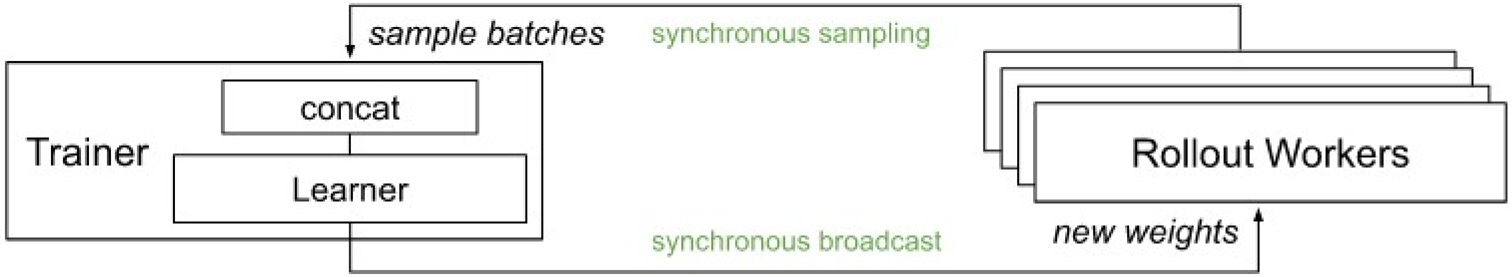
※ 결과 파일은 ~/ray_results 에 기록

(params.json : 하이퍼파라미터, result.json : training summary, tensorboard 파일: 훈련 과정 시각화 등)

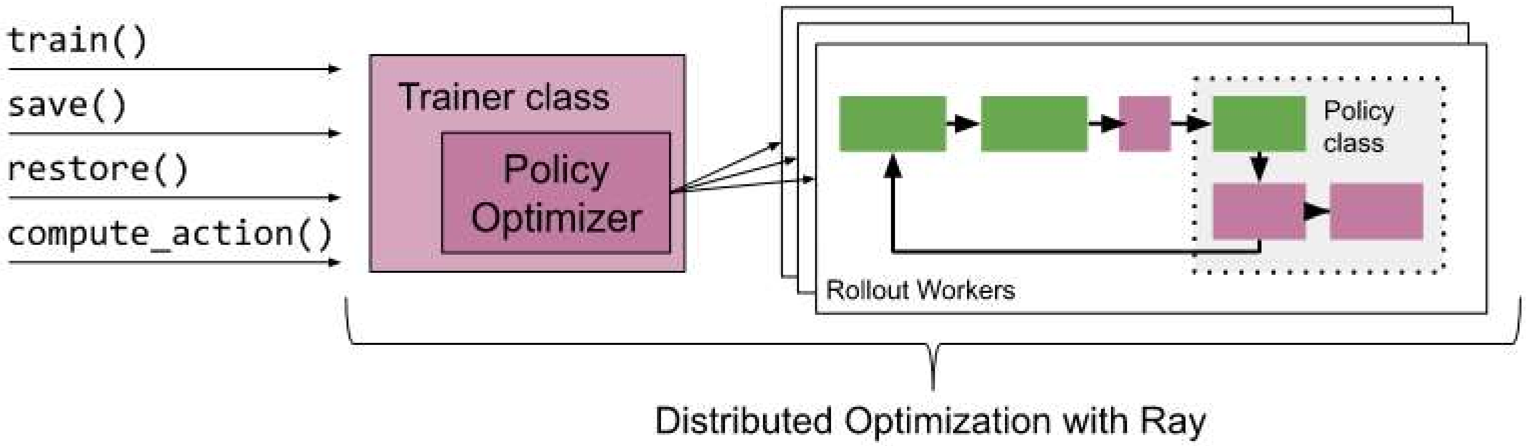
Evaluating Trained Policies

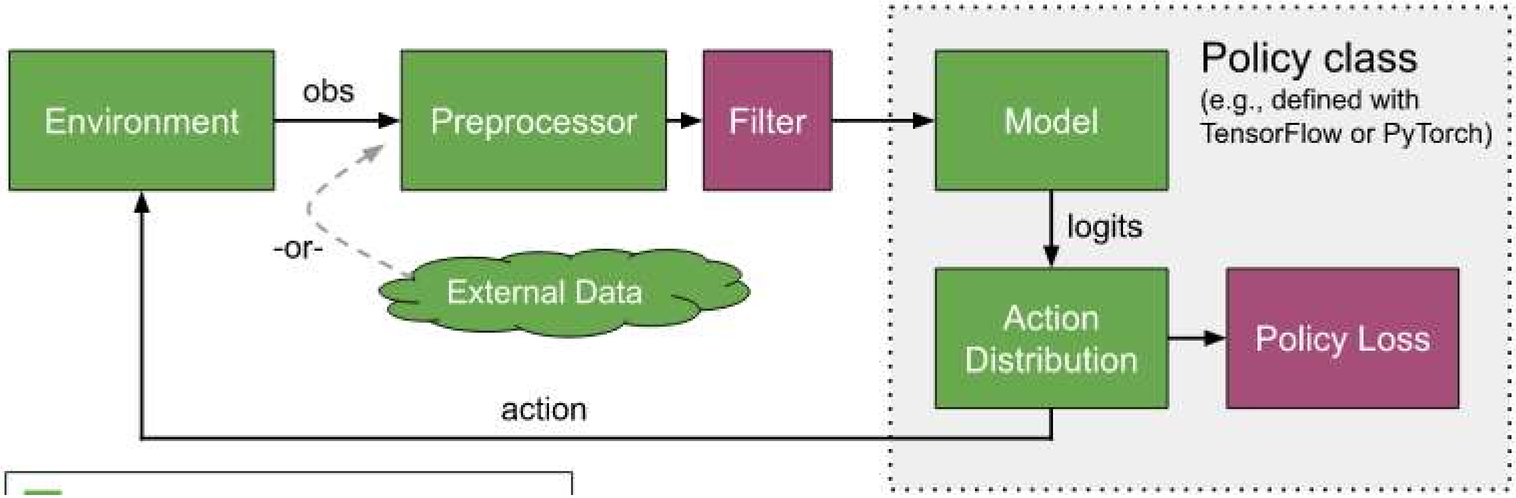
[%] rllib rollout ~/ray_results/default/DQN_CartPole-v0_0upjmdgr0/checkpoint_1/checkpoint-1 ₩

--run DQN --env CartPole-v0 --steps 10000



Synchronous Sampling (e.g., A2C, PG, PPO)





RLlib Training APIs : Configuration

- 하이퍼파라미터 설정 : resource, trainer process, model, deep learning, env, …
  – 좋은 설정 저장소에 존재

COMMON_CONFIG: TrainerConfigDict = {

=== Settings for Rollout Worker processes === num_worker, num_envs_per_worker, ,,,,

=== Settings for the Trainer process === gamma, lr, train_batch_size, model, optimizer,observation_space, action_space, ….

=== Debug Settings === log_level, callback, …

=== Deep Learning Framework Settings === frame_work, eager_tracing, …

=== Exploration Settings === explore, exploration_config, …

=== Evaluation Settings === evaluation_interval, evaluation_num_episode, evaluation_config, ,…

=== Advanced Rollout Settings === observation_filter, seed, ….

=== Resource Settings === num_gpus, num_cpus_per_worker, num_gpus_per_worker,….

=== Offline Datasets ===

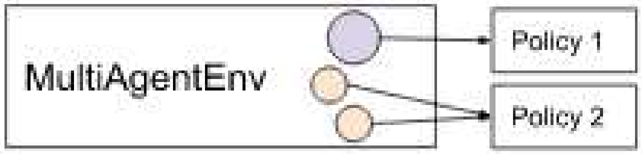
=== Settings for Multi-Agent Environments === policies, policy_mapping_루, policies_to_train, observation_fn, …

=== Logger ===

…. }

```
COMMON_CONFIG: TrainerConfigDict = {
    # === Settings for Rollout Worker processes ===
    # Number of rollout worker actors to create for parallel sampling. Setting
    # this to 0 will force rollouts to be done in the trainer actor.
    "num_workers": 2,
    # Number of environments to evaluate vector-wise per worker. This enables
    # model inference batching, which can improve performance for inference
    # bottlenecked workloads.
    "num_envs_per_worker": 1,
    # When `num_workers` > 0, the driver (local_worker; worker-idx=0) does not
    # need an environment. This is because it doesn't have to sample (done by
```

RLlib Training APIs : Configuration



각 agent 별로 다른 policy 설정
(교차로(군)별로 다른 policy 설정 가능)

```python
# === Settings for Multi-Agent Environments ===
"multiagent": {
    # Map of type MultiAgentPolicyConfigDict from policy ids to tuples
    # of (policy_cls, obs_space, act_space, config). This defines the
    # observation and action spaces of the policies and any extra config.
    "policies": {},
    # Function mapping agent ids to policy ids.
    "policy_mapping_fn": None,
    # Optional list of policies to train, or None for all policies.
    "policies_to_train": None,
    # Optional function that can be used to enhance the local agent
    # observations to include more state.
    # See rllib/evaluation/observation_function.py for more info.
    "observation_fn": None,
    # When replay_mode=lockstep, RLlib will replay all the agent
    # transitions at a particular timestep together in a batch. This allows
    # the policy to implement differentiable shared computations between
    # agents it controls at that timestep. When replay_mode=independent,
    # transitions are replayed independently per policy.
    "replay_mode": "independent",
    # Which metric to use as the "batch size" when building a
    # MultiAgentBatch. The two supported values are:
    # env_steps: Count each time the env is "stepped" (no matter how many
    #   multi-agent actions are passed/how many multi-agent observations
    #   have been returned in the previous step).
    # agent_steps: Count each individual agent step as one step.
    "count_steps_by": "env_steps",
},
```

skip

Multiagent 예 : RockPaperScissors

```python
def select_policy(agent_id, episode, **kwargs):
    if agent_id == "player1":
        return "learned"
    else:
        return random.choice(["always_same", "beat_last"])

config = {
    "env": RockPaperScissors,
    "gamma": 0.9,
    # Use GPUs iff `RLLIB_NUM_GPUS` env var set to > 0.
    "num_gpus": int(os.environ.get("RLLIB_NUM_GPUS", "0")),
    "num_workers": 0,
    "num_envs_per_worker": 4,
    "rollout_fragment_length": 10,
    "train_batch_size": 200,
    "multiagent": {
        "policies_to_train": ["learned"],
        "policies": {
            "always_same": (AlwaysSameHeuristic, Discrete(3), Discrete(3),
                            {}),
            "beat_last": (BeatLastHeuristic, Discrete(3), Discrete(3), {}),
            "learned": (None, Discrete(3), Discrete(3), {
                "model": {
                    "use_lstm": use_lstm
                },
                "framework": args.framework,
            }),
        },
        "policy_mapping_fn": select_policy,
    },
    "framework": args.framework,
}
```

```python
class RockPaperScissors(MultiAgentEnv):
    """Two-player environment for the famous rock paper scissors game.

    The observation is simply the last opponent action."""

    def __init__(self, config):
        self.sheldon_cooper = config.get("sheldon_cooper", False)
        self.action_space = Discrete(5 if self.sheldon_cooper else 3)
        self.observation_space = Discrete(5 if self.sheldon_cooper else 3)
        self.player1 = "player1"
        self.player2 = "player2"
        self.last_move = None
        self.num_moves = 0

        # For test-case inspections (compare both players' scores).
        self.player1_score = self.player2_score = 0

    def reset(self):
        self.last_move = (0, 0)
        self.num_moves = 0
        return {
            self.player1: self.last_move[1],
            self.player2: self.last_move[0],
        }
```

추정 :
반환되는 obs dict의 키로 이용된 값이 policy 선택을 위한 agent_id로 이용

Ref. https://github.com/ray-project/ray/blob/master/rllib/examples/rock_paper_scissors_multiagent.py (왼쪽)
https://github.com/ray-project/ray/blob/master/rllib/examples/env/rock_paper_scissors.py (오른쪽)

RLlib Training APIs : Training With Python

Basic Python API

Trainer 1 + worker 1

```
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

ray.init()
config = ppo.DEFAULT_CONFIG.copy()
config["num_gpus"] = 0
config["num_workers"] = 1
trainer = ppo.PPOTrainer(config=config, env="CartPole-v0")

# Can optionally call trainer.restore(path) to load a checkpoint.

for i in range(1000):
   # Perform one iteration of training the policy with PPO
   result = trainer.train()
   print(pretty_print(result))

   if i % 100 == 0:
       checkpoint = trainer.save()
       print("checkpoint saved at", checkpoint)
```

# of rollout worker

num_worker 가 0이면
단일 프로세스로 동작

Tune API

```
import ray
from ray import tune

ray.init()
tune.run(
    "PPO",
    stop={"episode_reward_mean": 200},
    config={
        "env": "CartPole-v0",
        "num_gpus": 0,
        "num_workers": 1,
        "lr": tune.grid_search([0.01, 0.001, 0.0001]),
    },
)
```

trainer

Checkppint_at_end=True,
Checkpoint_freq=100

```
== Status ==
Using FIFO scheduling algorithm.
Resources requested: 4/4 CPUs, 0/0 GPUs
Result logdir: ~/ray_results/my_experiment
PENDING trials:
 - PPO_CartPole-v0_2_lr=0.0001:     PENDING
RUNNING trials:
 - PPO_CartPole-v0_0_lr=0.01:       RUNNING [pid=21940], 16 s, 4
 - PPO_CartPole-v0_1_lr=0.001:      RUNNING [pid=21942], 27 s, 8
```

Tune이 스케줄링하여 병렬로 실행

RLlib Training APIs : Training With Python (Evaluation)

Tune을 이용한 훈련 결과 중 최적 선정

실행

```
# tune.run() allows setting a custom log directory (other than ``~/ray-results``)
# and automatically saving the trained agent
analysis = ray.tune.run(
    ppo.PPOTrainer,
    config=config,
    local_dir=log_dir,
    stop=stop_criteria,
    checkpoint_at_end=True)
```

```
# or simply get the last checkpoint (with highest "training_iteration")
last_checkpoint = analysis.get_last_checkpoint()
# if there are multiple trials, select a specific trial or automatically
# choose the best one according to a given metric
last_checkpoint = analysis.get_last_checkpoint(
    metric="episode_reward_mean", mode="max"
)
```

최적 선정

```
# list of lists: one list per checkpoint; each checkpoint list contains
# 1st the path, 2nd the metric value
checkpoints = analysis.get_trial_checkpoints_paths(
    trial=analysis.get_best_trial("episode_reward_mean"),
    metric="episode_reward_mean")
```

로딩

```
agent = ppo.PPOTrainer(config=config, env=env_class)
agent.restore(checkpoint_path)
```

훈련된 agent 이용하여 action 수행

```
# instantiate env class
env = env_class(env_config)

# run until episode ends
episode_reward = 0
done = False
obs = env.reset()
while not done:
    action = agent.compute_action(obs)
    obs, reward, done, info = env.step(action)
    episode_reward += reward
```

Obs를 Agent policy에 전달하기 전에 전처리 & 필터

Action을 반환하기 전에 normalize & clip

Callback

- Policy Evaluation 동안 특정 시점에 호출하는 Callback 제공
- Callback 들은 현재 Episode의 State에 접근하여 수행

ray.rllib.agents.callbacks.DefaultCallbacks(legacy_callbacks_dict: Dict[str, callable] = None)	
on_episode_start()	rollout worker에 대해 episode를 시작하기 전에 불리는 함수
on_episode_step()	Episode의 매 step 마다 불리는 함수
on_episode_end()	Episode가 끝날때 불리는 함수
on_postprocess_trajectory()	policy에서 policy의 postprocess_fn이 불리고 호출되는 함수로, batch를 처리하는 부분. 예를들어, MultiAgent에서 다른 Agent의 Observation을 처리하는 부분을 추가할 수 있음
on_sample_end()	RolloutWorker.sample()이 끝나고 호출되는 함수
on_learned_on_batch()	Policy.learn_on_batch()의 첫 부분에 호출되는 함수
on_train_result()	Trainer.train()으로 학습을 완료하고 호출되는 함수

Ref. https://github.com/ray-project/ray/blob/master/rllib/examples/custom_metrics_and_callbacks.py

Callbacks

- Customizing
  - DefaultCallbacks 를 상속 받아 정의
  - Config 를 이용하여 연결하여 활용

정의

```
class MyCallbacks(DefaultCallbacks):
  # Episode 가 시작되면 Pole의 각도를 저장할 리스트를 생성
  def on_episode_start(self, *, worker: RolloutWorker, base_env: BaseEnv,
                       policies: Dict[str, Policy],
                       episode: MultiAgentEpisode, env_index: int, **kwargs):
    print("episode {} (env-idx={}) started.".format(
        episode.episode_id, env_index))

    episode.user_data["pole_angles"] = []
    episode.hist_data["pole_angles"] = []
  # 매 Episode 마다 Pole의 각도를 저장
  def on_episode_step(self, *, worker: RolloutWorker, base_env: BaseEnv,
                      episode: MultiAgentEpisode, env_index: int, **kwargs):
    pole_angle = abs(episode.last_observation_for()[2])
    raw_angle = abs(episode.last_raw_obs_for()[2])
    assert pole_angle == raw_angle
    episode.user_data["pole_angles"].append(pole_angle)
```

활용

```
ray.init()
trials = tune.run(
    "PG",
    stop={"training_iteration": args.stop_iters},
    config={
      "env": "CartPole-v0",
      "num_envs_per_worker": 2,
      "callbacks": MyCallbacks, # <----------------!!!
    }
)
```

config 의 Callbacks 에 넣어 준다

```
'callbacks': MultiCallbacks([
    MyCustomStatsCallbacks, MyCustomVideoCallbacks, MyCusto
])
```

여러 callback을 동시에 등록할 수도 있다.

Exploration Behavior

- Trainer의 config 활용하여 Agent의 Exploration behavior를 customize

For Training

```
# 3) Example exploration_config usages:
# a) DQN: see rllib/agents/dqn/dqn.py
"explore": True,
"exploration_config": {
   # Exploration sub-class by name or full path to module+class
   # (e.g. "ray.rllib.utils.exploration.epsilon_greedy.EpsilonGreedy")
   "type": "EpsilonGreedy",
   # Parameters for the Exploration class' constructor:
   "initial_epsilon": 1.0,
   "final_epsilon": 0.02,
   "epsilon_timesteps": 10000,  # Timesteps over which to anneal epsilon.
},

# b) DQN Soft-Q: In order to switch to Soft-Q exploration, do instead:
"explore": True,
"exploration_config": {
   "type": "SoftQ",
   # Parameters for the Exploration class' constructor:
   "temperature": 1.0,
},

# c) All policy-gradient algos and SAC: see rllib/agents/trainer.py
# Behavior: The algo samples stochastically from the
# model-parameterized distribution. This is the global Trainer default
# setting defined in trainer.py and used by all PG-type algos (plus SAC).
"explore": True,
"exploration_config": {
   "type": "StochasticSampling",
   "random_timesteps": 0,  # timesteps at beginning, over which to act uniformly randomly
},
```

<u>Built-in Exploration subclasses</u>

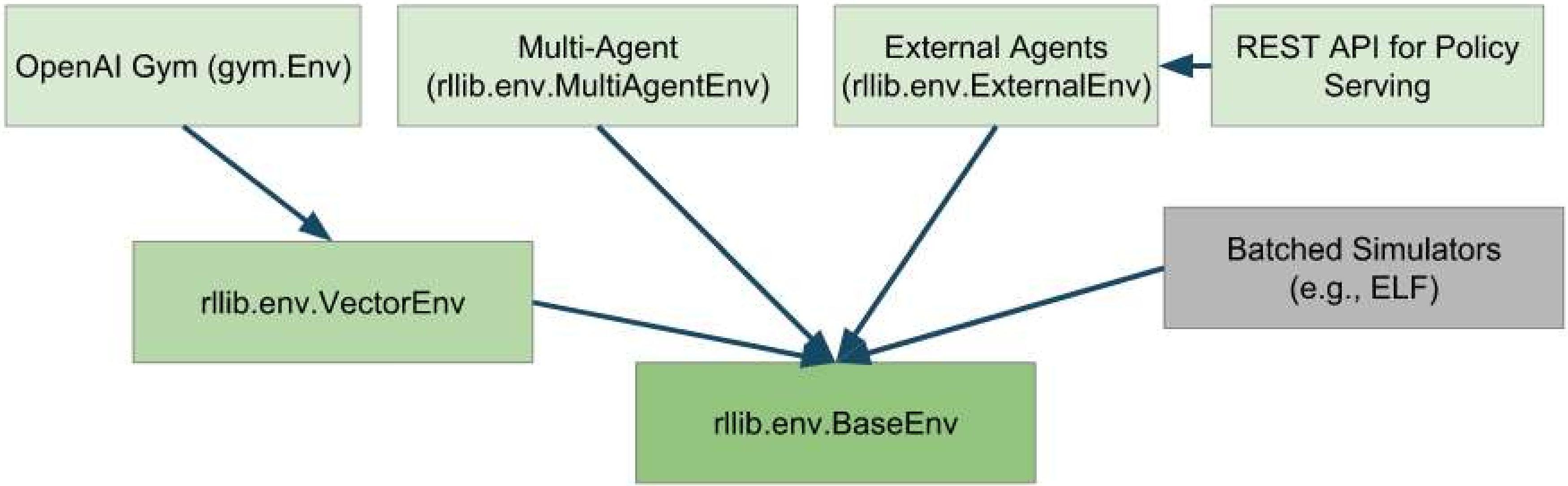
EpsilonGreedy
PerWorkerEpsilionGreedy
SoftQ
StochasticSampling
GaussianNoise
OrnsteinUhlenbeckNoise

For Evaluation

```
# Switching off exploration behavior for evaluation workers
# (see rllib/agents/trainer.py)
"evaluation_config": {
   "explore": False
}
```

Environments

- Rllib는 여러 유형의 Environment 상에서 동작
  - OpenAI Gym, User-defined, Multi-agent, batch env…



Environments

- 상위 클래스에서 상속 받아 정의 : __init__()/reset()/step() 함수 정의
- 사용
  - Python 클래스나 문자열 이름으로 Environment 명시
  - 사용자 정의 env 클래스 이용시 env_config(dict)를 이용하여 Environment에 전달할 인자 설정

```python
import gym, ray
from ray.rllib.agents import ppo

class MyEnv(gym.Env):
    def __init__(self, env_config):
        self.action_space = <gym.Space>
        self.observation_space = <gym.Space>
    def reset(self):
        return <obs>
    def step(self, action):
        return <obs>, <reward: float>, <done: bool>, <info: dict>

ray.init()
trainer = ppo.PPOTrainer(env=MyEnv, config={
    "env_config": {},  # config to pass to env class
})

while True:
    print(trainer.train())
```

환경 초기화하고, observation 반환

Action 수행하고, 다음 observation, reward, 종료 여부, 부가 정보 반환

Python 클래스

사용자 정의 환경 등록

```python
from ray.tune.registry import register_env

def env_creator(env_config):
    return MyEnv(...)  # return an env instance

register_env("my_env", env_creator)
trainer = ppo.PPOTrainer(env="my_env")
```

문자열

Environments : 참고: MultiAgentEnv

```
@PublicAPI
class MultiAgentEnv:
    """An environment that hosts multiple independent agents.

    Agents are identified by (string) agent ids.

    @PublicAPI
    def reset(self) -> MultiAgentDict:
        """Resets the env and returns observations from ready agents.

        Returns:
            obs (dict): New observations for each ready agent.
        """

    @PublicAPI
    def step(
            self, action_dict: MultiAgentDict
    ) -> Tuple[MultiAgentDict, MultiAgentDict, MultiAgentDict, MultiAgentDict]:
        """Returns observations from ready agents.

        The returns are dicts mapping from agent_id strings to values. The
        number of agents in the env can vary over time.

        Returns:
            Tuple[dict, dict, dict, dict]: Tuple with 1) new observations for
                each ready agent, 2) reward values for each ready agent. If
                the episode is just started, the value will be None.
                3) Done values for each ready agent. The special key
                "__all__" (required) is used to indicate env termination.
                4) Optional info values for each agent id.
        """
```

```
class SALTEnv(MultiAgentEnv):
    def __init__(self, ....):
        pass

    def reset(self):
        psss

    def step(self, action):
        pass
```

Ref. https://github.com/ray-project/ray/blob/master/rllib/env/multi_agent_env.py

Skip

Preprocessor

- Built-in Preprocessor
  - Discrete observations are one-hot encoded. E.g. Discrete(3) and value=1 ➔ [0, 1, 0]
  - MultiDiscrete obs.s are "multi" one-hot encodes. [3,4] and value=[1,0] ➔ [0 1 0 1 0 0 0]
  - Tuple and Dict obs.s are flatted
- Customizing
  - 현재는 complex observation space 를 다루기 위한 builtin-preprocessor 와 충돌이 나서 Deprecated
  - Preprocessor 대신 environment에 대한 wrapper class를 이용을 권장
    - Environment Wrapper Class를 이용하여 Environment의 Output을 preprocess 하자
      - ✓ https://github.com/openai/gym/tree/master/gym/wrappers 참고

예,

```
import gym
from ray.rllib.utils.numpy import one_hot

class OneHotEnv(gym.core.ObservationWrapper):
    # Override `observation` to custom process the original observation
    # coming from the env.
    def observation(self, observation):
        # E.g. one-hotting a float obs [0.0, 5.0[.
        return one_hot(observation, depth=5)
```

```
class ClipRewardEnv(gym.core.RewardWrapper):
    def __init__(self, env, min_, max_):
        super().__init__(env)
        self.min = min_
        self.max = max_

    # Override `reward` to custom process the original reward coming
    # from the env.
    def reward(self, reward):
        # E.g. simple clipping between min and max.
        return np.clip(reward, self.min, self.max)
```

Model : default model config setting

```
MODEL_DEFAULTS: ModelConfigDict = {
    # Experimental flag.
    # If True, try to use a native (tf.ke
    # model instead of our built-in Model
    # If False (default), use "classic" M
    # Note that this currently only works
    # 1) framework != torch AND
    # 2) fully connected and CNN default
    # auto-wrapped LSTM- and attention ne
    "_use_default_native_models": False,

    # === Built-in options ===
    # FullyConnectedNetwork (tf and torch
    # These are used if no custom model i
    # Number of hidden layers to be used.
    "fcnet_hiddens": [256, 256],
    # Activation function descriptor.
    # Supported values are: "tanh", "relu
    # "linear" (or None).
    "fcnet_activation": "tanh",

    # VisionNetwork (tf and torch): rllib
    # These are used if no custom model i
    # Filter config: List of [out_channel
    # Example:
    # Use None for making RLlib try to fi
    # observation space.
    "conv_filters": None,
    # Activation function descriptor.
    # Supported values are: "tanh", "relu
    # "linear" (or None).
    "conv_activation": "relu",

    # Some default models support a final
```

```
# == Attention Nets (experimental: torch-version is untested) ==
# Whether to use a GTrXL ("Gru transformer XL"; attention net) as the
# wrapper Model around the default Model.
"use_attention": False,
# The number of transformer units within GTrXL.
# A transformer unit in GTrXL consists of a) MultiHeadAttention module and
# b) a position-wise MLP.
"attention_num_transformer_units": 1,
# The input and output size of each transformer unit.
"attention_dim": 64,
# The number of attention heads within the MultiHeadAttention units.
"attention_num_heads": 1,
# The dim of a single head (within the MultiHeadAttention units).
"attention_head_dim": 32,
# The memory sizes for inference and training.
"attention_memory_inference": 50,
"attention_memory_training": 50,
# The output dim of the position-wise MLP.
"attention_position_wise_mlp_dim": 32,
# The initial bias values for the 2 GRU gates within a transformer unit.
"at
# W
"at
# W
"at

# =
# W
# "
# >
#
#
#
```

```
# === Options for custom models ===
# Name of a custom model to use
"custom_model": None,
# Extra options to pass to the custom classes. These will be available to
# the Model's constructor in the model_config field. Also, they will be
# attempted to be passed as **kwargs to ModelV2 models. For an example,
# see rllib/models/[tf|torch]/attention_net.py.
"custom_model_config": {},
# Name of a custom action distribution to use.
"custom_action_dist": None,
# Custom preprocessors are deprecated. Please use a wrapper class around
# your environment instead to preprocess observations.
"custom_preprocessor": None
```

Model : default model config setting

- 앞장의 것들을 Trainer config의 model 키를 이용하여 설정

```
algo_config = {
    # All model-related settings go into this sub-dict.
    "model": {
        # By default, the MODEL_DEFAULTS dict above will be used.

        # Change individual keys in that dict by overriding them, e.g.
        "fcnet_hiddens": [512, 512, 512],
        "fcnet_activation": "relu",
    },

    # ... other Trainer config keys, e.g. "lr" ...
    "lr": 0.00001,
}
```

Model : customizing

- How to provide own model logic
  - TFModelV2(for TensorFlow) or TorchModel2(for PyTorch)의 subclass로 정의(구현)
  - 모델 카탈로그에 등록
  - Config에서 명시 : { "model": { "custom_model": "MyModel", "custom_model_config":{}, ..}}

- Custom Model (TF만 설명 예정)
  - TFModelV2(TorchModelV2) 상속 받아 __init__(), forward() 메소드 구현, 다른 메소드 override
    - __init__() : 모델 구성
    - forward() : 입력(inpout tensor, state)을 받아서 model output 반환

```
import ray
import ray.rllib.agents.ppo as ppo
from ray.rllib.models import ModelCatalog
from ray.rllib.models.tf.tf_modelv2 import TFModelV2

class MyModelClass(TFModelV2):
    def __init__(self, obs_space, action_space, num_outputs, model_config, name): ...
    def forward(self, input_dict, state, seq_lens): ...
    def value_function(self): ...
```

정의

```
ModelCatalog.register_custom_model("my_tf_model", MyModelClass)
```

카탈로그에 등록

```
ray.init()
trainer = ppo.PPOTrainer(env="CartPole-v0", config={
    "model": {
        "custom_model": "my_tf_model",
        # Extra kwargs to be passed to your model's c'tor.
        "custom_model_config": {},
    },
})
```

Config에 명시

※ 참고 : 다양한 유형의 Custom Model 만드는 방법
https://docs.ray.io/en/master/rllib-models.html

Model : customizing

- Auto-wrapper 를 이용하여 custom model을 LSTM 또는 Attention Net으로 Wrapping

정의

```
# The custom model that will be wrapped by an LSTM.
class MyCustomModel(TorchModelV2):
    def __init__(self, obs_space, action
                 name):
        super().__init__(obs_space, acti
                         name)
        self.num_outputs = int(np.produc
        self._last_batch_size = None

    # Implement your own forward logic,
    # through an LSTM.
    def forward(self, input_dict, state,
        obs = input_dict["obs_flat"]
        # Store last batch size for valu
        self._last_batch_size = obs.shap
        # Return 2x the obs (and empty s
        # This will further be sent thro
        # LSTM head (b/c we are setting
        return obs * 2.0, []

    def value_function(self):
        return torch.from_numpy(np.zeros
```

등록

```
if __name__ == "__main__":
    ray.init()

    # Register the above custom model.
    ModelCatalog.register_custom_model("my_torch_model", MyCustomModel)

    # Create the Trainer.
    trainer = ppo.PPOTrainer(
        env="CartPole-v0",
        config={
            "framework": "torch",
            "model": {
                # Auto-wrap the custom(!) model with an LSTM.
                "use_lstm": True,
                # To further customize the LSTM auto-wrapper.
                "lstm_cell_size": 64,

                # Specify our custom model from above.
                "custom_model": "my_torch_model",
                # Extra kwargs to be passed to your model's c'tor.
                "custom_model_config": {},
            },
        })
    trainer.train()
```

또는 “use_attention”:True

연결

– Wrapper 를 이용하는 것이 아니라 Custom RNN, Attention Net 도 정의하여 사용 가능

Action Distribution

- Custom Model/preprocessor와 유사하게 Custom action distribution을 정의 & 활용

```python
import ray
import ray.rllib.agents.ppo as ppo
from ray.rllib.models import ModelCatalog
from ray.rllib.models.preprocessors import Preprocessor
```

```python
class MyActionDist(ActionDistribution):
    @staticmethod
    def required_model_output_shape(action_space, model_config):
        return 7  # controls model output feature vector size

    def __init__(self, inputs, model):
        super(MyActionDist, self).__init__(inputs, model)
        assert model.num_outputs == 7

    def sample(self): ...
    def logp(self, actions): ...
    def entropy(self): ...
```

상속 받아 정의

```python
ModelCatalog.register_custom_action_dist("my_dist", MyActionDist)
```

카탈로그에 등록

```python
ray.init()
trainer = ppo.PPOTrainer(env="CartPole-v0", config={
    "model": {
        "custom_action_dist": "my_dist",
    },
})
```

Config 이용하여 연결

Action Distribution : 모델의 출력을 해석하여 다음 동작 결정

Available Algorithms

Algorithm	Frameworks	Discrete Actions	Continuous Actions	Multi-Agent	Model Support	Multi-GPU
A2C, A3C	tf + torch	**Yes** +parametric	**Yes**	**Yes**	+RNN, +LSTM auto-wrapping, +Attention, +autoreg	A2C: tf + torch
ARS	tf + torch	**Yes**	**Yes**	No		No
BC	tf + torch	**Yes** +parametric	**Yes**	**Yes**	+RNN	torch
CQL	tf + torch	No	**Yes**	No		tf + torch
ES	tf + torch	**Yes**	**Yes**	No		No
DDPG, TD3	tf + torch	No	**Yes**	**Yes**		torch
APEX-DDPG	tf + torch	No	**Yes**	**Yes**		torch
Dreamer	torch	No	**Yes**	No	+RNN	torch
DQN, Rainbow	tf + torch	**Yes** +parametric	No	**Yes**		tf + torch
APEX-DQN	tf + torch	**Yes** +parametric	No	**Yes**		torch
IMPALA	tf + torch	**Yes** +parametric	**Yes**	**Yes**	+RNN, +LSTM auto-wrapping, +Attention, +autoreg	tf + torch
MAML	tf + torch	No	**Yes**	No		torch
MARWIL	tf + torch	**Yes** +parametric	**Yes**	**Yes**	+RNN	torch
MBMPO	torch	No	**Yes**	No		torch
PG	tf + torch	**Yes** +parametric	**Yes**	**Yes**	+RNN, +LSTM auto-wrapping, +Attention, +autoreg	tf + torch
PPO, APPO	tf + torch	**Yes** +parametric	**Yes**	**Yes**	+RNN, +LSTM auto-wrapping, +Attention, +autoreg	tf + torch
R2D2	tf + torch	**Yes** +parametric	No	**Yes**	+RNN, +LSTM auto-wrapping, +autoreg	torch
SAC	tf + torch	**Yes**	**Yes**	**Yes**		torch
SlateQ	torch	**Yes**	No	No		torch
LinUCB, LinTS	torch	**Yes** +parametric	No	**Yes**		No
AlphaZero	torch	**Yes** +parametric	No	No		No

Multi-Agent only Methods

Algorithm	Frameworks	Discrete Actions	Continuous Actions	Multi-Agent	Model Support
QMIX	torch	**Yes** +parametric	No	**Yes**	+RNN
MADDPG	tf	**Yes**	Partial	**Yes**	
Parameter Sharing	Depends on bootstrapped algorithm				
Fully Independent Learning	Depends on bootstrapped algorithm				
Shared Critic Methods	Depends on bootstrapped algorithm				

오늘은 여기까지....

Policies

- Encapsulate the core numerical components of RL alg.s
- includes
  - Policy model : determines actions to take
  - A trajectory postprocessor for experiences
  - A loss func to improve the policy given postprocessed experiences

Policy/Trainer customizing

- build_tf_policy()/build_trainer() 를 이용
  – Cf., Policy, TFPolicy, DynamicTFPolicy의 subclass로 Policy 를 정의

```
import tensorflow as tf
from ray.rllib.policy.sample_batch import SampleBatch

def policy_gradient_loss(policy, model, dist_class, train_batch):
    actions = train_batch[SampleBatch.ACTIONS]
    rewards = train_batch[SampleBatch.REWARDS]
    logits, _ = model.from_batch(train_batch)
    action_dist = dist_class(logits, model)
    return -tf.reduce_mean(action_dist.logp(actions) * rewards)
```

사용자 정의 손실 함수를 이용한 Policy 생성

```
from ray.rllib.policy.tf_policy_template import build_tf_policy

# <class 'ray.rllib.policy.tf_policy_template.MyTFPolicy'>
MyTFPolicy = build_tf_policy(
    name="MyTFPolicy",
    loss_fn=policy_gradient_loss)
```

사용자 정의 모델은?
사용자 정의 행동(action)은?

```
import ray
from ray import tune
from ray.rllib.agents.trainer_template import build_trainer

# <class 'ray.rllib.agents.trainer_template.MyCustomTrainer'>
MyTrainer = build_trainer(
    name="MyCustomTrainer",
    default_policy=MyTFPolicy)

ray.init()
tune.run(MyTrainer, config={"env": "CartPole-v0", "num_workers": 2})
```

사용자 정의 Policy를 이용한 Trainer 생성

ref. https://docs.ray.io/en/master/rllib-concepts.html

Policy/Trainer customizing

참고 : tf_policy_template.build_tf_policy() 함수

```python
def build_tf_policy(
        name: str,
        *,
        loss_fn: Callable[[
            Policy, ModelV2, Type[TFActionDistribution], SampleBatch
        ], Union[TensorType, List[TensorType]]],
        get_default_config: Optional[Callable[[None],
                                              TrainerConfigDict]] = None,
        postprocess_fn: Optional[Callable[[
            Policy, SampleBatch, Optional[Dict[AgentID, SampleBatch]],
            Optional["MultiAgentEpisode"]
        ], SampleBatch]] = None,
        stats_fn: Optional[Callable[[Policy, SampleBatch], Dict[
            str, TensorType]]] = None,
        optimizer_fn: Optional[Callable[[
            Policy, TrainerConfigDict
        ], "tf.keras.optimizers.Optimizer"]] = None,
        compute_gradients_fn: Optional[Callable[[
            Policy, "tf.keras.optimizers.Optimizer", TensorType
        ], ModelGradients]] = None,
        apply_gradients_fn: Optional[Callable[[
            Policy, "tf.keras.optimizers.Optimizer", ModelGradients
        ], "tf.Operation"]] = None,
        grad_stats_fn: Optional[Callable[[Policy, SampleBatch, ModelGradients],
                                         Dict[str, TensorType]]] = None,
        extra_action_out_fn: Optional[Callable[[Policy], Dict[
            str, TensorType]]] = None,
        extra_learn_fetches_fn: Optional[Callable[[Policy], Dict[
            str, TensorType]]] = None,
        validate_spaces: Optional[Callable[
            [Policy, gym.Space, gym.Space, TrainerConfigDict], None]] = None,
        before_init: Optional[Callable[
            [Policy, gym.Space, gym.Space, TrainerConfigDict], None]] = None,
        before_loss_init: Optional[Callable[[
            Policy, gym.spaces.Space, gym.spaces.Space, TrainerConfigDict
        ], None]] = None,
        after_init: Optional[Callable[
            [Policy, gym.Space, gym.Space, TrainerConfigDict], None]] = None,
        make_model: Optional[Callable[[
            Policy, gym.spaces.Space, gym.spaces.Space, TrainerConfigDict
        ], ModelV2]] = None,
        action_sampler_fn: Optional[Callable[[TensorType, List[
            TensorType]], Tuple[TensorType, TensorType]]] = None,
        action_distribution_fn: Optional[Callable[[
            Policy, ModelV2, TensorType, TensorType, TensorType
        ], Tuple[TensorType, type, List[TensorType]]]] = None,
        mixins: Optional[List[type]] = None,
        get_batch_divisibility_req: Optional[Callable[[Policy], int]] = None,
        # Deprecated args.
        obs_include_prev_action_reward=DEPRECATED_VALUE,
        extra_action_fetches_fn=None,  # Use `extra_action_out_fn`.
        gradients_fn=None,  # Use `compute_gradients_fn`.
) -> Type[DynamicTFPolicy]:
    """Helper function for creating a dynamic tf policy at runtime.
```

```
      make_model (Optional[Callable[[Policy, gym.spaces.Space,
          gym.spaces.Space, TrainerConfigDict], ModelV2]]): Optional callable
          that returns a ModelV2 object.
          All policy variables should be created in this function. If None,
          a default ModelV2 object will be created.
```

```
      action_sampler_fn (Optional[Callable[[TensorType, List[TensorType]],
          Tuple[TensorType, TensorType]]]): A callable returning a sampled
          action and its log-likelihood given observation and state inputs.
          If None, will either use `action_distribution_fn` or
          compute actions by calling self.model, then sampling from the
          so parameterized action distribution.
      action_distribution_fn (Optional[Callable[[Policy, ModelV2, TensorType,
          TensorType, TensorType],
          Tuple[TensorType, type, List[TensorType]]]]): Optional callable
          returning distribution inputs (parameters), a dist-class to
          generate an action distribution object from, and internal-state
          outputs (or an empty list if not applicable). If None, will either
          use `action_sampler_fn` or compute actions by calling self.model,
          then sampling from the so parameterized action distribution.
```


ref. https://github.com/ray-project/ray/blob/master/rllib/policy/tf_policy_template.py

Internal concepts

Policy/Trainer customizing

참고 : trainer_template.build_trainer() 함수

```python
@DeveloperAPI
def build_trainer(
        name: str,
        *,
        default_config: Optional[TrainerConfigDict] = None,
        validate_config: Optional[Callable[[TrainerConfigDict], None]] = None,
        default_policy: Optional[Type[Policy]] = None,
        get_policy_class: Optional[Callable[[TrainerConfigDict], Optional[Type[
            Policy]]]] = None,
        validate_env: Optional[Callable[[EnvType, EnvContext], None]] = None,
        before_init: Optional[Callable[[Trainer], None]] = None,
        after_init: Optional[Callable[[Trainer], None]] = None,
        before_evaluate_fn: Optional[Callable[[Trainer], None]] = None,
        mixins: Optional[List[type]] = None,
        execution_plan: Optional[Callable[[
            WorkerSet, TrainerConfigDict
        ], Iterable[ResultDict]]] = default_execution_plan) -> Type[Trainer]:
    """Helper function for defining a custom trainer.

    Functions will be run in this order to initialize the trainer:
        1. Config setup: validate_config, get_policy
        2. Worker setup: before_init, execution_plan
        3. Post setup: after_init
```

```python
    Args:
        name (str): name of the trainer (e.g., "PPO")
        default_config (Optional[TrainerConfigDict]): The default config dict
            of the algorithm, otherwise uses the Trainer default config.
        validate_config (Optional[Callable[[TrainerConfigDict], None]]):
            Optional callable that takes the config to check for correctness.
            It may mutate the config as needed.
        default_policy (Optional[Type[Policy]]): The default Policy class to
            use if `get_policy_class` returns None.
        get_policy_class (Optional[Callable[
            TrainerConfigDict, Optional[Type[Policy]]]]): Optional callable
            that takes a config and returns the policy class or None. If None
            is returned, will use `default_policy` (which must be provided
            then).
        validate_env (Optional[Callable[[EnvType, EnvContext], None]]):
            Optional callable to validate the generated environment (only
            on worker=0).
        before_init (Optional[Callable[[Trainer], None]]): Optional callable to
            run before anything is constructed inside Trainer (Workers with
            Policies, execution plan, etc..). Takes the Trainer instance as
            argument.
        after_init (Optional[Callable[[Trainer], None]]): Optional callable to
            run at the end of trainer init (after all Workers and the exec.
            plan have been constructed). Takes the Trainer instance as
            argument.
        before_evaluate_fn (Optional[Callable[[Trainer], None]]): Callback to
            run before evaluation. This takes the trainer instance as argument.
        mixins (list): list of any class mixins for the returned trainer class.
            These mixins will be applied in order and will have higher
            precedence than the Trainer class.
        execution_plan (Optional[Callable[[WorkerSet, TrainerConfigDict],
            Iterable[ResultDict]]]): Optional callable that sets up the
            distributed execution workflow.

    Returns:
        Type[Trainer]: A Trainer sub-class configured by the specified args.
    """
```

ref. https://github.com/ray-project/ray/blob/master/rllib/agents/trainer_template.py

Policy/Trainer customizing

사용자 정의 모델, action_sampler 정의 함수를 이용한 Policy

```
SimpleQPolicy = build_tf_policy(
    name="SimpleQPolicy",
    get_default_config=lambda: ray.rllib.agents.dqn.dqn.DEFAULT_CONFIG,
    make_model=build_q_models,
    action_sampler_fn=build_action_sampler,
    loss_fn=build_q_losses,
    extra_action_feed_fn=exploration_setting_inputs,
    extra_action_out_fn=lambda policy: {"q_values": policy.q_values},
    extra_learn_fetches_fn=lambda policy: {"td_error": policy.td_error},
    before_init=setup_early_mixins,
    after_init=setup_late_mixins,
    obs_include_prev_action_reward=False,
    mixins=[
        ExplorationStateMixin,
        TargetNetworkMixin,
    ])
```

action_distribution_fn=build_action_distribution

Policy, model, inpit_dic, obs_state, action,...

```
def build_q_models(policy, obs_space, a
    ...

    policy.q_model = ModelCatalog.get_m
        obs_space,
        action_space,
        num_outputs,
        config["model"],
        framework="tf",
        name=Q_SCOPE,
        model_interface=SimpleQModel,
        q_hiddens=config["hiddens"])

    policy.target_q_model = ModelCatalc
        obs_space,
        action_space,
        num_outputs,
        config["model"],
        framework="tf",
        name=Q_TARGET_SCOPE,
        model_interface=SimpleQModel,
        q_hiddens=config["hiddens"])

    return policy.q_model
```

```
        extra_learn_fetches_fn: Optional[Calla
            str, TensorType]]] = None,
        validate_spaces: Optional[Callable[
            [Policy, gym.Space, gym.Space, Tra
        before_init: Optional[Callable[
            [Policy, gym.Space, gym.Space, Tra
        before_loss_init: Optional[Callable[[
            Policy, gym.spaces.Space, gym.spac
        ], None]] = None,
        after_init: Optional[Callable[
            [Policy, gym.Space, gym.Space, Tra
        make_model: Optional[Callable[[
            Policy, gym.spaces.Space, gym.spac
        ], ModelV2]] = None,
        action_sampler_fn: Optional[Callable[[
            TensorType]], Tuple[TensorType, Te
        action_distribution_fn: Optional[Calla
            Policy, ModelV2, TensorType, Tenso
        ], Tuple[TensorType, type, List[Tensor
        mixins: Optional[List[type]] = None,
        get_batch_divisibility_req: Optional[C
        # Deprecated args.
        obs_include_prev_action_reward=DEPRECA
        extra_action_fetches_fn=None,  # Use `
        gradients_fn=None,  # Use `compute_gra
) -> Type[DynamicTFPolicy]:
    """Helper function for creating a dynamic
```

```
def build_action_sampler(policy, q_model, input_dict, obs_space, action_space,
                         config):
    # do max over Q values...
    ...
    return action, action_logp
```

ref. https://docs.ray.io/en/master/rllib-concepts.html

Policy/Trainer customizing

기존 Policy/Trainer 확장을 통한 Customizing

- Trainer / Policy 객체의 with_update() 메소드를 이용하여 일부 변경된 Trainer / Policy 객체 사본을 만듦
- build_tf_policy()/build_trainer() 에 전달되는 인자를 이용하여 달라져야 하는 부분 정의

```python
from ray.rllib.agents.ppo import PPOTrainer
from ray.rllib.agents.ppo.ppo_tf_policy import PPOTFPolicy

CustomPolicy = PPOTFPolicy.with_updates(
    name="MyCustomPPOTFPolicy",
    loss_fn=some_custom_loss_fn)

CustomTrainer = PPOTrainer.with_updates(
    default_policy=CustomPolicy)
```

loss_fn 변경

policy 변경

```python
    def with_updates(**overrides):
        """Allows creating a TFPolicy cls based on settings of another one.

        Keyword Args:
            **overrides: The settings (passed into `build_tf_policy`) that
                should be different from the class that this method is called
                on.

        Returns:
            type: A new TFPolicy sub-class.

        Examples:
        >> MySpecialDQNPolicyClass = DQNTFPolicy.with_updates(
        ..    name="MySpecialDQNPolicyClass",
        ..    loss_function=[some_new_loss_function],
        .. )
        """
        return build_tf_policy(**dict(original_kwargs, **overrides))
```

From https://github.com/ray-project/ray/blob/master/rllib/policy/tf_policy_template.py

ref. https://docs.ray.io/en/master/rllib-concepts.html

Policy Evaluation

- produces batches of experiences.
  - Efficient policy evaluation can be burdensome to get right
    - especially when leveraging vectorization, RNNs, or when operating in a multi-agent environment
  - RLlib provides a RolloutWorker class that manages all of this, and this class is used in most RLlib algorithms.
    - We can use rollout workers standalone to produce batches of experiences.
    - This can be done by calling worker.sample() on a worker instance, or worker.sample.remote() in parallel on worker instances created as Ray actors

- 예
  - (1) creating a set of rollout workers
  - (2) using them gather experiences in parallel
  - (3) The trajectories are concatenated
  - (4) the policy learns on the trajectory batch,
  - (5) then we broadcast the policy weights to the workers for the next round of rollouts:

```
# Setup policy and rollout workers
env = gym.make("CartPole-v0")
policy = CustomPolicy(env.observation_space, env.action_space, {})
workers = WorkerSet(
(1)     policy_class=CustomPolicy,
    env_creator=lambda c: gym.make("CartPole-v0"),
    num_workers=10)

while True:
    # Gather a batch of samples (3)
    T1 = SampleBatch.concat_samples(
        ray.get([w.sample.remote() for w in workers.remote_workers()]))
          (2)

    # Improve the policy using the T1 batch
(4) policy.learn_on_batch(T1)

    # Broadcast weights to the policy evaluation workers
(5) weights = ray.put({"default_policy": policy.get_weights()})
    for w in workers.remote_workers():
        w.set_weights.remote(weights)
```


ref. https://docs.ray.io/en/master/rllib-concepts.html

Execution Plans

- Represent the dataflow of RL Training Job
  - 일련의 스텝들을 통해 RL 알고리즘의 실행을 쉽게 표현할 수 있게 함
    - Learner에서 순차적으로 발생하거나 다수의 actor들을 통해 병렬로 발생하는 스텝
  - RLlib 이 plan 을 ray actor들 상에서 ray.get()/ray.wait() 연산자들로 변환
  - 저수준 ray actor 호출을 다룰 필요없이 고성능 알고리즘을 쉽게 만들수 있게 함
  - build_trainer()의 인자로 전달하여 Trainer Customizing

- 예 : A2C 알고리즘
  - 다음 3 스텝의 반복
    1. `ParallelRollouts`: Generate experiences from many envs in parallel using rollout workers.
    2. `ConcatBatches`: The experiences are concatenated into one batch for training.
    3. `TrainOneStep`: Take a gradient step with respect to the policy loss, and update the worker weights.
  - A2C 알고리즘의 Dataflow를 코드화 하면

```python
def execution_plan(workers: WorkerSet, config: TrainerConfigDict):
    # type: LocalIterator[SampleBatchType]
    rollouts = ParallelRollouts(workers, mode="bulk_sync")

    # type: LocalIterator[(SampleBatchType, List[LearnerStatsDict])]
    train_op = rollouts \
        .combine(ConcatBatches(
            min_batch_size=config["train_batch_size"])) \
        .for_each(TrainOneStep(workers))

    # type: LocalIterator[ResultDict]
    return StandardMetricsReporting(train_op, workers, config)
```

ref. https://docs.ray.io/en/master/rllib-concepts.html

Execution Plans

- Execution Plan에 사용될 수 있는 Operators

구분	설명
Rollout ops	Functions for generating and working with experiences • ParallelRollouts : for generating experiences synchronously or asynchronously • ConcatBatches : for combining batches together • SelectExperiences : for selecting relevant experiences in a multi-agent setting • AsyncGradients : for computing gradients over new experiences on the fly, asynchronously, as in A3C
Train ops	functions that improve the policy and update workers • TrainOneStep : take in as input a batch of experiences and emit metrics as output(basic op) • TrainTFMultiGPU : for multi-GPU optimization • ComputeGradients : to compute gradients without updating the policy • ApplyGradients : to apply computed gradients to a policy
Replay ops	• StoreToReplayBuffer : can save experiences batches to either a local replay buffer or a set of distribute d replay actors • Replay : produces a new stream of experiences replayed from one of the aforementioned replay buffers

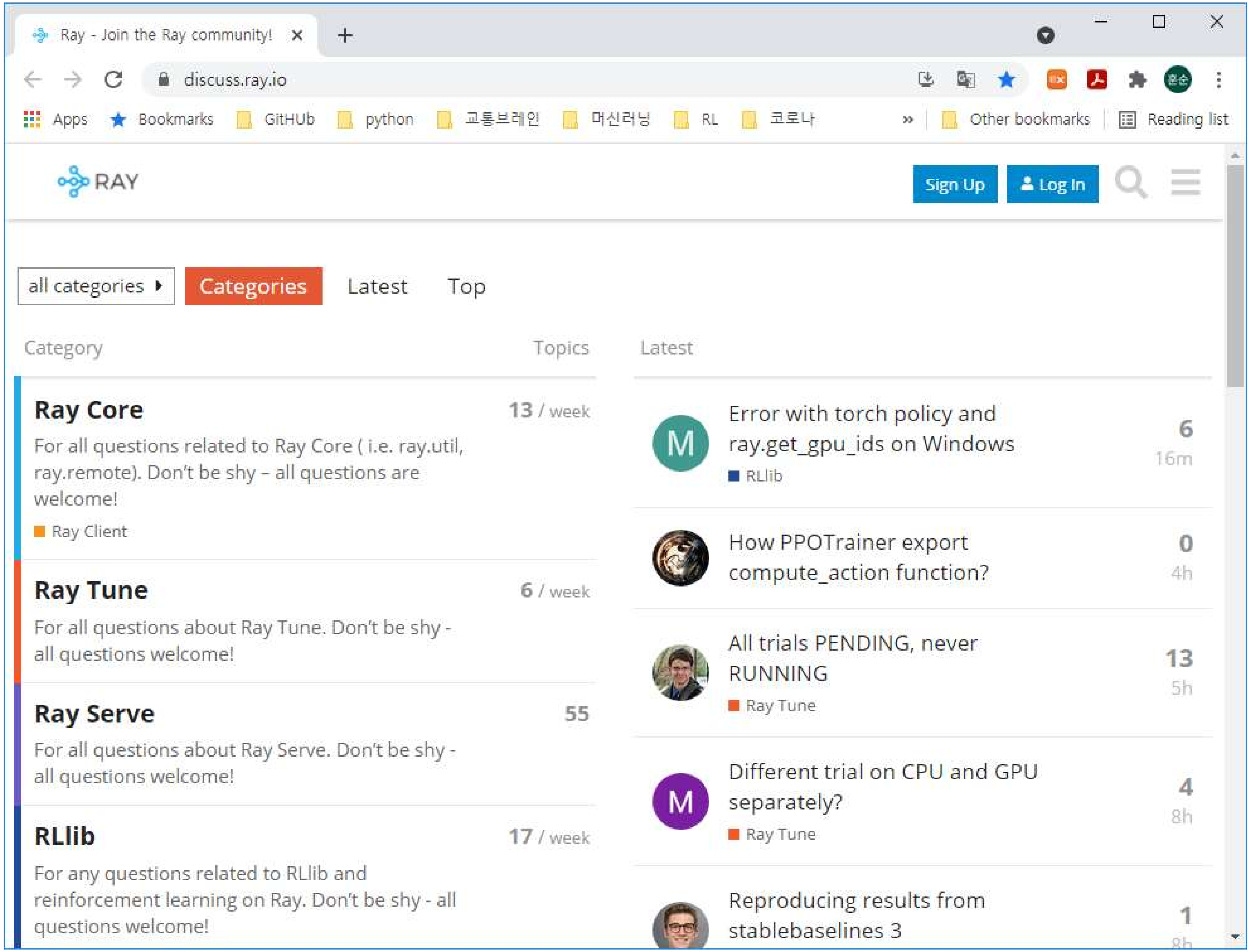
Concurrency ops	• Concurrently : composes multiple iterators (dataflows) into a single dataflow by executing them in an int erleaved fashion
	  • The output can be defined to be the mixture of the two dataflows, or filtered to that of one of the sub-d ataflows
	  • It has two modes:
	    • round_robin: Alternate taking items from each input dataflow.
	    • async: Execute each input dataflow as fast as possible without blocking.

Metric ops	• Execution plans should always end with this operator. • This metrics op also reports various internal performance metrics stored by other operators in the share d metrics context accessible via _get_shared_metrics(). • StandardMetricsReporting : collects training metrics from the rollout workers in a unified fashion, and re turns a stream of training result dicts.

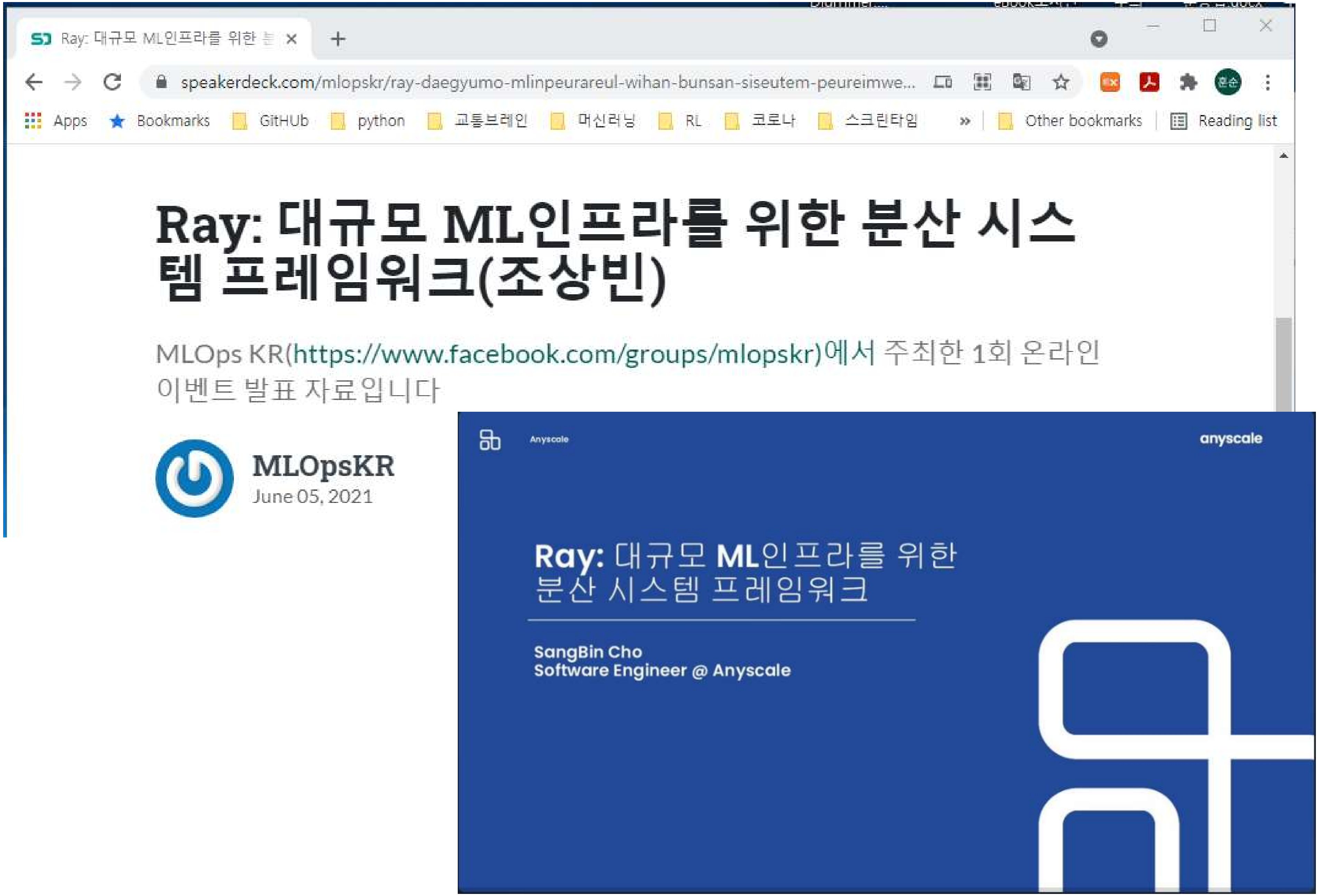
ref. https://docs.ray.io/en/master/rllib-concepts.html

RAY SUMMIT 2021, June 23 ~ 25, https://raysummit.anyscale.com/

https://raysummit.anyscale.com/content/Videos/Svaz4RtzmzQ6xWfpQ



https://discuss.ray.io/



https://speakerdeck.com/mlopskr/ray-daegyumo-mlinpeurareul-wihan-bunsan-siseutem-peureimweokeu-josangbin https://github.com/DLR-RM/stable-baselines3

OpenAI Baselines is a set of high-quality implementations of reinforcement learning algorithms.

Stable Baselines is a set of improved implementations of reinforcement learning algorithms based on OpenAI Baselines.

DLR-RM/stable-baselines3: PyT
github.com/DLR-RM/stable-baseli
Apps Bookmarks GitHUb python

README.md

pipeline passed | docs passing
coverage 96.00% | code style black

Stable Baselines3

Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch. It is the next major version of Stable Baselines.

You can read a detailed presentation of Stable Baselines3 in the v1.0 blog post.

These algorithms will make it easier for the research community and industry to replicate, refine, and identify new ideas, and will create good baselines to build projects on top of. We expect these tools will be used as a base around which new ideas can be added, and as a tool for comparing a new approach against existing ones. We also

Used by 339

+ 331

Contributors 48

+ 37 contributors

Languages

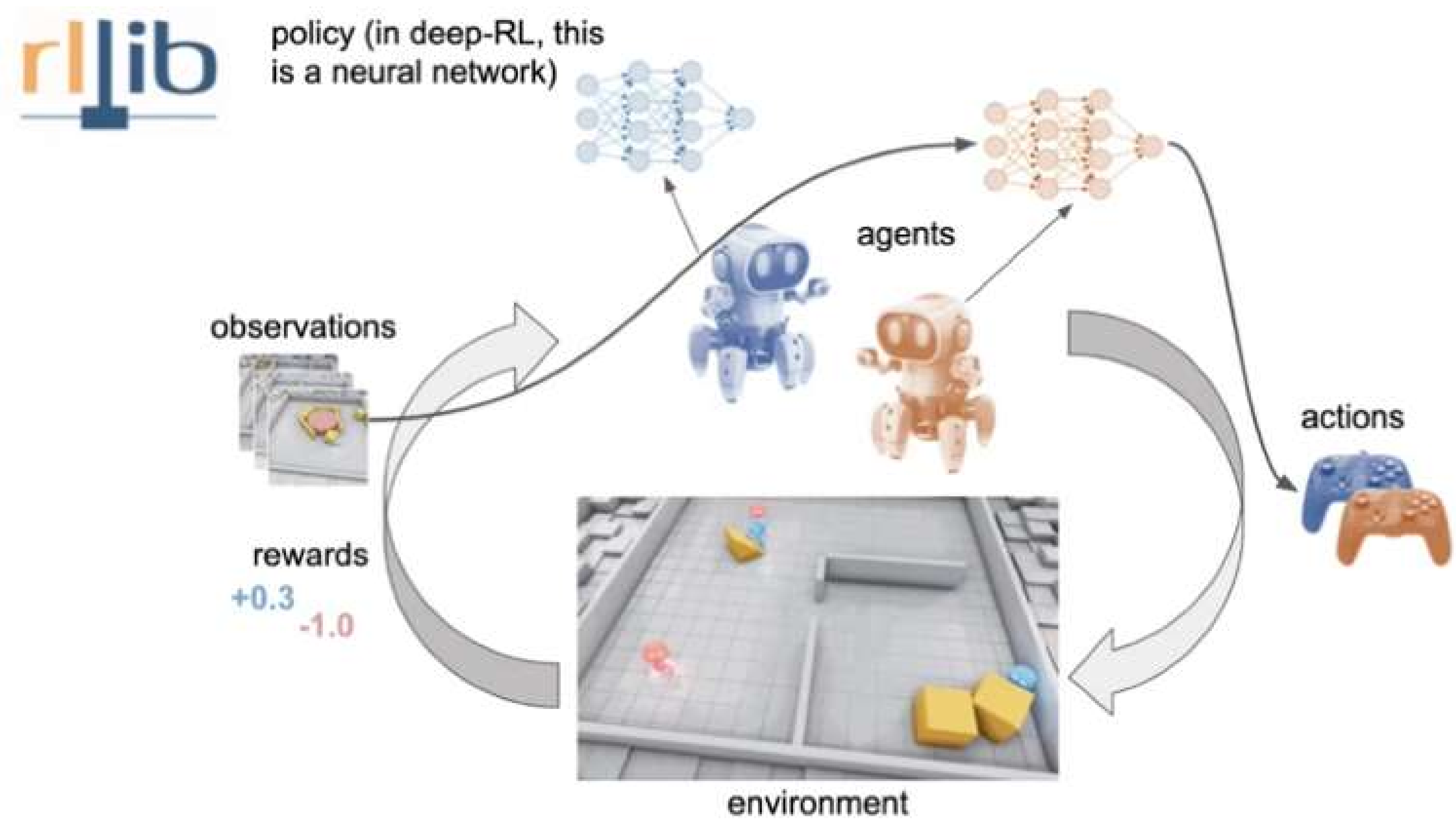
Python 99.4%

생각해 볼 것...

- (-) 디버깅이 어렵다
  - 분산 환경
  - 디버거로 따라가기 어렵다
    - 문제 단순화하여 점진적 확장
      - ✓ 환경 만들고, stable_baselines3 의 모델과 연동해서 디버깅 후에 진행
- (+) 잘 구현된 다양한 RL 알고리즘을 이용할 수 있다.
- (+) 학습 시간/튜닝 등에 대한 고민이 줄어든다.
  - 분산 병렬 실행
  - Tuner

전형적인 RL

RLlib이 제공하는 것(일부)



전형적인 RL

RLlib이 제공하는 것(일부)