2월 3주차 레포트

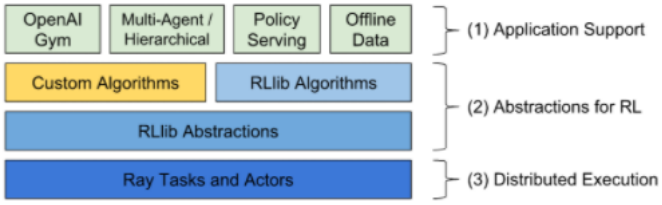
RLlib 기초 - 분석

1. Ray - RLlib

Ray의 전체적인 구조

분산처리를 위한 프로젝트로 파이썬으로 작성되어있다.

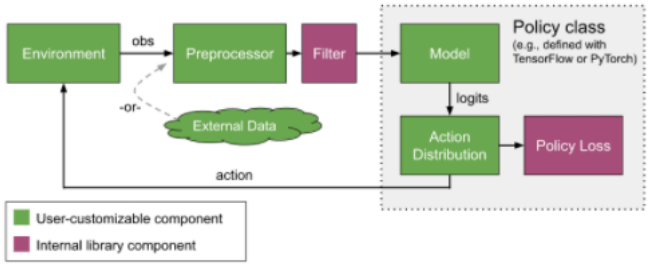
이를 바탕으로 시간이 오래 걸리는 강화학습을 더욱 효율적으로 할 수 있다.



RLlib

RLlib에서 바꿀 수 있는 부분은 다음과 같다.

해당 부분에 대해서 사용자가 커스터마이즈를 할 수 있다.



Filter와 Policy Loss를 제외하고 나머지 부분에서는 사용자가 설정할 수 있다.

Policy

Agent를 조종하기 위한 Policy의 구조

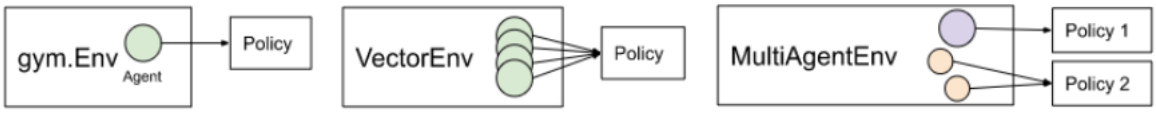
[ex]

single agent에 대해서 하나의 policy를 할당

multi agent에 대해서 하나의 policy를 할당

multi agent에 대해서 각각 다른 policy를 할당

>> 최종적인 목적은, multi agent setting에서 각각 다른 policy를 가지고 행동하는 것



제일 좌측 - single agent (단일 policy)

중간 - multi agent (단일 policy)

마지막 - multi agent (다른 policy)

Sample Batches

Episode에 대해서 Trajectory를 축적한다.

모든 정보들은 모두 이곳(Trajectory??)에 저장된다.

만일 MultiAgent 환경이라면, 각각의 Agent마다 Sample Batch를 가지게 된다.

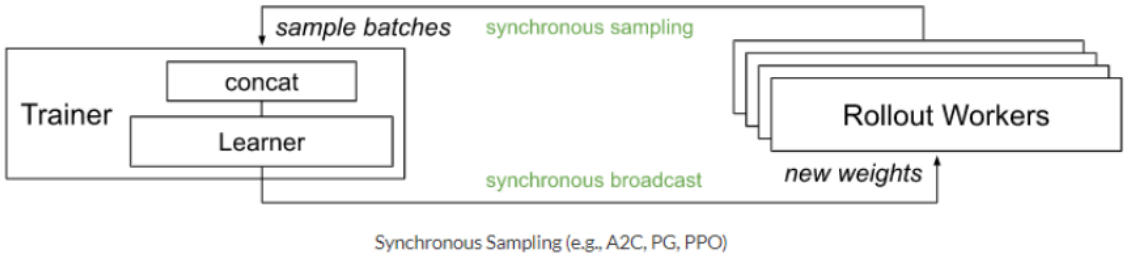
```
{ 'action_logp': np.ndarray((200,), dtype=float32, min=-0.701, max=-0.685, mean=-0.694),
  'actions': np.ndarray((200,), dtype=int64, min=0.0, max=1.0, mean=0.495),
  'dones': np.ndarray((200,), dtype=bool, min=0.0, max=1.0, mean=0.055),
  'infos': np.ndarray((200,), dtype=object, head={}),
  'new_obs': 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)}

>> action_logp : logp ??
>> actions : 그냥 액션
>> dones : 결과값?
>> infos : info는 또 정보인가?
>> new_obs : 업데이트 되어지는 정보
>> obs : 관찰되어진 정보?
>> reward : 보상 관련해서 주어진 Trajectory 정보를 지니게 된다.
>> t : 타임스탭?
```

Trainer

데이터 수집과 훈련은 동기화 & 병렬처리

어느정도 훈련된 모델로 데이터 수집 >> 다시 훈련 >> 반복



Leaner : 학습자

concat : ??

synchronous broadcast : (동시 발생하는 : 병렬처리)

Rollout Worker : Rollout으로 나누어져서 병렬로 Worker들이 데이터처리

new weights :

2. RLlib Basic Training

Training

[Trainer Class]의 기능

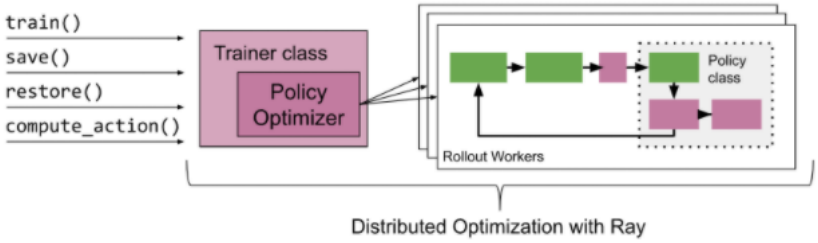
1. Policy Optimizer를 가지고 있으며, 외부 환경과 상호작용을 책임진다.
   >> Policy Optimizer : 정책 최적화

>> 출처 : https://gomguard.tistory.com/187
>> Optimization : 학습속도를 빠르고 안정저깅게 하는 것

2. Policy에 대해서 훈련, Checkpoint, 모델 파라미터 복구, 다음 action을 계산
>> train, save, restore, compute
>> Checkpoint : 추론 또는 학습의 재개를 위한 지점

[trainer]이 기능

1. 환경(Env)와 상호작용하면서 처리하는 모든 것들을 해준다.



Train 관련 코드

```
// 설치 관련 코드 //
pip install 'ray[rllib]'

// 간단한 훈련(Cmd 창) - 실행 //
rllib train --run DQN --env CartPole-v0
>> Algorithm : DQN (Deep Q-Network)
>> Envrionment : CartPole-v0 (Gym의 내장 환경)
  >> Gym에서 제공하는 툴을 이용하여 Env도 만들어 보자

// 간단한 훈련(Cmd 창) - 결과 //
== Status ==
Current time: 2022-02-15 12:58:12 (running for 00:04:25.44)
Memory usage on this node: 5.9/23.0 GiB
Using FIFO scheduling algorithm.
Resources requested: 1.0/16 CPUs, 0/0 GPUs, 0.0/11.54 GiB heap, 0.0/5.77 GiB objects
Result logdir: /home/tot4766/ray_results/default
Number of trials: 1/1 (1 RUNNING)
+-----------------------------+----------+------------------+--------+------------------+-------+----------+----------------------+--
| Trial name                  | status   | loc              |   iter |   total time (s) |    ts |   reward |   episode_reward_max |
|-----------------------------+----------+------------------+--------+------------------+-------+----------+----------------------+--
| DQN_CartPole-v0_e0872_00000 | RUNNING  | 192.168.0.3:34564 |     79 |          252.769 | 79000 |   161.69 |                  200 |
+-----------------------------+----------+------------------+--------+------------------+-------+----------+----------------------+--
```

>> Current time : 현재 시간
  >> running for : 실행된 시간
>> Memory usage on this node : 총 파일에 대한 현재 파일의 용량 진행도
>> Using FIFO scheduling algoritm : 단순 설명
>> Resources requested : Resources에서 요구된 사항들
  >> CPUs : 요구되는 CPUs 사용개수
  >> GPUs : 요구되는 GPUs 사용개수
  >> GiB heap : ??
  >> GiB objects: ??
>> Result logdir : 결과값이 저장되는 위치
>> Number of trials : 시행착오의 개수??

>> Trial name : Algorithm과 Environment의 합쳐진 최종 형식의 툴의 이름

>> status : 현재 상태 (running or complete)

>> loc : ?? IP 주소?

>> iter : ??

>> total time (s) : 현재 status까지 걸린 전체 시간

>> ts : ??

>> reward : 현재 보상

>> episode_reward_max : 지금 현재 에피소드에서 주어진 보상의 최대치

>> episode_reward_min : 지금 현재 에피소드에서 주어진 보상의 최소치

>> episdoe_len_mean : 지금 현재 에피소드에서의 주어진 제곱근 평균값

Train 결과 파일 구조

Command line에서 실행한 RLlib의 결과는 ray_results/에 저장된다.

>> params.json : 모델, 환경에 대한 정보

>> params.pkl : 모델 파라미터

>> result.json : ??

>> progress : 결과값을 CSV 파일 형식으로 저장

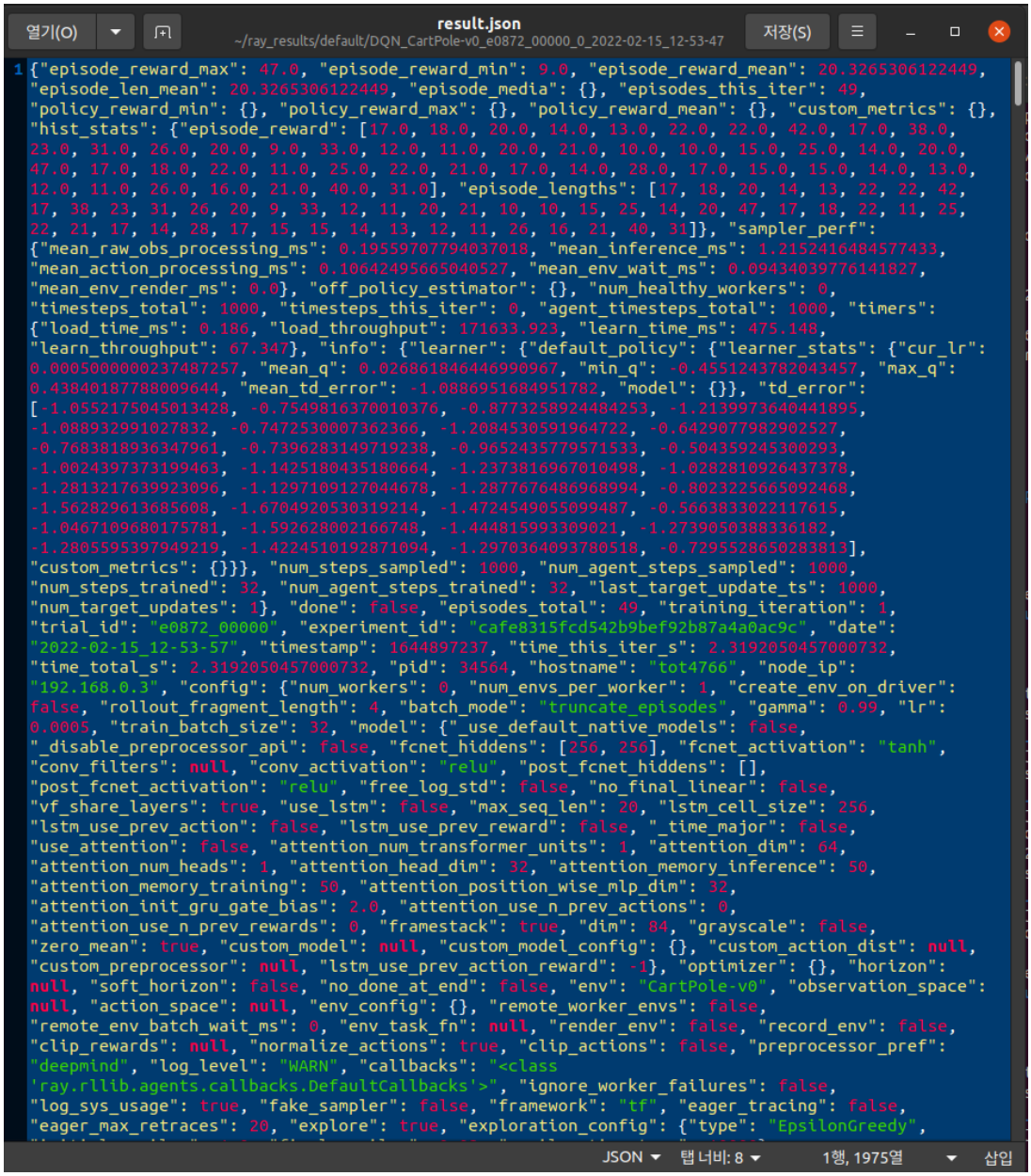
>> events.out.tfevents : Plot 가능한 값에 대해서 tensorborad로 확인 가능하다.

>> tensorboard —logdir .

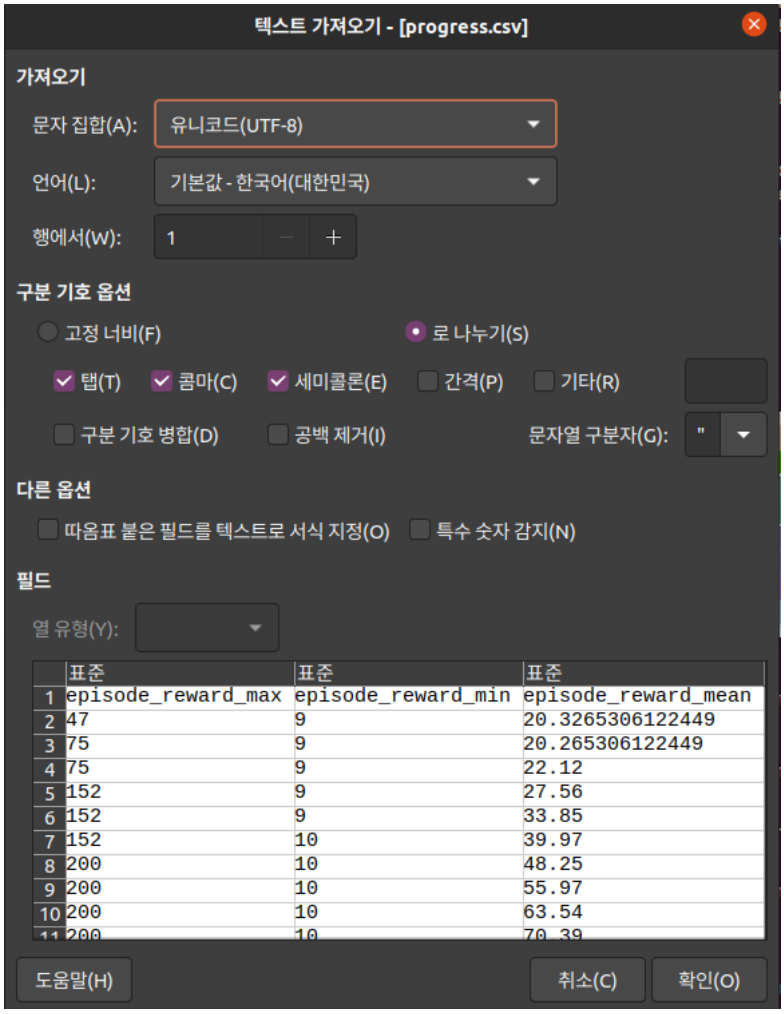
```
// params.json //
{
  "env": "CartPole-v0"
}

// params.pkl //
?? 확인 불가
```

// result.json //



// progress.csv //



	표준	표준	표준
1	episode_reward_max	episode_reward_min	episode_reward_mean
2	47	9	20.3265306122449
3	75	9	20.265306122449
4	75	9	22.12
5	152	9	27.56
6	152	9	33.85
7	152	10	39.97
8	200	10	48.25
9	200	10	55.97
10	200	10	63.54
11	200	10	70.39

Test 해보기 (Cmd 창)

[train] 대신에 [rollout]을 사용하면 환경에 대해서 simulation을 진행

알고리즘에 대해서 학습된 모델이 필요하기 때문에, checkpoint 정보를 줘야함

```
// Test 해보기 (Cmd 창) - 실행 //
rllib rollout \
    ~/ray_results/default/DQN_CartPole-v0_0upjmdgr0/checkpoint_1/checkpoint-1 \
    --run DQN --env CartPole-v0 --steps 10000
```

Basic Python API(.py File)

[Trainer]

직접 생성하는 방법으로 환경에 대해서 모델을 학습시킬 수 있다.

[Trainer] 안에는 호나경과 상호작용으로 배우는 모든 정보가 담겨져 있으므로, high level api로 생각할 수도 있을 것 같다.

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

# === initialize ray ===
ray.init()
config = ppo.DEFAULT_CONFIG.copy()
```

```
config["num_gpus"] = 0
config["num_workers"] = 1
config["eager"] = False

# === make trainer(ppo algorithm) ==
# ppo trainer을 생성하면서, 환경에 대한 정보, configuration을 같이 준다.
trainer = ppo.PPOTrainer(config=config, env="CartPole-v0")

# === Train ===
for i in range(1000):
   result = trainer.train()
   print(pretty_print(result))
   if i % 100 == 0:
       checkpoint = trainer.save()
       print("checkpoint saved at", checkpoint)

# === import from the checkpoint ===
trainer.import_model("my_weights.h5")
```

Tune API

[Trainer Class]

직접 호출해서 학습을 진행하는데, 만일 하이퍼 파라미터를 설정해야 한다면, [ray.tune]을 이용해서 파라미터를 설정하고 train을 하는데 필요한 정보들을 추가적으로 설정할 수 있다.

```
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]),
        "eager": False,
    },
)
```

3. Train Model with Ray Trainer

DQN with simple environment

주어진 환경에 대해서 빠르게 강화학습 코드를 돌리는 방법은 다음과 같다.

>> First Register Envrionment

>> Train the model with [tune.run]

필요한 라이브러리를 [import]한다.

```
import ray
from ray.tune.registry import register_env
from ray.rllib.agents import ppo
from ray import tune
from my_env import MyEnv

>> 여기서 MyEnv는 사용자가 만든 Env로 생각이 된다.
```

[gym environment]를 불러오고, [ray]에 등록해줘야 한다.

[distributed system]을 위해서 반드시 등록 해저야 한다.

또한, Model도 등록을 해줘야 한다.

물론 [CartPole-v0] 또는 [DQN]처럼 많이 사용되는 환경과 모델은 이미 등록되어 있다.

```
# Init ray and register custom env
ray.init()
register_env("my_env", lambda config : MyEnv(config))


# Run tune with DQN
tune.run("DQN",
        stop = {"training_iteration": 100},
        config= {'env':'my_env',
                "timesteps_per_iteration": 1000,
                "buffer_size": 10000,
                "train_batch_size": 64,
                'lr' : 1e-4
                }
        )
```

Trainer

위에서 나온 [DQN]은 Trainer이다.

environment로부터 observation을 받아서, 처리하고 action을 compute 해주며, 훈련을 진행한다.

총 4가지 함수가 있는데 마지막 [compute_action()]만 살펴보자

1. train()
2. save()
3. restore()
4. compute_action()

Compute Action

trainer의 compute_action을 사용하면, Observation에 대해서 어떠한 actioon을 취해야 하는지 결정할 수 있다.

```
import ray
from my_env import MyEnv
from ray.rllib.agents import ppo
from ray.tune.registry import register_env

# === init ray ===
ray.init()

# === register env ===
register_env("my_env", lambda config : MyEnv(config))

# === agent and env ===
agent = ppo.PPOTrainer(env="my_env")
env = MyEnv()

# === initial values ===
obs = env.reset()
done = False
episode_reward = 0

# === choose actions with agent.compute_action(obs) ===
while not done:
    action = agent.compute_action(obs)
    obs, reward, done, info = env.step(action)
    episode_reward += reward
    print(episode_reward)
```

Inside of compute_action()

Trainer의 [compute_action(...)] 함수는 [observation]을 처리하고
내부의 [policy]를 이용해서 [action]을 계산하는 역할이다.

```
class Trainer(Trainable):
    @publicAPI
    def compute_action(self,
                       observation,
                       state=None,
                       prev_action=None,
                       prev_reward=None,
                       info=None,
                       policy_id=DEFAULT_POLICY_ID,
                       full_fetch=False):

        if state is None:
            state = []
        # === observation에 대해서 전처리를 진행한다 ===
        preprocessed = self.workers.local_worker().preprocessors[policy_id].transform(observation)
        filtered_obs = self.workers.local_worker().filters[policy_id](preprocessed, update=False)

        # === Trainer 안에 있는 policy를 이용해서 action을 계산한다.  ===
        if state:
            return self.get_policy(policy_id).compute_single_action(...)

        res = self.get_policy(policy_id).compute_single_action(...)
        if full_fetch:
            return res
        else:
            return res[0]
```

Inside policy, Model (Neural Network)

[policy] 안에는 [neural network]가 있어서, 모델을 통해서 다음 행동을 결정할 수도 있다.

만일 [Rule Base]로 행동을 결정하고 싶다면 이는 [compute_action]에서 이루어지고 [model]과는 별개로 진행된다.

```
import ray
import numpy as np
from ray.rllib.agents.ppo import PPOTrainer

## === setup ===
ray.init()
trainer = PPOTrainer(env="CartPole-v0")
policy = trainer.get_policy()

# === feed a minibatch of size 1 ===
logits, _ = policy.model.from_batch({"obs":np.array([[0.1, 0.2, 0.3, 0.4]])})
dist = policy.dist_class(logits, policy.model)

print(type(policy.model))  # <class 'ray.rllib.models.tf.fcnet.FullyConnectedNetwork'>
print(dist.sample())       # Tensor("Squeeze:0", shape=(1,), dtype=int64)
print(dist.logp)           # bound method Categorical.logp
print(policy.model.value_function())     # Tensor("Reshape:0", shape=(1,), dtype=float32)
print(policy.model.base_model.summary()) # Model: "functional_1"
```

Getting Q-values from a DQN

모델을 좀 더 세부적으로 파고든다면, [input batch]를 넣은 상태에서 값들을 추출해낼 수 있다.

```
import ray
from ray.rllib.agents.dqn import DQNTrainer
import numpy as np

# === setup ===
ray.init()
trainer = DQNTrainer(env="CartPole-v0")
model = trainer.get_policy().model

# === feed to model ===
```

```
model_out = model.from_batch({"obs":np.array([[0.1, 0.2, 0.3, 0.4]])})
model_out_dist = model.get_q_value_distributions(model_out)

# === print values ===
print(model_out) # Tuple
print(model_out_dist)  # List
print(model.get_state_value(model_out)) # Tensor


"""
(<tf.Tensor 'functional_1/fc_out/Tanh:0' shape=(1, 256) dtype=float32>, [])

[<tf.Tensor 'BiasAdd_1:0' shape=(1, 2) dtype=float32>,
<tf.Tensor 'default_policy/ExpandDims_10:0' shape=(1, 2, 1) dtype=float32>,
<tf.Tensor 'default_policy/ExpandDims_1_1:0' shape=(1, 2, 1) dtype=float32>]

Tensor("BiasAdd_3:0", shape=(1, 1), dtype=float32)
"""
```

4. RLlib Callbacks

Trainer에서 Batch에 대해서 훈련을 진행하면서, 진행되는 전처리, 후처리를 모두 처리하는 부분이다.

아래의 [CallBack Method]들은 모두 상황에 따라서 필요한 경우가 다르겠지만, 훈련 중간에 각 부분에서 어떤 처리를 하고 싶다면 반드시 구현해야 하는 부분이다.

(ex)

> [MultiAgent] 환경에서 각각의 [Agent]의 [Observation]을 합치고 싶다면, [on_postprocess_trajectory()] 부분을 구현해서 [Batch]의 형태를 바꿔주면 된다.

RLlib Callbacks

Aa CallBack Class Functions	Description
on_spisode_start()	rollout worker에 대해서 episode를 시작하기 전에 불리는 함수입니다.
on_episode_step()	episode의 매 step마다 불리는 함수입니다.
on_episode_end()	episode가 끝날 때 불리는 함수입니다.
on_postprocess_trajectory()	`policy`에서 `policy'postprocess_fn`이 불리고 호출되는 함수로, `batch`를 처리하는 부분입니다. 예를 들어서 `MultiAgent`에서 다른 `Agent`의 Observation을 처리하는 부분을 추가할 수 있습니다.
on_sample_end()	`RolloutWorer.sample()`이 끝나고 호출되는 함수입니다.
on_learned_on_batch()	`Policy.learn_on_batch()`의 첫 부분에 호출되는 함수입니다.
on_train_result()	`Trainer.train()`으로 학습을 완료하고 호출되는 함수입니다.

CallBack Class 사용법

[CartPole-v0] 환경에서 [CallBack]을 적용해보자.

아래의 기능들은 밑에 [Class MyCallbacks]에 구현이 되어 있다.

1. 에피소드가 시작되면 [Pole]의 각도를 저장할 리스트를 생성한다.
2. 에피소드의 매 Timestep 마다 [Pole]의 각도를 저장한다.
3. 에피소드가 끝나면 [Pole]의 각도를 Timestep에 대해서 평균내고 저장한다.
4. [Rollout Worker]에서 샘플링을 진행하면, 샘플의 사이즈를 출력한다.
5. 훈련이 종료되면 [Callback]이 완료되었음을 저장한다.
6. mini batch에 대해서 훈련을 시작하기 전에, action을 평균낸다.

7. Policy를 Mapping 후 Batch를 처리할 때, 처리된 개수를 저장한다.

```
class MyCallbacks(DefaultCallbacks):
    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"] = []

    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)

    def on_episode_end(self, *, worker: RolloutWorker, base_env: BaseEnv,
                       policies: Dict[str, Policy], episode: MultiAgentEpisode,
                       env_index: int, **kwargs):
        pole_angle = np.mean(episode.user_data["pole_angles"])
        print("episode {} (env-idx={}) ended with length {} and pole "
              "angles {}".format(episode.episode_id, env_index, episode.length,
                                 pole_angle))
        episode.custom_metrics["pole_angle"] = pole_angle
        episode.hist_data["pole_angles"] = episode.user_data["pole_angles"]

    def on_sample_end(self, *, worker: RolloutWorker, samples: SampleBatch,
                      **kwargs):
        print("returned sample batch of size {}".format(samples.count))

    def on_learn_on_batch(self, *, policy: Policy, train_batch: SampleBatch,
                          result: dict = {}, **kwargs) -> None:
        result["sum_actions_in_train_batch"] = np.sum(train_batch["actions"])
        print("policy.learn_on_batch() result: {} -> sum actions: {}".format(
            policy, result["sum_actions_in_train_batch"]))

    def on_postprocess_trajectory(
            self, *, worker: RolloutWorker, episode: MultiAgentEpisode,
            agent_id: str, policy_id: str, policies: Dict[str, Policy],
            postprocessed_batch: SampleBatch,
            original_batches: Dict[str, SampleBatch], **kwargs):
        print("postprocessed {} steps".format(postprocessed_batch.count))
        if "num_batches" not in episode.custom_metrics:
            episode.custom_metrics["num_batches"] = 0
        episode.custom_metrics["num_batches"] += 1

    def on_train_result(self, *, trainer, result: dict, **kwargs):
        print("trainer.train() result: {} -> {} episodes".format(
            trainer, result["episodes_this_iter"]))
        # you can mutate the result dict to add new fields to return
        result["callback_ok"] = True
```

CallBack 사용법

[CallBack Class]를 정의하고 난 후에는, [tune]을 하면서
[config]의 [callbacks]에 넣어주면 된다.

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

5. Explore and Curriculum Learning

Explore

모든 [Policy]는 [Exploration Object]를 가지고 있다.

[Policy]가 Model에 의해서 정해지지 않고 탐색을 하는 방식으로 장기적으로 [return]값을 높이기 위해서 강화학습에 필수적이다.

아래와 같은 방식으로 [Explore]를 설정할 수 있다.

```
# DQN
config = {
    "explore": True,
    'expolration_config : {
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.02,
        "epsilon_timesteps": 10000
    }
}

# StochasticSampling
config = {
    "explore": True,
    "exploration_config": {
        "type": "StochasticSampling",
        "random_timesteps": 0,
    }
}
```

Curriculum Learning

Curriculum Learning은 학습을 진행하면서, 난이도가 유동적으로 변하는 경우이다.

난이도가 너무 높으 task에 대해서는 모델이 학습하기 어려우며, 쉬운 문제부터 차근차근 배워서 향후 어려운 task에 대해서도 높은 성능 목표로 한다.

Trainer()는 그대로 유지한 상태에서 특정 조건을 만족할 때마다 environment 자체만 바꿔주면 된다.

1. Using modification of train

```
import ray
from ray import tune
from ray.rllib.agents.ppo import PPOTrainer

# === define curriculum ===
def train(config, reporter):
    trainer = PPOTrainer(config=config, env=YourEnv)
    while True:
        result = trainer.train()
        reporter(**result)
        if result["episode_reward_mean"] > 200:
            phase = 2
        elif result["episode_reward_mean"] > 100:
            phase = 1
        else:
            phase = 0
        trainer.workers.foreach_worker(
            lambda ev: ev.foreach_env(
                lambda env: env.set_phase(phase)))

num_gpus = 0
num_workers = 2

# === tune ===
ray.init()
tune.run(
    train,
    config={
        "num_gpus": num_gpus,
        "num_workers": num_workers,
```

```
    },
    resources_per_trial=tune.PlacementGroupFactory(
        [{"CPU": 1}, {"GPU": num_gpus}] + [{"CPU": 1}] * num_workers
    ),
)
```

2. Using Callback

Callback에서는 Train이 끝날 때마다 작동하는 [on_train_result()]함수가 있다.

이 함수를 변경해서 좀 더 쉽게 Curriculum Learning을 할 수 있다.

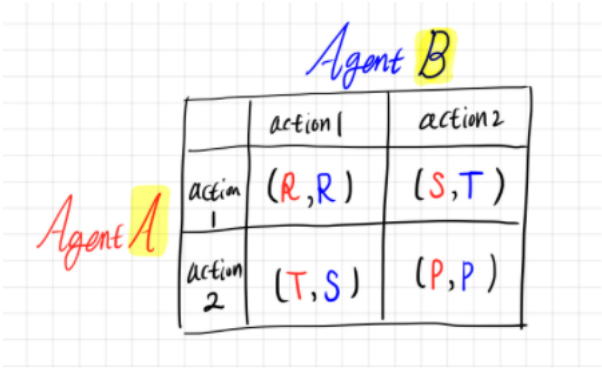
```
import ray
from ray import tune

def on_train_result(info):
    result = info["result"]
    if result["episode_reward_mean"] > 200:
        phase = 2
    elif result["episode_reward_mean"] > 100:
        phase = 1
    else:
        phase = 0
    trainer = info["trainer"]
    trainer.workers.foreach_worker(
        lambda ev: ev.foreach_env(
            lambda env: env.set_phase(phase)))

ray.init()
tune.run(
    "PPO",
    config={
        "env": YourEnv,
        "callbacks": {
            "on_train_result": on_train_result,
        },
    },
)
```

6. Multi Agent Two Step Game and MADDPG

Two Step Gamd



Actions

각각의 Agent는 0 또는 1을 선택할 수 있고, 결과는 state에 따라서 다르다.

State

State

state	global_reward	done
0	0	False(0,*)->state(1)(1,*)->state(2)
1	7	True
2	(0,1)->0(1,0)->8otherwise->1	True

>> (a,b)에서 [a] : agent_1's action, [b] : agent_2's action

Observations

각 Agent마다 Observation을 가지게 된다.

[np.concatenate([self.state,[agent_index]])로 구현되며,

[state2] 상태의 2번째 [Agent]라면, [0,0,1,2] 형태로 나온다.

[state0] 상태의 1번째 [Agent]라면, [1,0,0,1] 형태로 나온다.

구현에서는 [dict] 타입으로 observation을 반환하며, state를 따로 줄 수도 있다.

```
def _obs(self):
        if self.with_state:
            return {
                self.agent_1:{
                    "obs": self.agent_1_obs(),
                    ENV_STATE:self.state
                },
                self.agent_2:{
                    "obs":self.agent_2_obs(),
                    ENV_STATE:self.state
                }
            }
        else:
            return {
                self.agent_1 : self.agent_1_obs(),
                self.agent_2 : self.agent_2_obs()
            }

# === obsevation of each agent ===
def agent_1_obs(self):
        if self.one_hot_state_encoding:
            return np.concatenate([self.state, [1]])
        else:
            return np.flatnonzero(self.state)[0]
def agent_2_obs(self):
    if self.one_hot_state_encoding:
        return np.concatenate([self.state, [2]])
    else:
        return np.flatnonzero(self.state)[0] + 3
```

Rewards

Reward는 State에서 구한 Global Reward 값의 절반이다.

```
rewards = {
            self.agent_1: global_rew / 2.0,
            self.agent_2: global_rew / 2.0
        }
```

MADDPG with TwoStepGAme MultiAgent Environment

[rllib/contrib/maddpg]에는 이미 [MADDPG] 모델이 구현되어 있다.

Two Step Game 환경에서 모델을 학습시키자.

1. configuration

multi agent 환경이므로 [multiagent] 키를 추가해준다.

또한, [agent_id]에 대해서 policy를 mapping 해주는 [policy_mapping_fn]도 필요하다.

```
config = {
        "learning_starts": 100,
        "env_config": {
            "actions_are_logits": True,
        },
        "multiagent": {
            "policies": {
                "pol1": (None, Discrete(6), TwoStepGame.action_space, {"agent_id": 0}),
                "pol2": (None, Discrete(6), TwoStepGame.action_space, {"agent_id": 1}),
            },
            "policy_mapping_fn": lambda x: "pol1" if x == 0 else "pol2",
        },
        "framework": "torch" if args.torch else "tf",
        # Use GPUs iff `RLLIB_NUM_GPUS` env var set to > 0.
        "num_gpus": int(os.environ.get("RLLIB_NUM_GPUS", "0")),
    }
```

2. tune

config에 [env]를 추가해준다.

위의 코드에서 [env: ~~]로 추가할 수도 있다.

여기서는 dictionary에 새로운 원소를 추가하는 부분이라서 그대로 사용했다.

```
ray.init(num_cpus=args.num_cpus or None)
    stop = {
        "episode_reward_mean": args.stop_reward,
        "timesteps_total": args.stop_timesteps,
    }
    config = dict(config, **{"env": TwoStepGame})

    # === train ===
    results = tune.run("contrib/MADDPG", stop=stop, config=config, verbose=1)
```

만일 제대로 동작하지 않는다면
[pip3 install tensorflow-probability==0.7.0]
으로 [tfp]를 설치해야 한다.

Appendix

1. 정리

(a). MADDPG의 Output은 Action에 대한 logit값이다.

(b). [Dict.get(”name”,False)는 “name”에 대한 값을 가져오며
키가 없다면, 두 번째 파라미터를 반환한다.

(c). Constant 값 [ENV_STATE]는 “state”값이다.

(d). state가 같이 있다면, [observation_space]는 [dict] 타입이고,
아니면, [gym.space.MultiDiscrete] 타입이다.

(e). np.flatnonzero(self.state)[0] + 3
give the index of nonzero parts

```
np.flatnonzero([0,0,100,200])
# [2, 3]
```

2. MADDPG full code

```
"""The two-step game from QMIX: https://arxiv.org/pdf/1803.11485.pdf

Configurations you can try:
    - normal policy gradients (PG)
    - contrib/MADDPG
    - QMIX

See also: centralized_critic.py for centralized critic PPO on this game.
"""
import argparse
from gym.spaces import Tuple, MultiDiscrete, Dict, Discrete
import os

import ray
from ray import tune
from ray.tune import register_env, grid_search
from ray.rllib.env.multi_agent_env import ENV_STATE
from ray.rllib.examples.env.two_step_game import TwoStepGame
from ray.rllib.utils.test_utils import check_learning_achieved

parser = argparse.ArgumentParser()
parser.add_argument("--num-cpus", type=int, default=0)
parser.add_argument("--as-test", action="store_true")
parser.add_argument("--torch", action="store_true")
parser.add_argument("--stop-reward", type=float, default=7.0)
parser.add_argument("--stop-timesteps", type=int, default=50000)

if __name__ == "__main__":
    args = parser.parse_args()
    config = {
        "learning_starts": 100,
        "env_config": {
            "actions_are_logits": True,
        },
        "multiagent": {
            "policies": {
                "pol1": (None, Discrete(6), TwoStepGame.action_space, {"agent_id": 0}),
                "pol2": (None, Discrete(6), TwoStepGame.action_space, {"agent_id": 1}),
            },
            "policy_mapping_fn": lambda x: "pol1" if x == 0 else "pol2",
        },
        "framework": "torch" if args.torch else "tf",
        # Use GPUs iff `RLLIB_NUM_GPUS` env var set to > 0.
        "num_gpus": int(os.environ.get("RLLIB_NUM_GPUS", "0")),
    }


    # === setup ===
    ray.init(num_cpus=args.num_cpus or None)
    stop = {
        "episode_reward_mean": args.stop_reward,
        "timesteps_total": args.stop_timesteps,
    }
    config = dict(config, **{"env": TwoStepGame})

    # === train ===
    results = tune.run("contrib/MADDPG", stop=stop, config=config, verbose=1)
    if args.as_test:
        check_learning_achieved(results, args.stop_reward)

    ray.shutdown()
```

RLlib 예제

MuJoCo and Atari benchmarks

출처 : https://github.com/ray-project/rl-experiments

간단한 예제

간단한 예제