EFFICIENT REINFORCEMENT LEARNING EXPERIMENTATION IN PYTORCH

Anonymous authors
Paper under double-blind review

1 SUPPLEMENTARY MATERIAL

The online documentation of the package can be found at: To be disclosed

1.1 EXPERIMENTS ON ATARI 2600 ENVIRONMENTS

We used the ensuing parameters, following Schulman et al. (2017). We also used the same CNN network architecture:

Table 1: PPO + GAE Agent - hyperparameters values.

PARAMETER	VALUE
ADAM LR	2.5E-4
CLIP-PARAM	0.15
GAMMA	0.99
FRAME-SKIP	4
FRAME-STACK	4
NUM-STEPS	128
NUM-MINI-BATCH	4
NUM-EPOCHS	3
ENTROPY-COEF	0.01
VALUE-LOSS-COEF	1.0
ENVIRONMENT VECTOR LENGTH	8
GAE-LAMBDA	0.95
MAX-GRAD-NORM	0.5

The experiment was executed with the following training script. During training, the learning rate and the PPO clipping parameter are decayed also following Schulman et al. (2017):

```
import sys
import ray
import numpy as np

from xxxxxxxxx import utils
from xxxxxxxxx import Learner
from xxxxxxxxx.scheme import Scheme
from xxxxxxxxx.agent.algorithms import PPO
from xxxxxxxxx.agent.env import VecEnv
from xxxxxxxxx.agent.storages import GAEBuffer
from xxxxxxxxx.agent.actors import OnPolicyActor, get_feature_extractor
from xxxxxxxxx.envs import atari_train_env_factory

seed = 1
env_id = "PongNoFrameskip-v4"
log_dir = "/tmp/atari/{}/{}".format(env_id, seed)
col_workers_communication = "synchronous"
grad_workers_communication = "synchronous"

if __name__ == "__main__":

    utils.cleanup_log_dir(log_dir)
    ray.init()

    # 1. Define Train Vector of Envs
    train_envs_factory, action_space, obs_space = \
        VecEnv.create_factory(
            env_fn=atari_train_env_factory,
            env_kwargs={
                "env_id": env_id,
```

```
                "frame_stack": 4,
                "seed": seed},
            vec_env_size=8, log_dir=log_dir,
            info_keywords=('rr', 'rrr', 'lives'))

    # 2. Define RL training algorithm
    algo_factory, algo_name = PPO.create_factory(
        lr=2.5e-4, num_epochs=3, clip_param=0.1,
        entropy_coef=0.01, value_loss_coef=1.0,
        max_grad_norm=0.5, num_mini_batch=4,
        gamma=0.99)

    # 3. Define RL Policy
    actor_factory = OnPolicyActor.create_factory(
        obs_space, action_space, algo_name,
        get_feature_extractor("CNN"))

    # 4. Define rollouts storage
    storage_factory = GAEBuffer.create_factory(
        size=128, gae_lambda=0.95)

    # 5. Define scheme
    params = {}

    # add core modules
    params.update({
        "algo_factory": algo_factory,
        "actor_factory": actor_factory,
        "storage_factory": storage_factory,
        "train_envs_factory": train_envs_factory})

    # add collection specs
    params.update({
        "num_col_workers": 1,
        "col_workers_communication":
            col_workers_communication,
        "col_workers_resources": {
            "num_cpus": 8, "num_gpus": 0.5}})

    # add gradient specs
    params.update({
        "num_grad_workers": 1,
        "grad_workers_communication":
            grad_workers_communication,
        "grad_workers_resources": {
            "num_cpus": 1.0, "num_gpus": 0.5}})

    scheme = Scheme(**params)

    # 6. Define learner
    learner = Learner(
        scheme, target_steps=10000000, log_dir=log_dir)

    # 7. Define train loop
    updates = 0
    iterations = 0

    # lr and clip parameters decay values
    total_updates = np.ceil((10000000 * 3 * 4) / (8 * 128))
    alpha = np.linspace(1.0, 0.0, int(total_updates))

    while not learner.done():

        learner.step()
        learner.print_info()

        if iterations % 100 == 0:
            save_name = learner.save_model()

        iterations += 1
        learner.update_algo_parameter(
            "lr", alpha[updates] * 2.5e-4)
        learner.update_algo_parameter(
            "clip_param", alpha[updates] * 0.1)
```

Listing 1: Training script Atari 2600

1.2 Experiments on PyBullet environments

We used the following parameters, hand-picked to perform well on the single-threaded regime. As a feature extractor, we used a multilayer perceptron with two hidden layers of 256 each neurons and ReLU activation function:

Table 2: SAC Agent - hyperparameters values.

PARAMETER	VALUE
ADAM LR Q NETWORKS	1E-3
ADAM LR POLICY	1E-4
ADAM LR ALPHA	1E-05
GAMMA	0.98
INITIAL ALPHA	0.2
ENVIRONMENT VECTOR LENGTH	1
POLIAK UPDATE	0.995
BUFFER SIZE	1500000
MINI-BATCH SIZE	256
START STEPS	10000
NUM UPDATES	32
UPDATEEVERY	128

We used the following training script:

```
import ray
import time
from xxxxxxxxx import utils
from xxxxxxxxx import Learner
from xxxxxxxxx.scheme import Scheme
from xxxxxxxxx.agent.algos import SAC
from xxxxxxxxx.agent.env import VecEnv
from xxxxxxxxx.agent.storages import ReplayBuffer
from xxxxxxxxx.envs import pybullet_train_env_factory
from xxxxxxxxx.agent.actors import OffPolicyActor, get_feature_extractor

seed = 1
max_time = 3600 * 8  # 8h
env_id = "AntBulletEnv-v0"
log_dir = "/tmp/pybullet/{}/{}".format(env_id, seed)
num_col_workers = 1
num_grad_workers = 1
col_workers_communication = "synchronous"
grad_workers_communication = "synchronous"

if __name__ == "__main__":

    utils.cleanup_log_dir(log_dir)
    ray.init()

    # Define Environment Vector
    train_envs_factory, action_space, obs_space = \
        VecEnv.create_factory(
            vec_env_size=1,
            log_dir=log_dir,
            env_fn=pybullet_train_env_factory,
            env_kwargs={"env_id": env_id, "seed": seed})

    # Define RL training algorithm
    algo_factory, algo_name = SAC.create_factory(
        lr_pi=0.0001, lr_q=0.001, lr_alpha=1e-05,
        initial_alpha=0.1, gamma=0.98,
        polyak=0.995, num_updates=8,
        update_every=128, start_steps=5000,
        mini_batch_size=400)

    # Define RL Actor
    actor_factory = OffPolicyActor.create_factory(
        obs_space, action_space, algo_name,
        get_feature_extractor("MLP"))

    # Define rollouts storage
    storage_factory = ReplayBuffer.create_factory(
        size=15000000)

    # Core components params
    scheme_parameters = {
        "algo_factory": algo_factory,
        "actor_factory": actor_factory,
        "storage_factory": storage_factory,
        "train_envs_factory": train_envs_factory}

    # Collection operation params
    scheme_parameters.update({
        "num_col_workers": num_col_workers,
        "col_workers_communication":
            col_workers_communication})

```

```
    # Gradient computation operation params
    scheme_parameters.update({
        "num_grad_workers": num_grad_workers,
        "grad_workers_communication":
            grad_workers_communication})

    scheme = Scheme(**scheme_parameters)

    # 6. Define learner
    learner = Learner(
        scheme, target_steps=10000000, log_dir=log_dir)

    # 7. Define train loop
    iterations = 0
    start_time = time.time()
    while not learner.done():
        learner.step()
        if iterations % 1 == 0:
            learner.print_info()
        if iterations % 250000 == 0:
            learner.save_model()
        if max_time != -1 and (
                time.time() - start_time) > max_time:
            break
        iterations += 1
```

Listing 2: Training script PyBullet

1.3 Experiments on Obstacle Tower Unity3D Challenge environment

Table 3: Classification accuracies for naive Bayes and flexible Bayes on various data sets.

Parameter	Value
Adam lr	4e-4
clip-param	0.15
gamma	0.99
frame-skip	2
frame-stack	4
num-steps	800
num-mini-batch	6
num-epochs	4
entropy-coef	0.01
value-loss-coef	0.2
num-workers	1
environment vector length	16
gae-lambda	0.95
max-grad-norm	0.5
recurrent network	True
recurrent hidden size	256

We use the network architecture proposed in Espeholt et al. (2018) but we initialize its weights according to Fixup Zhang et al. (2019). We end our network with a gated recurrent unit (GRU) Cho et al. (2014) with a hidden layer of size 256 neurons. Gradients are computed using Adam optimizer Kingma & Ba (2014), with a starting learning rate of 4e-4 decayed by a factor of 0.25 both after 100 million steps and 400 million steps. The value coefficient, the entropy coefficient and the clipping parameters of the PPO algorithm are set to 0.2, 0.01 and 0.15 respectively. We use a discount factor gamma of 0.99. Furthermore, the horizon parameters is set to 800 steps and rollout collections are parallelized using environment vectors of size 16. Gradients are computed in minibatches of size 1600 for 2 epochs. Finally we also use generalized advantage estimation Schulman et al. (2015) with lambda 0.95. We use frame skip 2 and frame stack 4. We restart each new episode at a randomly selected floor between 0 and the higher floor reached in the previous episode.

We reshape the reward received from the environment in order to increase training efficiency. The reward upon the agent completing a floor is $+1$, and $+0.1$ is provided for opening doors, solving puzzles, or picking up keys. We additionally reward the agent with an extra $+1$ to pick up keys, $+0.002$ to detect boxes, $+0.001$ to find box intended locations, $+1.5$ to place the boxes target locations and +0.002 for any event that increases remaining time. We also reduce the action set from

the initial 54 actions to 6 (rotate camera in both directions, go forward, and go forward while turning left, turning right or jumping).

We used the following training script:

```
import time
from xxxxxxxxx import Learner
from xxxxxxxxx.scheme import Scheme
from xxxxxxxxx.agent.algorithms import PPO
from xxxxxxxxx.agent.env import VecEnv
from xxxxxxxxx.agent.storages import GAEBuffer
from xxxxxxxxx.envs import obstacle_train_env_factory
from xxxxxxxxx.agent.actors import OnPolicyActor, \
    get_feature_extractor

# Define Train Vector of Envs
envs_factory, action_space, obs_space = \
    VecEnv.create_factory(
        env_fn=obstacle_train_env_factory,
        env_kwargs={"frame_skip": 2, "frame_stack": 4},
        log_dir='/tmp/obstacle_tower_agent',
        vec_env_size=8, info_keywords=('floor', 'start', 'seed'))

# Define RL training algorithm
algo_factory, algo_name = PPO.create_factory(
    lr=2.5e-5, num_epochs=2, clip_param=0.15,
    entropy_coef=0.01, value_loss_coef=0.2,
    max_grad_norm=0.5, num_mini_batch=8,
    use_clipped_value_loss=True, gamma=0.99)

# Define RL Policy
actor_factory = OnPolicyActor.create_factory(
    obs_space, action_space, algo_name,
    get_feature_extractor("Fixup"),
    recurrent_policy=True,
    recurrent_hidden_size=256)

# Define rollouts storage
storage_factory = GAEBuffer.create_factory(
    size=800, gae_lambda=0.95)

# 6. Define scheme
params = {}

# add core modules
params.update({
    "algo_factory": algo_factory,
    "actor_factory": actor_factory,
    "storage_factory": storage_factory,
    "train_envs_factory": envs_factory})

# add collection specs
params.update({
    "num_col_workers": 4,
    "col_communication": "asynchronous",
    "col_worker_resources": {
        "num_cpus": 1, "num_gpus": 0.5,
        "object_store_memory": 2.5 * 1024 ** 3,
        "memory": 2.5 * 1024 ** 3}})

# add gradient specs
# num_cpus is equal to 32 - 4 to force ray to locate
# 4 collection workers and 1 gradient worker in each
# machine with 32 cpus
params.update({
    "num_grad_workers": 2,
    "grad_communication": "synchronous",
    "grad_worker_resources": {
        "num_cpus": 32 - 4,
        "num_gpus": 1.0,
        "object_store_memory": 2.5 * 1024 ** 3,
        "memory": 2.5 * 1024 ** 3}})

scheme = Scheme(**params)

# 7. Define learner
learner = Learner(
    scheme, target_steps=600000000,
    log_dir='/tmp/obstacle_tower_agent')

# 8. Define train loop
iterations = 0
num_lr_updates = 0
start_time = time.time()
while not learner.done():

    learner.step()
    learner.print_info()

    if iterations % 100 == 0:
        save_name = learner.save_model()
```

```
    if learner.num_samples_collected > 100000000 and num_lr_updates == 0:
        learner.update_algo_parameter("lr", 4e-4 * 0.25)
        num_lr_updates += 1

    elif learner.num_samples_collected > 400000000 and num_lr_updates == 1:
        learner.update_algo_parameter("lr", 4e-4 * 0.25 ** 2)
        num_lr_updates += 1

    iterations += 1
```

Listing 3: Training script Obstacle Tower Unity3D Challenge

1.4 SCHEME CONFIGURATION OF POPULAR RL AGENTS

Table 4 shows the architecture configurations used by some popular RL Agents with which we have experimented in this work. All these architectures can be configured via our *Scheme* class.

Table 4: *Scheme* configurations to match the underlying training architectures of some popular RL Agents.

AGENT ARCHITECTURE	NUMBER OF GRADIENT WORKERS	NUMBER OF COLLECTION WORKER (PER EACH GRADIENT WORKER)	GRADIENT WORKERS COMMUNICATION	COLLECTION WORKERS COMMUNICATION
SINGLE THREADED	1	1	SYNCHRONOUS	SYNCHRONOUS
DPPO	≥ 1	1	SYNCHRONOUS	SYNCHRONOUS
APPO	≥ 1	1	AYNCHRONOUS	SYNCHRONOUS
IMPALA / APEX	1	≥ 1	SYNCHRONOUS	ASYNCHRONOUS
RAPID	≥ 1	≥ 1	SYNCHRONOUS	ASYNCHRONOUS
ASYNC RAPID	≥ 1	≥ 1	ASYNCHRONOUS	ASYNCHRONOUS

REFERENCES

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.