ElegantRL Documentation

Release 0.3.1

ElegantRL

Aug 17, 2023

CONTENTS

ElegantRL is an open-source massively parallel library for deep reinforcement learning (DRL) algorithms, implemented in PyTorch. We aim to provide a *next-generation* framework that leverage recent techniques, e.g., massively parallel simulations, ensemble methods, population-based training, and showcase exciting scientific discoveries.

ElegantRL features strong **scalability**, **elasticity** and **lightweightness**, and allows users to conduct **efficient** training on either one GPU or hundreds of GPUs:

- **Scalability**: ElegantRL fully exploits the parallelism of DRL algorithms at multiple levels, making it easily scale out to hundreds or thousands of computing nodes on a cloud platform, say, a SuperPOD platform with thousands of GPUs.
- **Elasticity**: ElegantRL can elastically allocate computing resources on the cloud, which helps adapt to available resources and prevents over/under-provisioning/under-provisioning.
- **Lightweightness**: The core codes <1,000 lines (check elegantrl_helloworld).
- **Efficient**: in many testing cases, it is more efficient than Ray RLlib.

ElegantRL implements the following DRL algorithms:

- **DDPG, TD3, SAC, A2C, PPO, REDQ for continuous actions**
- **DQN, DoubleDQN, D3QN, PPO-Discrete for discrete actions**
- **QMIX, VDN; MADDPG, MAPPO, MATD3 for multi-agent RL**

For beginners, we maintain ElegantRL-HelloWorld as a tutorial. It is a lightweight version of ElegantRL with <1,000 lines of core codes. More details are available here.

CHAPTER

ONE

INSTALLATION

ElegantRL generally requires:

- Python>=3.6
- PyTorch>=1.0.2
- gym, matplotlib, numpy, pybullet, torch, opencv-python, box2d-py.

You can simply install ElegantRL from PyPI with the following command:

```
pip3 install erl --upgrade
```

Or install with the newest version through GitHub:

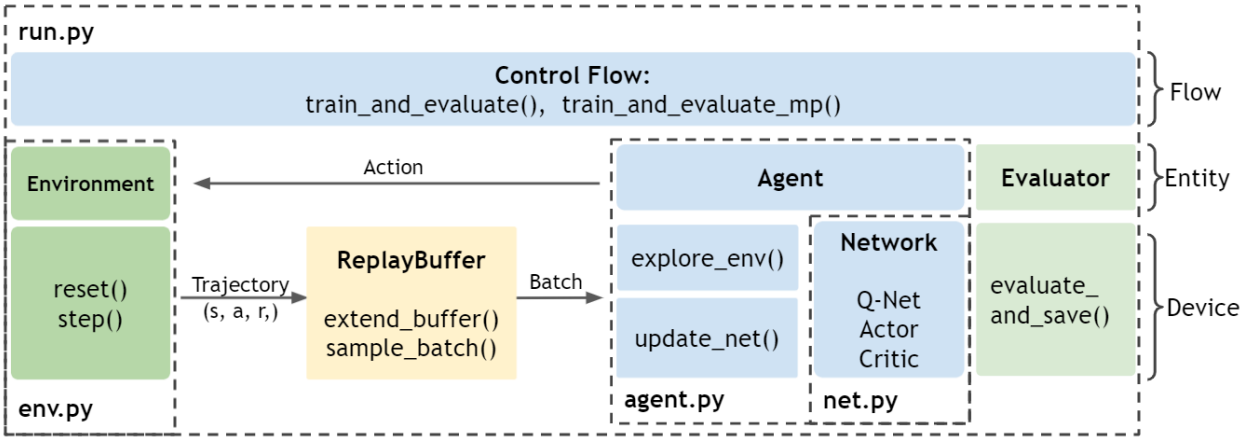
```
git clone https://github.com/AI4Finance-Foundation/ElegantRL.git
cd ElegantRL
pip3 install .
```

1.1 Hello, World!

We will help you understand DRL and get hands-on experiences through ElegantRL-HelloWorld.

Table of Contents

1.1.1 File Structure ("Net-Agent-Env-Run")



One sentence summary: an agent (*agent.py*) with Actor-Critic networks (*net.py*) is trained (*run.py*) by interacting with an environment (*env.py*).

As a high-level overview, the file structure is as follows. Initialize an environment from *env.py* and an agent from *agent.py*. The agent is constructed with Actor and Critic networks from *net.py*. In each training step from *run.py*, the agent interacts with the environment, generating transitions that are stored into a Replay Buffer. Then, the agent fetches transitions from the Replay Buffer to train its networks. After each update, an evaluator evaluates the agent's performance and saves the agent if the performance is good.

net.py

Our net.py contains three types of networks. Each type of networks includes a base network for inheritance and a set of variations for algorithms.

- Q-Net
- Actor Network
- Critic Network

agent.py

agent.py contains classes of different DRL agent, where each agent corresponds to a DRL algorithms. In addition, it also contains the Replay Buffer class for data storage.

In this HelloWorld, we focus on DQN, SAC, and PPO, which are the most representative and commonly used DRL algorithms.

For a complete list of DRL algorithms, please go to here.

env.py

env.py contains a wrapper class that preprocesses the Gym-styled environment (env).

Refer to OpenAI's explanation to better understand the how a Gym-styled environment is formulated.

run.py

run.py contains basic functions for the training and evaluating process. In the training process `train_and_evaluate`, there are two major steps:

1. Initialization:
    - hyper-parameters args.
    - env = PreprocessEnv() : creates an environment (in the OpenAI gym format).
    - agent = agent.XXX() : creates an agent for a DRL algorithm.
    - evaluator = Evaluator() : evaluates and stores the trained model.
    - buffer = ReplayBuffer() : stores the transitions.
2. Then, the training process is controlled by a while-loop:
    - agent.explore_env(...): the agent explores the environment within target steps, generates transitions, and stores them into the ReplayBuffer.
    - agent.update_net(...): the agent uses a batch from the ReplayBuffer to update the network parameters.
    - evaluator.evaluate_save(...): evaluates the agent's performance and keeps the trained model with the highest score.

The while-loop will terminate when the conditions are met, e.g., achieving a target score, maximum steps, or manual breaks.

In **run.py**, we also provide an evaluator to periodically evaluate and save the model.

demo.py

demo.py contains four demo functions:

- discrete action + off-policy algorithm
- discrete action + on-policy algorithm
- continuous action + off-policy algorithm
- continuous action + on-policy algorithm

1.1.2 Run the Code

In demo.py, there are four functions that are available to run in the main function. You can see `demo_continuous_action_on_policy()` called at the bottom of the file.

```
if __name__ == '__main__':

    ENV_ID = 3  # int(sys.argv[2])
    # demo_continuous_action_off_policy()
```

(continues on next page)

(continued from previous page)

```
    demo_continuous_action_on_policy()
    # demo_discrete_action_off_policy()
    # demo_discrete_action_on_policy()
```

Inside each of the four functions, we provide three tasks as demos to help you get start. You can choose the task you want to train on by setting the `env_id`.

- Pendulum id: 1
- LunarLanderContinuous-v2 id: 2
- BipedalWalker-v3 id: 3

If everything works, congratulations!

Enjoy your journey to the DRL world with ElegantRL!

1.2 Networks: *net.py*

In ElegantRL, there are three basic network classes: Q-net, Actor, and Critic. Here, we list several examples, which are the networks used by DQN, SAC, and PPO algorithms.

The full list of networks are available here

1.2.1 Q Net

1.2.2 Actor Network

1.2.3 Critic Network

1.3 Agents: *agent.py*

In this HelloWorld, we focus on DQN, SAC, and PPO, which are the most representative and commonly used DRL algorithms.

1.3.1 Agents

1.3.2 Replay Buffer

1.4 Environment: *env.py*

1.5 Main: *run.py*

1.5.1 Hyper-parameters

1.5.2 Train and Evaluate

1.5.3 Evaluator

1.6 Quickstart

As a quickstart, we select the Pendulum task from the demo.py to show how to train a DRL agent in ElegantRL.

1.6.1 Step 1: Import packages

```
from elegantrl_helloworld.demo import *

gym.logger.set_level(40) # Block warning
```

1.6.2 Step 2: Specify Agent and Environment

```
env = PendulumEnv('Pendulum-v0', target_return=-500)
args = Arguments(AgentSAC, env)
```

1.6.3 Part 3: Specify Hyper-parameters

```
args.reward_scale = 2 ** -1  # RewardRange: -1800 < -200 < -50 < 0
args.gamma = 0.97
args.target_step = args.max_step * 2
args.eval_times = 2 ** 3
```

1.6.4 Step 4: Train and Evaluate the Agent

```
train_and_evaluate(args)
```

Try by yourself through this Colab!

Tip:

- By default, it will train a stable-SAC agent in the Pendulum-v0 environment for 400 seconds.
- It will choose to utilize CPUs or GPUs automatically. Don't worry, we never use `.cuda()`.
- It will save the log and model parameters file in `'./{Environment}_{Agent}_{GPU_ID}'`.
- It will print the total reward while training. (Maybe we should use TensorBoardX?)
- The code is heavily commented. We believe these comments can answer some of your questions.

1.7 Key Concepts and Features

One sentence summary: in deep reinforcement learning (DRL), an agent learns by continuously interacting with an unknown environment, in a trial-and-error manner, making sequential decisions under uncertainty and achieving a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The lifecycle of a DRL application consists of three stages: *simulation*, *learning*, and *deployment*. Our goal is to leverage massive computing power to address three major challenges existed in these three stages:

- simulation speed bottleneck;
- sensitivity to hyper-parameters;
- unstable generalization ability.

ElegantRL is a massively parallel framework for cloud-native DRL applications implemented in PyTorch:

- We embrace the accessibility of cloud computing platforms and follow a cloud-native paradigm in the form of containerization, microservices, and orchestration, to ensure fast and robust execution on a cloud.
- We fully exploit the parallelism of DRL algorithms at multiple levels, namely the worker/learner parallelism within a container, the pipeline parallelism (asynchronous execution) over multiple microservices, and the inherent parallelism of the scheduling task at an orchestrator.
- We take advantage of recent technology breakthroughs in massively parallel simulation, population-based training that implicitly searches for optimal hyperparameters, and ensemble methods for variance reduction.

ElegantRL features strong scalability, elasticity and stability and allows practitioners to conduct efficient training from one GPU to hundreds of GPUs on a cloud:

Scalable: the multi-level parallelism results in high scalability. One can train a population with hundreds of agents, where each agent employs thousands of workers and tens of learners. Therefore, ElegantRL can easily scale out to a cloud with hundreds or thousands of nodes.

Elastic: ElegantRL features strong elasticity on the cloud. The resource allocation can be made according to the numbers of workers, learners, and agents and the unit resource assigned to each of them. We allow a flexible adaptation to meet the dynamic resource availability on the cloud or the demands of practitioners.

Stable: With the massively computing power of a cloud, ensemble methods and population-based training will greatly improve the stability of DRL algorithms. Furthermore, ElegantRL leverages computing resource to implement the

Hamiltonian-term as an add-on regularization to model-free DRL algorithms. Such an add-on H-term utilizes computing power (can be computed in parallel on GPU) to search for the "minimum-energy state", corresponding to the stable state of a system. Altogether, ElegantRL demonstrates a much more stable performance compared to Stable-Baseline3, a popular DRL library devote to stability.

Accessible: ElegantRL is a highly modularized framework and maintains ElegantRL-HelloWorld for beginners to get started. We also help users overcome the learning curve by providing API documentations, Colab tutorials, frequently asked questions (FAQs), and demos, e.g., on OpenAI Gym, MuJoCo, Isaac Gym.

1.8 Cloud-native Paradigm

To the best of our knowledge, ElegantRL is the first open-source cloud-native framework that supports millions of GPU cores to carry out massively parallel DRL training at multiple levels.

In this article, we will discuss our motivation and cloud-native designs.

1.8.1 Why cloud-native?

When you need more computing power and storage for your task, running on a cloud may be a more preferable choice than buying racks of machines. Due to its accessible and automated nature, the cloud has been a disruptive force in many deep learning tasks, such as natural langauge processing, image recognition, video synthesis, etc.

Therefore, we embrace the cloud computing platforms to:

- build a serverless application framework that performs the entire life-cycle (simulate-learn-deploy) of DRL applications on low-cost cloud computing power.
- support for single-click training for sophisticated DRL problems (compute-intensive and time-consuming) with automatic hyper-parameter tuning.
- provide off-the-shelf APIs to free users from full-stack development and machine learning implementations, e.g., DRL algorithms, ensemble methods, performance analysis.

Our goal is to allow for wider DRL applications and faster development life cycles that can be created by smaller teams. One simple example of this is the following workflow.

A user wants to train a trading agent using minute-level NASDAQ 100 constituent stock dataset, a compute-intensive task as the dimensions of the dataset increase, e.g., the number of stocks, the length of period, the number of features. Once the user finishes constructing the environment/simulator, she can directly submit the job to our framework. Say the user has no idea which DRL algorithms she should use and how to setup the hyper-parameters, the framework can automatically initialize agents with different algorithms and hyper-parameter to search the best combination. All data is stored in the cloud storage and the computing is parallized on cloud clusters.

1.8.2 A cloud-native solution

ElegantRL follows the cloud-native paradigm in the form of microservice, containerization, and orchestration.

Microservices: ElegantRL organizes a DRL agent as a collection of microservices, including orchestrator, worker, learner, evaluator, etc. Each microservice has specialized functionality and connects to other microservices through clear-cut APIs. The microservice structure makes ElegantRL a highly modularized framework and allows practitioners to use and customize without understanding its every detail.

Containerization: An agent is encapsulated into a pod (the basic deployable object in Kubernetes (K8s)), while each microservice within the agent is mapped to a container (a lightweight and portable package of software). On the cloud, microservice and containerization together offer significant benefits in asynchronous parallelism, fault isolation, and security.

Orchestration: ElegantRL employs K8s to orchestrate pods and containers, which automates the deployment and management of the DRL application on the cloud. Our goal is to free developers and practitioners from sophisticated distributed machine learning.

We provide two different scheduling mechanism on the cloud, namely generational evolution and tournament-based evolution.

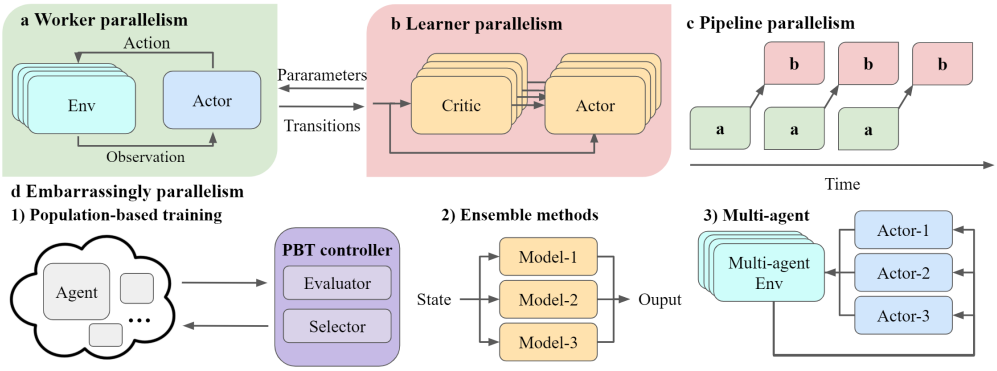
A tutorial on generational evolution is available here.

A tutorial on tournament-based evolution is available here.

1.9 Muti-level Parallelism

ElegantRL is a massively parallel framework for DRL algorithms. In this article, we will explain how we map the multi-level parallelism of DRL algorithms to a cloud, namely the worker/learner parallelism within a container, the pipeline parallelism (asynchronous execution) over multiple microservices, and the inherent parallelism of the scheduling task at an orchestrator.

Here, we follow a *bottom-up* approach to describe the parallelism at multiple levels.



An overview of the multi-level parallelism supported by ElegantRL. ElegantRL decomposes an agent into worker (a) and learner (b) and pipes their executions through the pipeline parallelism (c). Besides, ElegantRL emphasizes three types of inherent parallelism in DRL algorithms, including population-based training (PBT) (d1), ensemble methods (d2), and multi-agent DRL (d3).

1.9.1 Worker/Learner parallelism

ElegantRL adopts a worker-learner decomposition of a single agent, decoupling the data sampling process and model learning process. We exploit both the worker parallelism and learner parallelism.

Worker parallelism: a worker generates transitions from interactions of an actor with an environment. As shown in the figure a, ElegantRL supports the recent breakthrough technology, *massively parallel simulation*, with a simulation speedup of 2 ~ 3 orders of magnitude. One GPU can simulate the interactions of one actor with thousands of environments, while existing libraries achieve parallel simulation on hundreds of CPUs.

Advantage of massively parallel simulation:

- Running thousands of parallel simulations, since the manycore GPU architecture is natually suited for parallel simulations.
- Speeding up the matrix computations of each simulation using GPU tensor cores.
- Reducing the communication overhead by bypassing the bottleneck between CPUs and GPUs.

- Maximizing GPU utilization.

To achieve massively parallel simulation, ElegantRL supports both user-customized and imported simulator, namely Issac Gym from NVIDIA.

A tutorial on how to create a GPU-accelerated VecEnv is available here.

A tutorial on how to utilize Isaac Gym as an imported massively parallel simulator is available here.

Note: Besides massively parallel simulation on GPUs, we allow users to conduct worker parallelism on classic environments through multiprocessing, e.g., OpenAI Gym and MuJoCo.

Learner parallelism: a learner fetches a batch of transitions to train neural networks, e.g., a critic net and an actor net in the figure b. Multiple critic nets and actor nets of an ensemble method can be trained simultaneously on one GPU. It is different from other libraries that achieve parallel training on multiple CPUs via distributed SGD.

1.9.2 Pipeline parallelism

We view the worker-learner interaction as a *producer-consumer* model: a worker produces transitions and a learner consumes. As shown in figure c, ElegantRL pipelines the execution of workers and learners, allowing them to run on one GPU asynchronously. We exploit pipeline parallelism in our implementations of off-policy model-free algorithms, including DDPG, TD3, SAC, etc.

1.9.3 Inherent parallelism

ElegantRL supports three types of inherent parallelism in DRL algorithms, including *population-based training*, *ensemble methods*, and *multi-agent DRL*. Each features strong independence and requires little or no communication.

- Population-based training (PBT): it trains hundreds of agents and obtains a powerful agent, e.g., generational evolution and tournament-based evolution. As shown in figure d1, an agent is encapsulated into a pod on the cloud, whose training is orchestrated by the evaluator and selector of a PBT controller. Population-based training implicitly achieves massively parallel hyper-parameter tuning.
- Ensemble methods: it combines the predictions of multiple models and obtains a better result than each individual result, as shown in figure d2. ElegantRL implements various ensemble methods that perform remarkably well in the following scenarios:
  1. take an average of multiple critic nets to reduce the variance in the estimation of Q-value;
  2. perform a minimization over multiple critic nets to reduce over-estimation bias;
  3. optimize hyper-parameters by initializing agents in a population with different hyper-parameters.
- Multi-agent DRL: in the cooperative, competitive, or mixed settings of MARL, multiple parallel agents interact with the same environment. During the training process, there is little communication among those parallel agents.

1.10 Example 1: LunarLanderContinuous-v2

LunarLanderContinuous-v2 is a robotic control task. The goal is to get a Lander to rest on the landing pad. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Detailed description of the task can be found at OpenAI Gym. Our Python code is available here.

When a Lander takes random actions:

1.10.1 Step 1: Install ElegantRL

```
pip install git+https://github.com/AI4Finance-LLC/ElegantRL.git
```

1.10.2 Step 2: Import packages

- ElegantRL
- OpenAI Gym: a toolkit for developing and comparing reinforcement learning algorithms (collections of environments).

```
from elegantrl.run import *

gym.logger.set_level(40) # Block warning
```

1.10.3 Step 3: Get environment information

```
get_gym_env_args(gym.make('LunarLanderContinuous-v2'), if_print=True)
```

Output:

```
env_args = {
    'env_num': 1,
    'env_name': 'LunarLanderContinuous-v2',
    'max_step': 1000,
    'state_dim': 8,
    'action_dim': 4,
    'if_discrete': True,
    'target_return': 200,
    'id': 'LunarLanderContinuous-v2'
}
```

1.10.4 Step 4: Initialize agent and environment

- agent: chooses a agent (DRL algorithm) from a set of agents in the directory.
- env_func: the function to create an environment, in this case, we use `gym.make` to create LunarLanderContinuous-v2.
- env_args: the environment information.

```
env_func = gym.make
env_args = {
    'env_num': 1,
    'env_name': 'LunarLanderContinuous-v2',
    'max_step': 1000,
    'state_dim': 8,
    'action_dim': 4,
    'if_discrete': True,
    'target_return': 200,
    'id': 'LunarLanderContinuous-v2'
}

args = Arguments(AgentModSAC, env_func=env_func, env_args=env_args)
```

1.10.5 Step 5: Specify hyper-parameters

A list of hyper-parameters is available here.

```
args.target_step = args.max_step
args.gamma = 0.99
args.eval_times = 2 ** 5
```

1.10.6 Step 6: Train your agent

In this tutorial, we provide a single-process demo to train an agent:

```
train_and_evaluate(args)
```

Try by yourself through this Colab!

Performance of a trained agent:

1.11 Example 2: BipedalWalker-v3

BipedalWalker-v3 is a classic task in robotics that performs a fundamental skill: moving forward as fast as possible. The goal is to get a 2D biped walker to walk through rough terrain. BipedalWalker is considered to be a difficult task in the continuous action space, and there are only a few RL implementations that can reach the target reward. Our Python code is available here.

When a biped walker takes random actions:

1.11.1 Step 1: Install ElegantRL

```
pip install git+https://github.com/AI4Finance-LLC/ElegantRL.git
```

1.11.2 Step 2: Import packages

- ElegantRL
- OpenAI Gym: a toolkit for developing and comparing reinforcement learning algorithms (collections of environments).

```
from elegantrl.run import *

gym.logger.set_level(40) # Block warning
```

1.11.3 Step 3: Get environment information

```
get_gym_env_args(gym.make('BipedalWalker-v3'), if_print=False)
```

Output:

```
env_args = {
    'env_num': 1,
    'env_name': 'BipedalWalker-v3',
    'max_step': 1600,
    'state_dim': 24,
    'action_dim': 4,
    'if_discrete': False,
    'target_return': 300,
}
```

1.11.4 Step 4: Initialize agent and environment

- agent: chooses a agent (DRL algorithm) from a set of agents in the directory.
- env_func: the function to create an environment, in this case, we use `gym.make` to create BipedalWalker-v3.
- env_args: the environment information.

```
env_func = gym.make
env_args = {
    'env_num': 1,
    'env_name': 'BipedalWalker-v3',
    'max_step': 1600,
    'state_dim': 24,
    'action_dim': 4,
    'if_discrete': False,
    'target_return': 300,
    'id': 'BipedalWalker-v3',
}
```

(continues on next page)

(continued from previous page)

```
args = Arguments(AgentPPO, env_func=env_func, env_args=env_args)
```

1.11.5 Step 5: Specify hyper-parameters

A list of hyper-parameters is available here.

```
args.target_step = args.max_step * 4
args.gamma = 0.98
args.eval_times = 2 ** 4
```

1.11.6 Step 6: Train your agent

In this tutorial, we provide four different modes to train an agent:

- **Single-process**: utilize one GPU for a single-process training. No parallelism.
- **Multi-process**: utilize one GPU for a multi-process training. Support worker and learner parallelism.
- **Multi-GPU**: utilize multi-GPUs to train an agent through model fusion. Specify the GPU ids you want to use.
- **Tournament-based ensemble training**: utilize multi-GPUs to run tournament-based ensemble training.

```
flag = 'SingleProcess'

if flag == 'SingleProcess':
    args.learner_gpus = 0
    train_and_evaluate(args)

elif flag == 'MultiProcess':
    args.learner_gpus = 0
    train_and_evaluate_mp(args)

elif flag == 'MultiGPU':
    args.learner_gpus = [0, 1, 2, 3]
    train_and_evaluate_mp(args)

elif flag == 'Tournament-based':
    args.learner_gpus = [[i, ] for i in range(4)]  # [[0,], [1, ], [2, ]] or [[0, 1], [2,
↪ 3]]
    python_path = '.../bin/python3'
    train_and_evaluate_mp(args, python_path)

else:
    raise ValueError(f"Unknown flag: {flag}")
```

Try by yourself through this Colab!

Performance of a trained agent:

Check out our **video** on bilibili: Crack the BipedalWalkerHardcore-v2 with total reward 310 using IntelAC.

1.12 How to create a VecEnv on GPUs

ElegantRL supports massively parallel simulation through GPU-accelerated VecEnv.

Here, we talk about how to create a VecEnv on GPUs from scratch and go through a simple chasing example, a deterministic environment with continuous actions and continuous state space. The goal is to move an agent to chase a randomly moving robot. The reward depends on the distance between agent and robot. The environment terminates when the agent catches the robot or the max step is reached.

To keep the example simple, we only use two packages, PyTorch and Numpy.

```
import torch
import numpy as np
```

Now, we start to create the environment, which usually includes initialization function, reset function, and step function.

For **initialization function**, we specify the number of environments `env_num`, the GPU id `device_id`, and the dimension of the chasing space `dim`. In the chasing environment, we keep track of positions and velocities of the agent and the robot.

```
class ChasingVecEnv:
    def __init__(self, dim=2, env_num=4096, device_id=0):
        self.dim = dim
        self.init_distance = 8.0

        # reset
        self.p0s = None  # position
        self.v0s = None  # velocity
        self.p1s = None
        self.v1s = None

        self.distances = None
        self.current_steps = None

        '''env info'''
        self.env_name = 'ChasingVecEnv'
        self.state_dim = self.dim * 4
        self.action_dim = self.dim
        self.max_step = 2 ** 10
        self.if_discrete = False
        self.target_return = 6.3

        self.env_num = env_num
        self.device = torch.device(f"cuda:{device_id}")
```

The second step is to implement a **reset function**. The reset function is called at the beginning of each episode and sets initial state to current state. To utilize GPUs, we use data structures for multi-dimensional tensors provided by the torch package.

```
def reset(self):
    self.p0s = torch.zeros((self.env_num, self.dim), dtype=torch.float32, device=self.
↪device)
    self.v0s = torch.zeros((self.env_num, self.dim), dtype=torch.float32, device=self.
↪device)
    self.p1s = torch.zeros((self.env_num, self.dim), dtype=torch.float32, device=self.
```

(continues on next page)

(continued from previous page)

```
↪device)
    self.v1s = torch.zeros((self.env_num, self.dim), dtype=torch.float32, device=self.
↪device)

    self.current_steps = np.zeros(self.env_num, dtype=np.int)

    for env_i in range(self.env_num):
        self.reset_env_i(env_i)

    self.distances = ((self.p0s - self.p1s) ** 2).sum(dim=1) ** 0.5

    return self.get_state()
```

The last function is the **step function**, that includes a transition function and a reward function, and signals the terminal state. To compute the transition function, we utilize mathematical operations from the torch package over the data (tensors). These operations allow us to compute transitions and rewards of thousands of environments in parallel.

Note: Unlike computing the transition function and reward function in parallel, we check the terminal state in a sequential way. Since sub-environments may terminate at different time steps, when a sub-environment is at terminal state, we have to reset it manually.

```
def step(self, action1s):
    '''transition function'''
    action0s = torch.rand(size=(self.env_num, self.dim), dtype=torch.float32,␣
↪device=self.device)
    action0s_l2 = (action0s ** 2).sum(dim=1, keepdim=True) ** 0.5
    action0s = action0s / action0s_l2.clamp_min(1.0)

    self.v0s *= 0.50
    self.v0s += action0s
    self.p0s += self.v0s * 0.01

    action1s_l2 = (action1s ** 2).sum(dim=1, keepdim=True) ** 0.5
    action1s = action1s / action1s_l2.clamp_min(1.0)

    self.v1s *= 0.75
    self.v1s += action1s
    self.p1s += self.v1s * 0.01

    '''reward function'''
    distances = ((self.p0s - self.p1s) ** 2).sum(dim=1) ** 0.5
    rewards = self.distances - distances - action1s_l2.squeeze(1) * 0.02
    self.distances = distances

    '''check terminal state'''
    self.steps += 1  # array
    dones = torch.zeros(self.env_num, dtype=torch.float32, device=self.device)
    for env_i in range(self.env_num):
        done = 0
        if distances[env_i] < 1:
            done = 1
```

(continues on next page)

(continued from previous page)

```
            rewards[env_i] += self.init_distance
        elif self.steps[env_i] == self.max_step:
            done = 1

        if done:
            self.reset_env_i(env_i)
        dones[env_i] = done

    '''next_state'''
    next_states = self.get_state()
    return next_states, rewards, dones, None
```

For more information about the chasing environment, we provide a Colab version to play with, and its code can be found here.

1.13 How to run worker parallelism: Isaac Gym

In the previous tutorial, we present how to create a GPU-accelerated VecEnv that takes a batch of actions and returns a batch of transitions for every step.

Besides the user-customized VecEnv, ElegantRL supports external VecEnv, e.g., NVIDIA Isaac Gym. In this tutorial, we select Isaac Gym as an example to show how to utilize such a VecEnv to realize the massively parallel simulation (worker parallelism) in ElegantRL.

1.13.1 What is NVIDIA Isaac Gym?

NVIDIA Isaac Gym is NVIDIA's physics simulation environment for reinforcement learning research, an end-to-end high performance robotics simulation platform. It leverages NVIDIA PhysX to provide a GPU-accelerated simulation back-end and enables thousands of environments to run in parallel on a single workstation, achieving 2-3 orders of magnitude of training speed-up in continuous control tasks.

Features:

- Implementation of multiple highly complex robotic manipulation environments which can be simulated at hundreds of thousands of steps per second on a single GPU.
- High-fidelity GPU-accelerated robotics simulation with a variety of environment sensors - position, velocity, force, torque, etc.
- A Tensor API in Python providing direct access to physics buffers by wrapping them into PyTorch tensors without going through any CPU bottlenecks.
- Support for importing URDF and MJCF files with automatic convex decomposition of imported 3D meshes for physical simulation.

Here is a visualization of humanoid in Isaac Gym:

For more information, please view its recently released paper at https://arxiv.org/abs/2108.10470.

To install Isaac Gym, please follow the instructions at https://developer.nvidia.com/isaac-gym.

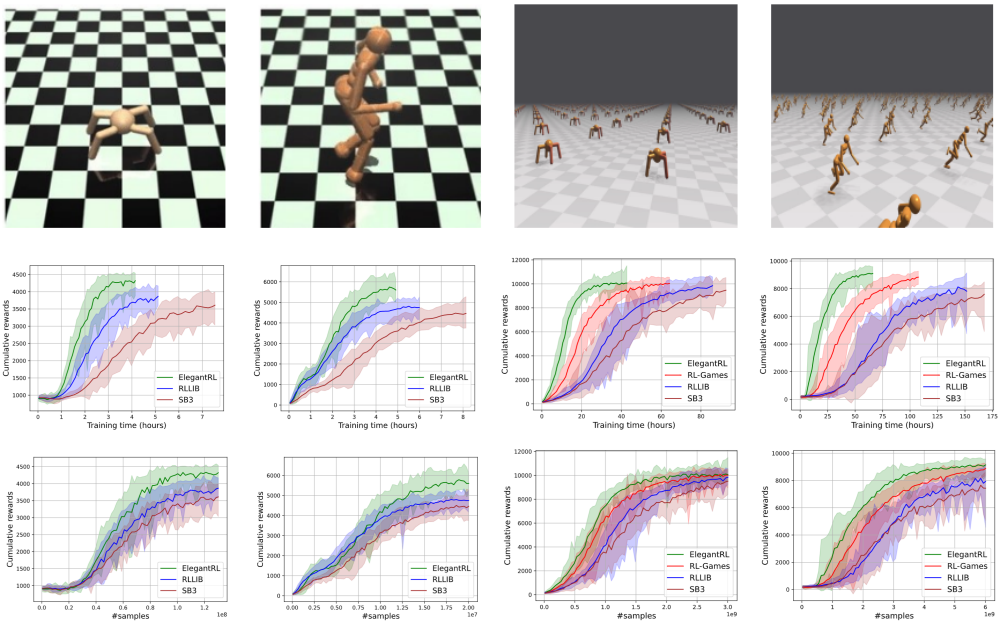
1.13.2 Experiments on Ant and Humanoid

Ant and humanoid are two canonical robotic control tasks that simulate an ant and a humanoid, respectively, where each task has both MuJoCo version and Isaac Gym version. The ant task is a simple environment to simulate due to its stability in the initial state, while the humanoid task is often used as a testbed for locomotion learning. Even though the implementations of MuJoCo and Isaac Gym are slightly different, the objective of both is to have the agent move forward as fast as possible.

On one DGX-2 server, we compare ElegantRL-podracer with RLlib, since both support multiple GPUs. ElegantRL-podracer used PPO from ElegantRL, while in RLlib we used the Decentralized Distributed Proximal Policy Optimization (DD-PPO) algorithm that scales well to multiple GPUs. For fair comparison, we keep all adjustable parameters and computing resources the same, such as the depth and width of neural networks, total training steps/time, number of workers, and GPU and CPU resources. Specifically, we use a batch size of 1024, learning rate of 0.001, and a replay buffer size of 4096 across tasks.

We employ two different metrics to evaluate the agent's performance:

1. Episodic reward vs. training time (wall-clock time): we measure the episodic reward at different training time, which can be affected by the convergence speed, communication overhead, scheduling efficiency, etc.
2. Episodic reward vs. #samples: from the same testings, we also measure the episodic reward at different number of samples. This result can be used to investigate the massive parallel simulation capability of GPUs, and also check the algorithm's performance.



1.13.3 Running NVIDIA Isaac Gym in ElegantRL

ElegantRL provides a wrapper `IsaacVecEnv` to process an Isaac Gym environment:

```
from elegantrl.envs.IsaacGym import IsaacVecEnv, IsaacOneEnv
import isaacgym
import torch  # import torch after import IsaacGym modules

env_func = IsaacVecEnv
env_args = {
```

(continues on next page)

(continued from previous page)

```
    'env_num': 4096,
    'env_name': 'Ant',
    'max_step': 1000,
    'state_dim': 60,
    'action_dim': 8,
    'if_discrete': False,
    'target_return': 14000.0,

    'device_id': None,  # set by worker
    'if_print': False,  # if_print=False in default
}
```

Once we have the env_func and env_args, we can follow the same training procedure as we listed in the Bipedal-Walker and LunarLander examples.

Initialize agent and environment, specify hyper-parameters, and start training:

```
from elegantrl.agents.AgentPPO import AgentPPO
from elegantrl.run import train_and_evaluate_mp

args = Arguments(agent=AgentPPO, env_func=env_func, env_args=env_args)

'''set one env for evaluator'''
args.eval_env_func = IsaacOneEnv
args.eval_env_args = args.env_args.copy()
args.eval_env_args['env_num'] = 1

'''set other hyper-parameters'''
args.net_dim = 2 ** 9
args.batch_size = args.net_dim * 4
args.target_step = args.max_step
args.repeat_times = 2 ** 4

args.save_gap = 2 ** 9
args.eval_gap = 2 ** 8
args.eval_times1 = 2 ** 0
args.eval_times2 = 2 ** 2

args.worker_num = 1
args.learner_gpus = 0
train_and_evaluate_mp(args)
```

1.14 How to run learner parallelism: REDQ

1.15 How to learn stably: H-term

Stability plays a key role in productizing DRL applications to real-world problems, making it a central concern of DRL researchers and practitioners. Recently, a lot of algorithms and open-source software have been developed to address this challenge. A popular open-source library Stable-Baselines3 offers a set of reliable implementations of DRL algorithms that match prior results.

In this article, we introduce a **Hamiltonian-term (H-term)**, a generic add-on in ElegantRL that can be applied to existing model-free DRL algorithms. The H-term essentially trades computing power for stability.

1.15.1 Basic Idea

In a standard RL problem, a decision-making process can be modeled as a Markov Decision Process (MDP). The Bellman equation gives the optimality condition for MDP problems:

$$Q(\pi|s,a) = R^{a}_{s,s'} + \gamma \sum_{a'} \pi(s',a') Q(\pi|s',a'),$$

The above equation is inherently recursive, so we expand it as follows:

$$Q(\pi|s_t,a_t) = R^{a_t}_{s_t,s_{t+1}} + \gamma \sum_{a_{t+1}} R^{a_{t+1}}_{s_{t+1},s_{t+2}} \pi_{t+1} + \gamma^2 \sum_{a_{t+1}} \sum_{a_{t+2}} R^{a_{t+2}}_{s_{t+2},s_{t+3}} \pi_{t+1}\pi_{t+2} + \cdots$$

In practice, we aim to find a policy that maximizes the Q-value. By taking a variational approach, we can rewrite the Bellman equation into a Hamiltonian equation. Our goal then is transformed to find a policy that minimizes the energy of a system. (Check our paper for details).

$$\begin{aligned} H'(\pi) &= -\sum_{k=1}^{K} \left(\sum_{\mu_{t+k}}^{S\times A} \left(\cdots \left(\sum_{\mu_{t+1}}^{S\times A} \mathcal{C}^{(k)}_{\mu_{t+1},\ldots,\mu_{t+k}} \pi(\mu_{t+1}) \right) \cdots \right) \pi(\mu_{t+k}) \right) \\ &= -\sum_{k=1}^{K} \Big(\Big(\cdots \Big(\mathcal{C}^{(k)} \times_1 \pi \Big) \times_2 \cdots \Big) \times_k \pi \Big) = -\sum_{k=1}^{K} \Big\langle \mathcal{C}^{(k)}, \underbrace{(\pi \otimes \pi \otimes \cdots \otimes \pi)}_{k \text{ times}} \Big\rangle. \end{aligned}$$

1.15.2 A Simple Add-on

The derivations and physical interpretations may be a little bit scary, however, the actual implementation of the H-term is super simple. Here, we present the pseudocode and make a comparison (marked in red) to the Actor-Critic algorithms:

Algorithm 1 Actor-Critic DRL Algorithms with H-term

Input: N, K, G, B, M,L γ, τ
Output: policy network $\theta^{\mu'}$
Randomly initialize critic net $Q(s,a|\theta^Q)$ and actor net $\mu(s|\theta^\mu)$ with parameters θ^Q and θ^μ
Initialize target nets Q' and μ' with parameters $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $\mathcal{D}_1$ and $\mathcal{D}_2$
for $m = 1, ..., M$ **do** % each iteration
  Receive initial observation state s_1
  for $n = 1, ..., N$ **do** % each environment step
    Take action $a_n \sim \mu(\cdot|s_n)$
    Execute action a_n, observe reward R_n, and observe new state s_{n+1}
    Store transition (s_n, a_n, R_n, s_{n+1}) in $\mathcal{D}_1$
  end
  Store trajectory $(s_1, a_1, R_1, s_2, ..., s_N, a_N, R_N, s_{N+1})$ in $\mathcal{D}_2$
  for $g = 1, ..., G$ **do** % perform G updates
    Sample a random minibatch of B transitions (s_t, a_t, R_t, s_{t+1}) from $\mathcal{D}_1$
    Set target value $y_t = R_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$
    Using a random batch of transitions from $\mathcal{D}_1$ to update θ^Q by minimizing the loss

$$\mathcal{L}_Q = \frac{1}{B}\sum_{t=1}^{B}\left(y_t - Q(s_t, a_t|\theta^Q)\right)^2 \tag{13}$$

    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{B}\sum_{t=1}^{B} \nabla_a Q\left(s_t, a_t|\theta^Q\right) \nabla_{\theta^\mu}\mu\left(s_t|\theta^\mu\right) \tag{14}$$

    Sort the trajectories in $\mathcal{D}_2$ in a descending order according to the K-step return.
    Select the top L trajectories from $\mathcal{D}_2$ to update θ^μ by minimizing

$$\widehat{H} = -\sum_{\ell=1}^{L}\sum_{k=1}^{K} \gamma^k R^{\ell}_{s_{t+k}, s_{t+k+1}} \mu(s_{t+k}, a_{t+k}|\theta^\mu) \tag{15}$$

    Update target nets:

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \end{aligned} \tag{16}$$

  end
end

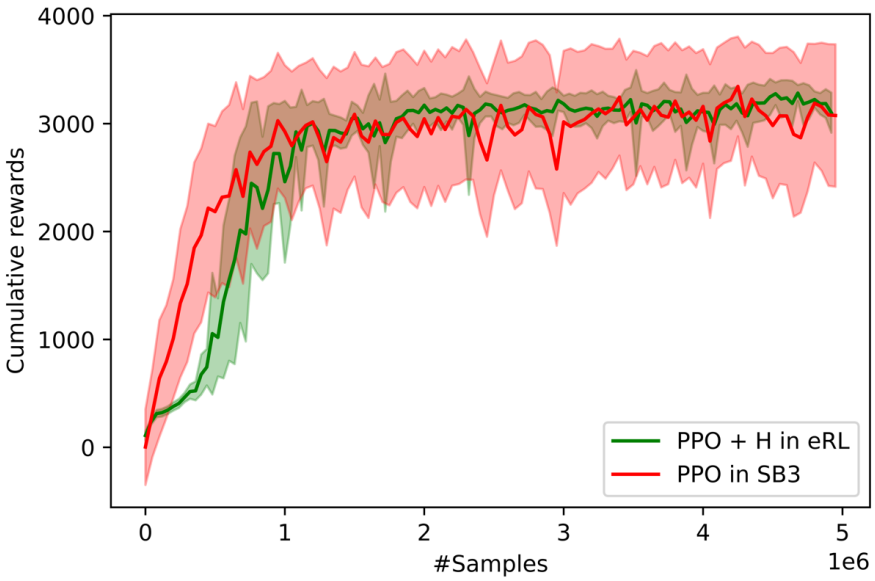
As marked out in lines 19–20, we include an additional update of the policy network, in order to minimize the H-term. Different from most algorithms that optimize on a single step (batch of transitions), we emphasize the importance of the sequential information from a trajectory (batch of trajectories).

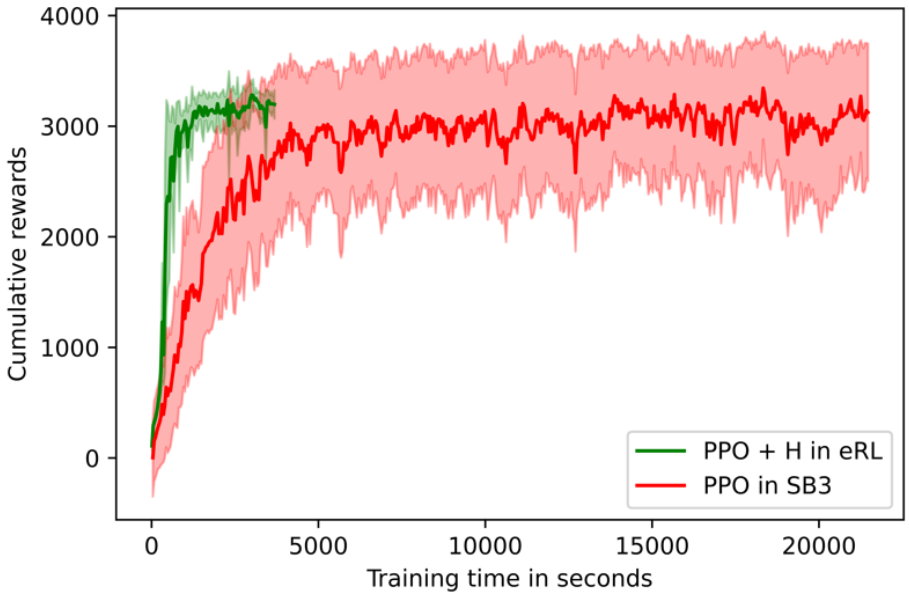
It is a fact that optimizing the H-term is compute-intensive, controlled by the hyper-parameter L (the number of selected trajectories) and K (the length of each trajectory). Fortunately, ElegantRL fully supports parallel computing from a single GPU to hundreds of GPUs, which provides the opportunity to trade computing power for stability.

1.15.3 Example: Hopper-v2

Currently, we have implemented the H-term into several widely-used DRL algorithms, PPO, SAC, TD3, and DDPG. Here, we present the performance on a benchmark problem Hopper-v2 using PPO algorithm.

The implementations of PPO+H in here





In terms of variance, it is obvious that ElegantRL substantially outperforms Stable-Baseline3. The variance over 8 runs is much smaller. Also, the PPO+H in ElegantRL completed the training process of 5M samples in about 6x faster than Stable-Baseline3.

1.16 Cloud Example 1: Generational Evolution

In this section, we provide a tutorial of *generational evolution mechanism* with an ensemble method, to show ElegantRL's scalability on hundreds of computing nodes on a cloud platform, say, hundreds of GPUs.

For detailed description, please check our recent paper:

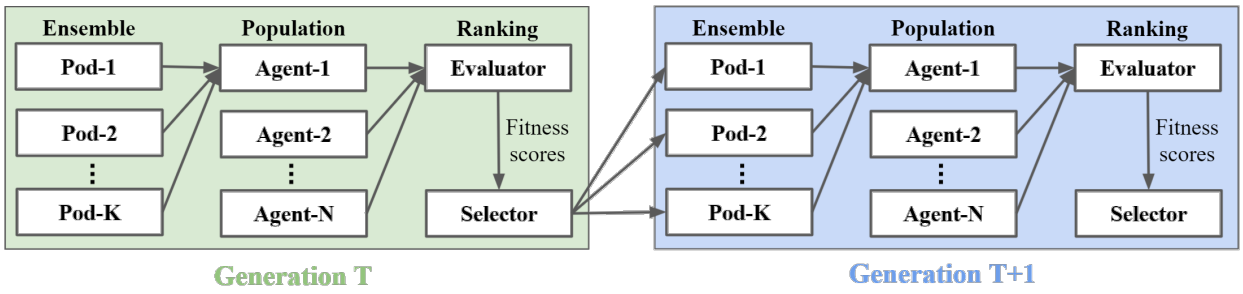
Zechu Li, Xiao-Yang Liu, Jiahao Zheng, Zhaoran Wang, Anwar Walid, and Jian Guo. FinRL-podracer: High performance and scalable deep reinforcement learning for quantitative finance. *ACM International Conference on AI in Finance (ICAIF)*, 2021.

The codes are available on GitHub.

1.16.1 What is a generational evolution mechanism?

A generational evolution mechanism with an ensemble method is a way to coordinate parallel agents in the population-based training (agent parallelism in ElegantRL). Under such a mechanism, we can initialize hundreds or even thousands of parallel agents with different hyper-parameter setups, thus performing hyper-parameter search on hundreds of GPUs of the cloud.

In the generational evolution, we periodically update every agent in parallel to form generations, where each period can be a certain number of training steps or a certain amount of training time. For each generation, it is composed of population ranking and model ensemble, as shown in the figure below.



1.16.2 Population ranking

The population ranking is scheduled by an evaluator and a selector.

At every generation,

1. A population of N agents is trained for a certain number of training steps or a certain amount of training time.
2. The evaluator calculates agents' scores, e.g., episodic rewards.
3. The selector ranks agents based on their scores and redistributes training files of agents with the highest scores to form a new population
4. The new population of N agents continues to be trained in the next generation.

1.16.3 Model ensemble

In the training of each agent, we provide an ensemble method, model fusion, to stabilize its learning process. In the model fusion, we concurrently run K pods (training processes) to train each agent in parallel, where all K pods are initialized with the same hyper-parameters but different random seeds. The stochasticity brought by different random seeds increases the diversity of data collection, thus improving the stability of the learning process. After all K pods finish training, we fuse K trained models and optimzers to obtain a single model and optimizer for that agent.

At present, we achieve the model fusion in a similar fashion to the soft update of target network in DRL. For example, for models and optimizers, we have:

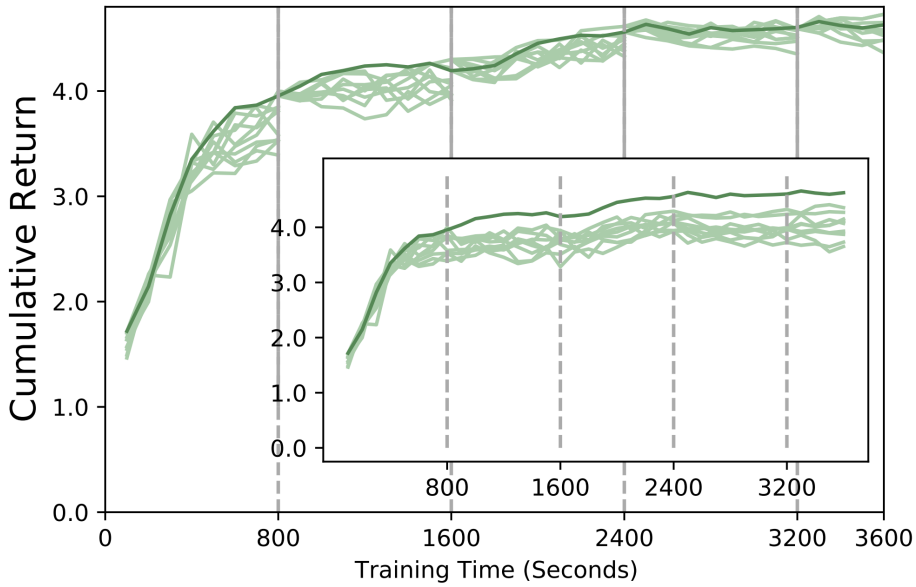
```
def avg_update_net(dst_net, src_net, device):
  for dst, src in zip(dst_net.parameters(), src_net.parameters()):
      dst.data.copy_((dst.data + src.data.to(device)) * 0.5)
      # dst.data.copy_(src.data * tau + dst.data * (1 - tau))

def avg_update_optim(dst_optim, src_optim, device):
  for dst, src in zip(get_optim_parameters(dst_optim), get_optim_parameters(src_optim)):
      dst.data.copy_((dst.data + src.data.to(device)) * 0.5)
```

1.16.4 Example: stock trading

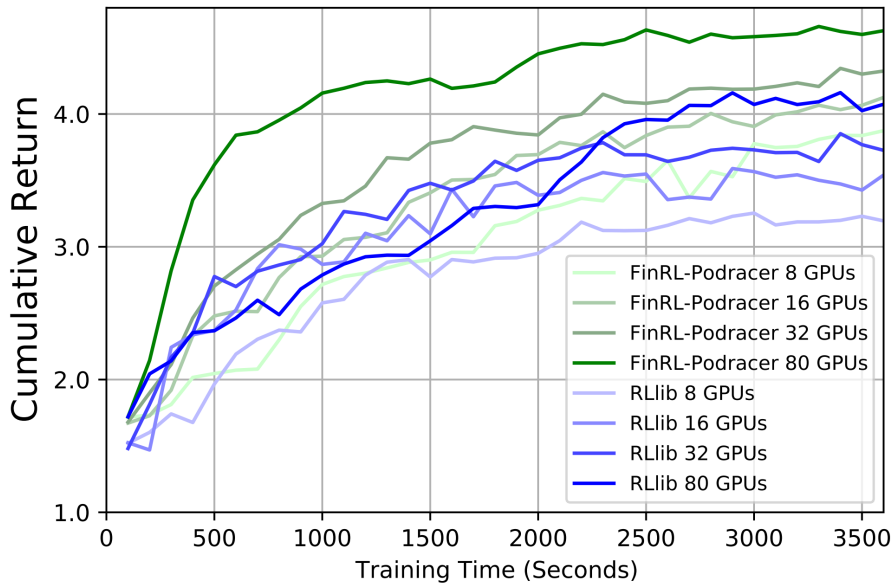
Finance is a promising and challenging real-world application of DRL algorithms. Therefore, we select a stock trading task as an example, which aims to train a DRL agent that decides *where to trade*, *at what price and what quantity* in a stock market.

We select the minute-level dataset of the NASDAQ-100 constituent stocks and follow a training-backtesting pipeline to split the dataset into two sets: the data from 01/01/2016 to 05/25/2019 for training, and the data from 05/26/2019 to 05/26/2021 for backtesting. To ensure that we do not use any future information from backtesting dataset, we store the model snapshots at different training time, say every 100 seconds, then later we use each snapshot model to perform inference on the backtesting dataset and obtain the generalization performance, namely, the cumulative return.

First, we empirically investigate the generational evolution mechanism. The figure below explicitly demonstrates an evolution of N (= 10) agents on 80 A100 GPUs, where the selector chooses the best agent to train in the next generation every 800 seconds. The inner figure depicts the generalization curves of the ten agents in the first generation (without using the agent evolution mechanism). The curve with the generational evolution mechanism (the thick green curve) is substantially higher than the other ten curves.



We compare our generational evolution mechanism with RLlib on a varying number of A100 GPUs, i.e., 8, 16, 32, and 80.



1.17 Cloud Example 2: Tournament-based Ensemble Training

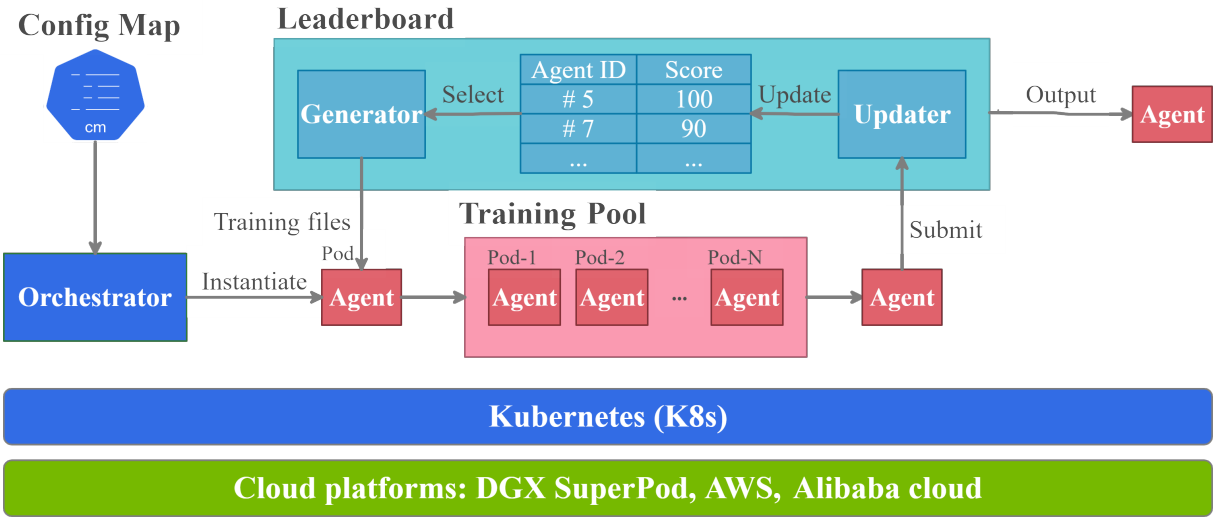
In this section, we provide a tutorial of *tournament-based ensemble training*, to show ElegantRL's scalability on hundreds of computing nodes on a cloud platform, say, hundreds of GPUs.

For detailed description, please check our recent paper:

Xiao-Yang Liu, Zechu Li, Zhuoran Yang, Jiahao Zheng, Zhaoran Wang, Anwar Walid, Jiang Guo, and Michael I. Jordan. ElegantRL-Podracer: Scalable and Elastic Library for Cloud-Native Deep Reinforcement Learning. *Deep Reinforcement Learning Workshop at NeurIPS*, 2021.

1.17.1 What is a tournament-based ensemble training?

The key of the tournament-based ensemble training scheme is the interaction between a *training pool* and a *leaderboard*. The training pool contains hundreds of agents that 1) are trained in an asynchronous manner, and 2) can be initialized with different DRL algorithms/hyper-parameter setup for an ensemble purpose. The leaderboard records the agents with high performance and continually updates as more agents (pods) are trained.



As shown in the figure above, the tournament-based ensemble training proceeds as follows:

1. An *orchestrator* instantiates a new agent and put it into a training pool.
2. A *generator* initializes an agent with networks and optimizers selected from a leaderboard. The generator is a class of subordinate functions associated with the leaderboard, which has different variations to support different evolution strategies
3. An *updater* determines whether and where to insert an agent into the leaderboard according to its performance, after a pod has been trained for a certain number of steps or certain amount of time.

1.17.2 Comparison with generational evolution

In generational evolution, the entire population of agents is simultaneously updated for each generation. However, this paradigm scales poorly on the cloud since it requires to finish training of every member of a large population before any further evolution can occur, imposing a significant computational burden.

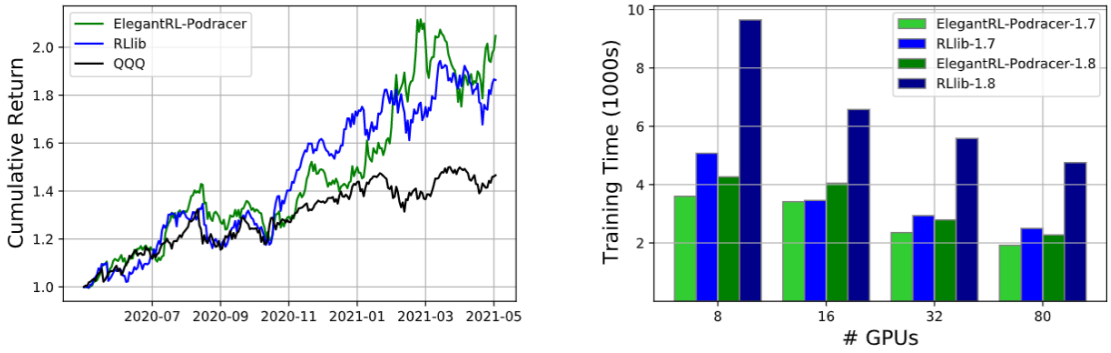
Our tournament-based ensemble training updates agents asynchronously, which decouples population evolution and singleagent learning. Such an asynchronously distributed training reduce waiting time among parallel agents and reduce the agent-to-agent communication overhead.

1.17.3 Example: Stock Trading

Finance is a promising and challenging real-world application of DRL algorithms. We apply ElegantRL-podracer to a stock trading task as an example to show its potential in quantitative finance.

We aim to train a DRL agent that decides where to trade, at what price and what quantity in a stock market, thus the objective of the problem is to maximize the expected return and minimize the risk. We model the stock trading task as a Markov Decision Process (MDP) as in FinRL. We follow a training-backtesting pipeline and split the dataset into two sets: the data from 01/01/2016 to 05/25/2020 for training, and the data from 05/26/2020 to 05/26/2021 for backtesting.

The experiments were executed using NVIDIA DGX-2 servers in a DGX SuperPOD cloud, a cloud-native infrastructure.



Left: cumulative return on minute-level NASDAQ-100 constituents stocks (initial capital $1, 000, 000, transaction cost 0.2%). Right: training time (wall-clock time) for reaching cumulative rewards 1.7 and 1.8, using the model snapshots of ElegantRL-podracer and RLlib.

	Cumul. return	Annual return	Annual volatility	Max. drawdown	Sharpe ratio	Calmar ratio
ElegantRL-podracer	**104.743%**	**103.591%**	35.357%	-17.187%	**2.20**	6.02
RLlib Liang et al. [2018]	86.274%	85.364%	34.319%	-13.689%	1.98	**6.24**
Invesco QQQ ETF	46.586%	46.146%	**23.39%**	**-12.749%**	1.75	3.62

All DRL agents can achieve a better performance than the market benchmark with respect to the cumulative return, demonstrating the algorithm's effectiveness. We observe that ElegantRL-podracer has a cumulative return of 104.743%, an annual return of 103.591%, and a Sharpe ratio of 2.20, which outperforms RLlib substantially. However, ElegantRL-podracer is not as stable as RLlib during the backtesting period: it achieves annual volatility of 35.357%, max. drawdown 17.187%, and Calmar ratio 6.02. There are two possible reasons to account for such instability:

1. the reward design in the stock trading environment is mainly related to the cumulative return, thus leading the agent to take less care of the risk;
2. ElegantRL-podracer holds a large number of funds around 2021–03, which naturally leads to a larger slip.

We compare the training performance on a varying number of GPUs, i.e., 8, 16, 32, and 80. We measure the required training time to obtain two cumulative returns of 1.7 and 1.8, respectively. Both ElegantRL-podracer and RLlib require less training time to achieve the same cumulative return as the number of GPUs increases, which directly demonstrates the advantage of cloud computing resources on the DRL training. For ElegantRL-podracer with 80 GPUs, it requires (1900s, 2200s) to reach cumulative returns of 1.7 and 1.8. ElegantRL-podracer with 32 and 16 GPUs need (2400s, 2800s) and (3400s, 4000s) to achieve the same cumulative returns. It demonstrates the high scalability of ElegantRL-podracer and the effectiveness of our cloud-oriented optimizations. For the experiments using RLlib, increasing the number of GPUs does not lead to much speed-up.

1.17.4 Run tournament-based ensemble training in ElegantRL

Here, we provide a demo code to run the Isaac Gym Ant with tournament-based ensemble training in ElegantRL.

```
import isaacgym
import torch  # import torch after import IsaacGym modules
from elegantrl.train.config import Arguments
from elegantrl.train.run import train_and_evaluate_mp
from elegantrl.envs.IsaacGym import IsaacVecEnv, IsaacOneEnv
from elegantrl.agents.AgentPPO import AgentPPO

'''set vec env for worker'''
env_func = IsaacVecEnv
env_args = {
    'env_num': 2 ** 10,
    'env_name': 'Ant',
    'max_step': 1000,
    'state_dim': 60,
    'action_dim': 8,
    'if_discrete': False,
    'target_return': 14000.0,

    'device_id': None,  # set by worker
    'if_print': False,  # if_print=False in default
}

args = Arguments(agent=AgentPPO(), env_func=env_func, env_args=env_args)
args.agent.if_use_old_traj = False  # todo

'''set one env for evaluator'''
args.eval_env_func = IsaacOneEnv
args.eval_env_args = args.env_args.copy()
args.eval_env_args['env_num'] = 1

'''set other hyper-parameters'''
args.net_dim = 2 ** 9
args.batch_size = args.net_dim * 4
args.target_step = args.max_step
args.repeat_times = 2 ** 4

args.save_gap = 2 ** 9
args.eval_gap = 2 ** 8
args.eval_times1 = 2 ** 0
args.eval_times2 = 2 ** 2
```

(continues on next page)

(continued from previous page)

```
args.worker_num = 1  # VecEnv, worker number = 1
args.learner_gpus = [(i,) for i in range(0, 8)]  # 8 agents (1 GPU per agent) performing␣
↪tournament-based ensemble training

train_and_evaluate_mp(args, python_path='.../bin/python3')
```

1.18 DQN

Deep Q-Network (DQN) is an off-policy value-based algorithm for discrete action space. It uses a deep neural network to approximate a Q function defined on state-action pairs. This implementation starts from a vanilla Deep Q-Learning and supports the following extensions:

- Experience replay: ✓
- Target network (soft update): ✓
- Gradient clipping: ✓
- Reward clipping:
- Prioritized Experience Replay (PER): ✓
- Dueling network architecture: ✓

Note: This implementation has no support for reward clipping because we introduce the hyper-paramter `reward_scale` for reward scaling as an alternative. We believe that the clipping function may omit information since it cannot map the clipped reward back to the original reward; however, the reward scaling function is able to manipulate the reward back and forth.

Warning: PER leads to a faster learning speed and is also critical for environments with sparse rewards. However, a replay buffer with small size may hurt the performance of PER.

1.18.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentDQN import AgentDQN

# train and save
args = Arguments(env=build_env('CartPole-v0'), agent=AgentDQN())
args.cwd = 'demo_CartPole_DQN'
args.target_return = 195
args.agent.if_use_dueling = True
train_and_evaluate(args)

# test
```

(continues on next page)

(continued from previous page)

```
agent = AgentDQN()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('CartPole-v0')
state = env.reset()
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.18.2 Parameters

class elegantrl.agents.AgentDQN.**AgentDQN**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

Deep Q-Network algorithm. "Human-Level Control Through Deep Reinforcement Learning". Mnih V. et al.. 2015.

net_dims: the middle layer dimension of MLP (MultiLayer Perceptron) state_dim: the dimension of state (the number of state vector) action_dim: the dimension of action (or the number of discrete action) gpu_id: the gpu_id of the training device. Use CPU when cuda is not available. args: the arguments for agent training. *args = Config()*

explore_one_env(*env*, *horizon_len: int*, *if_random: bool = False*) → Tuple[torch.Tensor, ...]

Collect trajectories through the actor-environment interaction for a **single** environment instance.

env: RL training environment. env.reset() env.step(). It should be a vector env. horizon_len: collect horizon_len step while exploring to update networks if_random: uses random action for warn-up exploration return: *(states, actions, rewards, undones)* for off-policy

num_envs == 1 states.shape == (horizon_len, num_envs, state_dim) actions.shape == (horizon_len, num_envs, action_dim) rewards.shape == (horizon_len, num_envs) undones.shape == (horizon_len, num_envs)

explore_vec_env(*env*, *horizon_len: int*, *if_random: bool = False*) → Tuple[torch.Tensor, ...]

Collect trajectories through the actor-environment interaction for a **vectorized** environment instance.

env: RL training environment. env.reset() env.step(). It should be a vector env. horizon_len: collect horizon_len step while exploring to update networks if_random: uses random action for warn-up exploration return: *(states, actions, rewards, undones)* for off-policy

states.shape == (horizon_len, num_envs, state_dim) actions.shape == (horizon_len, num_envs, action_dim) rewards.shape == (horizon_len, num_envs) undones.shape == (horizon_len, num_envs)

get_obj_critic_per(*buffer:* ReplayBuffer, *batch_size: int*) → Tuple[torch.Tensor, torch.Tensor]

Calculate the loss of the network and predict Q values with **Prioritized Experience Replay (PER)**.

Parameters

- **buffer** – the ReplayBuffer instance that stores the trajectories.
- **batch_size** – the size of batch data for Stochastic Gradient Descent (SGD).

Returns

the loss of the network and Q values.

get_obj_critic_raw(*buffer:* ReplayBuffer, *batch_size: int*) → Tuple[torch.Tensor, torch.Tensor]

Calculate the loss of the network and predict Q values with **uniform sampling**.

Parameters

- **buffer** – the ReplayBuffer instance that stores the trajectories.
- **batch_size** – the size of batch data for Stochastic Gradient Descent (SGD).

Returns

the loss of the network and Q values.

1.18.3 Networks

class elegantrl.agents.net.**QNet**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**QNetDuel**(**args: Any*, ***kwargs: Any*)

1.19 Double DQN

Double Deep Q-Network (Double DQN) is one of the most important extensions of vanilla DQN. It resolves the issue of overestimation via a simple trick: decoupling the max operation in the target into **action selection** and **action evaluation**.

Without having to introduce additional networks, we use a Q-network to select the best among the available next actions and use the target network to evaluate its Q-value. This implementation supports the following extensions:

- Experience replay: ✓
- Target network: ✓
- Gradient clipping: ✓
- Reward clipping:
- Prioritized Experience Replay (PER): ✓
- Dueling network architecture: ✓

1.19.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentDoubleDQN import AgentDoubleDQN

# train and save
args = Arguments(env=build_env('CartPole-v0'), agent=AgentDoubleDQN())
args.cwd = 'demo_CartPole_DoubleDQN'
args.target_return = 195
train_and_evaluate(args)

# test
agent = AgentDoubleDQN()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('CartPole-v0')
state = env.reset()
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.19.2 Parameters

1.19.3 Networks

class elegantrl.agents.net.**QNetTwin**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**QNetTwinDuel**(**args: Any*, ***kwargs: Any*)

1.20 DDPG

Deep Deterministic Policy Gradient (DDPG) is an off-policy Actor-Critic algorithm for continuous action space. Since computing the maximum over actions in the target is a challenge in continuous action space, DDPG deals with this using a policy network to compute an action. This implementation provides DDPG and supports the following extensions:

- Experience replay: ✓
- Target network: ✓
- Gradient clipping: ✓
- Reward clipping:
- Prioritized Experience Replay (PER): ✓
- Ornstein–Uhlenbeck noise: ✓

Warning: In the DDPG paper, the authors use time-correlated Ornstein-Uhlenbeck Process to add noise to the action output. However, as shown in the later works, the Ornstein-Uhlenbeck Process is an overcomplication that does not have a noticeable effect on performance when compared to uncorrelated Gaussian noise.

1.20.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentDDPG import AgentDDPG

# train and save
args = Arguments(env=build_env('Pendulum-v0'), agent=AgentDDPG())
args.cwd = 'demo_Pendulum_DDPG'
args.env.target_return = -200
args.reward_scale = 2 ** -2
train_and_evaluate(args)

# test
agent = AgentDDPG()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('Pendulum-v0')
state = env.reset()
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
```

(continues on next page)

(continued from previous page)

```
    else:
        state = next_state
    env.render()
```

1.20.2 Parameters

class elegantrl.agents.AgentDDPG.**AgentDDPG**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

DDPG(Deep Deterministic Policy Gradient) "Continuous control with deep reinforcement learning". T. Lillicrap et al.. 2015."

net_dims: the middle layer dimension of MLP (MultiLayer Perceptron) state_dim: the dimension of state (the number of state vector) action_dim: the dimension of action (or the number of discrete action) gpu_id: the gpu_id of the training device. Use CPU when cuda is not available. args: the arguments for agent training. *args = Config()*

1.20.3 Networks

class elegantrl.agents.net.**Actor**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**Critic**(**args: Any*, ***kwargs: Any*)

1.21 TD3

Twin Delayed DDPG (TD3) is a successor of DDPG algorithm with the usage of three additional tricks. In TD3, the usage of **Clipped Double-Q Learning**, **Delayed Policy Updates**, and **Target Policy Smoothing** overcomes the overestimation of Q-values and smooths out Q-values along with changes in action, which shows improved performance over baseline DDPG. This implementation provides TD3 and supports the following extensions:

- Experience replay: ✓
- Target network: ✓
- Gradient clipping: ✓
- Reward clipping:
- Prioritized Experience Replay (PER): ✓

Note: With respect to the clipped Double-Q learning, we use two Q-networks with shared parameters under a single Class `CriticTwin`. Such an implementation allows a lower computational and training time cost.

Warning: In the TD3 implementation, it contains a number of highly sensitive hyper-parameters, which requires the user to carefully tune these hyper-parameters to obtain a satisfied result.

1.21.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentTD3 import AgentTD3

# train and save
args = Arguments(env=build_env('Pendulum-v0'), agent=AgentTD3())
args.cwd = 'demo_Pendulum_TD3'
args.env.target_return = -200
args.reward_scale = 2 ** -2
train_and_evaluate(args)

# test
agent = AgentTD3()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('Pendulum-v0')
state = env.reset()
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.21.2 Parameters

class elegantrl.agents.AgentTD3.**AgentTD3**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

Twin Delayed DDPG algorithm. Addressing Function Approximation Error in Actor-Critic Methods. 2018.

1.21.3 Networks

class elegantrl.agents.net.**Actor**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**CriticTwin**(**args: Any*, ***kwargs: Any*)

1.22 SAC

Soft Actor-Critic (SAC) is an off-policy Actor-Critic algorithm for continuous action space. In SAC, it introduces an entropy regularization to the loss function, which has a close connection with the trade-off of the exploration and exploitation. In our implementation, we employ a **learnable entropy regularization coefficienct** to dynamic control the scale of the entropy, which makes it consistent with a pre-defined target entropy. SAC also utilizes **Clipped Double-Q Learning** (mentioned in TD3) to overcome the overestimation of Q-values. This implementation provides SAC and supports the following extensions:

- Experience replay: ✓
- Target network: ✓
- Gradient clipping: ✓
- Reward clipping:
- Prioritized Experience Replay (PER): ✓
- Leanable entropy regularization coefficient: ✓

Note: Inspired by the delayed policy update from TD3, we implement a modified version of SAC `AgentModSAC` with a dynamic adjustment of the frequency of the policy update. The adjustment is based on the loss of critic networks: a small loss leads to a high update frequency and vise versa.

1.22.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentSAC import AgentSAC

# train and save
args = Arguments(env=build_env('Pendulum-v0'), agent=AgentSAC())
args.cwd = 'demo_Pendulum_SAC'
args.env.target_return = -200
args.reward_scale = 2 ** -2
train_and_evaluate(args)

# test
agent = AgentSAC()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('Pendulum-v0')
```

(continues on next page)

(continued from previous page)

```
state = env.reset()
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.22.2 Parameters

class elegantrl.agents.AgentSAC.**AgentSAC**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

class elegantrl.agents.AgentSAC.**AgentModSAC**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

1.22.3 Networks

class elegantrl.agents.net.**ActorSAC**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**CriticTwin**(**args: Any*, ***kwargs: Any*)

1.23 A2C

Advantage Actor-Critic (A2C) is a synchronous and deterministic version of Asynchronous Advantage Actor-Critic (A3C). It combines value optimization and policy optimization approaches. This implementation of the A2C algorithm is built on PPO algorithm for simplicity, and it supports the following extensions:

- Target network: ✓
- Gradient clipping: ✓
- Reward clipping:
- Generalized Advantage Estimation (GAE): ✓
- Discrete version: ✓

Warning: The implementation of A2C serves as a pedagogical goal. For practitioners, we recommend using the PPO algorithm for training agents. Without the trust-region and clipped ratio, hyper-parameters in A2C, e.g., `repeat_times`, need to be fine-tuned to avoid performance collapse.

1.23.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentA2C import AgentA2C

# train and save
args = Arguments(env=build_env('Pendulum-v0'), agent=AgentA2C())
args.cwd = 'demo_Pendulum_A2C'
args.env.target_return = -200
args.reward_scale = 2 ** -2
train_and_evaluate(args)

# test
agent = AgentA2C()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('Pendulum-v0')
state = env.reset()
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.23.2 Parameters

class elegantrl.agents.AgentA2C.**AgentA2C**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

A2C algorithm. “Asynchronous Methods for Deep Reinforcement Learning”. Mnih V. et al.. 2016.

class elegantrl.agents.AgentA2C.**AgentDiscreteA2C**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

1.23.3 Networks

class elegantrl.agents.net.**ActorPPO**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**ActorDiscretePPO**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**CriticPPO**(**args: Any*, ***kwargs: Any*)

1.24 PPO

Proximal Policy Optimization (PPO) is an on-policy Actor-Critic algorithm for both discrete and continuous action spaces. It has two primary variants: **PPO-Penalty** and **PPO-Clip**, where both utilize surrogate objectives to avoid the new policy changing too far from the old policy. This implementation provides PPO-Clip and supports the following extensions:

- Target network: ✓
- Gradient clipping: ✓
- Reward clipping:
- Generalized Advantage Estimation (GAE): ✓
- Discrete version: ✓

Note: The surrogate objective is the key feature of PPO since it both regularizes the policy update and enables the reuse of training data.

A clear explanation of PPO algorithm and implementation in ElegantRL is available here.

1.24.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentPPO import AgentPPO

# train and save
args = Arguments(env=build_env('BipedalWalker-v3'), agent=AgentPPO())
args.cwd = 'demo_BipedalWalker_PPO'
args.env.target_return = 300
args.reward_scale = 2 ** -2
train_and_evaluate(args)

# test
agent = AgentPPO()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('BipedalWalker-v3')
state = env.reset()
```

(continues on next page)

(continued from previous page)

```
episode_reward = 0
for i in range(2 ** 10):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.24.2 Parameters

class elegantrl.agents.AgentPPO.**AgentPPO**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

PPO algorithm. "Proximal Policy Optimization Algorithms". John Schulman. et al.. 2017.

net_dims: the middle layer dimension of MLP (MultiLayer Perceptron) state_dim: the dimension of state (the number of state vector) action_dim: the dimension of action (or the number of discrete action) gpu_id: the gpu_id of the training device. Use CPU when cuda is not available. args: the arguments for agent training. *args = Config()*

explore_one_env(*env*, *horizon_len: int*, *if_random: bool = False*) → Tuple[torch.Tensor, ...]

Collect trajectories through the actor-environment interaction for a **single** environment instance.

env: RL training environment. env.reset() env.step(). It should be a vector env. horizon_len: collect horizon_len step while exploring to update networks return: *(states, actions, rewards, undones)* for off-policy

env_num == 1 states.shape == (horizon_len, env_num, state_dim) actions.shape == (horizon_len, env_num, action_dim) logprobs.shape == (horizon_len, env_num, action_dim) rewards.shape == (horizon_len, env_num) undones.shape == (horizon_len, env_num)

explore_vec_env(*env*, *horizon_len: int*, *if_random: bool = False*) → Tuple[torch.Tensor, ...]

Collect trajectories through the actor-environment interaction for a **vectorized** environment instance.

env: RL training environment. env.reset() env.step(). It should be a vector env. horizon_len: collect horizon_len step while exploring to update networks return: *(states, actions, rewards, undones)* for off-policy

states.shape == (horizon_len, env_num, state_dim) actions.shape == (horizon_len, env_num, action_dim) logprobs.shape == (horizon_len, env_num, action_dim) rewards.shape == (horizon_len, env_num) undones.shape == (horizon_len, env_num)

class elegantrl.agents.AgentPPO.**AgentDiscretePPO**(*net_dims: [<class 'int'>], state_dim: int, action_dim: int, gpu_id: int = 0, args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

explore_one_env(*env*, *horizon_len: int*, *if_random: bool = False*) → Tuple[torch.Tensor, ...]

Collect trajectories through the actor-environment interaction for a **single** environment instance.

env: RL training environment. env.reset() env.step(). It should be a vector env. horizon_len: collect horizon_len step while exploring to update networks return: *(states, actions, rewards, undones)* for off-policy

env_num == 1 states.shape == (horizon_len, env_num, state_dim) actions.shape == (horizon_len, env_num, action_dim) logprobs.shape == (horizon_len, env_num, action_dim) rewards.shape == (horizon_len, env_num) undones.shape == (horizon_len, env_num)

explore_vec_env(*env*, *horizon_len: int*, *if_random: bool = False*) → Tuple[torch.Tensor, ...]

Collect trajectories through the actor-environment interaction for a **vectorized** environment instance.

env: RL training environment. env.reset() env.step(). It should be a vector env. horizon_len: collect horizon_len step while exploring to update networks return: *(states, actions, rewards, undones)* for off-policy

states.shape == (horizon_len, env_num, state_dim) actions.shape == (horizon_len, env_num, action_dim) logprobs.shape == (horizon_len, env_num, action_dim) rewards.shape == (horizon_len, env_num) undones.shape == (horizon_len, env_num)

1.24.3 Networks

class elegantrl.agents.net.**ActorPPO**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**ActorDiscretePPO**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**CriticPPO**(**args: Any*, ***kwargs: Any*)

1.25 REDQ

Randomized Ensembled Double Q-Learning: Learning Fast Without a Model (REDQ) has three carefully integrated ingredients to achieve its high performance:

- update-to-data (UTD) ratio >> 1.
- an ensemble of Q functions.
- in-target minimization across a random subset of Q functions.

This implementation is based on SAC.

1.25.1 Code Snippet

```
import torch
from elegantrl.run import train_and_evaluate
from elegantrl.config import Arguments
from elegantrl.train.config import build_env
from elegantrl.agents.AgentREDQ import AgentREDQ

# train and save
args = Arguments(env=build_env('Hopper-v2'), agent=AgentREDQ())
args.cwd = 'demo_Hopper_REDQ'
train_and_evaluate(args)
```

(continues on next page)

(continued from previous page)

```
# test
agent = AgentREDQ()
agent.init(args.net_dim, args.state_dim, args.action_dim)
agent.save_or_load_agent(cwd=args.cwd, if_save=False)

env = build_env('Pendulum-v0')
state = env.reset()
episode_reward = 0
for i in range(125000):
    action = agent.select_action(state)
    next_state, reward, done, _ = env.step(action)

    episode_reward += reward
    if done:
        print(f'Step {i:>6}, Episode return {episode_reward:8.3f}')
        break
    else:
        state = next_state
    env.render()
```

1.25.2 Parameters

1.25.3 Networks

class elegantrl.agents.net.**ActorSAC**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**Critic**(**args: Any*, ***kwargs: Any*)

1.26 MADDPG

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) is a multi-agent reinforcement learning algorithm for continuous action space:

- Implementation is based on DDPG ✓
- Initialize n DDPG agents in MADDPG ✓

1.26.1 Code Snippet

```
def update_net(self, buffer, batch_size, repeat_times, soft_update_tau):
    buffer.update_now_len()
    self.batch_size = batch_size
    self.update_tau = soft_update_tau
    rewards, dones, actions, observations, next_obs = buffer.sample_batch(self.batch_
↪size)
    for index in range(self.n_agents):
        self.update_agent(rewards, dones, actions, observations, next_obs, index)
```

(continues on next page)

(continued from previous page)

```
    for agent in self.agents:
        self.soft_update(agent.cri_target, agent.cri, self.update_tau)
        self.soft_update(agent.act_target, agent.act, self.update_tau)

    return
```

1.26.2 Parameters

class elegantrl.agents.AgentMADDPG.**AgentMADDPG**

Bases: AgentBase

Multi-Agent DDPG algorithm. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive". R Lowe. et al.. 2017.

Parameters

- **net_dim[int]** – the dimension of networks (the width of neural networks)
- **state_dim[int]** – the dimension of state (the number of state vector)
- **action_dim[int]** – the dimension of action (the number of discrete action)
- **learning_rate[float]** – learning rate of optimizer
- **gamma[float]** – learning rate of optimizer
- **n_agents[int]** – number of agents
- **if_per_or_gae[bool]** – PER (off-policy) or GAE (on-policy) for sparse reward
- **env_num[int]** – the env number of VectorEnv. env_num == 1 means don't use VectorEnv
- **agent_id[int]** – if the visible_gpu is '1,9,3,4', agent_id=1 means (1,9,4,3)[agent_id] == 9

explore_one_env(*env*, *target_step*) → list

Exploring the environment for target_step. param env: the Environment instance to be explored. param target_step: target steps to explore.

save_or_load_agent(*cwd*, *if_save*)

save or load training files for Agent

Parameters

- **cwd** – Current Working Directory. ElegantRL save training files in CWD.
- **if_save** – True: save files. False: load files.

select_actions(*states*)

Select continuous actions for exploration

Parameters

state – states.shape==(n_agents,batch_size, state_dim,)

Returns

actions.shape==(n_agents,batch_size, action_dim,), -1 < action < +1

update_agent(*rewards*, *dones*, *actions*, *observations*, *next_obs*, *index*)

Update the single agent neural networks, called by update_net.

Parameters

- **rewards** – reward list of the sampled buffer
- **dones** – done list of the sampled buffer
- **actions** – action list of the sampled buffer
- **observations** – observation list of the sampled buffer
- **next_obs** – next_observation list of the sample buffer
- **index** – ID of the agent

update_net(*buffer*, *batch_size*, *repeat_times*, *soft_update_tau*)

Update the neural networks by sampling batch data from `ReplayBuffer`.

Parameters

- **buffer** – the ReplayBuffer instance that stores the trajectories.
- **batch_size** – the size of batch data for Stochastic Gradient Descent (SGD).
- **repeat_times** – the re-using times of each trajectory.
- **soft_update_tau** – the soft update parameter.

1.26.3 Networks

class elegantrl.agents.net.**Actor**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**Critic**(**args: Any*, ***kwargs: Any*)

1.27 MATD3

Multi-Agent TD3 (MATD3) uses double centralized critics to reduce overestimation bias in multi-agent environments. It combines the improvements of TD3 with MADDPG.

1.27.1 Code Snippet

```
def update_net(self, buffer, batch_size, repeat_times, soft_update_tau):
    """
    Update the neural networks by sampling batch data from ``ReplayBuffer``.

    :param buffer: the ReplayBuffer instance that stores the trajectories.
    :param batch_size: the size of batch data for Stochastic Gradient Descent (SGD).
    :param repeat_times: the re-using times of each trajectory.
    :param soft_update_tau: the soft update parameter.
    :return Nonetype
    """
    buffer.update_now_len()
    self.batch_size = batch_size
```

(continues on next page)

(continued from previous page)

```
    self.update_tau = soft_update_tau
    rewards, dones, actions, observations, next_obs = buffer.sample_batch(self.batch_
↪size)
    for index in range(self.n_agents):
        self.update_agent(rewards, dones, actions, observations, next_obs, index)

    for agent in self.agents:
        self.soft_update(agent.cri_target, agent.cri, self.update_tau)
        self.soft_update(agent.act_target, agent.act, self.update_tau)

    return
```

1.27.2 Parameters

1.27.3 Networks

class elegantrl.agents.net.**Actor**(**args: Any*, ***kwargs: Any*)

class elegantrl.agents.net.**CriticTwin**(**args: Any*, ***kwargs: Any*)

1.28 QMix

QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning is a value-based method that can train decentralized policies in a centralized end-to-end fashion. QMIX employs a network that estimates joint action-values as a complex non-linear combination of per-agent values that condition only on local observations.

- Experience replay: ✓
- Target network: ✓
- Gradient clipping:
- Reward clipping:
- Prioritized Experience Replay (PER): ✓
- Ornstein–Uhlenbeck noise:

1.28.1 Code Snippet

```
def train(self, batch, t_env: int, episode_num: int, per_weight=None):
    rewards = batch["reward"][:, :-1]
    actions = batch["actions"][:, :-1]
    terminated = batch["terminated"][:, :-1].float()
    mask = batch["filled"][:, :-1].float()
    mask[:, 1:] = mask[:, 1:] * (1 - terminated[:, :-1])
    avail_actions = batch["avail_actions"]

    self.mac.agent.train()
```

(continues on next page)

(continued from previous page)

```
    mac_out = []
    self.mac.init_hidden(batch.batch_size)
    for t in range(batch.max_seq_length):
        agent_outs = self.mac.forward(batch, t=t)
        mac_out.append(agent_outs)
    mac_out = th.stack(mac_out, dim=1)

    chosen_action_qvals = th.gather(mac_out[:, :-1], dim=3, index=actions).squeeze(3)  #␣
↪Remove the last dim
    chosen_action_qvals_ = chosen_action_qvals

    with th.no_grad():
        self.target_mac.agent.train()
        target_mac_out = []
        self.target_mac.init_hidden(batch.batch_size)
        for t in range(batch.max_seq_length):
            target_agent_outs = self.target_mac.forward(batch, t=t)
            target_mac_out.append(target_agent_outs)

        target_mac_out = th.stack(target_mac_out, dim=1)  # Concat across time

        mac_out_detach = mac_out.clone().detach()
        mac_out_detach[avail_actions == 0] = -9999999
        cur_max_actions = mac_out_detach.max(dim=3, keepdim=True)[1]
        target_max_qvals = th.gather(target_mac_out, 3, cur_max_actions).squeeze(3)

        target_max_qvals = self.target_mixer(target_max_qvals, batch["state"])

        if getattr(self.args, 'q_lambda', False):
            qvals = th.gather(target_mac_out, 3, batch["actions"]).squeeze(3)
            qvals = self.target_mixer(qvals, batch["state"])

            targets = build_q_lambda_targets(rewards, terminated, mask, target_max_qvals,
↪ qvals,
                                self.args.gamma, self.args.td_lambda)
        else:
            targets = build_td_lambda_targets(rewards, terminated, mask, target_max_
↪qvals,
                                              self.args.n_agents, self.args.gamma,␣
↪self.args.td_lambda)

    chosen_action_qvals = self.mixer(chosen_action_qvals, batch["state"][:, :-1])

    td_error = (chosen_action_qvals - targets.detach())
    td_error2 = 0.5 * td_error.pow(2)

    mask = mask.expand_as(td_error2)
    masked_td_error = td_error2 * mask


    if self.use_per:
        per_weight = th.from_numpy(per_weight).unsqueeze(-1).to(device=self.device)
```

(continues on next page)

(continued from previous page)

```
        masked_td_error = masked_td_error.sum(1) * per_weight

    loss = L_td = masked_td_error.sum() / mask.sum()


    self.optimiser.zero_grad()
    loss.backward()
    grad_norm = th.nn.utils.clip_grad_norm_(self.params, self.args.grad_norm_clip)
    self.optimiser.step()
```

1.28.2 Parameters

1.28.3 Networks

class elegantrl.agents.net.**Critic**(**args: Any*, ***kwargs: Any*)

1.29 VDN

Value Decomposition Networks (VDN) trains individual agents with a novel value decomposition network architecture, which learns to decompose the team value function into agent-wise value functions.

1.29.1 Code Snippet

```
def train(self, batch, t_env: int, episode_num: int):

    # Get the relevant quantities
    rewards = batch["reward"][:, :-1]
    actions = batch["actions"][:, :-1]
    terminated = batch["terminated"][:, :-1].float()
    mask = batch["filled"][:, :-1].float()
    mask[:, 1:] = mask[:, 1:] * (1 - terminated[:, :-1])
    avail_actions = batch["avail_actions"]

    # Calculate estimated Q-Values
    mac_out = []
    self.mac.init_hidden(batch.batch_size)
    for t in range(batch.max_seq_length):
        agent_outs = self.mac.forward(batch, t=t)
        mac_out.append(agent_outs)
    mac_out = th.stack(mac_out, dim=1)  # Concat over time

    # Pick the Q-Values for the actions taken by each agent
    chosen_action_qvals = th.gather(mac_out[:, :-1], dim=3, index=actions).squeeze(3)  #␣
↪Remove the last dim

    # Calculate the Q-Values necessary for the target
    target_mac_out = []
```

(continues on next page)

(continued from previous page)

```
    self.target_mac.init_hidden(batch.batch_size)
    for t in range(batch.max_seq_length):
        target_agent_outs = self.target_mac.forward(batch, t=t)
        target_mac_out.append(target_agent_outs)
```

1.29.2 Parameters

1.29.3 Networks

1.30 MAPPO

Multi-Agent Proximal Policy Optimization (MAPPO) is a variant of PPO which is specialized for multi-agent settings. MAPPO achieves surprisingly strong performance in two popular multi-agent testbeds: the particle-world environments and the Starcraft multi-agent challenge.

- Shared network parameter for all agents ✓

MAPPO achieves strong results while exhibiting comparable sample efficiency.

1.30.1 Code Snippet

```
def ppo_update(self, sample, update_actor=True):

   share_obs_batch, obs_batch, rnn_states_batch, rnn_states_critic_batch, actions_batch,␣
↪\
   value_preds_batch, return_batch, masks_batch, active_masks_batch, old_action_log_
↪probs_batch, \
   adv_targ, available_actions_batch = sample

   old_action_log_probs_batch = check(old_action_log_probs_batch).to(**self.tpdv)
   adv_targ = check(adv_targ).to(**self.tpdv)
   value_preds_batch = check(value_preds_batch).to(**self.tpdv)
   return_batch = check(return_batch).to(**self.tpdv)
   active_masks_batch = check(active_masks_batch).to(**self.tpdv)

   # Reshape to do in a single forward pass for all steps
   values, action_log_probs, dist_entropy = self.policy.evaluate_actions(share_obs_batch,
                                                                          obs_batch,
                                                                          rnn_states_batch,
                                                                          rnn_states_
↪critic_batch,
                                                                          actions_batch,
                                                                          masks_batch,
                                                                          available_
↪actions_batch,
                                                                          active_masks_
↪batch)
   # actor update
   imp_weights = torch.exp(action_log_probs - old_action_log_probs_batch)
```

(continues on next page)

(continued from previous page)

```
   surr1 = imp_weights * adv_targ
   surr2 = torch.clamp(imp_weights, 1.0 - self.clip_param, 1.0 + self.clip_param) * adv_
↪targ
```

1.30.2 Parameters

1.30.3 Networks

1.31 Overview

One sentence summary: ElegantRL_Solver is a high-performance RL Solver.

We aim to find high-quality optimum, or even (nearly) global optimum, for nonconvex/nonlinear optimizations (continuous variables) and combinatorial optimizations (discrete variables). We provide pretrained neural networks to perform real-time inference for nonconvex optimization problems, including combinatorial optimization problems.

This project is built on [ElegantRL](https://github.com/AI4Finance-Foundation/ElegantRL) and OpenAI Gym.

The following two key technologies are under active development:

- Massively parallel simuations of gym-environments on GPU, using thousands of CUDA cores and tensor cores.
- Podracer scheduling on a GPU cloud, e.g., DGX-2 SuperPod.

Key references:

- Mazyavkina, Nina, et al. "Reinforcement learning for combinatorial optimization: A survey." Computers & Operations Research 134 (2021): 105400.
- Bengio, Yoshua, Andrea Lodi, and Antoine Prouvost. "Machine learning for combinatorial optimization: a methodological tour d'horizon." European Journal of Operational Research 290.2 (2021): 405-421.
- Makoviychuk, Viktor, et al. "Isaac Gym: High performance GPU based physics simulation for robot learning." Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2). 2021.
- Nair, Vinod, et al. "Solving mixed integer programs using neural networks." arXiv preprint arXiv:2012.13349 (2020).

Environments:

- MIMO Beamforming in 5G/6G.
- Classical NP-Hard problems.
- Classical Simulation of Quantum Circuits.
- Compressive Sensing.
- Portfolio Management.
- OR-Gym.

File Structure: ` -RLSolver -├── opt_methods -| ├──branch-and-bound.py -| └──cutting_plane.py -├── helloworld -| ├──maxcut.py -| ├──maxcut_env.py -└── rlsolver (main folder) - ├── envs - | ├── _base - | └── maxcut - | └── tsp - | ├── portfolio_management - |── rlsolver_learn2opt - | ├── mimo - | ├── tensor_train - └── utils - └──

graph_partitioning.py - └── graph_partitioning_gurobi.py - └── maxcut.py - └── maxcut_gurobi.py - └── tsp.py - └── tsp_gurobi.py `

ElegantRL_Solver features high-performance and stability:

High-performance: it can find high-quality optimum, or even (nearly) global optimum.

Stable: it leverages computing resource to implement the Hamiltonian-term as an add-on regularization to DRL algorithms. Such an add-on H-term utilizes computing power (can be computed in parallel on GPU) to search for the "minimum-energy state", corresponding to the stable state of a system.

1.32 Configuration: *config.py*

1.32.1 Arguments

The `Arguments` class contains all parameters of the training process, including environment setup, model training, model evaluation, and resource allocation. It provides users an unified interface to customize the training process.

The class should be initialized at the start of the training process. For example,

```
from elegantrl.train.config import Arguments
from elegantrl.agents.AgentPPO import AgentPPO
from elegantrl.train.config import build_env
import gym

args = Arguments(build_env('Pendulum-v1'), AgentPPO())
```

The full list of parameters in `Arguments`:

1.32.2 Environment registration

elegantrl.train.config.**build_env**(*env_class=None*, *env_args: Optional[dict] = None*, *gpu_id: int = -1*)

1.32.3 Utils

elegantrl.train.config.**kwargs_filter**(*function*, *kwargs: dict*) → dict

1.33 Run: *run.py*

In *run.py*, we provide functions to wrap the training (and evaluation) process.

In ElegantRL, users follow a **two-step procedure** to train an agent in a lightweight and automatic way.

1. Initializing the agent and environment, and setting hyper-parameters up in `Arguments`.
2. Passing the `Arguments` to functions for the training process, e.g., `train_and_evaluate` for single-process training and `train_and_evaluate_mp` for multi-process training.

Let's look at a demo for the simple two-step procedure.

```
from elegantrl.train.config import Arguments
from elegantrl.train.run import train_and_evaluate, train_and_evaluate_mp
from elegantrl.envs.Chasing import ChasingEnv
from elegantrl.agents.AgentPPO import AgentPPO

# Step 1
args = Arguments(agent=AgentPPO(), env_func=ChasingEnv)

# Step 2
train_and_evaluate_mp(args)
```

1.33.1 Single-process

1.33.2 Multi-process

1.33.3 Utils

1.34 Worker: *worker.py*

Deep reinforcement learning (DRL) employs a trial-and-error manner to collect training data (transitions) from agent-environment interactions, along with the learning procedure. ElegantRL utilizes `Worker` to generate transitions and achieves worker parallelism, thus greatly speeding up the data collection.

1.34.1 Implementations

1.35 Replay Buffer: *replay_buffer.py*

ElegantRL provides `ReplayBuffer` to store sampled transitions.

In ElegantRL, we utilize `Worker` for exploration (data sampling) and `Learner` for exploitation (model learning), and we view such a relationship as a "producer-consumer" model, where a worker produces transitions and a learner consumes, and a learner updates the actor net at worker to produce new transitions. In this case, the `ReplayBuffer` is the storage buffer that connects the worker and learner.

Each transition is in a format (state, (reward, done, action)).

Note: We allocate the `ReplayBuffer` on continuous RAM for high performance training. Since the collected transitions are packed in sequence, the addressing speed increases dramatically when a learner randomly samples a batch of transitions.

1.35.1 Implementations

class elegantrl.train.replay_buffer.**ReplayBuffer**(*max_size: int*, *state_dim: int*, *action_dim: int*, *gpu_id: int = 0*, *num_seqs: int = 1*, *if_use_per: bool = False*, *args: ~elegantrl.train.config.Config = <elegantrl.train.config.Config object>*)

device

The struction of ReplayBuffer (for example, num_seqs = num_workers * num_envs == 2*4 = 8 ReplayBuffer: worker0 for env0: sequence of sub_env0.0 self.states = Tensor[s, s, …, s, …, s]

self.actions = Tensor[a, a, …, a, …, a] self.rewards = Tensor[r, r, …, r, …, r] self.undones = Tensor[d, d, …, d, …, d]

<——max_size——> <-cur_size->

↑ pointer

sequence of sub_env0.1 s, s, …, s a, a, …, a r, r, …, r d, d, …, d sequence of sub_env0.2 s, s, …, s a, a, …, a r, r, …, r d, d, …, d sequence of sub_env0.3 s, s, …, s a, a, …, a r, r, …, r d, d, …, d

worker1 for env1: sequence of sub_env1.0 s, s, …, s a, a, …, a r, r, …, r d, d, …, d

sequence of sub_env1.1 s, s, …, s a, a, …, a r, r, …, r d, d, …, d sequence of sub_env1.2 s, s, …, s a, a, …, a r, r, …, r d, d, …, d sequence of sub_env1.3 s, s, …, s a, a, …, a r, r, …, r d, d, …, d

D: done=True d: done=False sequence of transition: s-a-r-d, s-a-r-d, s-a-r-D s-a-r-d, s-a-r-d, s-a-r-d, s-a-r-d, s-a-r-D s-a-r-d, …

<——trajectory——-> <————trajectory————————> <————

per_beta

PER. Prioritized Experience Replay. Section 4 alpha, beta = 0.7, 0.5 for rank-based variant alpha, beta = 0.6, 0.4 for proportional variant

1.35.2 Multiprocessing

1.35.3 Initialization

1.35.4 Utils

1.36 Evaluator: *evaluator.py*

In the course of training, ElegantRL provide an `evaluator` to periodically evaluate agent's performance and save models.

For agent evaluation, the evaluator runs agent's actor (policy) network on the testing environment and outputs corresponding scores. Commonly used performance metrics are mean and variance of episodic rewards. The score is useful in following two cases:

- Case 1: the score serves as a goal signal. When the score reaches the target score, it means that the goal of the task is achieved.
- Case 2: the score serves as a criterion to determine overfitting of models. When the score continuously drops, we can terminate the training process early to mitigate the performance collapse and the waste of computing power brought by overfitting.

Note: ElegantRL supports a tournament-based ensemble training scheme to empower the population-based training (PBT). We maintain a leaderboard to keep track of agents with high scores and then perform a tournament-based evolution among these agents. In this case, the score from the evaluator serves as a metric for leaderboard.

For model saving, the evaluator saves following three types of files:

- actor.pth: actor (policy) network of the agent.
- plot_learning_curve.jpg: learning curve of the agent.
- recorder.npy: log file, including total training steps, reward average, reward standard deviation, reward exp, actor loss, and critic loss.

We implement the `evaluator` as a microservice, which can be ran as an independent process. When an evaluator is running, it can automatically monitors parallel agents, and provide evaluation when any agent needs, and communicate agent information with the leaderboard.

1.36.1 Implementations

class elegantrl.train.evaluator.**Evaluator**(*cwd: str*, *env*, *args: Config*, *if_tensorboard: bool = False*)

1.36.2 Utils

1.37 FAQ

Version
1.0

Date
12-31-2021

Contributors
Steven Li, Xiao-Yang Liu

1.37.1 Description

This document contains the most frequently asked questions related to the ElegantRL Library, based on questions posted on the slack channels and Github issues.

1.37.2 Outline

- *Section 5 Common issues/bugs*

1.37.3 Section 1 Where to start?

- Get started with ElegantRL-helloworld, a lightweight and stable subset of ElegantRL.
  - Read the introductary post of ElegantRL-helloworld.
  - Read the post to learn how an algorithm is implemented.
  - Read the posts (Part I, Part II) to learn a demo of ElegantRL-helloworld on a stock trading task.
- Read the post and the paper that describe our cloud solution, ElegantRL-Podracer.
- Run the Colab-based notebooks on simple Gym environments.
- Install the library following the instructions at the official Github repo.
- Run the demos from MuJoCo to Isaac Gym provided in the library folder.
- Enter on the AI4Finance slack.

1.37.4 Section 2 What to do when you experience problems?

- If any questions arise, please follow this sequence of activities:
  - Check if it is not already answered on this FAQ
  - Check if it is not posted on the Github repo issues.
  - If you cannot find your question, please report it as a new issue or ask it on the AI4Finance slack (Our members will get to you ASAP).

1.37.5 Section 3 Most frequently asked questions related to the ElegantRL Library

- ElegantRL supports any gym-style environment and provides wrappers for MuJoCo and Isaac Gym.
- You can use VecEnv imported from Isaac Gym or write your own VecEnv by yourself. There is no VecEnv wrapper to process a non-VecEnv to VecEnv.
- It is a tutorial-level implementation for users (e.g., beginners) who do not have a demand for parallel computing.
- In the folder, we currently have DQN, DDQN, DDPG, TD3, SAC, A2C, REDQ, and PPO.
- ElegantRL support parallelism of DRL algorithms at multiple levels, including agent parallelism of population-based training and worker-learner parallelism of a single agent.
- Agent parallelism is to train hundreds of agents in parallel through population-based training (PBT), which offers a flexibility for ensemble methods.

- Worker parallelism is to generate transitions in parallel, thus accelerating the data collection. We currently support two different parallelism to adapt different types of environments.
  - use a VecEnv to generate transitions in batch.
  - if the environment is not a VecEnv, use multiple workers to generate transitions in parallel.
- Learner parallelism is to train multiple-critics and multiple actors running in parallel for ensemble DRL methods. Due to the stochastic nature of the training process (e.g., random seeds), an ensemble DRL algorithm increases the diversity of the data collection, improves the stability of the learning process, and reduces the overestimation bias.
- We currently support three ensemble methods, which are weighted average, model fusion, and tournament-based ensemble training scheme.
- Tournament-based ensemble training scheme is our cloud orchestration mechanism, scheduling the interactions between a leaderboard and a training pool with hundreds of agents (pods). More details are available in the post and the paper.
- Yes, you can load a model to continue the training. A tutorial is coming soon.
- No, we cannot support Tensorboard.
- Yes, we are implementing MARL algorithms and adapting them to the massively parallel framework. Currently, we provide several MARL algorithms, such as QMix, MADDPG, MAPPO, and VDN. The tutorials are coming soon.
- ElegantRL supports flexible resource allocation from zero to hundreds of GPUs.
- Of course! You can use ElegantRL-helloworld for non-GPU training or use ElegantRL by setting GPU_ids to None (you cannot use GPU-accelerated VecEnv in this case).
- You can participate on the slack channels, check the current issues and the roadmap, and help any way you can (sharing the library with others, testing the library of different applications, contributing with code development, etc).

1.37.6 Section 4 References for diving deep into Deep Reinforcement Learning (DRL)

1.37.7 Subsection 4.1 Open-source softwares and materials

- **OpenAI Gym**
  https://gym.openai.com/
- **MuJoCo**
  https://mujoco.org/
- **Isaac Gym**
  https://developer.nvidia.com/isaac-gym
- **OpenAI Spinning Up**
  https://spinningup.openai.com/en/latest/
- **Stable Baselines3**
  https://github.com/DLR-RM/stable-baselines3
- **Ray RLlib**
  https://docs.ray.io/en/master/rllib.html
- **Tianshou**
  https://github.com/thu-ml/tianshou
- **ChainerRL**
  https://github.com/chainer/chainerrl
- **MushroomRL**
  https://github.com/MushroomRL/mushroom-rl/tree/master
- **ACME**
  https://github.com/deepmind/acme
- **PFRL**
  https://github.com/pfnet/pfrl
- **SURREAL**
  https://github.com/SurrealAI/surreal
- **rlpyt**
  https://github.com/astooke/rlpyt
- **MAlib**
  https://github.com/sjtu-marl/malib
- **Policy gradient algorithms**
  https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

1.37.8 Subsection 4.2 DRL algorithms

- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. *Mastering the game of Go without human knowledge*. Nature, 550(7676):354–359, 2017.
- V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. *Playing atari with deep reinforcement learning*. ArXiv, abs/1312.5602, 2013.
- H. V. Hasselt, Arthur Guez, and David Silver. *Deep reinforcement learning with double q-learning*. ArXiv, abs/1509.06461, 2016.
- Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. *Continuous control with deep reinforcement learning*. In ICLR, 2016.
- J. Schulman, F. Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal policy optimization algorithms*. ArXiv, abs/1707.06347, 2017.
- Matteo Hessel, Joseph Modayil, H. V. Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan,Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. *Rainbow: Combining improvements in deepreinforcement learning*. In AAAI, 2018.
- Scott Fujimoto, Herke Hoof, and David Meger. *Addressing function approximation error in actor-critic methods*. In International Conference on Machine Learning, pages 1587–1596. PMLR, 2018.
- Tuomas Haarnoja, Aurick Zhou, P. Abbeel, and Sergey Levine. *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*. In ICML, 2018.
- Xinyue Chen, Che Wang, Zijian Zhou, and Keith W. Ross. *Randomized ensembled double q-learning: Learning fast without a model*. In International Conference on Learning Representations, 2021.

1.37.9 Subsection 4.2 Other resources

- Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. IEEE Transactions on Neural Networks, 16:285–286, 2005.
- Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charlie Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. *Massively parallel methods for deep reinforcement learning*. ArXiv, abs/1507.04296, 2015.
- Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. *Ray: A distributed framework for emerging ai applications*. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 561–577, 2018.
- Lasse Espeholt, Rapha¨el Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. *Seed rl: Scalable and efficient deep-rl with accelerated central inference*. In International Conference on Machine Learning. PMLR, 2019.
- Agrim Gupta, Silvio Savarese, Surya Ganguli, and Fei-Fei Li. *Embodied intelligence via learning and evolution*. Nature Communications, 2021.
- Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. *Podracer architectures for scalable reinforcement learning*. arXiv preprint arXiv:2104.06272, 2021.
- Zechu Li, Xiao-Yang Liu, Jiahao Zheng, Zhaoran Wang, Anwar Walid, and Jian Guo. *FinRL-podracer: High performance and scalable deep reinforcement learning for quantitative finance*. ACM International Conference on AI in Finance (ICAIF), 2021.

- Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. *Learning to walk in minutes using massively parallel deep reinforcement learning*. In Conference on Robot Learning, 2021.
- Brijen Thananjeyan, Kirthevasan Kandasamy, Ion Stoica, Michael I. Jordan, Ken Goldberg, and Joseph Gonzalez. *Resource allocation in multi-armed bandit exploration: Overcoming nonlinear scaling with adaptive parallelism*. In ICML, 2021.

1.37.10 Section 5 Common issues/bugs

- **When running Isaac Gym, found error *ImportError: libpython3.7m.so.1.0: cannot open shared object file: No such file or directory*:**
  Run the following code in bash to add the path of Isaac Gym conda environment.

  export LD_LIBRARY_PATH=$PATH$

  For example, the name of Isaac Gym conda environment is rlgpu:

  export LD_LIBRARY_PATH=/xfs/home/podracer_steven/anaconda3/envs/rlgpu/lib

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

S

U