Lamarckian Platform: Pushing the Boundaries of Evolutionary Reinforcement Learning towards Asynchronous Commercial Games

Hui Bai, Ruimin Shen, Yue Lin, Botian Xu, and Ran Cheng, Senior Member, IEEE

Abstract—Despite the emerging progress of integrating evolutionary computation into reinforcement learning, the absence of a high-performance platform endowing composability and massive parallelism causes non-trivial difficulties for research and applications related to asynchronous commercial games. Here we introduce Lamarckian - an open-source platform featuring support for evolutionary reinforcement learning scalable to distributed computing resources. To improve the training speed and data efficiency, Lamarckian adopts optimized communication methods and an asynchronous evolutionary reinforcement learning workflow. To meet the demand for an asynchronous interface by commercial games and various methods, Lamarckian tailors an asynchronous Markov Decision Process interface and designs an object-oriented software architecture with decoupled modules. In comparison with the state-of-the-art RLlib, we empirically demonstrate the unique advantages of Lamarckian on benchmark tests with up to 6000 CPU cores: i) both the sampling efficiency and training speed are doubled when running PPO on Google football game; ii) the training speed is 13 times faster when running PBT+PPO on Pong game. Moreover, we also present two use cases: i) how Lamarckian is applied to generating behavior-diverse game AI; ii) how Lamarckian is applied to game balancing tests for an asynchronous commercial game.

Index Terms—reinforcement learning, evolutionary computation, evolutionary reinforcement learning, asynchronous commercial games, platform.

I. INTRODUCTION

Reinforcement learning (RL), as a powerful tool for sequential decision-making, has achieved remarkable successes in a number of challenging tasks varying from board games [1], arcade games [2], robot control [3], scheduling problems [4] to autonomous driving [5]. Despite that RL algorithms have been widely assessed on game benchmarks (e.g., Atari games [6], ViZDoom [7], and DeepMind Lab [8]), the applications of RL in commercial games (e.g., StarCraft I [9] & II [10],

H. Bai and R. Shen contribute equally to this work.

H. Bai, B. Xu, and R. Cheng are with the Guangdong Key Laboratory of Brain-Inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China. E-mail: huibaimonky@163.com, btx0424@outlook.com, ranchengcn@gmail.com. (Corresponding author: Ran Cheng)

R. Shen and Y. Lin are with the NetEase Games AI Lab, Guangzhou 510653, China. E-mail: shenruimin@corp.netease.com, gzlinyue@corp.netease.com.

This work was supported by the National Natural Science Foundation of China (No. 61906081), the Shenzhen Science and Technology Program (No. RCBS20200714114817264), the Guangdong Provincial Key Laboratory (No. 2020B121201001), and the Program for Guangdong Introducing Innovative and Entrepreneurial Teams (Grant No. 2017ZT07X386).

 $^{\rm l}$ The code and demonstrational setup of Lamarckian are publicly available at https://github. com/lamarckian/lamarckian.

and Dota2 [11]) have raised new issues to be considered, e.g., partially observed maps, large state space and action space, delayed credit assignment, etc. Besides, when applying RL to real-world scenarios, there are also a number of technical challenges such as brittle convergence properties caused by sensitive hyperparameters, temporal credit assignment with long time horizons and sparse rewards, difficult credit assignments in multi-agent reinforcement learning, lack of diverse exploration, a set of conflicting objectives for rewards, etc.

To meet the above challenges, recently, there has been an emerging progress in integrating RL with evolutionary computation (EC) to address the above challenges. In firstperson multiplayer games, the Population Based Training (PBT) trains a population of agents to dynamically optimize hyperparameters for self-play [12]. In various benchmark games of OpenAI Gym [6], the Evolutionary Reinforcement Learning [13] and Collaborative Evolutionary Reinforcement Learning [14] take the advantages of EC to make up for some deficiencies in RL, such as difficult credit assignment, lack of effective exploration, and brittle convergence through a fitness metric. EMOGI generates desirable styles of game artificial intelligence (AI) by a multi-objective EC algorithm [15]. Wuji combines EC and RL for automated game testing to detect game bugs by exploring states as much as possible [16]. Besides, under the names of neuroevolution/derivativefree reinforcement learning, Evolution strategies (ESs) and Genetic Algorithms (GAs) have been used to optimize policy networks, which avoids the gradient vanishing problem [17]. In brief, the literature has demonstrated promising potentials for integrating EC to RL, despite that the development of related research and application is still in its infancy.

While reinforcement learning enjoys the advances of several state-of-the-art platforms or frameworks [18], [19], [20], [21], the literature is still in the absence of a platform or framework specifically tailored to evolutionary reinforcement learning (EvoRL)². Moreover, current distributed platforms do not use computational resources efficiently, and thus the training of RL is extremely time-consuming for complex commercial games. Therefore, Lamarckian exactly meets the rigid demand for such a highly decoupled, high-performance, scalable implementation of a distributed architecture, to support research and engineering in EvoRL. Most importantly, Lamarckian fully supports the implementations of evolutionary multi-objective

²In this work, the general *evolutionary reinforcement learning* is abbreviated as *EvoRL* for short.

optimization, which has demonstrated promising potentials in solving specific RL problems involving multiple objectives to be considered simultaneously [22], [23].

Intrinsically, EvoRL can be seen as an evolutionary distributed RL paradigm requiring delicate management of computing resources for asynchronous training. Hence, in addition to meeting the rigid demand of EvoRL as mentioned above, Lamarckian is also dedicated to improving data efficiency and training speed of general distributed RL. Despite that some recent works have successfully scaled RL methods (e.g. PPO [24] and IMPALA [25]) to large-scale distributed computing systems [11], the policy-lag [25] is still an open issue. In distributed RL, policy-lag happens when a learner policy is several updates ahead of an actor's policy when an update occurs, thus causing severe defects in convergence and stability. To address the issue of policy-lag, methods such as clipped surrogate objective [24] and V-trace [25] have been proposed. Apart from the delicate designs from methodology perspective, it has been evidenced that the improvements from the engineering perspective could also significantly alleviate policy-lag, as did in the case of SEED RL [26]. Specifically, Lamarckian is tailored for such a purpose from two aspects. First, an asynchronous tree-shaped data broadcasting method is proposed to reduce the policy-lag and alleviate the communication bottleneck of learners. Second, having inherited high scalability of the state-of-the-art Ray [27], Lamarckian further improves the efficiency of distributed computing and increases throughput by coupling Ray with ZeroMQ.

From the engineering perspective, game environmental interfaces provide a system of API components allowing players to interact with the game story and break into the game space. For example, the interfaces receive actions from players and return the next state of the game environment to the players. In this paper, game environmental interfaces can be divided into two types: synchronous interfaces and asynchronous interfaces. In synchronous interfaces (e.g., OpenAI Gym), a tick moves to the next tick only when it receives actions from all players. In asynchronous interfaces (e.g., most commercial games), each player is controlled independently and asynchronously. Consequently, if a player is off-line, the other players and the main tick will continue moving without waiting. To meet such requirements of asynchronous commercial game environments, players with different roles are expected to interact with the environment independently. Thus, Lamarckian is designed on the basis of an asynchronous Markov Decision Process (MDP, e.g., modeled by game playing) interface, which is highly decoupled by object-oriented programming. With these tailored designs, Lamarckian is shipped with a number of representative algorithms in EC and RL algorithms as summarized in TABLE I. By simple configurations, EC algorithms can be easily integrated with RL algorithms in Lamarckian. Meanwhile, Lamarckian has good scalability in terms of adding new algorithms, new environments, or new DNN models. In brief, our main contributions of Lamarckian can be summarized as:

- A highly decoupled, high-performance, scalable platform tailored for EvoRL is delivered.
- To accelerate training speed in large-scale distributed

- computing, two methods are proposed from the engineering perspectives: i) broadcasting data from the learner to actors by an asynchronous tree-shaped data broadcasting method; and ii) coupling high scalability of Ray with high efficiency of ZeroMQ.
- A highly decoupled, asynchronous MDP interface is designed for asynchronous commercial game environments.
- An object-oriented software architecture with decoupled modules is developed to support various problems and algorithms in RL and EC.

The organization of the paper is as follows. Section II gives the background and related work. Section III describes the asynchronous distributed designs. Section IV provides the benchmark experiments, and Section V provides two use cases. Finally, Section VI concludes the paper.

TABLE I
THE MAIN-STREAM ALGORITHMS AND BENCHMARKS IMPLEMENTED IN
LAMARCKIAN.

Modules	Implementations			
RL algorithms EvoRL algorithms RL benchmarks EC algorithms EC benchmarks	A3C [28], PPO [24], IMPALA [25], and DQN [29], etc. PBT+self-play [12], EMOGI [15], Wuji [16], etc. OpenAI Gym [6], Google football [30], etc. single-objective GA [31], multi-objective NSGA-II [32], etc. multi-objective DTLZ [33] and ZDT [34], etc.			

II. BACKGROUND AND RELATED WORK

In this section, we first present the definition of MDP and the formulation of RL in Section II-A, and then describe EC and EvoRL in Section II-B. Next, we discuss the previous RL platforms and frameworks in Section II-C. Finally, we summarise and discuss essential terminologies in Section II-D.

A. RL and MDP

RL is a special branch of AI considering the interactions between agents and an environment towards maximum rewards. The problem that an agent acts in a stochastic environment by sequentially choosing actions over a sequence of time steps to maximise a cumulated reward can be modeled as a Markov Decision Process.

Definition 1. (Markov Decision Process) An MDP is usually defined as $\langle S, A, T, R, \rho_0, \gamma \rangle$, with a state space S, an action space A, a stochastic transition function $T: S \times A \to P(S)$ representing the probability distribution over possible next states, a reward function $R: S \times A \to \mathbb{R}$, an initial state distribution $\rho_0: S \to \mathbb{R}_{\in [0,1]}$, and a discount factor $\gamma \in [0,1)$.

At each discrete time step t, given the current state $s_t \in S$, the agent selects actions $a_t \in A$ according to its policy π_θ : $S \to P(A)$, where P(A) is the set of probability measures on A and $\theta \in \mathbb{R}^n$ is a vector of n parameters, and $\pi_\theta(a_t|s_t)$ is the conditional probability density at a_t given the input s_t associated with the policy. The agent's objective is to learn a policy to maximize the expected cumulative discounted reward from the start state:

$$J(\pi) = \mathbb{E}_{\rho_0, \pi, T} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right], \tag{1}$$

where $s_0 \sim \rho_0(s_0)$, $a_t \sim \pi(s_t)$, $s_{t+1} \sim T(\cdot | s_t, a_t)$, and $r_t = R(s_t, a_t)$.

In practice, asynchronous commercial games often involve multiple players, where the main tick will continue moving without waiting off-line players. In this case, an asynchronous MDP interface will be required for dealing with the interactions between agents and the environment.

B. EC and EvoRL

Evolutionary computation (EC) generally refers to the family of population-based stochastic optimization algorithms (e.g., Population Based Training (PBT) [35], Evolutionary Strategy (ES) [1], Genetic Algorithm (GA) [36], etc.) inspired by natural evolution. Specifically, an EC algorithm first initializes a population of candidate solutions, and then it enters an iterative loop: the candidate solutions in the current population are pairwisely mated to undergo the permutation operations (i.e., crossover and mutation) to generate new offspring candidate solutions; the offspring candidate solutions are evaluated by task-related performance indicators to obtain their fitness values (a.k.a. objective values); the offspring candidate solutions are merged with the ones in the current population to be selected for the next iteration (a.k.a generation). After a number of generations as above, ideally, an EC algorithm will end up with a population of candidate solutions approximating the global optima of the optimization problems as given.

The optimization problems in RL tasks often involve complex characteristics, while EC has been found to be a powerful tool for dealing with them [17]. On the one hand, EC algorithms require no gradient information and is widely applicable to problems without explicit objective functions by quality diversity (QD) [37] or novelty search (NS) [38]. On the other hand, thanks to the population-based nature, EC algorithms are inherently robust to dynamic changes that widely exist in real-world applications of RL (e.g., sim-to-real transfer in robot control [39]).

As an emerging research direction of RL, EvoRL is exactly dedicated to meeting the challenges of various key research problems in RL research by dealing the complex optimization problems as involved, including but not limited to policy search [40], reward shaping [41], exploration [13], hyperparameter optimization [12], meta-RL [42], multi-objective RL [43], etc.

TABLE II
SUMMARY OF LAMARCKIAN AND THREE REPRESENTATIVE DISTRIBUTED REINFORCEMENT LEARNING FRAMEWORKS OR PLATFORMS.

Frameworks /Platforms	Single Agent	Multi- Agent	Tailored EvoRL Framework/Workflow	Asynchronous MDP Interface
TLeague [44]	√	√	×	×
Acme [45]	✓	✓	×	×
RLlib [21]	✓	✓	×	×
Lamarckian	\checkmark	\checkmark	✓	✓

C. RL Platforms and Frameworks

Since an RL agent is trained on a large number of samples generated by interacting with its environment, the training speed is highly related to the sampling efficiency. Hence, the sampling efficiency is an important indicator to measure RL frameworks or platforms [21], [45]. This motivates the distributed framework to sample in parallel, where each of the multiple actors interacts with its environment independently. Except for the sampling efficiency, the sample staleness reflecting the policy-lag is another essential indicator [11]. To keep learners and actors as consistent as possible, various synchronous methods are employed in state-of-the-art distributed frameworks and platforms. Specially, RLlib uses the synchronous data broadcasting where the learner stops policy optimization when sending data to actors [21]. Acme applies the data storage system Reverb and its rate limiter to control the policy-lag, where the rate limiter will block faster processes until slower processes catch up [45]. However, since both RLlib and Acme may have blocking or waiting processes for synchronization, they are low-efficient in largescale scenarios.

From the engineering perspective, most existing reinforcement learning platforms or frameworks tend to scale at long-running program replicas for distributed execution [18], [19], [20], [46], [45], thus causing difficulties in generalizing to complex architectures. In contrast, RLlib scales well at short-running tasks by a Ray-based hierarchical control model [21]. Despite the high scalability, however, RLlib suffers slow training efficiencies on large-scale distributed computing environments, and therefore RLlib cannot generalize well to complex AI systems requiring highly parallel data transmission between the learner and actors, such as OpenAI Five [11].

Among other state-of-the-art frameworks and platforms, SEED RL [26] and TorchBeast [20] are two high-performance scalable implementations of IMPALA [25]. SURREAL [47] focuses on continuous control agents in robot manipulation tasks by a distributed training framework. The recently proposed frameworks, e.g., Arena [48], TLeague [44], and MALib [49], target at multi-agent reinforcement learning.

Despite the various platforms or frameworks as introduced above, none of them is highly parallel or tailored for evolutionary reinforcement learning, particularly, facing the applications of asynchronous commercial games. TABLE II summarizes the main features of Lamarckian, in comparison with several representative RL platforms or frameworks from four aspects.

D. Discussions

From the perspective of engineering designs, an asynchronous system has a non-blocking architecture where multiple operations can run concurrently without waiting others to complete; a distributed system is a computing environment where independent components run on different machines to achieve a common goal. Hence, distributed RL and asynchronous RL refer to engineering implementations of RL in the system level.

On the contrary, *EvoRL* aims to adopt EC methods to deal with the challenging issues for RL in the *methodology level*. Nonetheless, considering the population-based property of EvoRL, delicate *asynchronous* system designs are particularly important for the high performance of EvoRL.

Therefore, we are motivated to improve efficiency for EvoRL as well as conventional RL by adopting a series of asynchronous system designs, including: asynchronous distributed EvoRL workflow allowing several operations of EvoRL to execute in different machines concurrently without communication or information exchange, asynchronous sampling allowing a learner to continue learning when sending its policy to actors; asynchronous data broadcasting allowing the simultaneous data transmission and reception mode based on asynchronous sampling, and asynchronous MDP interfaces compatible with asynchronous commercial games.

III. ASYNCHRONOUS DISTRIBUTED DESIGNS

To provide highly parallel and scalable implementations of distributed evolutionary reinforcement learning algorithms, Lamarckian is designed to be asynchronous distributed by considering four main aspects: the distributed EvoRL workflow in Section III-A, the acceleration of distributed computing in Section III-B, the asynchronous MDP interface in Section III-C, and the object-oriented software architecture in Section III-D.

A. Distributed EvoRL Workflow

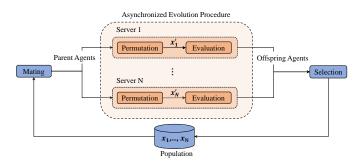


Fig. 1. Distributed workflow of EvoRL. In the *Asynchronous Evolution Procedure*, each pair of candidate solutions (a.k.a parents) are distributed in an independent server to conduct permutation and evaluation in an asynchronous manner.

Existing RL platforms merely support very limited singleobjective EC algorithms such as PBT. However, On the one hand, there are rich scenarios involving multiple (instead of single) objectives to be optimized simultaneously in RL tasks; on the other hand, EvoRL requires tailored workflow design for general high-performance implementations.

Fig. 1 is the overview of the proposed distributed workflow for EvoRL: first, a population of candidate solutions (\mathbf{x}_1 , \mathbf{x}_2 , ..., \mathbf{x}_N) is initialized; then, each candidate solution enters the mating process to generate parent pairs; afterward, permutation operations are performed on each pair of mated candidate solutions to generate offspring candidate solution; finally, the offspring candidate solution is evaluated by an evaluator encapsulating configurable RL algorithms or other task-related performance indicators.

Considering computational efficiencies, the evolution workflow is asynchronous on each step, i.e., permutation, training, and evaluation. Particularly, after evaluation, each offspring candidate solution is sent to the collector server independently. Once the expected number of offspring candidates solutions are obtained, a synchronized selection will be performed to have the promising candidate solutions survive to the next generation.

B. Acceleration of Distributed Computing

On top of the proposed distributed workflow, we further improve the computing efficiency from two engineering perspectives.

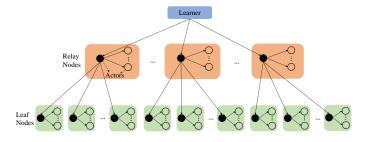


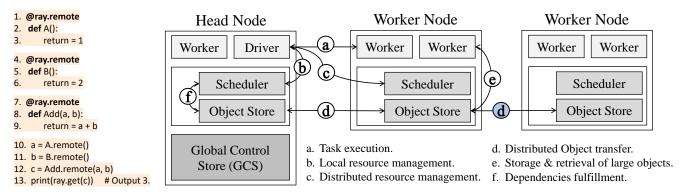
Fig. 2. Asynchronous tree-shaped data broadcasting. First, all actors are grouped by the machines where they are located. In the same machine (i.e. each box), actors fetch data by high-performance communication. Then, the learner sends data to relay nodes. Finally, each relay node sends data to its connected leaf nodes.

1) Asynchronous Tree-shaped Data Broadcasting: It is common in an RL algorithm to require estimating a term in the form of $\mathbf{E}_{\tau \sim \pi} \left[f(\tau) \right]$, where each $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$ is a trajectory generated by carrying out the policy π . Since the expectation is taken under the distribution induced by τ , it is expected for on-policy RL algorithms that the samples used for training should be as up-to-date as possible to give an unbiased estimate.

For large-scale on-policy RL, when a learner is associated with a large number of actors, a learner policy is potentially several updates ahead of an actor's policy when an update occurs, thus causing the policy-lag phenomenon. Since shorter data broadcasting time will directly lead to a smaller policy-lag for asynchronous broadcasting, we propose a tree-shaped data broadcasting method shown in Fig. 2.

The tree-shaped architecture is built automatically and implicitly by getting available computational resources. The architecture is a Complete N-ary Tree with no more than three layers for nodes of any scale. Hence, the out-degree of each node is approximated by \sqrt{n} (n is the number of nodes). Each node represents a machine with multiple CPU cores and/or GPUs. The GPU is given priority to the learner for a large amount of model inference. In Fig. 2, we describe the process of data broadcasting in three steps: first, all actors are grouped by the machines where they are located, and in the same machine, actors fetch data by high-performance communication (e.g., Inter-Process Communication or Shared Memory); then, the learner sends data to relay nodes; finally, each relay node sends data to its connected leaf nodes.

With a flat layout, directly broadcasting to n nodes would incur O(n) traffic, and the head node (usually the learner) becomes a bandwidth bottleneck. In the worst case, it suffers an O(n) delay until the data to be received by all nodes. But with a tree-shaped structure, it faces only $O(\sqrt{n})$ traffic,



(a) Code rewriting by Ray.

(b) A Ray cluster.

Fig. 3. (a) is an example of using Ray to rewrite the code that outputs the result of a+b. (b) is an example of a Ray cluster. The cluster architecture consists of a head node and two worker nodes, where the head node has a global control store (GCS) managing the system metadata, and each node has its local worker, scheduler, object store. The protocols between these components are presented in "a" to "f" in circles.

and the second-layer relay nodes can operate in parallel. Consequently, this results in a much smaller $O(\sqrt{n})$ delay. The effectiveness of this improvement is demonstrated in the experiment section.

2) Ray plus ZeroMQ: Ray is a distributed computing framework for the easy scaling of Python programs, which provides a simple and universal API and only requires minimum modification to build distributed applications. An example of using Ray to rewrite the code that outputs the result of a+b is provided in Fig. 3a, where the functions execute as parallel and remote tasks by using the ray.remote primitive. The tasks can distribute in different nodes in a Ray cluster as in Fig. 3b, where the Add() task executes in the head node, and the A() and B() tasks execute in the worker nodes, respectively. The diagram illustrates an example of the cluster architecture and protocols in brief. The head node has a global control store (GCS) managing the system metadata. Each node has its local worker, scheduler, and object store. The protocols between these components, as presented in "a" to "f" in the subfigure, are mostly over the Remote Procedure Call (RPC) framework of gRPC. The details of Ray can refer to the public document.

Since the HTTP/2-based gRPC has large overhead, latency and complexity of request/response chain in large-scale scenarios, the communication efficiency is quite limited. In contrast, ZeroMO is a powerful messaging library featuring high-throughput and low latency. Moreover, it is recognized that ZeroMQ is faster and more stable than HTTP/2 in communication. Thus ZeroMQ is employed as transports in Ray to construct highly efficient distributed computing framework for Lamarckian. Ray is used for creating and managing remote objects (e.g., workers that collect trajectories for training), and spawning processes correspondingly. The underlying communication is through ZeroMQ sockets that implement various messaging patterns, for example, Publish-Subscription for broadcasting model weights and Push-Pull for distributing rollout tasks. Benefiting from ZeroMQ, the large object transfer between worker nodes is enabled to support data transfer between relay nodes and leaf nodes in the treeshaped data broadcasting, as demonstrated by "d" in Fig. 3b.

C. Asynchronous MDP Interface

states = env.reset()
while True:

To achieve high performance, flexible assembly, and scalability, Lamarckian adopts the asynchronous MDP interface with highly decoupled properties.

```
    actions = [agent(state) for state, agent in zip(states, agents)]
    states, rewards, done, info = env.step(actions)
    if done:

            break

    (a) Synchronous MDP interface of Gym.
    async def train(controller, agent):

            state = controller.get state()
```

```
3.
         rewards = []
4.
    while True:
         script = agent(*state['inputs'])
         for action in script:
             exp = await controller(discrete=action)
         state = controller.get_state()
         reward = controller.get_reward()
10.
         rewards.append(reward)
11.
         if exp['done']:
12.
             break
13.
         return rewards
14. async def evaluate(controller, agent):
         state = controller.get_state()
15.
         with torch.no grad():
16.
17.
         while True:
             script = agent(*state['inputs'])
18.
19.
             for action in script:
                 exp = await controller(discrete=action)
20.
21.
                 state = controller.get state()
             if exp['done']:
22.
23.
                 break
24. mdp = MDP()
25. battle = mdp.reset(0, 1)
                                 # A battle with double agents(id=0, 1).
26. with contextlib.closing(battle):
         loop = asyncio.get_event_loop()
27.
         rewards = loop.run until complete(asyncio.gather(
             train(battle.controllers[0], agent0).
                                                       # Create trainina Coroutine.
             evaluate(battle.controllers[1], agent1), # Create sparring Coroutine.
              battle.ticks
```

(b) Asynchronous MDP interface of Lamarckian.

Fig. 4. During the interaction between the environment and agents: the synchronous MDP interface (a) is highly coupled, thus having poor compatibility with asynchronous commercial games; in contrast, the asynchronous MDP interface (b) decouples the agents with different functions, using Coroutine-based controllers to control agents asynchronously.

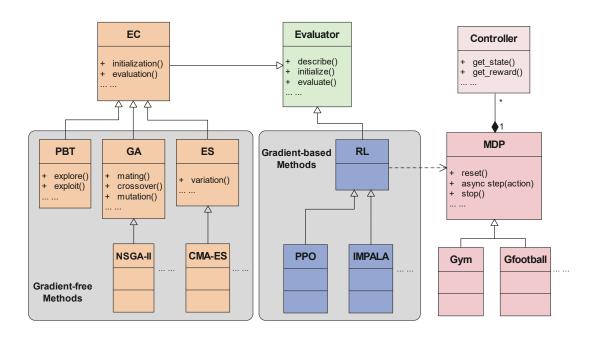


Fig. 5. The object-oriented software architecture of Lamarckian: EC module provides a general interface for EC algorithms; Evaluator module provides an unified abstract interface for EC module and RL module; MDP module provides an asynchronous interface that decouples the agents with different functions, using Coroutine-based controllers to control agents asynchronously.

Most existing MDP interfaces are based on the procedureoriented OpenAI Gym, where a code example is given in (Fig. 4a). The Gym-based MDP interfaces are synchronous in terms of two aspects: 1) the environment requires all generated actions from all agents before it moves a step (line 3); 2) the environment simultaneously returns states, rewards, and other information of all agents until the agents perform all actions (line 4). Despite their simple implementations, such Gym-based MDP interfaces suffer from three main limitations:

- poor compatibility with asynchronous commercial game environments;
- difficulties in implementing some specific RL techniques such as data skipping technique [50] and scripted actions [11];
- computational redundancies caused by close couplings.

To address the above limitations, we have designed an objectoriented, highly decoupled and asynchronous MDP interface (Fig. 4b). The proposed MDP interface is somehow similar to OpenAI Gym, but it discriminates training agents and sparring agents by decoupling (line 28), where the training agent and the sparring agent can be respectively implemented in two functions (i.e., the *train* function and the *evaluate* function) according to their requirements (e.g., whether they need to return rewards or not). One can define multiple controllers for an MDP, where each controller controls an agent by a Coroutine asynchronously. In contrast with Thread, Coroutine is more efficient since it involves no scheduling overhead or synchronization overhead on the level of operating systems.

Meanwhile, the independence and asynchrony of the proposed MDP interface enable easy implementations of some specific RL techniques such as scripted actions (lines 5-

7 and 18-20). Moreover, the proposed MDP interface well supports multiple inputs of states (lines 5 and 18), multiple types of actions (e.g., discrete or continuous actions, lines 7 and 20), and multi-head value estimation [51]. Consequently, each agent can independently focus on its programming and execution without considering others.

We demonstrate how to design an asynchronous MDP by users. First, users should determine the types of agents according to their actions and returned information. Second, each type of agent is implemented in a separate function using Python Coroutine Syntax (e.g., async; await; asyncio.get_event_loop; asyncio.gather). Specially, as exemplified by the Gym codes in Fig. 4a, lines 3 and 4 are extended to line 2-10 in the *train* function or line 15-21 in the *evaluate* function. Third, in the *main* function (line 28), a training agent and a sparring agent are initialized and executed asynchronously, and then only the training agent returns the rewards.

D. Object-Oriented Software Architecture

As shown in Fig. 5 by UML [52], Lamarckian is designed on the basis of an object-oriented software architecture comprehensively covering key modules involved in EvoRL. In the subsection, we will introduce each module one by one.

Evaluator module provides an unified abstract interface for **EC** module and **RL** module, which includes three general methods: describe(), initialize(), and evaluate(). Specifically, describe() returns the coding of a candidate solution (e.g., integer coding, real coding, and neural network coding); initialize() initializes a candidate solution with specific coding schemes; evaluate() evaluates the performance of a candidate solution on specific objective/cost function(s).

Essentially, **Evaluator** provides a bridge between **EC** module and **RL** module such that gradient-free EC algorithms and gradient-based RL algorithms can work collaboratively in RL tasks.

EC module includes two general methods: *initialization()* and *evaluation()*, which provides a general interface for EC algorithms (e.g., PBT, GA, ES, etc.). Specifically, *initialization()* initializes a population of candidate solutions with corresponding coded decision variables³; *evaluation()* evaluates the performance of a population of candidate solutions by an instantiated evaluator.

PBT module includes two general methods explore() and exploit(). Implementation of any PBT-like algorithm falls into this module.

GA module includes three general methods mating(), crossover() and mutation(). Implementation of any GA based algorithm (e.g., NSGA-II [32]) falls into this module.

ES module includes a general method *variation*(). Implementation of any ES based algorithm (e.g., CMA-ES [53]) falls into this module.

RL module provides a general interface and primitives for state-of-the-art RL algorithms, such as A3C, PPO and DQN.

MDP module provides an asynchronous interface. Each MDP can include multiple controllers, where each controller controls an agent. Any game environment can adapt to this asynchronous MDP interface by modifications.

Lamarckian can independently support EC algorithms by providing shared modules (e.g., GA and ES) and functions (e.g., initialization, evaluation, crossover and mutation). Thus, users can easily implement their EC algorithms by inheriting and reusing the existing modules and functions. To efficiently test the accuracy of implementing EC algorithms, users can also use the existing numerical optimization problems or implement their own problems by inheriting the Evaluator module. In addition, benefiting from the decoupled and object-oriented software architecture of the platform, users can flexibly apply different EC algorithms to different tasks by configuration commands or files.

IV. BENCHMARK EXPERIMENTS

In this section, we conduct benchmark experiments to assess the performance of Lamarckian on three benchmark games as shown in Fig. 6 in Section IV-B to Section IV-E.



Fig. 6. Benchmark games. From left to right: Pendulum, Pong, Google football.

A. Experimental Design

To assess the performance of Lamarckian, we conduct benchmark experiments against the state-of-the-art library RLlib across different cluster scales and environments. For some of the commonly used environments in RL research (e.g., OpenAI gym) that provide only synchronous interfaces, we wrap them with the proposed asynchronous interface to be compatible with Lamarckian. Note that for the inherently synchronous environments, this modification merely affects sampling efficiency as the sampled trajectories would remain the same for the same action sequence. To sufficiently demonstrate the efficiency of gathering and broadcasting, we adopt an asynchronous variant of PPO built on top of an architecture similar to that of IMPALA with the proposed Ray+ZeroMQ and tree-shaped broadcasting method. All results are averaged over 5 independent runs of the corresponding experiments.

For small-scale experiments (10 to 160 CPU cores), we train agents to play Atari Pendulum and the image input version of Pong by PPO. For large-scale experiments (2000 to 6000 CPU cores), we train agents to play 1) Google football, a challenging video game based on physics simulation with complex interactions and a sparse reward by PPO, and 2) vector-input version of Pong by using the PBT to adjust the learning rate and loss ratio of PPO.

In all experiments except Pendulum, only a single GPU is used in the learner process. The hyperparameters used in the experiments and the model implementations are listed in TABLE III. All experiments were carried out on a cluster where each computation node has 40 Intel Xeon Gold 6148 CPU @2.4GHz. The learner always uses an NVIDIA Tesla A100-40GB GPU.

TABLE III

HYPERPARAMETER SETTING AND MODEL IMPLEMENTATION FOR ALL BENCHMARK INSTANCES. FOR PPO, WE USE THE NORMALIZED ADVANTAGE ESTIMATE AND NO KL PENALTY. INTERVALS INDICATE THE SEARCH RANGES OF THE CORRESPONDING PARAMETER IN PBT. THE CRITIC NETWORKS ARE BUILT ON TOP OF THE POLICY NETWORKS, SHARING THE BOTTOM PART. LEAKYRELU ACTIVATION IS APPLIED TO EACH LAYER.

	PPO					
Benchmarks	Discount	Clip	Learning rate			
Pendulum	0.99	0.2	0.01			
Image Pong	0.99	0.2	0.01			
Vector Pong	0.99	0.2	(0, 0.1]			
Gfootball	0.993	0.2	0.0005			
Benchmarks	Batch size	Batch reuse	Loss weight (policy, critic, entropy)			
Pendulum	8192	1	1, 0.5, 0.01			
Image Pong	8192	1	1, 0.5, 0.01			
Vector Pong	8192	1	1, (0, 1], 0.01			
Gfootball	32768	2	1, 0.5, 0.01			
Benchmarks	Policy models (hidden layers)		Critic models (hidden layers)			
Pendulum	FC(256, 128)		FC(256, 128, 128, 64)			
Image Pong	$(ConvK4S2) \times 4$, $FC(256)$		$(ConvK4S2) \times 4$, $FC(256)$			
Vector Pong	FC(256, 128)		FC(256, 128)			
Gfootball	FC(512, 256, 128)		FC(512, 256, 128, 128, 64)			

B. Sampling Efficiency

Sampling efficiency is measured by the average number of frames consumed per second (fps) during the training and

³In the context of RL, a *decision variable* generally refers to a *weight* or *hyperparameter*.

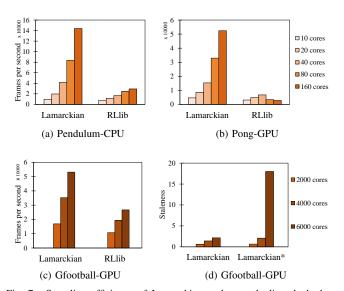


Fig. 7. Sampling efficiency of Lamarckian scales nearly linearly both on small-scale (10 to 160 CPU cores) and large-scale (2000 to 6000 CPU cores) training instances, which is only true for RLlib in (a) and (c). Besides, the staleness metric of Lamarckian* in (d) sharply increases as the number of CPUs grows.

TABLE IV
THE AVERAGE CPU UTILIZATION OF LAMARCKIAN AND RLLIB WHEN RUNNING PPO ON IMAGE PONG.

Platforms	CPU cores 10 20 40 80 160						
Lamarckian	32.2%	62.1%	96.8%	97.2%	69.4%		
RLlib	10.7%	21.2%	23.9%	7.5%	6.5%		

evaluation processes. Higher sampling efficiency leads to a faster training speed of agents generally. As shown in Fig. 7, Lamarckian has near-linear scalability in all cases, which is only true for RLlib in Fig. 7a and Fig. 7c. In detail, on the Pendulum game, Lamarckian achieves 10k samples per second on 10 cores. Compared with the sampling efficiency with the minimum number of cores, Lamarckian reaches about 2, 4, 8, and 15 times sampling efficiency on 20, 40, 80, and 160 cores, respectively. On the Pong game, Lamarckian achieves similar acceleration on these cores. Moreover, on the complex Gfootball game, Lamarckian obtains about 17k samples per second on 2000 cores, which becomes more advantageous as the number of cores increases, reaching over 53k samples per second on 6000 cores. The results show that even in the case communicating on thousands of cores, Lamarckian can still achieve near-linear scalability. In constrast, RLlib shows generally lower performance and degenerated in the experiment on Image Pong, suspiciously due to its complex remote object managing mechanism and limited communication efficiency.

The average CPU utilization of Lamarckian is much higher than that of RLlib when running PPO on Image Pong in TABLE IV. Specially, Lamarckian can achieve CPU utilization of 96.8% and 97.2% with one machine and two machines, respectively. However, the CPU utilization of both platforms decreases with an increasing number of machines, further indicating the potential improvement brought by Lamarckian.

C. Data Broadcasting Efficiency

Sample staleness, defined to measure the degree of policylag, is calculated as the version difference between the policy used to draw the samples and the policy being optimized. Note that RLlib's implementation of PPO employs synchronous sampling (the learner stalls to wait for new samples), thus always having a reference staleness of one. Fractional values are possible since PPO takes more than one gradient step on each batch of data. In an asynchronous sampling scheme, the staleness is mainly determined by the latencies of broadcasting updated policies and gathering samples.

To verify the efficiency of the proposed tree-shaped data broadcasting, we conduct ablation experiments by comparing it against asynchronous broadcasting with a flat layout as discussed in Section III-B1, termed Lamarckian*. Moreover, we study how the staleness influences the agents' learning efficiency, as measured by the performance achieved given the same amount of samples.

As shown in Fig. 7d, the staleness of Lamarckian* sharply increases as the number of CPUs grows, which matches our expectation on its scaling behavior. And the out-of-date samples also lead to worse performance, which can be observed from Fig. 8. Besides, although the staleness of Lamarckian is larger when on 4000 cores or more, the performance of trained agents is consistently better. Therefore, we can conclude that the proposed tree-shaped broadcasting can effectively reduce the side effects brought by the policy-lag in asynchronous learning schemes.

D. Performance and Training Speed

We investigate the agents' performance and the training speed of Lamarckian and RLlib by the learning curves in fixed numbers of environment frames (i.e., 150M frames for PPO on Gfootball, and 1G frames for PBT+PPO on Pong) on the three large-scale instances. Fig. 8 and Fig. 9 provide the learning curves from two perspectives, with total frames and time as the horizontal axes respectively. The two subfigures in each column represent the same instance of CPU cores.

As shown in the two figures, Lamarckian achieves competitive scores on both PPO for Gfootball and PBT+PPO for Pong. Moreover, the training speed of Lamarckian is much faster than that of RLlib in both games: i) in Fig. 8, with a larger number of CPU cores, Lamarckian achieves as twice fast training speed as RLlib on 6000 cores; ii) in Fig. 9, Lamarckian is 13 times faster than RLlib on 6000 cores.

However, when consuming the same frames, RLlib converges faster than Lamarckian on Pong with 4000 and 6000 cores in the first row of Fig. 9, but the observation is not true in Fig. 8. The different observations of the performance and training speed from the two figures are highly related to the difficulties of two tasks, i.e., tasks of different difficulties have different sensitivities to the abundance of samples. In the experiments, the task of Gfootball, with sparse rewards, is a much more challenging RL task than the task of Pong. Consequently, Lamarckian achieved significantly faster convergence (as well as higher scores) than RLlib on Gfootball due to the abundant samples generated by the asynchronous

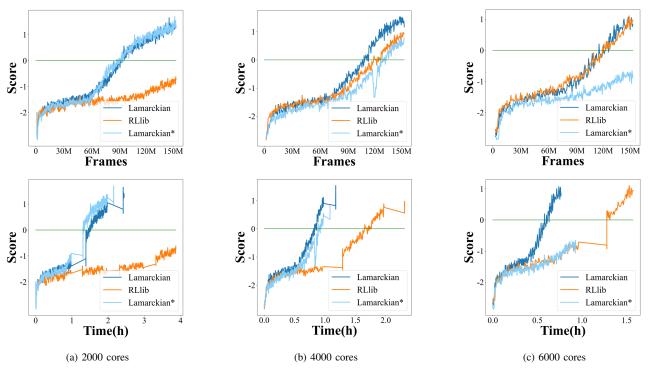


Fig. 8. The agents' performance and the training speed of Lamarckian, RLlib and Lamarckian* by the learning curves in 150M environment frames when running PPO on Gfootball.

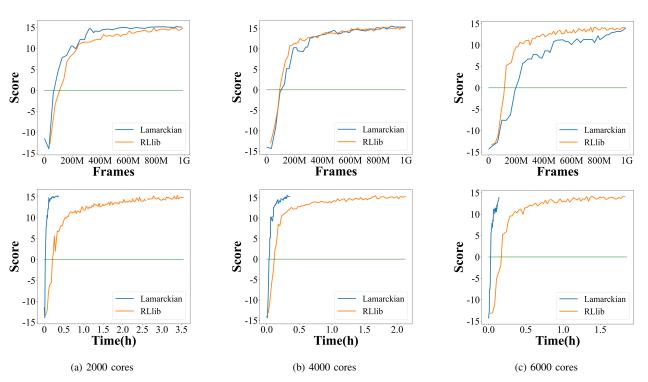


Fig. 9. The agents' performance and the training speed of Lamarckian and RLlib by the learning curves in 1G environment frames when running PBT+PPO on Pong.

sampling, while the limited samples generated by the synchronous sampling in RLlib seem good enough for the task of Pong. Besides, as shown in Fig. 8, RLlib stagnates on 2000 cores while converging better on 4000 and 6000 cores reflects. It can be attributed to the fact that the inefficient and unstable communication mechanism of RLlib can lead to its unstable performance on large-scale CPU cores.

E. Discussion

In asynchronous RL, higher sampling efficiency does not necessarily lead to faster convergence or higher performance in frames, but staleness plays the key role. In Lamarckian, the staleness of samples increases as the transfer of samples takes longer; by contrast, RLlib adopts a synchronous learning mechanism where the staleness is fixed no matter how long it takes to transfer the samples. Hence, despite that Lamarckian has significantly higher sampling efficiency, its staleness also becomes larger as the number of machines increases. Nonetheless, the proposed tree-shaped broadcasting is able to effectively reduce the side effects brought by the policy-lag and the communication bottleneck of learners in the asynchronous learning procedure, thus leading to competitive performance (i.e., scores) with a faster training speed. Furthermore, the distributed EvoRL workflow can also accelerate the training speed for EvoRL methods. In common practice, staleness is often fixed by synchronous sampling, while maintaining stable staleness is particularly crucial and challenging in asynchronous RL.

V. USE CASES

In this section, we provide two use cases of Lamarckian: first, we provide a use case of how to generate behavior-diverse game AI by implementing a state-of-the-art algorithm in Lamarckian; then, we provide a use case of how Lamarckian is applied to game balancing tests for an asynchronous RTS game.

A. Generating Behavior-Diverse Game AI

In commercial games, *game AI* entertains users via humanlike interactions. For high-quality game AI, *diversity of behaviors* is among the most important criteria to meet. However, generating behavior-diverse game AI often involves rich domain knowledge and exhaustive human labors. For example, the behavior tree [54] is a rule-based method, which requires abundant expert knowledge and labor costs in designing rules. In contrast, the EC-based methods can generate strong game AI beyond human common sense by requiring little prior human knowledge [55], [56], [15].

The EMOGI [15] is a recently proposed EvoRL approach for generating diverse behaviors for game AI with little prior knowledge. The basic idea of EMOGI is to guide the AI agent to learn towards desired behaviors automatically by tailoring a reward function with multiple objectives, where a multi-objective optimization algorithm is adopted to obtain policies trading-off between the multiple objectives.

We implement EMOGI in Lamarckian and show a use case on the Atari Pong game. Specifically, apart from maximizing the win-rate/score, a Pong agent is designed by considering *active* or *lazy* behavior styles according to its willingness to make movements during a game epoch. The two styles are formulated as a reward involving two objectives:

$$\mathbf{r}(s,a) = [f_1(s,a), f_2(s,a)]^T,$$
(2)

with

$$\begin{cases} f_1(s, a) = win(s) + activeness(a) \\ f_2(s, a) = win(s) + laziness(a) \end{cases},$$
 (3)

and

$$\begin{cases} \text{activeness}(a) = w_1 \cdot \text{move}(a) \\ \text{laziness}(a) = w_2 \cdot (1 - \text{move}(a)) \end{cases}, \tag{4}$$

where $f_1(s,a)$ and $f_2(s,a)$ are the *activeness*-related objective and the *laziness*-related objective respectively; win(s) returns 1 iff s is a winning state, otherwise 0; move(a) returns $\frac{1}{T}$ iff there is any movement in action a, otherwise 0, with T denoting the total number of actions in an epoch; w_1 and w_2 are weight parameters set by users. Intuitively, the reward as formulated above encourages an agent to either play in an active manner or, conversely, try to win the game with a minimal number of movements.

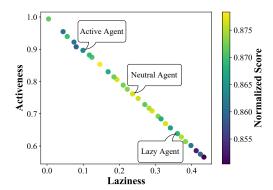


Fig. 10. The Pareto front of activeness vs. laziness values of the AI agents obtained by EMOGI using Lamarckian on Atari Pong game. Colors indicate the average normalized scores achieved by each agent.

As evidenced in Fig. 10, the AI agents formed different play styles in terms of activeness/laziness, while achieving similar normalized scores (defined as $\frac{score_{self} - score_{enemy} + score_{max}}{2score_{max}} \in [0,1]$). On the one hand, the agents with relatively lower competitiveness (i.e. those on the two corners) have survived due to their distinguished behavior styles – either too active or too lazy. On the other hand, the agents with relatively higher competitiveness ((i.e. those in the center) have also survived despite that the behavior styles are neutral. In practice, such wide spectrum of behavior-diverse game AI would enable richer user experience.

B. Game Balancing in RTS Game

In commercial games, user satisfaction is always influenced by game balancing [57]. In an imbalanced game, there maybe exist an unbeatable strategy or a role getting the player frustrated. The game designers aim to develop a balanced game by weakening the most imbalanced strategies or roles obtained



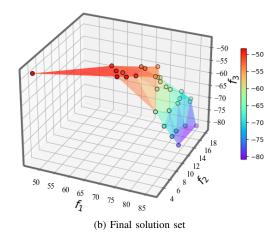


Fig. 11. (a) is a typical hero-led fighting arena of The Lord of the Rings: Rise to War. (b) is the final solution set obtained by Lamarckian to a three-objective optimization problem in the game balancing test. f_1 is the battle damage difference, f_2 is the remaining economic resources multiples, and f_3 is the strength of the weakest team.

by the game balancing test. Traditionally, the enumeration methods or reinforcement methods have been adopted to the problem in industry. However, they are low-efficient since only one solution is obtained in each run. In contrast, the EC-based methods are more efficient by obtaining a set of diverse solutions in one run.

From the optimization point of view, the game balancing test is a non-differentiable optimization problem that involves multiple objectives to be considered simultaneously. We apply Lamarckian to a three-objective optimization problem in the game balancing test for the asynchronous commercial RTS game, The Lord of the Rings: Rise to War, aiming to obtain the most imbalanced and strongest lineups. First, we encode a lineup as the decision variables of a candidate solution, including heroes (51 types) and soldiers (two to three types); each hero is armed with four types of skills and 79 types of equipment. Thus, the search space is up to 50,000 units. A typical hero-led fighting arena is shown Fig. 11a. Second, we formulate the game balancing test problem as a multi-objective combinatorial optimization (maximization) problem associated with three objectives: i) the battle damage difference (f_1) calculated by the remaining strength difference between the strongest team and the weakest team in a lineup, ii) the remaining economic resources multiples (f_2) calculated by the remaining economic resources of the strongest team divided by those of the weakest team, and iii) the strength of the weakest team (f_3) to improve the overall strength of a lineup. Third, we use crossover and mutation operators of discrete variables similar to those in the classic Traveling Salesman Problem to ensure the validity of offspring and the non-repetitiveness of teams. Then, to evaluate a candidate solution, all teams in the solution should play against each other to obtain the first two objectives, and then battle with the baseline lineups to determine the strength of the weakest team. Finally, we adopt NSGA-II, a classic EC algorithm for multi-objective optimization, as the optimizer to iteratively perform nondominated sorting on a population of candidate solutions

evolving towards the Pareto front. The final population is an approximation to the Pareto optimal set. The solutions in the Pareto optimal set mean that there is no solution better than the other solutions on all the objectives and they are Pareto non-dominated.

As shown in Fig. 11b, a solution with a remaining economic resources multiple larger than two (i.e., $f_2 > 2$) is considered *imbalanced*. Apart from the solution set itself, we also obtain some interesting observations. For example, the outlier solution at the left corner indicates the strong conflicts between f_1 and f_3 , and f_2 and f_3 , respectively. It means that a set of overall very strong lineup leads to less battle damage differences and remaining economic resources multiples. Eventually, the game designers will select the solutions with overall strong lineups (i.e., the red points) and modify the game parameters to weaken these lineups. The above testing and modification processes will loop until the game is balanced. The game balancing test of each loop only takes about ten minutes by a tailored game simulator, which is acceptable in the industry.

VI. CONCLUSION

This paper introduces Lamarckian – an open-source highperformance scalable platform tailored for evolutionary reinforcement learning. To meet the requirements of applications in large-scale distributed computing environments (e.g. the asynchronous commercial game environments), Lamarckian adopts a tree-shaped data broadcasting method as well as the asynchronous Coroutine-based MDP interface. To accelerate the training of agents, Lamarckian couples Ray with ZeroMQ to take advantage of both. From the software engineering perspective, Lamarckian also provides good flexibility and extensibility by well decoupled objective-oriented designs. The performance of Lamarckian has been evaluated on large-scale benchmark tests with up to 6000 CPU cores, in comparison with the state-of-the-art RLlib. Additionally, we provide two use cases of Lamarckian. In the first use case, we apply Lamarckian to generating behavior-diverse game AI by implementing a recently proposed EvoRL algorithm. In the second use case, we apply Lamarckian to multi-objective game balancing test for an asynchronous commercial real-time strategy game.

In order to match the asynchronous MDP interface in Lamarckian, users need to transfer the original synchronous MDP interfaces into asynchronous ones by following unified workflow. However, it is worthy of such additional implementation labor as it brings substantial performance improvement. Further, the ready-to-use modules in Larmackian also brings extra benefits to users.

In summary, Lamarckian is a high-performance, easy-touse, and scalable platform for researchers and engineers to take instant adventures.

REFERENCES

- D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, 2016.
- [2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *The International Conference on Learning Representations* (ICLR), 2016.
- [4] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi, "Learning to dispatch for job shop scheduling via deep reinforcement learning," in Advances in Neural Information Processing Systems (NeurIPS), 2020.
- [5] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yo-gamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, 2021.
- [6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," arXiv preprint arXiv:1606.01540, 2016.
- [7] M. Wydmuch, M. Kempka, and W. Jaskowski, "Vizdoom competitions: Playing doom from pixels," *IEEE Transactions on Games*, vol. 11, no. 3, pp. 248–259, 2019.
- [8] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik et al., "Deepmind lab," arXiv preprint arXiv:1612.03801, 2016.
- [9] S. Xu, H. Kuang, Z. Zhi, R. Hu, Y. Liu, and H. Sun, "Macro action selection with deep reinforcement learning in starcraft," in *Proceedings* of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, vol. 15, no. 1, 2019, pp. 94–99.
- [10] R.-Z. Liu, H. Guo, X. Ji, Y. Yu, Z.-J. Pang, Z. Xiao, Y. Wu, and T. Lu, "Efficient reinforcement learning for starcraft by abstract forward models and transfer learning," *IEEE Transactions on Games*, 2021.
- [11] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv* preprint arXiv:1912.06680, 2019.
- [12] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castaneda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman *et al.*, "Human-level performance in 3D multiplayer games with population-based reinforcement learning," *Science*, vol. 364, no. 6443, pp. 859–865, 2019. [Online]. Available: http: //dx.doi.org/10.1126/science.aau6249
- [13] S. Khadka and K. Tumer, "Evolution-guided policy gradient in reinforcement learning," in *International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.
- [14] S. Khadka, S. Majumdar, T. Nassar, Z. Dwiel, E. Tumer, S. Miret, Y. Liu, and K. Tumer, "Collaborative evolutionary reinforcement learning," *International Conference on Machine Learning (ICML)*, 2019.
- [15] R. Shen, Y. Zheng, J. Hao, Z. Meng, Y. Chen, C. Fan, and Y. Liu, "Generating behavior-diverse game AIs with evolutionary multi-objective deep reinforcement learning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

- [16] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 772–784.
- [17] H. Qian and Y. Yu, "Derivative-free reinforcement learning: A review," Frontiers of Computer Science, 2021.
- [18] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare, "Dopamine: A research framework for deep reinforcement learning," *CoRR*, vol. abs/1812.06110, 2018. [Online]. Available: http://arxiv.org/abs/1812.06110
- [19] D. Hafner, J. Davidson, and V. Vanhoucke, "Tensorflow agents: Efficient batched reinforcement learning in tensorflow," CoRR, vol. abs/1709.02878, 2017. [Online]. Available: http://arxiv.org/abs/1709.02878
- [20] H. Küttler, N. Nardelli, T. Lavril, M. Selvatici, V. Sivakumar, T. Rocktäschel, and E. Grefenstette, "Torchbeast: A pytorch platform for distributed RL," *CoRR*, vol. abs/1910.03552, 2019. [Online]. Available: http://arxiv.org/abs/1910.03552
- [21] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference* on Machine Learning (ICML), 2018. [Online]. Available: http://proceedings.mlr.press/v80/liang18b.html
- [22] P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker, "Empirical evaluation methods for multiobjective reinforcement learning algorithms," *Machine Learning*, vol. 84, pp. 51–80, 2010.
- [23] A. Abels, D. Roijers, T. Lenaerts, A. Nowé, and D. Steckelmacher, "Dynamic weights in multi-objective deep reinforcement learning," in International Conference on Machine Learning. PMLR, 2019, pp. 11– 20.
- [24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017
- [25] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning et al., "IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning (ICML)*, 2018.
- [26] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski, "SEED RL: Scalable and efficient deep-rl with accelerated central inference," in *International Conference on Learning Representations* (ICLR), 2020. [Online]. Available: https://openreview.net/forum?id= rkgvXlrKwH
- [27] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz
- [28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016. [Online]. Available: http://proceedings.mlr. press/v48/mniha16.html
- [29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," arXiv preprint arXiv:1312.5602, 2013.
- [30] K. Kurach, A. Raichuk, P. Stańczyk, M. Zając, O. Bachem, L. Espeholt, C. Riquelme, D. Vincent, M. Michalski, O. Bousquet et al., "Google research football: A novel reinforcement learning environment," in Proceedings of the AAAI Conference on Artificial Intelligence, 2020.
- [31] H. Ishibuchi, Y. Nojima, and T. Doi, "Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures," in *IEEE Congress on Evolutionary Computation*, Vancouver, BC, Canada, July 2006, pp. 3959–3966.
- [32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.
- [33] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable test problems for evolutionary multi-objective optimization," in *Evolutionary Multiobjective Optimization*, ser. Advanced Information and Knowledge Processing. Springer, 2005, pp. 105–145.
- [34] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation*, vol. 8, no. 2, pp. 173–195, June 2000.

- [35] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan et al., "Population based training of neural networks," arXiv preprint arXiv:1711.09846, 2017.
- [36] D. Whitley, "A genetic algorithm tutorial," Statistics and computing, vol. 4, no. 2, pp. 65–85, 1994.
- [37] J. K. Pugh, L. B. Soros, and K. O. Stanley, "Quality diversity: A new frontier for evolutionary computation," *Frontiers in Robotics and AI*, vol. 3, p. 40, 2016. [Online]. Available: https://www.frontiersin.org/article/10.3389/frobt.2016.00040
- [38] J. Lehman and K. O. Stanley, "Abandoning objectives: Evolution through the search for novelty alone," *Evolutionary computation*, vol. 19, no. 2, pp. 189–223, 2011.
- [39] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: a survey," in 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2020, pp. 737–744.
- [40] A. Pourchot and O. Sigaud, "Cem-rl: Combining evolutionary and gradient-based methods for policy search," arXiv preprint arXiv:1810.01222, 2018.
- [41] S. Liu, G. Lever, J. Merel, S. Tunyasuvunakool, N. Heess, and T. Graepel, "Emergent coordination through competition," in *International Conference on Learning Representations (ICLR)*, 2019.
- [42] J. D. Co-Reyes, Y. Miao, D. Peng, E. Real, Q. V. Le, S. Levine, H. Lee, and A. Faust, "Evolving reinforcement learning algorithms," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=0XXpJ4OtjW
- [43] R. Yang, X. Sun, and K. Narasimhan, "A generalized algorithm for multi-objective reinforcement learning and policy adaptation," in Advances in Neural Information Processing Systems (NeurIPS), H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Aché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 14610–14621. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/4a46fbfca3f1465a27b210f4bdfe6ab3-Abstract.html
- [44] P. Sun, J. Xiong, L. Han, X. Sun, S. Li, J. Xu, M. Fang, and Z. Zhang, "TLeague: A framework for competitive self-play based distributed multi-agent reinforcement learning," arXiv preprint arXiv:2011.12895, 2020
- [45] M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli et al., "Acme: A research framework for distributed reinforcement learning," arXiv preprint arXiv:2006.00979, 2020.
- [46] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," *International Conference on Machine Learning (ICML)*, 2016. [Online]. Available: http://arxiv.org/abs/1604.06778
- [47] L. Fan, Y. Zhu, J. Zhu, Z. Liu, O. Zeng, A. Gupta, J. Creus-Costa, S. Savarese, and L. Fei-Fei, "SURREAL: Open-source reinforcement learning framework and robot manipulation benchmark," in *Conference* on Robot Learning, 2018.
- [48] Y. Song, A. Wojcicki, T. Lukasiewicz, J. Wang, A. Aryan, Z. Xu, M. Xu, Z. Ding, and L. Wu, "Arena: A general evaluation platform and building toolkit for multi-agent intelligence," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 7253–7260.
- [49] M. Zhou, Z. Wan, H. Wang, M. Wen, R. Wu, Y. Wen, Y. Yang, W. Zhang, and J. Wang, "Malib: A parallel framework for population-based multi-agent reinforcement learning," arXiv preprint arXiv:2106.07551, 2021.
- [50] I. Oh, S. Rho, S. Moon, S. Son, H. Lee, and J. Chung, "Creating pro-level AI for a real-time fighting game using deep reinforcement learning," *IEEE Transactions on Games*, 2021.
- [51] D. Ye, G. Chen, W. Zhang, S. Chen, B. Yuan, B. Liu, J. Chen, Z. Liu, F. Qiu, H. Yu, Y. Yin, B. Shi, L. Wang, T. Shi, Q. Fu, W. Yang, L. Huang, and W. Liu, "Towards playing full moba games with deep reinforcement learning," in *Advances in Neural Information Processing Systems* (NeurIPS), 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/06d5ae105ea1bea4d800bc96491876e9-Paper.pdf
- [52] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, "Modeling software architectures in the unified modeling language," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 11, no. 1, pp. 2–57, 2002.
- [53] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)," *Evolutionary computation*, vol. 11, no. 1, pp. 1–18, 2003.
- [54] I. Millington and J. Funge, Artificial intelligence for games. CRC Press,

- [55] J.-B. Mouret and J. Clune, "Illuminating search spaces by mapping elites," arXiv preprint arXiv:1504.04909, 2015.
- [56] A. Agapitos, J. Togelius, S. M. Lucas, J. Schmidhuber, and A. Konstantinidis, "Generating diverse opponents with multiobjective evolution," in 2008 IEEE Symposium On Computational Intelligence and Games. IEEE, 2008, pp. 135–142.
- [57] F. d. Mesentier Silva, R. Canaan, S. Lee, M. C. Fontaine, J. Togelius, and A. K. Hoover, "Evolving the hearthstone meta," in *IEEE Conference on Games (CoG)*, 2019.



Hui Bai received the B.Sc. and M.Sc. degrees in software engineering from Xiangtan University, Xiangtan, China, in 2014 and 2017 respectively. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, Southern University of Science and Technology, China.

Her main research interests include evolutionary algorithms and their applications to reinforcement learning.



Ruimin Shen received the Ph.D. degree in applied mathematics and M.Sc. degree in computer science from Xiangtan University of China, in 2015 and 2012, respectively. He is currently the researcher of the Game AI research team of NetEase Games AI Lab, Guangzhou, China.

His research interests include evolutionary algorithms, reinforcement learning and their applications to online games.



Yue Lin received the M.Sc. degree in computer science from Zhejiang University, Hangzhou, China in 2013, and the B.Sc. degree in control science and engineering from Zhejiang University, Hangzhou, China in 2010. He is currently the director of NetEase Games AI Lab, Guangzhou, China.

His research interests include computer vision, data mining, reinforcement learning and their applications to online games.



Botian Xu received his bachelor's degree in Computer Science from Southern University of Science and Technology. He is currently a researcher at Institution of Interdisciplinary Information Science, Tsinghua University.

His interests focus on deep reinforcement learning and its applications.



Ran Cheng (M'2016-SM'2021) received the B.Sc. degree from Northeastern University, Shenyang, China, in 2010, and the Ph.D. degree from the University of Surrey, Guildford, U.K., in 2016. He is currently an Associate Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology, China.

His research interests mainly fall into the interdisciplinary fields across evolutionary computation and other major AI branches such as statistical learning and deep learning, aiming to provide end-

to-end solutions to optimization/modeling problems in scientific research and engineering applications.

He is a recipient of the 2019 IEEE Computational Intelligence Society Outstanding Ph.D. Dissertation Award, the 2018 and 2021 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION Outstanding Paper Awards, and the 2020 IEEE Computational Intelligence Magazine Outstanding Paper Award. He serves as Associate Editors of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, the IEEE TRANSACTIONS ON ARTIFICIAL INTELLIGENCE, and the IEEE TRANSACTIONS ON COGNITIVE AND DEVELOPMENTAL SYSTEMS. He is the Chair of IEEE Computational Intelligence Society Shenzhen Chapter.