Review, Analyze, and Design a Comprehensive Deep Reinforcement Learning Framework

Ngoc Duy Nguyen, Thanh Thi Nguyen, Hai Nguyen, Saeid Nahavandi, *Senior Member, IEEE*

***Abstract*—Reinforcement learning (RL) has emerged as a standard approach for building an intelligent system, which involves multiple self-operated agents to collectively accomplish a designated task. More importantly, there has been a great attention to RL since the introduction of deep learning that essentially makes RL feasible to operate in high-dimensional environments. However, current research interests are diverted into different directions, such as multi-agent and multi-objective learning, and human-machine interactions. Therefore, in this paper, we propose a comprehensive software architecture that not only plays a vital role in designing a connect-the-dots deep RL architecture but also provides a guideline to develop a realistic RL application in a short time span. By inheriting the proposed architecture, software managers can foresee any challenges when designing a deep RL-based system. As a result, they can expedite the design process and actively control every stage of software development, which is especially critical in agile development environments. For this reason, we designed a deep RL-based framework that strictly ensures flexibility, robustness, and scalability. Finally, to enforce generalization, the proposed architecture does not depend on a specific RL algorithm, a network configuration, the number of agents, or the type of agents.**

***Index Terms*—software architecture, deep learning, reinforcement learning, learning systems, multi-agent systems, human-machine interactions.**

I. Introduction

REINFORCEMENT learning (RL) has attracted a great deal of research attention for decades. It is desirable because RL conducts a learning procedure by allowing agents to directly interact with the environment. As a result, an RL agent can imitate human learning process to achieve a designated goal, *i.e.*, the agent conducts trial-and-error learning (exploration) and draws on "experience" (exploitation) to improve its behaviors [1], [2]. Therefore, RL is used in countless domains, such as IT resources management [3], cyber-security [4], robotics [5], [6], [7], [8], control systems [9], [10], recommendation systems [11], bidding and advertising campaigns [12], and video games [13], [14], [15]. However, traditional RL methods and dynamic programming [16], which use a *bootstrapping* mechanism to approximate the objective function, cease to work in high-dimensional environments due to memory and processing power limitations. This so-called issue, *the curse of dimensionality*, creates a major challenge in RL literature.

Ngoc Duy Nguyen and Saeid Nahavandi are with the Institute for Intelligent Systems Research and Innovation, Deakin University, Waurn Ponds Campus, Geelong, Victoria, Australia (e-mails: duy.nguyen@deakin.edu.au and saeid.nahavandi@deakin.edu.au).

Thanh Thi Nguyen is with the School of Information Technology, Deakin University, Burwood Campus, Melbourne, Victoria, Australia (e-mail: thanh.nguyen@deakin.edu.au).

Hai Nguyen is with Khoury College of Computer Science, Notheastern University, Boston, USA (e-mail: hainguyen@ccs.neu.edu)

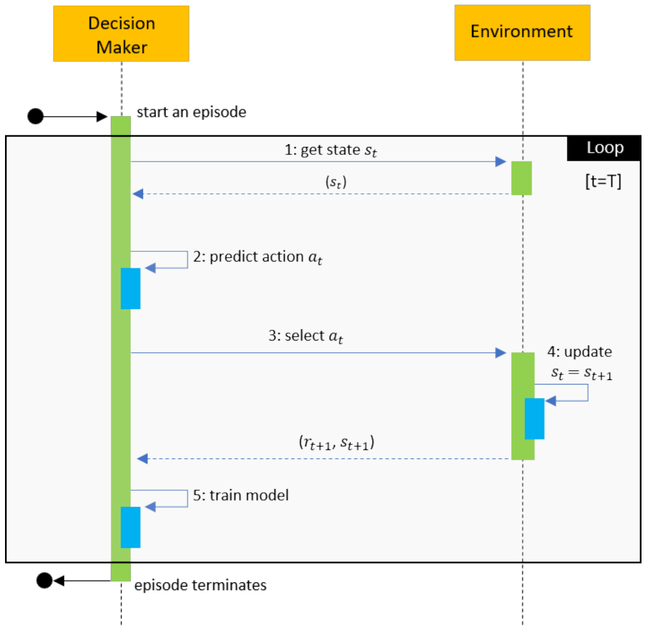

Fig. 1: Using a UML sequential diagram to describe an RL problem.

Fig. 1 describes an RL problem by using a *Unified Modeling Language* (UML) [17] sequential diagram. Specifically, the problem includes two entities: a decision maker and the environment. The environment can be an artificial simulator or a wrapper of the real-world environment. While the environment is a *passive* entity, the decision maker is an *active* entity that periodically interacts with the environment. In the RL context, a decision maker and an agent can be interchangeable, though they can be two identified objects from a software design perspective.

At first, the decision maker perceives a state s_t from the environment. Then it uses its internal model to select the corresponding action a_t. The environment interacts with the chosen action a_t by sending a numerical reward r_{t+1} to the decision maker. The environment also brings the decision maker to a new state s_{t+1}. Finally, the decision maker uses the current transition $\vartheta = \{a_t, s_t, r_{t+1}, s_{t+1}\}$ to update its decision model. This process is iterated until t equals T, where s_T denotes the terminal state of an episode. There are different methods to develop a decision model, such as fuzzy logic [18], genetic algorithms [19], [20], or dynamic programming [21].

In this paper, however, we consider a deep neural network as the decision model.

The previous diagram infers that RL is *online* learning because the model is updated with incoming data. However, RL can be performed offline via a *batch learning* [22] technique. In particular, the current transition ϑ can be stored in an *experience replay* [23] and retrieved later to train the decision model. Finally, the goal of an RL problem is to maximize the expected sum of discounted reward R_t, *i.e.*,

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T,$$

where γ denotes the discounted factor and $0 < \gamma \leq 1$.

In 2015, Google DeepMind [24] announced a breakthrough in RL by combining it with deep learning to create an intelligent agent that can beat a professional human player in a series of 49 Atari games. The idea was to use a deep neural network with convolutional layers [25] to directly process raw images (states) of the game screen to estimate subsequent actions. The study is highly valued because it opened a new era of RL with deep learning, which partially solves the curse of dimensionality. In other words, deep learning is a great complement for RL in a wide range of complicated applications. For instance, Google DeepMind created a program, AlphaGo, which beat the Go grandmaster, Lee Sedol, in the best-of-five tournament in 2016 [26]. AlphaGo is a full-of-tech AI that is based on Monte Carlo Tree Search [27], a hybrid network (policy network and value network), and a self-taught training strategy [28]. Other applications of deep RL can be found in self-driving cars [29], [30], helicopters [31], or even NP-hard problems such as *Vehicle Routing Problem* [32] and combinatorial graph optimization [33].

As stated above, deep RL is crucial owing to its appealing learning mechanism and widespread applications in the real world. In this study, we further delve into practical aspects of deep RL by analyzing challenges and solutions while designing a deep RL-based system. Furthermore, we consider a real-world scenario where multiple agents, multiple objectives, and human-machine interactions are involved. Firstly, if we can take advantage of using multiple agents to accomplish a designated task, we can shorten the *wall time*, *i.e.*, the computational time to execute the assigned task. Depending on the task, the agents can be *cooperative* or *competitive*. In the cooperative mode, agents work in *parallel* or in *pipeline* to achieve the task [34]. In the case of competition, agents are scrambled, which basically raises the *resource hunting* problem [35]. However, in contrast to our imagination, competitive learning can be fruitful. Specifically, the agent is trained continually to place the opponent into a disadvantaged position and the agent is improved over time. Because the opponent is also improved over time, this phenomenon eventually results in the *Nash equilibrium* [36]. Moreover, competitive learning originates the self-taught strategy (*e.g.* AlphaGo) and a series of techniques such as the *Actor-Critic architecture* [37], [38], opponent modeling [39], [40], and *Generative Adversarial Networks* [41]. Finally, we notice the problem of *moving target* [42], [43] in multi-agent systems, which describes a scenario when the decision of an agent depends on other agents, thus the optimal policy becomes *non-stationary* [43], [44].

Secondly, a real-world objective is often complicated as it normally comprises of multiple sub-goals. It is straightforward if sub-goals are *non-conflicting* because they can be seen as a single composite objective. The more difficult case is when there are *conflicting objectives*. One solution is to convert a multi-objective problem into a single objective counterpart by applying *scalarization* through an application of a linear weighted sum for individual objective [45] or non-linear methods [46]. These approaches are categorized as *single-policy methods*. In contrast, the *multi-policy methods* [47] seek multiple optimal policies at the same time. Although the number of multi-policy methods is restricted, it can be powerful. For instance, the *Convex Hull Value Iteration* algorithm [48] computes a set of objective combinations to retrieve all deterministic optimal policies. Finally, to benchmark a multi-objective method, we find or approximate a boundary surface, namely *Pareto dominance*, which presents the maximum performance of different weights (if scalarization is used) [49]. Recent studies have tried to integrate multi-objective mechanisms into deep RL [50], [51], [52].

Last but not least, human-machine interaction is another key factor in a deep RL-based system. A self-driving car, for example, should accept human intervention in emergency cases [53], [54]. Therefore, it is critical to ensure a certain level of safety while designing a hybrid system in which humans and machines can work together. Due to its importance, Google DeepMind and OpenAI have presented novel ways that have encouraged a number of inventions in recent years [55]. For instance, Christiano *et al.* [56] propose a novel scheme that accepts human feedback during the training process. However, the method requires an operator to constantly observe the agent's behavior, which is an onerous and error-prone task. Recent work [57] provides a more practical approach by introducing a behavioral control system. The system is used to control multiple agents in real time via human dictation. Table I summarizes key terminologies that are widely used in RL contexts.

In summary, the study contributes the following key factors:

- The paper presents an overall picture of contemporary deep RL. We briefly overview state-of-the-art deep RL methods considering three key factors of a real-world application such as multi-agent learning, multi-objective problems, and human-machine interactions. Thereafter, the paper offers a checklist for software managers, a guideline for software designers, and a technical document for software programmers.
- We analyze challenges and difficulties while designing a deep RL-based system and hence mitigate possible mistakes during the development process. In other words, software designers can inherit the proposed design, foresee difficulties, and eventually expedite the entire development procedure, especially in agile software development.
- Finally, the source code of the proposed framework can be found in [58]. Based on this template, RL beginners can prototype an RL method and develop an RL-based

TABLE I: Key RL terminologies

Term	Description	Pros	Cons
Model-free RL	The environment is a black box. Agents mostly conduct a trial-and-error procedure to learn on its own. They use rewards to update their decision models	The algorithm does not need a model of the environment	Requires a large amount of samples
Model-based RL	Agents construct a model that simulates the environment and use it to generate future episodes. By using the model, agents can estimate not only actions but also future states	Speed up learning and improve sample efficiency	Having an accurate and useful model is often challenging
Temporal difference learning	Use TD error to estimate the value function. For example, in Q-learning, $Q(s_t, a_t) = Q(s_t, a_t) + \beta(r_t + \gamma \max_a Q(s_{t+1}, a))$	Fast convergence as it does not need to wait until the episode ends	Estimates can be biased
Monte-Carlo method	Estimate the value function by obtaining the average of the same values in different episodes $Q(s_t, a_t) = \lim_{N->\infty} \sum_{i=1}^{N} Q(s_t^i, a_t^i)$	The values are non-biased estimates	Slow convergence and the estimates have high variances. Has to wait until episode ends to do updates
Continuous action space	The number of control actions is continuous	A policy-based method can be used	Cannot use a value-based method
Discrete action space	The number of control actions is discrete and finite	Both policy-based method and value-based method can be used	Intractable if the number of actions is large
Deterministic policy	The policy maps each state to a specific action	Reduce data sampling	Vulnerable to noise and stochastic environments
Stochastic policy	The policy maps each state to a probability distribution over actions	Better exploration	Requires a large amount of samples
On-policy method	Improve the current policy that the agent is using to make decisions	Safer to explore	May be stuck in local minimum solutions
Off-policy method	Learn the optimal policy (while samples are generated by the behavior policy)	Instability. Often used with an experience replay	Might be unsafe because the agent is free to explore
Fully observable environment	All agents can observe the complete states of the environment	Easier to solve than partially observable environments	The number of state can be large
Partially observable environment	Each agent only observes a limited observation of the environment	More practical in real-world applications	More difficult to solve as the agents require to remember past states

application in a short time span. As a result, the paper is a key factor to share deep RL to a wider community.

The paper has the following sections. Section II conducts a brief survey of state-of-the-art deep RL methods in different research directions. Section III presents our proposed system architecture, which supports multiple agents, multiple objectives, and human-machine interactions. Finally, we conclude the paper in Section IV.

II. Literature Review

A. Single-Agent Method

The first advent of deep RL, *Deep Q-Network* (DQN) [24], [59], basically uses a deep neural network to estimate values of state-action pairs via a *Q-value function* (*a.k.a.*, *action-value function* or $Q(s, a)$). Thereafter, a number of variants based on DQN were introduced to improve the original algorithm. Typical extensions can be examined such as *Double DQN* [60], *Dueling Network* [61], *Prioritized Experience Replay* [62], *Recurrent DQN* [63], *Attention Recurrent DQN* [64], and an ensemble method named *Rainbow* [65]. These approaches use an experience replay to store historical transitions and retrieve them in batches to train the network. Moreover, a separate network *target network* can be used to mitigate the correlation of the sequential data and prevent the training network from overfitting.

Instead of estimating the action-value function, we can directly approximate the agent's policy $\pi(s)$. This approach is known as the *policy gradient* or *policy-based* method. *Asynchronous Advantage Actor-Critic* (A3C) [66] is one of the first policy-based deep RL methods to appear in the literature. In particular, A3C includes two networks: an actor network that is used to estimate the agent policy $\pi(s)$ and a critic network that is used to estimate the *state-value function* $V(s)$. Additionally, to stabilize the learning process, A3C uses the *advantage function*, *i.e.*, $A(s, a) = Q(s, a) - V(s)$. There is a synchronous version of A3C, namely A2C [66], which has the advantage of being simpler but with comparable or better performance. A2C mitigates the risk of multiple learners which might be overlapping when updating the weights of the global

TABLE II: Key deep RL methods in literature

Method	Description and Advantage	Technical Requirements	Drawbacks	Implementation
Value-based method				
DQN	Use a deep convolutional network to directly process raw graphical data and approximate the action-value function	● Experience replay ● Target network ● Q-learning	○ Excessive memory usage ○ Learning instability ○ Only for discrete action space	[81] [82] [83] [84]
Double DQN	Mitigate the DQN's maximization bias problem by using two separate networks: one for estimating the value, one for selecting action.	● Double Q-learning	○ Inherit DQN's drawbacks	[84]
Prioritized Experience Replay	Prioritize important transitions so that they are sampled more frequently. Improve sample efficiency	●Importance sampling	○ Inherit DQN's drawbacks ○ Slower than non-prioritized experience replay (speed)	[81] [84]
Dueling Network	Separate the DQN architecture into two streams: one estimates state-value function and one estimates the advantage of each action	● Dueling network architecture ● Prioritized replay	○ Inherit DQN's drawbacks	[84]
Recurrent DQN	Integrate recurrency into DQN Extend the use of DQN in partially observable environments	● Long Short Term Memory	○ Inherit DQN's drawbacks	[84]
Attention Recurrent DQN	Highlight important regions of the environment during the training process	● Attention mechanism ● Soft attention ● Hard attention	○ Inherit DQN's drawbacks	[64]
Rainbow	Combine different techniques in DQN variants to provide the state-of-the-art performance on Atari domain	● Double Q-learning ● Prioritized replay ●Dueling network ●Multi-step learning ●Distributional RL ●Noisy Net	○ Inherit DQN's drawbacks	[81]
Policy-based method				
A3C/A2C	Use actor-critic architecture to estimate directly the agent policy. A3C enables concurrent learning by allowing multiple learners to operate at the same time	● Multi-step learning ● Actor-critic model ● Advantage function ● Multi-threading	○ Policy updates exhibit high variance	[82] [83] [84]
UNREAL	Use A3C and multiple unsupervised reward signals to improve learning efficiency in complicated environments	● Unsupervised reward signals	○ Policy updates exhibit high variance	[85]
DDPG	Concurrently learn a deterministic policy and a Q-function in DQN's fashion	● Deterministic policy gradient	○ Support only continuous action space	[83] [84]
TRPO	Limit policy update variance by using the conjugate gradient to estimate the natural gradient policy TRPO is better than DDPG in terms of sample efficiency	● Kullback-Leibler divergence ● Conjugate gradient ● Natural policy gradient	○ Computationally expensive ○ Large batch of rollouts ○ Hard to implement	[83] [84]
ACKTR	Inherit the A2C method Use Kronecker-Factored approximation to reduce computational complexity of TRPO ACKTR outperforms TRPO and A2C	● Kronecker-factored approximate curvature	○ Still complex	[83]
ACER	Integrate an experience replay into A3C Introduce a light-weight version of TRPO ACER outperforms TRPO and A3C	● Importance weight truncation & bias correction ● Efficient TRPO	○ Excessive memory usage ○ Still complex	[83] [84]
PPO	Simplify the implementation of TRPO by using a surrogate objective function Achieve the best performance in continuous control tasks	● Clipped objective ● Adaptive KL penalty coefficient	○ Require network tuning	[82] [83] [84]

networks.

There have been a great number of policy gradient methods since the development of A3C. For instance, *UNsupervised REinforcement and Auxiliary Learning* (UNREAL) [67] uses multiple unsupervised pseudo-reward signals at the same time to improve the learning efficiency in complicated environments. Rather than estimating a stochastic policy, *Deterministic Policy Gradient* [68] (DPG) finds a deterministic policy, which significantly reduces data sampling. Moreover, *Deep Deterministic Policy Gradient* [69] (DDPG) combines DPG with DQN to enable the learning of a deterministic policy in a continuous action space using the actor-critic architecture. The authors in [70] even propose *Multi-agent DDPG* (MADDPG), which employs DDPG in multi-agent environments. To further stabilize the training process, the authors in [71] introduce the *Trust Region Policy Optimization* (TRPO) method, which integrates the *Kullback–Leibler divergence* [72] into the training procedure. However, the implementation of the method is complicated. In 2017, Wu *et al.* [73] proposed *Actor-Critic using Kronecker-Factored Trust Region* (ACKTR), which applies Kronecker-factored approximation curvature into gradient update steps. Additionally, the authors in [74] introduced an efficient off-policy sampling method based on A3C and an experience replay, namely *Actor-Critic with Experience Replay* (ACER). To simplify the implementation of TRPO, ACKTR, and ACER, *Proximal Policy Optimization* (PPO) [75] is introduced by using a clipped "surrogate" objective function together with stochastic gradient ascent. Finally, some studies combine a policy-based and value-based method such as [76], [77], [78] or an on-policy and off-policy method such as [79], [80]. Table II summarizes key deep RL methods and their reliable implementation repositories. Based on specific application domains, software managers can select a suitable deep RL method to act as a baseline for the target system.

B. Multi-Agent Method

In multi-agent learning, there are two widely used schemes in the literature: *individual* and *mutual*. In the first case, each agent in the system can be considered as an independent decision maker and other agents as a part of the environment. In this way, any deep RL methods in the previous subsection can be used in multi-agent learning. For instance, Tampuu *et al.* [86] used DQN to create an independent policy for each agent. The authors analyze the behavioral convergence of the involved agents with respect to cooperation and competition. Similarly, Leibo *et al.* [87] introduced a sequential social dilemma, which basically uses DQN to analyze the agent's strategy in Markov games such as Prisoner's Dilemma, Fruit Gathering, and Wolfpack. However, the approach limits the number of agents because the computational complexity increases with the number of policies. To overcome this obstacle, Nguyen *et al.* developed a behavioral control system [57] in which homogeneous agents can share the same policy. As a result, the method is robust and scalable. Another problem in multi-agent learning is the use of an experience replay, which amplifies the non-stationary problem that occurs due to asynchronous data sampling of different agents [88], [89]. A lenient approach [90] can subdue the problem by mapping transitions into decaying temperature values, which basically controls the magnitude of updating different policies.

In mutual scheme, agents can "speak" with each other via a settled communication channel. Moreover, agents are often trained in a centralized manner but eventually operate in a decentralized fashion when deployed [91]. In other words, a multi-agent RL problem can be divided into two sub-problems: a goal-directed problem and a communication problem. Specifically, Foerster *et al.* [92] introduced two communication schemes based on the centralized-decentralized rationale: *Reinforced Inter-Agent Learning* (RIAL) and *Differentiable Inter-Agent Learning* (DIAL). While RIAL reinforces agents' learning by sharing parameters, DIAL allows *inter-communication* between agents via a shared medium. Both methods, however, operate with a discrete number of communication actions. As opposed to RIAL and DIAL, the authors in [93] introduce a novel network architecture, namely *Communication Neural Net* (CommNet), which enables communication by using a continuous vector. As a result, the agents are trained to learn to communicate by backpropagation. However, CommNet limits the number of agents due to the increase of computational complexity. To make it scalable, Gupta *et al.* [94] introduced a parameter sharing method that can handle a large number of agents. However, the method only works with homogeneous systems. Finally, Nguyen *et al.* [57] extended the Gupta's study to heterogeneous systems by designing a behavioral control system. For rigorous study, a complete survey on multi-agent RL can be found in [95], [96], [97].

In summary, it is critical to address the following factors in multi-agent learning because they have a great impact on the target software architecture:

- It is preferable to employ the centralized-decentralized rationale in a multi-agent RL-based system because the training process is time-consuming and computationally expensive. A working system may require hundreds to thousands of training sessions by searching through the hyper-parameter space to find the optimal solution.
- The communication between agents can be *realistic* or *imaginary*. In realistic communication, agents "speak" with each other using an established communication protocol. However, there is no actual channel in imaginary communication. The agents are trained to collaborate using a specialized network architecture. For instance, OpenAI [98] proposes an actor-critic architecture where the critic is augmented with other agents' policy information. As a result, the two methods can differentiate how to design an RL-based system.
- A partially observable environment has a great impact on designing a multi-agent system because each agent has its own unique perspective of the environment. Therefore, it is important to first carefully examine the environment and application type to avoid any malfunction in the design.

C. RL Challenges

In this subsection, we briefly review major challenges while designing a deep RL-based system and corresponding solutions. To remain concise, the proposed framework is not concerned with these techniques but it is straightforward to extend the architecture to support these rectifications.

Catastrophic forgetting is a problem that occurs in *continual learning* and *multi-task learning*, *i.e.*, an another task is trained after learning the first task by using the same neural network. In this case, the neural network gradually forgets the knowledge of the first task to adopt the new one. One solution is to use regularization [99], [100] or a dense neural network [101], [102]. However, these approaches are only feasible with a limited number of tasks. Recent studies introduce more scalable approaches such as *Elastic Weight Consolidation* (EWC) [103] or *PathNet* [104]. While EWC finds a configuration of the network that yields the best performance in different tasks, PathNet uses a "super" neural network to fulfill the knowledge of different tasks in different paths.

Policy distillation [105] or *transfer learning* [106], [107] can be used to train an agent to learn individual tasks and collectively transfer the knowledge to a single network. Transfer learning is often used when the actual experiment is expensive and intractable. In this case, the network is trained with simulations and later is deployed into the target experiment. However, a *negative transfer* may occur when the performance of the learner is lower than the trainer. The authors in [108] introduced *Hierarchical Prioritized Experience Replay* that uses high-level features of the task and selects important data from the experience replay to mitigate the negative transfer. One recent study [109] aligned the mutual learning to achieve a comparable performance between the actual experiment and simulations.

Another obstacle in RL is dealing with *long-horizon* environments with *sparse rewards*. In such tasks, the agent hardly receives any reward and easily gets stuck in local minimum solutions. One straightforward solution is to use *reward shaping* [110] that continuously instructs the agent to achieve the objective. The problem can also be divided into a hierarchical tree of sub-problems where the parent problem has a higher abstraction than the child problem (*Hierarchical RL*) [111]. To encourage self-exploration, the authors in [112] introduced intrinsic reward signals to reinforce the agent to make a generic decision. State-of-the-art methods of *intrinsic motivation* can be found in [113], [114], [115]. Finally, Andrychowicz *et al.* [116] propose *Hindsight Experience Replay* that implicitly simulates *curriculum learning* [117] by creating imagined trajectories in the experience replay with positive rewards. In this way, the agent can learn from failures and automatically generalize a solution in success cases.

Finally, a variety of RL-related techniques are proposed to make RL feasible in large-scale applications. One approach is to augment the neural network with a "memory" to enhance sample efficiency in complicated environments [118], [119]. Additionally, to enforce scalability, many distributed methods have been proposed such as *Distributed Experience Replay* [120], deep RL acceleration [121], and distributed deep RL [122]. Finally, *imitation learning* can be used together with *inverse RL* to speed up training by directly learning from expert demonstrations and extracting the expert's cost function [123].

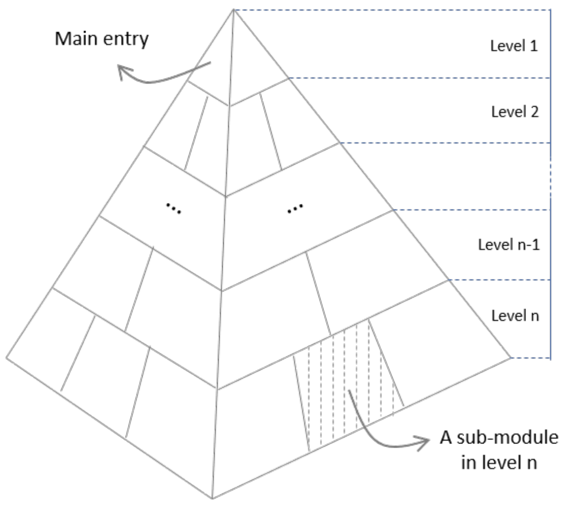

Fig. 2: A "pyramid" software architecture.

D. Deep RL Framework

In this subsection, we discuss the latest deep RL frameworks in the literature. We select the libraries based on different factors including Python-based implementation, clear documentation, reliability, and active community. Based on our analysis, software managers can select a suitable framework depending on project requirements.

- *Chainer* – Chainer [84] is a powerful and flexible framework for neural networks. The framework is currently supported by IBM, Intel, Microsoft, and Nvidia. It provides an easy way to manipulate neural networks such as by creating a customized network, visualizing a computational graph, and supporting a debug mode. It also implements a variety of deep RL methods. However, the Chainer's architecture is complicated and requires a great effort to develop a new deep RL method. The number of integrated environments is also limited, *e.g.*, Atari [124], OpenAI Gym [125], and Mujoco [126].
- *Keras-RL* – Keras-RL [127] is a friendly deep RL library, which is recommended for deep RL beginners. However, the library provides a limited number of deep RL methods and environments.
- *TensorForce* – TensorForce [129] is an ambitious project that targets both industrial applications and academic research. The library has the best modular architecture we have reviewed so far. Therefore, it is convenient to use the framework to integrate customized environments, modify network configurations, and tweak deep RL algorithms. However, the framework has a deep software stack ("pyramid" model) that includes many abstraction layers, as shown in Fig. 2. This hinders novice readers from prototyping a new deep RL method.
- *OpenAI Baselines* – OpenAI Baselines [83] is a high-quality framework for contemporary deep RL. In contrast to TensorForce, the library is suitable for researchers who want to reproduce original results. However, OpenAI Baselines is unstructured and incohesive.
- *RLLib* – RLLib [82] is a well-designed deep RL frame-

work that provides a means to deploy deep RL in distributed systems. Therefore, the usage of RLLib is not friendly for RL beginners.

- *RLLab* – RLLab [132] provides a diversity of deep RL methods including TRPO, DDPG, Cross-Entropy Method, and Evolutionary Strategy. The library is friendly to use but not straightforward for modifications.

In summary, most deep RL frameworks focus on the performance of deep RL methods. As a result, those frameworks limits code legibility, which basically restricts RL beginners from readability and modifications. In this paper, we propose a comprehensive framework that has the following properties:

- Allow new users to prototype a deep RL method in a short period of time by following a modular design. As opposed to TensorForce, we limit the number of abstraction layers and avoid the pyramid structure.
- The framework is friendly with a simplified user interface. We provide an API based on three key concepts: policy network, network configuration, and learner.
- Enforce scalability and generalization while supporting multiple agents, multiple objectives, and human-machine interactions.
- Finally, we introduce a concept of unification and transparency by creating plugins. Plugins are gateways that extract learners from other libraries and plug them into our proposed framework. In this way, users can interact with different frameworks using the same interface.

III. Software Architecture

In this section, we examine core components towards designing a comprehensive deep RL framework, which basically employs generality, flexibility, and interoperability. We aim to support a broad range of RL-related applications that involve multiple agents, multiple objectives, and human-agent interaction. We use the following pseudocode to describe a function signature:

- **function_name**($[param1, param2, ...]$) $\rightarrow$ $[return1, return2, ...]$ or $\{return1, return2, ...\}$ or A

where $\rightarrow$ denotes a return operation, A is a scalar value, [...] denotes an array, and {...} denotes a list of possible values of a single variable.

A. Environment

First, we create a unique interface for the *environment* to establish a communication channel between the framework and agents. However, to reduce complexity, we put any human-related communication into the environment. As a result, human interactions can be seen as a part of the environment and are hidden from the framework, *i.e.*, the environment provides two interfaces: one for the framework and one for human, as shown in Fig. 3. While the framework interface is often in programming level (functions), the human interface has a higher abstraction mostly in human understanding forms such as voice dictation, gesture recognition, or control system.

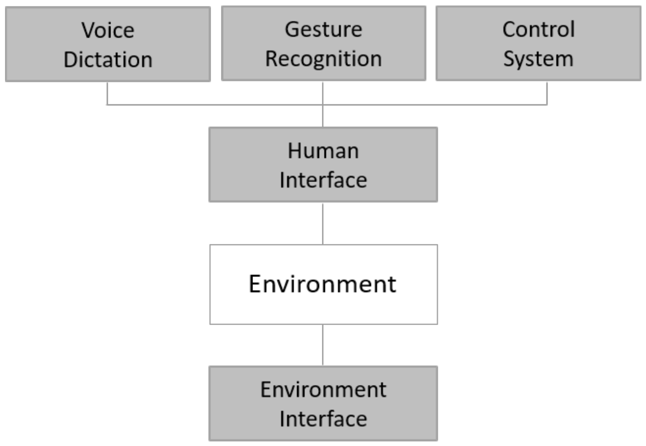

Fig. 3: A conceptual model of the environment with a human interface.

Essentially, the environment's framework interface should provide the following functions:

- **clone**(): the environment can duplicate itself. The function is useful when an RL algorithm requires multiple learners at the same time (*e.g.* A3C).
- **reset**(): reset the environment to the initial state. The function must be called after or before an episode.
- **step**($[a_1, a_2, ..., a_N]$) $\rightarrow [r_1, r_2, ..., r_M]$: executes N specified actions of N agents in the environment. The function returns M rewards, each of which represents an objective function.
- **get_state**() $\rightarrow [s_1, s_2, ..., s_N]$: retrieves the current states of the environment. If the environment is a partially observable MDP, the function returns N states, which each presents the current state of an agent. However, if the environment is a fully observable MDP, we have $s_1 = s_2 = ... = s_N = s$.
- **is_terminal**() $\rightarrow$ {True, False}: checks if the episode is terminated.
- **get_number_of_objectives**() $\rightarrow M$: is a helper function that indicates the number of objectives in the environment.
- **get_number_of_agents**() $\rightarrow N$: is a helper function that indicates the number of agents in the environment.

Finally, it is important to consider the following questions while designing an environment component as they have a great impact on subsequent design stages:

- *Is it a simulator or a wrapper?* In the case of a wrapper, the environment is already developed and configured. Our duty is to develop a wrapper interface that can compatibly interact with the framework. In contrast to the wrapper, developing a simulator is complicated and requires expert knowledge. In real-time applications, we may first develop a simulator in C/C++ (for better performance) and then create a Python wrapper interface (for easier integration). In this case, we need to develop both a simulator and a wrapper.
- *Is it stochastic or deterministic?* Basically, a stochastic environment is more challenging to implement than a

deterministic one. There are potential factors that are likely to contribute to the randomness of the environment. For example, a company intends to open a bike rental service. *N* bikes are equally distributed into *M* potential places. However, at a specific time, place *A* still has a plenty of bikes because there is no customer. As a result, bikes in place *A* are delivered to other places where the demand is higher. The company seeks development of an algorithm that can balance the number of bikes in each place over time. In this example, the bike rental service is a stochastic environment. We can start building a simple stochastic model based on Poisson distribution to represent the rental demand in each place. We end up with a complicated model based on a set of observable factors such as rush hour, weather, weekend, festival, etc. Depending on the stochasticity of the model, we can decide to use a model-based or model-free RL method.

- *Is it complete or incomplete?* A complete environment at any time provides sufficient information to construct a branch of possible moves in the future (*e.g.* Chess or Go). The completeness can help to decide an effective RL method later. For instance, a complete environment can be solved with a careful planning rather than a trial-and-error approach.
- *Is it fully observable or partially observable?* The observability of environment is essential when designing a deep neural network. Partially observable environments might require recurrent layers or an attention mechanism to enhance the network's capacity during the training. A self-driving scenario is partially observable while a board game is fully observable.
- *Is it continuous or discrete?* As described in Table I, this factor is important to determine the type of methods used such as policy-based or value-based methods or the network configuration such as actor-critic architectures.
- *How many objectives does it have?* Real-world applications often have multiple objectives. If the importance weights between objectives can be identified in the beginning, it is reasonable to use a single-policy RL method. Alternatively, a multi-policy RL method can prioritize the importance of an objective in real time.
- *How many agents does it have?* A multi-agent RL-based system is much more complicated than a single-agent counterpart. Therefore, it is essential to analyze the following factors of a multi-agent system before delving into the design: the number of agents, the type of agents, communication abilities, cooperation strategies, and competitive potentials.

B. Network

The *neural network* is also a key module of our proposed framework, which includes a network configuration and a policy network, as illustrated in Fig. 4. A network configuration defines the deep neural network architecture (*e.g.* CNN or LSTM), loss functions (*e.g.* Mean Square Error or Cross Entropy Loss), and optimization methods (*e.g.* Adam or SGD). Depending on the project's requirements, a configuration can be divided to different abstraction layers, where the lower abstraction layer is used as a mapping layer for the higher abstraction layer. In the lowest abstraction level (programming language), a configuration is implemented by a deep learning library, such as Pytorch [130] (with dynamic graph) or TensorFlow (with static graph). The next layer is to use a scripting language, such as *xml* or *json*, to describe the network configuration. This level is useful because it provides a faster and easier way to configure a network setting. For those who do not have much knowledge of implementation details such as system analysts, a graphical user interface can assist them. However, there is a trade-off here: the higher abstraction layer achieves better usability and productivity but has a longer development cycle.

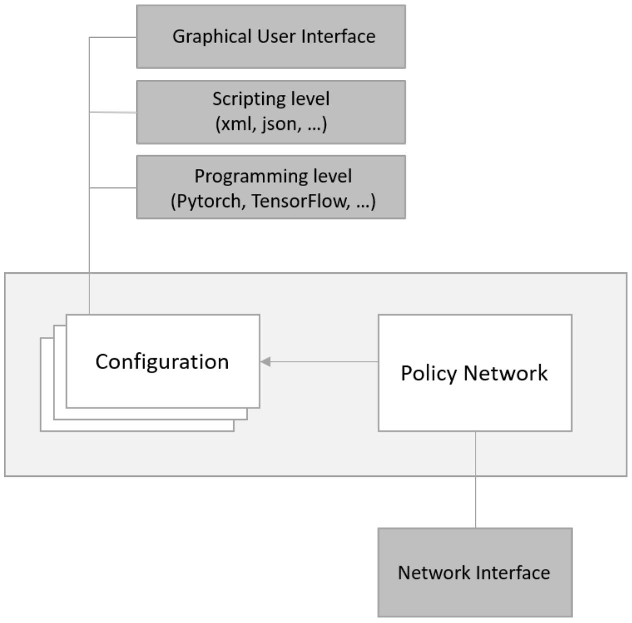

Fig. 4: A neural network module includes a network configuration and a policy network.

A policy network is a composite component that includes a number of network configurations. However, the dependency between a policy network and a configuration can be weak, *i.e.*, an *aggregation* relationship. The policy network's objective is twofold. It provides a high-level interface that maintains connectivity with other modules in the framework, and it initializes the network, saves the network's parameters into checkpoints, and restores the network's parameters from checkpoints. Finally, the neural network interface should provide the following functions:

- **create_network**() $\rightarrow [\theta_1, \theta_2, ..., \theta_K]$: instantiates a deep neural network by using a set of network configurations. The function returns the network's parameters (references) $\theta_1, \theta_2, ..., \theta_K$.
- **save_model**(): saves the current network's parameters into a checkpoint file.
- **load_model**([chk]): restores the current network's parameters from a specified checkpoint file *chk*.
- **predict**($[s_1, s_2, ..., s_N]$) $\rightarrow [a_1, a_2, ..., a_N]$: given the current states of N agents $s_1, s_2, ..., s_N$, the function uses the

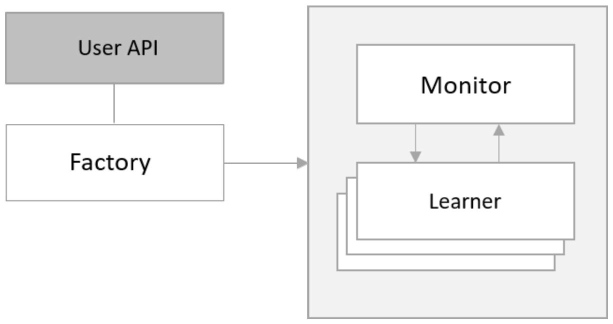

Fig. 5: A high-level design of a learner module.

network to predict the next N actions $a_1, a_2, ..., a_N$.
- **train_network**([data_dict]): trains the network by using the given data dictionary. The data dictionary often includes the current states, current actions, next states, terminal flags, and miscellaneous information (global time step or objective weights) of N agents.

C. Learner

The last key module of our proposed framework is a *learner*, as shown in Fig. 5. While the environment module and the network module create the application's shell, the learner plays as an engine that allows the system to operate properly. The three modules jointly create the backbone of the system. In particular, the learner uses the environment module to generate episodes. It manages the experience replay memory and defines the RL implementation details, such as multi-step learning, multi-threading, or reward shaping. The learner is often created together with a monitor. The monitor is used to manage multiple learners (if multi-threading is used) and collect any data from the learners during training, such as performance information for debugging purposes and post-evaluation reports. Finally, the learner collects necessary data, packs it into a dictionary, and sends it to the network module for training.

Additionally, a *factory* pattern [131] can be used to hide the operating details between the monitor and the learners. As a result, the factory component promotes higher abstraction and usability through a simplified user API as below:

- **create_learner**([monitor_dict, learner_dict]) $\rightarrow$ obj: The factory creates a learner by using the monitor's data dictionary (batch size, the number of epochs, report frequency, etc) and the learner's data dictionary (the number of threads, epsilon values, reward clipping thresholds, etc.).
- **train**(): trains the generated learner.
- **evaluate**(): evaluates the generated learner.

D. Plugin

There have been a great number of RL methods in the literature. Therefore, it is impractical to implement all of them. However, we can reuse the implementation from existing libraries such as TensorForce, OpenAI Baselines, or RLLab. To enforce flexibility and interoperability, we introduce a concept of unification by using plugins. A plugin is a piece of program that extracts learners or network configurations from third party libraries and plugs them into our framework. As a result, the integrated framework provides a unique user API but supports a variety of RL methods. In this way, users do not need to learn different libraries. The concept of unification is described in Fig. 6.

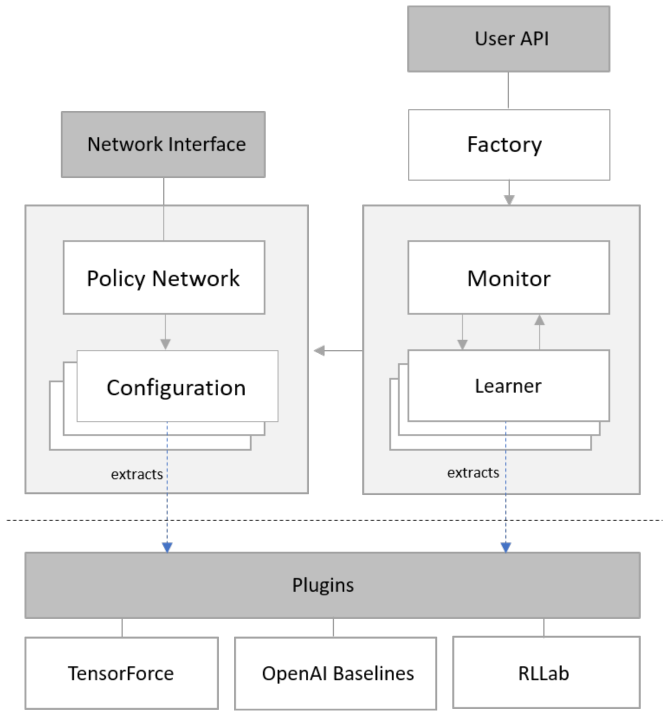

Fig. 6: A unification of different RL libraries by using plugins.

A plugin can also act as a conversion program that converts the environment's interface of this library into the environment's interface of other libraries. As a result, the proposed framework can work with any environments in third party libraries and vice versa. Therefore, a plugin should include the following functions:

- **convert_environment**([source_env]) $\rightarrow$ target_env: converts the environment's interface from the source library to the environment's interface defined in the target library.
- **extract_learner**([param_dict]) $\rightarrow$ learner: extracts the learner from the target library.
- **extract_configuration**([param_dict]) $\rightarrow$ config: extracts the network configuration from the target library.

E. Overall Structure

Putting everything together, we have a sequential diagram of the training process, as described in Fig. 6. The workflow breaks down the training process into smaller procedures. Firstly, the factory instantiates a specified learner (or a plugin) and sends its reference to the monitor. The monitor clones the learner into multiple learner threads. Each learner thread is run until the number of epochs exceeds a predefined value, K. The second loop within the learner thread is used to generate episodes. In each episode, a learner thread perceives the current states of the environment and predicts next actions

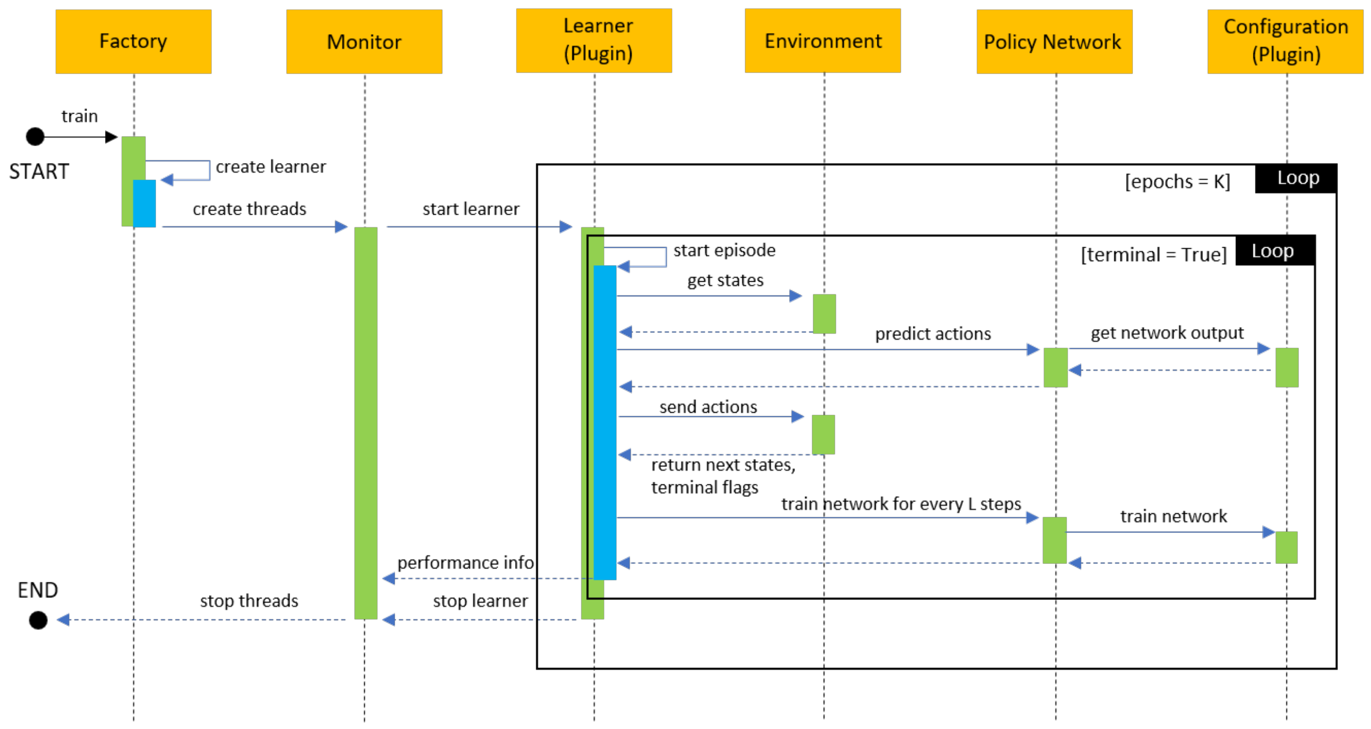

Fig. 7: A UML sequential diagram of the training process.

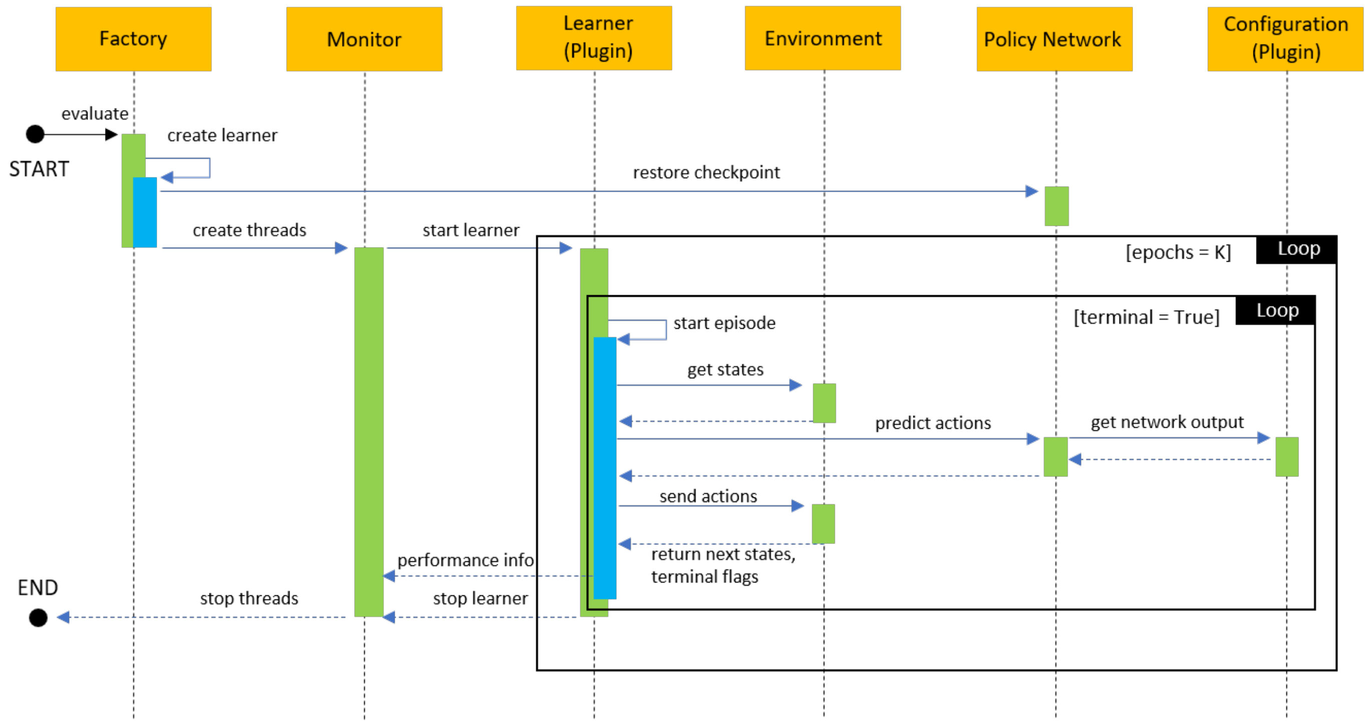

Fig. 8: A UML sequential diagram of the evaluation process.

using the policy network and configuration network. The next actions are applied to the environment. The environment returns next states and a terminal flag. Finally, the policy network is trained for every L-step. There are minor changes in the evaluation process, as shown in Fig. 8. First, the policy network's parameters are restored from a specified checkpoint file while initializing the learner. Second, all training procedure calls are discarded while generating episodes.

To enhance usability and reduce redundancy, it is advisable to implement the framework in *Object-Oriented Programming* (OOP). In this way, a new learner (configuration) can be easily developed by inheriting existing learners (configurations) in

TABLE III: Demonstration codes of different use cases [58].

Use case	Description	Source Code
1. How to inherit an existing learner?	Develop a Monte-Carlo learner that inherits the existing Q-Learning learner	fruit/learners/mc.py
2. Develop a new environment	Create a Grid World [1] environment that follows the framework's interface	fruit/envs/games/grid_world/
3. Develop a multi-agent environment with human-agent interaction	Create a Tank Battle [57] game in which humans and AI agents can play together	fruit/envs/games/tank_battle/
4. Multi-objective environment and multi-objective RL	Use a multi-objective learner (MO Q-Learning) to train an agent to play Mountain Car [47]	fruit/samples/basic/multi_objectives_test.py
5. Multi-agent learner with human-agent interaction	Create a multi-agent RL method based on A3C [94] and apply it to Tank Battle	fruit/samples/tutorials/chapter_6.py
6. How to use a plugin?	Develop a TensorForce plugin, extract the PPO learner, and train an agent to play Cart Pole [1]	fruit/plugins/quick_start.py

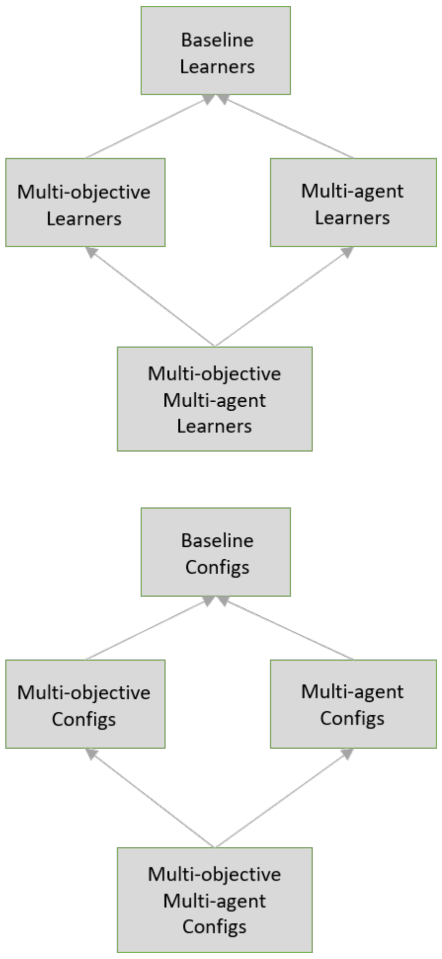

Fig. 9: An inheritance relationship between learners and configurations.

the framework, as shown in Fig. 9.

IV. Conclusions

In this paper, we briefly review recent advances in the RL literature with respect to multi-agent learning, multi-objective learning, and human-machine interactions. We also examine different deep RL libraries and analyze their limitations. Finally, we propose a novel design of a deep RL framework that exhibits great potential in terms of usability, flexibility, and interoperability. We highlight any considerable notices during the design so that software managers can avoid possible mistakes while designing an RL-based application.

The proposed framework can be considered as a *template* to design a real-world RL-based application. Because the framework is developed in OOP, it is beneficial to utilize OOP principles, such as inheritance, polymorphism, and encapsulation to expedite the whole development process. We advisedly created a "soft" software layer stack, where the number of modules is minimal while maintaining a certain level of cohesion. As a result, the learning curve is not steep. By providing a simplified API, the framework is suitable for novice readers who are new to deep RL, especially software engineers. Finally, the framework acts as a bridge to connect different RL communities around the world.

Our ultimate goal is to build an educational software platform for deep RL. The next development milestone includes three steps:

- Implement a variety of plugins for the proposed framework.
- Develop a GUI application that can configure the neural network, modify the learner, and visualize the RL workflow.
- Complete documentation including tutorials and sample codes.

Appendix

To keep the paper brief, we provide documentation of the proposed framework as online materials [132]. These include an installation guide, code samples, benchmark scores, tutorials, an API reference guide, a class diagram, and a package diagram. Table III lists demonstration codes of different use cases (codebase [58]).

Acknowledgement

The authors wish to thank our colleagues in the Institute for Intelligent Systems Research and Innovation for their comments and helpful discussions. We truly appreciate Nguyen Chau, a principle IT product manager at Atlassian, who shared his expertise in the field to eradicate any misunderstandings in this paper. We also thank Dr. Thanh Nguyen, University of Chicago, for being an active adviser in the design process. Finally, we send our gratefulness to the RL community who provided crucial feedback during the project's beta testing.

REFERENCES

[1] R. S. Sutton and G. B. Andrew, *Introduction to Reinforcement Learning*. Cambridge: MIT press, 1998.

[2] N. D. Nguyen, T. Nguyen, and S. Nahavandi, "System design perspective for human-level agents using deep reinforcement learning: A survey," *IEEE Access*, vol. 5, pp. 27091–27102, 2017.

[3] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 50–56, 2016.

[4] T. T. Nguyen, and V. J. Reddi, "Deep reinforcement learning for cyber security," *arXiv preprint arXiv:1906.05799*, 2019.

[5] D. Fox, W. Burgard, H. Kruppa, and S. Thrun, "A probabilistic approach to collaborative multi-robot localization," *Autonomous Robots*, vol. 8, no. 3, pp. 325–344, 2000.

[6] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Autonomous Robots*, vol. 27, no. 1, pp. 55–73, 2009.

[7] K. Mülling, J. Kober, O. Kroemer, and J. Peters, "Learning to select and generalize striking movements in robot table tennis," *International Journal of Robotics Research*, vol. 32, no. 3, pp. 263–279, 2013.

[8] T. G. Thuruthel et al., "Model-based reinforcement learning for closed-loop dynamic control of soft robotic manipulators," *IEEE Transactions on Robotics*, vol. 35, pp.124–134, 2018.

[9] R. H. Crites and A. G. Barto, "Elevator group control using multiple reinforcement learning agents," *Machine Learning*, vol. 33, no. 2–3, pp. 235–262, 1998.

[10] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intelligent Transport Systems*, vol 4, no. 2, pp. 128–135, 2010.

[11] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li, "DRN: A deep reinforcement learning framework for news recommendation," In *Proceedings of the 2018 World Wide Web Conference*, pp. 167–176, 2018.

[12] J. Jin, C. Song, H. Li, K. Gai, J. Wang, and W. Zhang, "Real-time bidding with multi-agent reinforcement learning in display advertising.," In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pp. 2193–2201, 2018.

[13] M. Campbell, A. J. Hoane, and F. H. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 57–83, 2002.

[14] G. Tesauro and G. R. Galperin, "On-line policy improvement using monte-carlo search," in *Advances in Neural Information Processing Systems*, pp. 1068–1074, 1997.

[15] G. Tesauro, "Temporal difference learning and td-gammon," *Communication*, vol. 38, no. 3, pp. 58–68, 1995.

[16] R. Bellman, *Dynamic Programming*. Princeton: Princeton University Press, 2010.

[17] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.

[18] T. J. Ross, *Fuzzy Logic with Engineering Applications*. John Wiley & Sons, 2005.

[19] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 4, pp. 355–366, 2014.

[20] T. Salimans, J. Ho, X. Chen, A. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning, *arXiv preprint arXiv:1703.03864*, 2017.

[21] D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Belmont, MA: Athena scientific, 1995.

[22] J. Duchi and Y. Singer, "Efficient online and batch learning using forward backward splitting," *Journal of Machine Learning Research*, vol. 10, pp. 2899–2934, 2009.

[23] S. Adam, L. Busoniu, and R. Babuska, "Experience replay for real-time reinforcement learning control, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 2, pp. 201–212, 2011.

[24] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, 2015.

[25] A. Krizhevsky, S. Ilya, and E. H. Geoffrey, "Imagenet classification with deep convolutional neural networks." In *Advances in Neural Information Processing Systems*, 2012.

[26] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no 7587, 2016.

[27] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2011.

[28] G. Tesauro, "TD-Gammon, a self-teaching backgammon program, achieves master-level play, *Neural Computation*, vol. 6, no. 2, pp. 215–219, 1992.

[29] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 19, pp. 70–76, 2017.

[30] S. S.-Shwartz, S. Shaked, and S. Amnon, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[31] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," *Experimental Robotics IX*, pp. 363–372, 2006.

[32] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takac, "Reinforcement learning for solving the vehicle routing problem," In *Advances in Neural Information Processing Systems*, pp. 9839–9849, 2018.

[33] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[34] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[35] J. Z. Leibo et al., "Multi-agent reinforcement learning in sequential social dilemmas," In *Conference on Autonomous Agents and Multi-Agent Systems*, 2017.

[36] X. Wang and T. Sandholm, "Reinforcement learning to play an optimal Nash equilibrium in team Markov games," In *Advances in Neural Information Processing Systems*, pp. 1603–1610, 2003.

[37] J. Peters and S. Stefan, "Natural actor-critic," *Neurocomputing*, vol. 71, no 7–9, pp. 1180-1190, 2008.

[38] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," In *Advances in Neural Information Processing Systems*, pp. 1008–1014, 2000.

[39] H. He, J. Boyd-Graber, K. Kwok, and H. Daume III, "Opponent modeling in deep reinforcement learning," In *International Conference on Machine Learning*, pp. 1804–1813, 2016.

[40] F. Southey, M. P. Bowling, B. Larson, C. Piccione, N. Burch, D. Billings, and C. Rayner, "Bayes' bluff: Opponent modelling in poker," *arXiv preprint arXiv:1207.1411*, 2012.

[41] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," In *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.

[42] G. Palmer et al., "Lenient multi-agent deep reinforcement learning," In *International Conference on Autonomous Agents and MultiAgent Systems*, 2018.

[43] L. Bu, B. Robert, and D.S. Bart, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 38, pp. 156–172, 2008.

[44] K. Tuyls and W. Gerhar, "Multiagent learning: Basics, challenges, and prospects," *AI Magazine*, vol. 33, 2012.

[45] S. Natarajan and P. Tadepalli, "Dynamic preferences in multi-criteria reinforcement learning," In *International Conference on Machine Learning*, pp. 601–608, 2005.

[46] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley, "A survey of multi-objective sequential decision-making," *Journal of Artificial Intelligence Research*, vol. 48, pp.67–113, 2013.

[47] K. Van Moffaert, and A. Nowe, "Multi-objective reinforcement learning using sets of pareto dominating policies," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3483–3512, 2014.

[48] L. Barrett and S. Narayanan, "Learning all optimal policies with multiple criteria," In *International Conference on Machine Learning*, pp. 41–47, 2008.

[49] P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker, "Empirical evaluation methods for multiobjective reinforcement learning algorithms," *Machine Learning*, vol. 84, no. 1–2, pp. 51–80, 2011.

[50] H. Mossalam, Y. M. Assael, D. M. Roijers, and S. Whiteson, "Multi-objective deep reinforcement learning," *arXiv preprint arXiv:1610.02707*, 2016.

[51] H. Van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang, "Hybrid reward architecture for reinforcement learning," In *Advances in Neural Information Processing Systems*, pp. 5392–5402, 2017.

[52] T. T. Nguyen, "A multi-objective deep reinforcement learning framework," *arXiv preprint arXiv:1803.02965*, 2018.

[53] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[54] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, pp. 70–76, 2017.

[55] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mane. Concrete problems in AI safety. In *arXiv:1606.06565*, 2016.

[56] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, pages 4302–4310, 2017.

[57] N. D. Nguyen, T. T. Nguyen, S. Nahavandi, "Multi-agent behavioral control system using deep reinforcement learning," *Neurocomputing*, 2019.

[58] N. D. Nguyen and T. T. Nguyen "Fruit-API," https://github.com/garlicdevs/Fruit-API, 2019.

[59] V. Mnih et al., "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[60] H. V. Hasselt, G. Arthur, and S. David, "Deep reinforcement learning with double q-learning," In *Conference on Artificial Intelligence*, 2016.

[61] Z. Wang et al., "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[62] T. Schaul et al., "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[63] M. Hausknecht and S. Peter, "Deep recurrent q-learning for partially observable mdps," In *AAAI Fall Symposium Series*, 2015.

[64] I. Sorokin et al., "Deep attention recurrent Q-network," *arXiv preprint arXiv:1512.01693*, 2015.

[65] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," In *AAAI Conference on Artificial Intelligence*, 2018.

[66] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning*, 2016.

[67] M. Jaderberg et al., "Reinforcement learning with unsupervised auxiliary tasks," *arXiv preprint arXiv:1611.05397*, 2016.

[68] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International Conference on Machine Learning*, 2014.

[69] T.P Lillicrap, J.Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[70] R. Lowe, Y. Wu, A. Tamar, J. Harb, O.P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," In *Advances in Neural Information Processing Systems*, pp. 6379–6390, 2017.

[71] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," In *International Conference on Machine Learning*, pp. 1889–1897, 2015.

[72] T. Van Erven and P. Harremos, "Renyi divergence and Kullback-Leibler divergence," *IEEE Transactions on Information Theory*, vol. 60, no. 7, pp. 3797–3820, 2014.

[73] Y. Wu, E. Mansimov, R.B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," In *Advances in Neural Information Processing Systems*, pp. 5279–5288, 2017.

[74] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224*, 2016.

[75] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[76] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, "Bridging the gap between value and policy based reinforcement learning," In *Advances in Neural Information Processing Systems*, pp. 2775–2785, 2017.

[77] B. O'Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih, "Combining policy gradient and Q-learning," *arXiv preprint arXiv:1611.01626*, 2016.

[78] J. Schulman, X. Chen, and P. Abbeel, "Equivalence between policy gradients and soft q-learning," *arXiv preprint arXiv:1704.06440*, 2017.

[79] A. Gruslys, W. Dabney, M.G. Azar, B. Piot, M. Bellemare, and R. Munos, "The Reactor: A fast and sample-efficient actor-critic agent for reinforcement learning," *arXiv preprint arXiv:1704.04651*, 2017.

[80] S.S Gu, T. Lillicrap, R.E. Turner, RZ. Ghahramani, B. Schlkopf, and S. Levine, "Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning," In *Advances in Neural Information Processing Systems*, pp. 3846–3855, 2017.

[81] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare, "Dopamine: A research framework for deep reinforcement learning," *arXiv preprint arXiv:1812.06110*, 2018.

[82] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, and I. Stoica, "Ray rllib: A composable and scalable reinforcement learning library," *arXiv preprint arXiv:1712.09381*, 2017.

[83] C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "OpenAI baselines," 2017.

[84] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," In *Proceedings of Workshop on Machine Learning Systems in Conference on Neural Information Processing Systems*, pp. 1–6, 2015.

[85] K. Miyoshi, 2017. Available: https://github.com/miyosuda/unreal.

[86] A. Tampuu *et al.*, "Multiagent cooperation and competition with deep reinforcement learning," *PloS One*, vol. 12, no. 4, Apr. 2017.

[87] J.Z. Leibo, v. Zambaldi, M. Lanctot, J. Marecki, and T. Graepel, "Multi-agent reinforcement learning in sequential social dilemmas," In *Conference on Autonomous Agents and MultiAgent Systems*, pp. 464–473, 2017.

[88] L. Bu, B. Robert, and D.S. Bart, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 38, pp. 156–172, 2008.

[89] K. Tuyls and W. Gerhar, "Multiagent learning: Basics, challenges, and prospects," *AI Magazine*, vol. 33, 2012.

[90] G. Palmer, K. Tuyls, D. Bloembergen, and R. Savani, "Lenient multi-agent deep reinforcement learning," In *International Conference on Autonomous Agents and MultiAgent Systems*, pp. 443–451, 2018.

[91] L. Kraemer and B. Banerjee, "Multi-agent reinforcement learning as a rehearsal for decentralized planning," *Neurocomputing*, pp. 82–94, 2016.

[92] J. Foerster, Y. M. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," in *Advances in Neural Information Processing Systems*, pp. 2137–2145, 2016.

[93] S. Sukhbaatar and R. Fergus, "Learning multiagent communication with backpropagation," in *Advances in Neural Information Processing Systems*, pp. 2244–2252, 2016.

[94] J. K. Gupta, M. Egorov, and M. Kochenderfer, "Cooperative multi-agent control using deep reinforcement learning," In *International Conference on Autonomous Agents and Multiagent Systems*, pp. 66–83, 2017.

[95] T. Nguyen, N. D. Nguyen, and S. Nahavandi, "Deep reinforcement learning for multi-agent systems: A review of challenges, solutions and applications," *arXiv preprint arXiv:1812.11794*, 2018.

[96] M. Egorov, "Multi-agent deep reinforcement learning," *CS231n: Convolutional Neural Networks for Visual Recognition*, 2016.

[97] L. Bu, B. Robert, and D.S. Bart, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 38, pp. 156–172, 2008.

[98] R. Lowe, Y. Wu, A. Tamar, J. Harb, O.P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," In *Advances in Neural Information Processing Systems*, pp. 6379–6390, 2017.

[99] F. Girosi, M. Jones, and T. Poggio, "Regularization theory and neural networks architectures," *Neural Comput.*, vol. 7, no. 2, pp. 219–269, 1995.

[100] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv:1312.6211 [cs, stat]*, Dec. 2013.

[101] S. Thrun and L. Pratt, *Learning to Learn*. Boston, MA: Kluwer Academic Publishers, 1998.

[102] A. A. Rusu *et al.*, "Progressive neural networks," *arXiv:1606.04671 [cs]*, 2016.

[103] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," in *Proc. Nat. Acad. Sci.*, pp. 3521–3526, 2017.

[104] C. Fernando *et al.*, "Pathnet: Evolution channels gradient descent in super neural networks," *arXiv preprint arXiv:1701.087*, 2017.

[105] A. A. Rusu *et al.*, "Policy distillation," *arXiv:1511.06295 [cs]*, Nov. 2015.

[106] H. Yin and S. J. Pan, "Knowledge transfer for deep reinforcement learning with hierarchical experience replay," in *Proc. AAAI Conf. Artif. Intell.*, pp. 1640–1646, Jan. 2017.

[107] E. Parisotto, J. L. Ba, and R. Salakhutdinov, “Actor-mimic: Deep multitask and transfer reinforcement learning,” *arXiv:1511.06342 [cs]*, 2015.
[108] H. Yin and S. J. Pan, “Knowledge transfer for deep reinforcement learning with hierarchical experience replay,” in *Proc. AAAI Conf. Artif. Intell.*, pp. 1640–1646, Jan. 2017.
[109] M. Wulfmeier, I. Posner, and P. Abbeel, “Mutual alignment transfer learning,” *arXiv preprint arXiv:1707.07907*, 2017.
[110] M. Grzes, and D. Kudenko, “Online learning of shaping rewards in reinforcement learning,” *Neural Networks*, vol. 23, no. 4, pp. 541–550, 2010.
[111] A. G. Barto and S. Mahadevan, “Recent advances in hierarchical reinforcement learning,” *Discrete Event Dyn. Syst.*, vol. 13, no. 4, pp. 341–379, 2003.
[112] T. D. .Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, “Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation,” in *Adv. Neural Inf. Process. Syst.*, pp. 3675–3683, 2016.
[113] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A.A. Efros, “Large-scale study of curiosity-driven learning,” *arXiv preprint arXiv:1808.04355*, 2018.
[114] D. Pathak, P. Agrawal, A.A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” In *Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17, 2017.
[115] G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos, “Count-based exploration with neural density models,” In *International Conference on Machine Learning*, pp. 2721–2730, 2017.
[116] M. Andrychowicz *et al.*, “Hindsight experience replay,” In *Advances in Neural Information Processing Systems*, 2017.
[117] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” In *International Conference on Machine Learning*, pp. 41–48, 2009.
[118] A. Santoro, R. Faulkner, D. Raposo, J. Rae, M. Chrzanowski, T. Weber, D. Wierstra, O. Vinyals, R. Pascanu, and T. Lillicrap, “Relational recurrent neural networks,” In *Advances in Neural Information Processing Systems*, pp. 7299–7310, 2018.
[119] E. Parisotto and R. Salakhutdinov, “Neural map: Structured memory for deep reinforcement learning,” *arXiv preprint arXiv:1702.08360*, 2017.
[120] D. Horgan, J. Quan, D. Budden, G Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver, “Distributed prioritized experience replay,” *arXiv preprint arXiv:1803.00933*, 2018.
[121] A. Stooke and P. Abbeel, “Accelerated methods for deep reinforcement learning,” *arXiv preprint arXiv:1803.02811*, 2018.
[122] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, and I. Stoica, “Rllib: Abstractions for distributed reinforcement learning,” *arXiv preprint arXiv:1712.09381*, 2017.
[123] J. Ho and S. Ermon, “Generative adversarial imitation learning,” In *Advances in Neural Information Processing Systems*, pp. 4565–4573, 2016.
[124] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, 2013.
[125] G. Brockman *et al.*, “OpenAI gym,” *arXiv:1606.01540 [cs]*, 2016.
[126] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” In *International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.
[127] M. Plappert, “keras-rl,” https://github.com/matthiasplappert/keras-rl, 2016.
[128] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, “Benchmarking deep reinforcement learning for continuous control,” In *International Conference on Machine Learning*, pp. 1329–1338, 2016.
[129] M. Schaarschmidt, A. Kuhnle, and K. Fricke, “TensorForce: A TensorFlow library for applied reinforcement learning,” 2017.
[130] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” In *Advances in Neural Information Processing Systems*, 2019.
[131] B. Ellis, J. Stylos, and B. Myers, “The factory pattern in API design: A usability evaluation,” In *International Conference on Software Engineering*, 2007.
[132] N. D. Nguyen and T. T. Nguyen, “FruitLAB,” http://fruitlab.org/, 2019.