electronics

Article

Transfer Learning Method from Parameter Scene to Physical Scene Based on Self-Game Theory

Nan Zhang [1,*], Guolai Yang [1], Bo Su [2,3] and Weilong Song [2,3]

[1] School of Mechanical Engineering, Nanjing University of Science and Technology, Nanjing 210094, China; yyanggl@njust.edu.cn
[2] China North Artificial Intelligence & Innovation Research Institute, Beijing 100072, China
[3] Collective Intelligence & Collaboration Laboratory, Beijing 100072, China
* Correspondence: zhang_n@njust.edu.cn

Abstract: Due to the inherent challenges in meeting the comprehensive training requirements of combat scenarios within a digital environment, this paper proposes innovative solutions. Firstly, a parametric scene training method is introduced, aiming to enhance the adaptability of the training process. Secondly, the utilization of the digital environment as a test environment is suggested, which can significantly improve the efficiency of iterative virtual-to-real conversion. Additionally, employing digital simulation as a pre-test environment for the physical setting can effectively reduce deployment costs and enhance the safety of virtual reality migration experiments. The proposed approach involves constructing a training model based on the parameterized environment and implementing a feedback loop utilizing the digital environment agent model. This framework enables continuous verification of the environment's parameters, ensuring the accuracy of decision-making strategies and evaluating the transferability of the decision-making model.

Keywords: transfer learning method; parameterizing scene to physical scene; agent model; game theory

check for updates

Citation: Zhang, N.; Yang, G.; Su, B.; Song, W. Transfer Learning Method from Parameter Scene to Physical Scene Based on Self-Game Theory. *Electronics* **2024**, *13*, 1327. https://doi.org/10.3390/electronics13071327

Academic Editor: Yue Wu

Received: 24 December 2023
Revised: 24 March 2024
Accepted: 26 March 2024
Published: 1 April 2024

1. Introduction

The general method of virtual and real transmission is to train in a digital environment and deploy directly in physical scenes. The disadvantages of this migration are high deployment costs and low iteration speed. In traditional experiments of virtual–real conversion, there is no test environment, only a training environment and model deployment. The training environment is usually a digital environment, not a parametric environment.

The ultimate goal of controlling unmanned platforms is to train strategies that can complete operational tasks in real-world physical environments. However, training dynamic sampling directly from the real world is costly, time-consuming, and insecure. Virtual game theory is a model of learning through self-game. In each iteration, the player chooses the best response strategy of his opponent's average strategy to act. In some types of games, the player's average strategy can converge to Nash equilibrium. For example, a two-person zero-sum game and multi-person potential game. Traditional virtual games are usually limited to regular games, and the efficiency of virtual games in extended games declines exponentially. Over the past few decades, Nash games have been widely applied in various fields such as enterprise management [1], supply chain management [2], control theory [3,4], economics [5], and engineering optimization [6–9].

Yu et al. proposed applying Nash games to market strategies between manufacturers and retailers, and they developed a strategy for finding Nash equilibrium using analytical methods, iterative methods, and genetic algorithms [2]. Hou et al. introduced the concept of group decision equilibrium and studied the relationship between group decision theory and game theory. They described experts' preferences using preference sequence vectors and used them to solve for Nash equilibrium [10]. Oliveira et al. proposed a Nash equilibrium solution strategy based on a fuzzy adaptive simulated annealing algorithm. They demonstrated that the game problem could be transformed into a constrained global optimization problem and proved its effectiveness through three examples [11]. Tang et al. proposed a gradient-free Nash equilibrium solution strategy based on the elite information exchange method and combined it with a global search algorithm to develop an iterative approach. They proved that the obtained result is indeed Nash equilibrium based on the fixed-point theorem [6,7].

Traditional fictitious play (FP) needs to calculate all game states during each iteration, which easily leads to dimension disaster in the case of large-scale game. The generalized weakening of virtual matching ensures convergence similar to that of traditional virtual matching and allows the approximate optimal response strategies and disturbance average strategy to be updated. Among them, strategy learning based on behavior strategies can be applied to extended games, and the linear time and space complexity of strategy solving can be guaranteed.

Fictitious self-play (FSP) calculates the optimal response strategy and updates the average strategy through reinforcement learning and supervised learning, respectively. FSP generates empirical datasets through self-game, and game intelligence stores quads in the experience pool for reinforcement learning training to calculate the optimal response strategy. The behavior tuples of agents themselves are stored in the experience buffer pool, which is used to supervise the learning and update the average strategy. Self-game constructs the reinforcement of learning memory of the subject by sampling, which is equivalent to the empirical data of Markov decision processes (MDPs), whose opponent adopts the average strategy. Therefore, the approximate optimal response strategy can be solved by a reinforcement learning algorithm to solve MDPs. Similarly, the agent's supervised learning memory can be used to approximate the agent's own average strategy experience data, and then the strategy can be solved through a supervised classification algorithm.

Neural fictitious self-play (NFSP) is a kind of neural network virtual self-game, which combines virtual self-game and deep reinforcement learning. It does not need prior knowledge M_{RL} and solves M_{SL} approximate Nash equilibrium strategies of incomplete information game problems in a scalable end-to-end manner. In NFSP, each iteration is mainly composed of three steps. Firstly, agents take actions to interact with other agents according to the strategy and store the sample data in the reinforcement learning experience buffer pool and the supervision learning experience buffer pool. Then, a reinforcement learning algorithm is used to update the optimal response strategy, and supervised learning memory is used to update the average strategy.

Su et al. proposed a new approach to approach phase-biased proportional navigation guidance law considering fuel consumption, effectively avoiding dangerous areas while consuming less fuel compared to traditional algorithms [12]. Zhang et al. augmented the traditional guidance law with collision detection and maneuvering correction instructions, ensuring the safety of unmanned aerial vehicles (UAVs) in collision avoidance scenarios [13]. Although these methods have low computational complexity, they tend to encounter local minima and result in infeasible solutions in complex scenarios with multiple types, quantities, and distributions of obstacles and threats. Additionally, it is challenging to obtain globally optimal solutions that satisfy mission requirements. It is worth mentioning that the Virtual-agent Artificial Potential Function constructed by Qian et al. improved the issue of local minima while reducing computation time, but such methods still lack long-term and global optimization capabilities [14].

Strategy learning in multi-agent games is inherently more complex than in single-agent games, as agents not only need to interact with the environment but also with dynamically changing strategies of other agents. The non-stationary characteristics of a multi-agent game will lead to the failure of most single-agent reinforcement learning algorithms. These algorithms do not consider the impact of the behavior of other agents, which will lead to the invalidity of the Markov property of MDPs. In addition, there are many problems in multi-agent games, such as dimensional disasters, credit allocation, and global exploration [15].

Currently, research on multi-agent reinforcement learning can be primarily categorized into four areas: analysis of emergency behavior, learning communication, learning cooperation, and agent modeling agents.

Credit allocation, sparse rewards, and sample efficiency are common issues in reinforcement learning algorithms. The traditional methods for solving sparse rewards include reward shaping, hierarchical reinforcement learning, and hindsight experience replay. Reward shaping usually involves modifying the reward function to provide denser reward signals to optimize the learning process. Layered reinforcement learning is the process of abstracting learning objectives into multi-layer objectives and then solving sparse reward problems by exploring high-level objectives. Post-experience replay is achieved by updating the target with previously sampled data in order to reuse failed explorations. This paper utilizes pre training of the strategy network to alleviate the problem of sparse rewards. There are two main ideas for pre training, one is to use Raw Self-Play to train strategies, the other is to use expert agent data for imitative learning. This project mainly uses Raw Self-Play for pre training. Both approaches aim to quickly learn a strategy network with a certain level, alleviate the problem of sparse rewards in the early stages of NFSP training and achieve warm start of NFSP. The Raw Self-Play method is simpler and more direct in strategy learning than NFSP. NFSP requires each agent to learn two strategies, namely the optimal response strategy and the average strategy, and select strategies based on probability in each action. However, in the Raw Self-Play method, each agent only needs to learn the optimal response strategy. In each game, it only needs to act directly based on the optimal response strategy. Although the convergence of global optima may not be as good as NFSP, the Raw Self-Play method can learn an initialization strategy with certain game capabilities faster than NFSP. Then, by utilizing NFSP for continued training, the advantages of each of the two self-play methods can be utilized while alleviating the sparse reward problem.

This study focuses on constructing a training model based on a parametric environment and implementing a feedback loop using a digital environment evaluation model. The objective is to continuously verify the accuracy of decision-making strategies in the parametric environment and assess the transferability of the decision-making model's strategies. The project acknowledges the challenges of utilizing digital environments to meet the training requirements for generalized battle scenes. To address this, the project proposes two innovative approaches. Firstly, it introduces a parametric scene training method. Secondly, it advocates for the utilization of a digital environment as a testing ground to enhance the efficiency of virtual-to-real conversion. Additionally, leveraging digital simulations as pre testing environments for physical settings can lead to reduced deployment costs and improved security in virtual–real migration experiments.

2. Materials and Methods

2.1. Algorithms Migration Task

This study uses a parameterized environment to reduce deployment costs and improve the security of virtual reality migration experiments. This method is easier to sample and train approximate strategies and can be expressed mathematically.

$$\hat{p}(s_{t+1}|s_t, a_t) \approx p^*(s_{t+1}|s_t, a_t) \tag{1}$$

where p^* represents strategies in a real environment; $\hat{p}$ represents a policy in a virtual environment; s_t represents information state of t; a_t represents action sequence of t.

However, due to modeling or dynamic changes in the environment, directly assigning strategies that have been successfully trained $\hat{p}(s_{t+1}|s_t, a_t, \lambda)$ in a non virtual environment to a ρ_λ real environment often leads to failed results. For example, many strategies trained in deep reinforcement learning usually lead to behaviors that cannot happen in real environments. We introduce a set of parameters λ, which parameterize the motion control model in a parameterized environment, and its strategy can be expressed as follows.

Therefore, we can further adjust the training objectives in the virtual environment to maximize the expected return of training in the motion control model:

$$\underset{\lambda\sim\rho\lambda}{E}[E_{\tau\sim p(\tau|\pi,\lambda)}[\sum_{t=0}^{T-1} r(s_t,a_t)]] \tag{2}$$

The final control strategy can be better extended to the real world by training strategies to adapt to the dynamic changes of the environment.

2.2. *Self-Game Training and Group Training*

In this project, the labor unit is a local unit. If the human unit uses a single rule strategy, it is difficult for the intelligent experience to train a robust strategy because the object of confrontation has not changed during training [16]. The design of a rich and reasonable behavior decision tree involves a lot of work, which is not the key point of this work. At the same time, the strategic strength of the behavior tree is hard to guarantee because of artificial design. At the same time, the human strategy itself is also very rich and complicated. Therefore, in this study, we used the method of self-play to learn people's strategies and fight against our team.

In addition, the idea of league training was used during the training process to store the latest strategies of some people and cars, avoiding the catastrophic forgetting problem of intelligent agents, which is the appearance of "dog biting tail" strategy distribution. The result of strategy optimization is not just to defeat the current enemy strategy but to gain advantages in the face of different types of enemy strategies in the training history. The same is true for the optimization of the enemy's strategies.

2.3. *Virtual Self Game*

The key point is that the convex combination of the standard regular form strategy can be modified to the convex combination with probability correlation, as shown below:

$$\sigma(s,a) \propto \lambda_1 x_{\pi_1}(s)\pi_1(s,a) + \lambda_2 x_{\pi_2}(s)\pi_2(s,a)\ \forall s,a \tag{3}$$

where $\lambda_1 x_{\pi_1}(s) + \lambda_2 x_{\pi_2}(s)$ is a normalized constant on the information sets. The above (s_t, a_t) formula M_{RL} not only defines the player's full-range average strategy update $(s_t, a_t, r_{t+1}, s_{t+1})$ in M_{RL} behavioral strategies but also specifies the data sampling method of this convex strategy combination.

The idea of repeating experience is introduced, and the optimal response strategy is solved by cement learning method of M_{RL} $(s_t, a_t, r_{t+1}, s_{t+1})$. Specifically, during the k-th iteration, the agent adopts a strategy to respond to the opponent's average strategy in order to achieve maximum returns, and agent I stores its own empirical data in the reinforcement learning experience buffer pool, which is stored in quadruple form, the experience pool size is set to a fixed value, and when the data volume reaches the upper limit, the old sample is replaced with new sample data. In addition, due to the use of off strategy reinforcement learning methods, after meeting the iterative condition, the next strategy update does not need to collect samples from the beginning, but it can still be updated with the data generated from the previous strategy in the experience pool, thus improving sample efficiency and balancing exploration and utilization problems in the optimization process of the optimal response policy. NFSP uses the self-game method to correctly combine average strategies and optimal response strategies to more accurately approximate the sample data distribution of the actions taken against the opponent's average strategy group. By strengthening the experience pool of learning, people can learn the Q function:

$$Q^i(s,a) \approx \mathbb{E}_{\beta^i,\sigma^{-i}}\left[G_t^i \middle| S_t = s, A_t = a\right] \tag{4}$$

The purpose of updating the Q function is to maximize the expected return $\pi - i$ when adopting the optimal response strategy β^i to the average strategy of the opponent;

that is, the greedy strategy is obtained based on the Q function. If $\beta = \epsilon - greedy(Q)$ is satisfied, the greedy strategy will take random behavior with probability ε, otherwise, it will maximize the Q value. The parameter ϵ Implemented a certain exploration of intelligent agents, avoiding the excessive utilization of the current optimal strategy. In short, solving the optimal response strategy is achieved by solving the optimal Q function.

2.4. Multi-Agent Virtual Self-Game

The algorithm for solving the optimal strategy of NFSP in multi-agent game systems is as follows:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,\mathrm{a}_t)\mapsto\rho_\pi}[r(s_t,\mathrm{a}_t)] \tag{5}$$

Specifically, for non-stationary problem in multi-agent games, centralized training and decentralized execution are used to solve the optimal strategy. To address the issue of credit allocation, baseline rewards are used to measure the contribution of agents to global rewards more accurately, simultaneously introducing the maximum entropy strategy gradient to balance exploration and utilization during the training process and enhance the robustness of the strategy. In addition, a pre trained average strategy network is used to alleviate the problem of sparse rewards.

A centralized training and decentralized execution training framework is adopted in the optimization process of the optimal strategy for multi-agent NFSP. When training intelligent agents, the global information shared by the agents and the actions of each agent are used as inputs, and the non-stationary nature of the game environment is eliminated by sharing agent actions. All agents share a critical valuation network. The centralized critical network loss function is:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \mathbb{E}_{\mathbf{x},a,r,\mathbf{X}}\left[\left(Q_i^\theta(\mathbf{x},a_1,\ldots,a_N) - y_l\right)^2\right] \tag{6}$$

$$y_i = r_i + \gamma\mathbb{E}_{a-\pi(o')}[Q_i^\theta(o',a_1',\ldots,a_N') - \alpha log(\pi(a\Big|o'))] \tag{7}$$

where $(x,\ a_1,\ldots,a_N)$ represents the global information shared during intelligent agent training and α represents the importance of the temperature coefficient used to control strategy entropy. The global losses of all agents are accumulated, and then the global critical network is updated through TD error. The global expected reward is used as the baseline reward, and simultaneously, the reward for taking a certain action and the baseline reward are used to solve the advantage function. Q represents the action return and b represents the global baseline reward after fixing other agent actions. The advantage function A can evaluate the actual gain brought by the agent taking a certain action on the global return. The specific calculation process of the advantage function A is:

$$A(\mathbf{x},a_i) = Q(\mathbf{x},a_i) - b(\mathbf{x},a_{-i}) \tag{8}$$

$$b(\mathbf{x},a_{-i}) = \mathbb{E}_{a_i\sim\pi_i(\mathbf{x}_i)}[Q_i(\mathbf{x},(a_i,a_{-i}))] = \sum_{a_i'\in Action_i} \pi\left(a_i'|\mathbf{x}_i\right)Q_i\left(\mathbf{x},\left(a_i',a_{-i}\right)\right) \tag{9}$$

Replacing the Q-value network with an advantage function for policy evaluation can more accurately guide the gradient update of the optimal strategy. The gradient of the policy network Actor is:

$$\nabla_{\phi_i}J(\phi_i) = \mathbb{E}_{o_i\sim D,a_i\sim\pi_i}\left[\nabla_{\phi_i}\log\pi_i(a_i|o_i)A(\mathbf{x},a_i)\right] \tag{10}$$

Due to the decentralized execution of the training method, each agent has a separate optimal policy network, and the Actor network updates independently through a global critical.

2.5. League Training

The gentle ague training based on multiple opponents adopts the latest neural network training method, i.e., population-based training (PBT), which is an asynchronous optimization algorithm. It simultaneously trains and optimizes the network of a group to quickly select the best hyperparameter set and model for the task. Most importantly, this method does not increase computational overhead. It can maximize performance and is easily integrated into existing machine learning processes.

From Go to Atari games to image recognition and language translation, neural networks have achieved tremendous success in various fields. However, it is often overlooked that the success of neural networks in a specific application often depends on a series of choices made at the beginning of the study, including the type of network and the data and methods used for training [17]. At present, these choices (called hyperparameters) are realized through experience, random search, or calculation-intensive search process.

The experimenter can quickly select the best hyperparameter set and model for the task. This technology is called population-based training (PBT), which simultaneously trains and optimizes a series of networks to find the optimal settings quickly. Most importantly, this method does not increase computational overhead, can be completed quickly like traditional technologies, and is easily integrated into existing machine learning processes.

This technique is a combination of the two most-used hyperparameter optimization methods: random search and manual tuning. In random search, the neural network population (cluster) is independently trained in parallel. At the end of the training, the model with the best performance is selected. Usually, this means that only a small part of the population will receive good hyperparameter training, and more parts will receive bad hyperparameter training, wasting computing resources.

With manual tuning, researchers must guess what the optimal hyperparameters are, train the model with them, and then evaluate the performance, as shown in Figure 1. This process is repeated until the researchers are satisfied with the performance of the network. Although this may lead to better performance, the disadvantage is that it takes a long time, sometimes weeks or even months, to find the perfect setting. Although there are some methods to automate this process, such as Bayesian optimization, it still takes a long time and requires a lot of continuous training to find the best hyperparameter. Manual tuning and Bayesian optimization methods change hyperparameters by observing many continuous running trainings, which makes these methods slow.

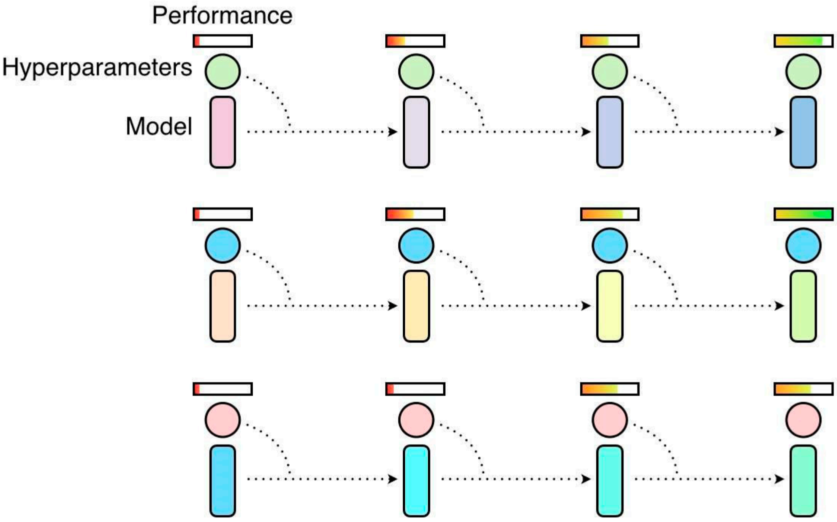


Figure 1. Hyperparameters random search.

PBT, like random search, starts by training many neural networks in parallel with random hyperparameters. However, these networks are not independent but use information from the rest of the group (cluster) to tune hyperparameters and guide computing resources to promising models. This is inspired by genetic algorithms, in which every member of the cluster (called agent) can use other people's information. For example, an agent can copy model parameters from a better-performing agent. It can also explore new hyperparameters by randomly changing the current value.

As shown in Figure 2, as the training of the neural network cluster continues, this process of development and exploration is also carried out periodically to ensure that all agents in the cluster have a good basic performance level and have been constantly exploring new hyperparameters. This means that PBT can quickly use good hyperparameters, allocate more training time to promising models, and, most importantly, adjust the hyperparameter value during the whole training process, thus automatically learn the optimal configuration.

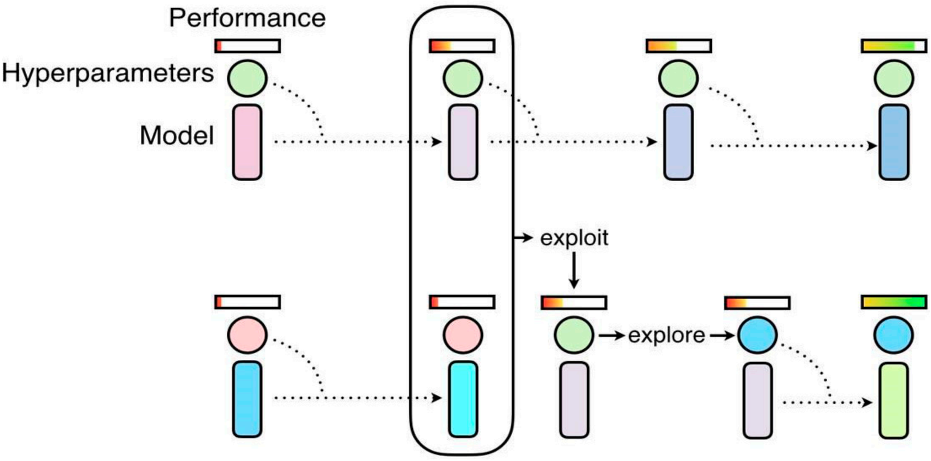


Figure 2. Population-based neural network training.

As illustrated in Figure 2, swarm-based neural network training is initially like random search but allows agents to use part of the results of other agents and explore new hyperparameters in the training process.

All stages of population-based neural network training are shown in Figure 3.

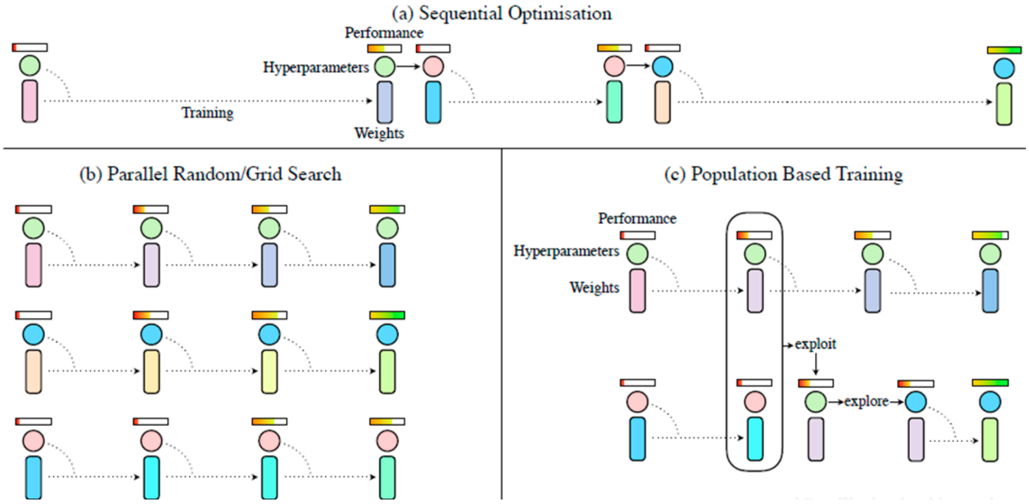


Figure 3. All stages of population-based neural network training.

3. Results

In this study, the method of course learning was adopted to gradually improve the ability of intelligent agents:

(1) Mini Map → Medium Map → Large Map

The final intelligent agent needs to complete the adversarial task on a 2 km × 2 km map, which has a large range and is difficult for the agent to explore from scratch. Therefore, this study uses the idea of course learning to cut out small maps from the large map, train the agent from 0 on the small map, and gradually increase the map range to train on the entire map after the performance stabilizes. The task design from easy to difficult simplifies the learning difficulty of intelligent agents and also reduces the time required for training.

(2) Avoiding obstacles and moving → Discovering and safely approaching competitors → Encircling and blocking to defeat competitors

Because the research scene itself is a large and complex task, involving motion planning, exploration, tactical decision-making and other aspects, the learning difficulty of these aspects is often inconsistent, and high-level tasks often rely on the learning of low-level tasks, so we separately design rewards for different tasks. At the initial stage of training, agents move randomly and are prone to encounter obstacles, so obstacle avoidance rewards can effectively assist the intelligent agent in learning obstacle avoidance behavior in the early stages of training [18]. After achieving good obstacle avoidance behavior, randomly encountering some enemy units will also trigger rewards. At this point, the learning goal of the intelligent agent will shift to discovering and safely approaching the enemy. Without touching obstacles, when safely approaching the enemy, choosing attack commands will further earn higher rewards. At this point, the intelligent agent learns how to kill more enemies based on such reward signals. This process involves disassembling the implementation process of the entire composite task and implementing the design of reward functions step by step, which can help the intelligent agent effectively smooth the learning curve and obtain effective reward signals for strategy optimization throughout the entire training process.

3.1. Overall Architecture

The agent training cloud platform is a basic platform supporting agent training, providing core functions such as large-scale heterogeneous computing resource scheduling management, distributed intelligent training engine, deep reinforcement algorithm support, and agent design and training and can be used to train and generate reinforcement learning agents controlled by unmanned equipment. The platform adopts a large-scale distributed architecture to achieve real-time data generation and model training and inference, consisting of five layers: hardware computing power layer, computing platform layer, intelligent engine layer, algorithm training layer and user layer. The detailed architecture diagram is shown in the following figure.

3.2. Platform Features

3.2.1. Hardware Computing Power Layer

The hardware computing power layer mainly includes a hardware server cluster composed of a central processing unit (CPU), a graphics processing unit (GPU), storage, and other resources, as well as corresponding 10 Gigabit networks, to meet the needs of large-scale operations in game confrontation decision making.

For hardware server clusters, the intelligent agent training cloud platform effectively organizes and manages the hardware server clusters by controlling the servers, achieving the implementation of various layer functions in the training platform. The control server consists of one main control device, one cloud management device, and two training management devices [19]. The system can connect to the client computer provided by the user, and the device connection method is shown in the following figure.

The deployment of components or modules at each layer of the intelligent agent training cloud platform in the hardware environment is shown in the Table 1.

Table 1. Component and model algorithm library deployment.

Serial Number	Hardware Devices	Software or Algorithm Library
1	Master control equipment	Intelligent agent development IDE components
2		Algorithm secondary development SDK components
3		Simulation environment integration SDK components
4		Platform management WEB interface components
5	Cloud management equipment	Container management components
6		Heterogeneous resource scheduling component
7		Distributed components
8		Virtual network management components
9	Training and management equipment	Large-scale data generation engine components
10		Distributed continuous learning engine components
11		High-performance prediction inference engine components
12		Core algorithm library
13		Network model library
14		Training method library
15	Hardware server cluster	Simulation and deduction environment
16		Large-scale data generation engine components
17		Distributed continuous learning engine components
18		High-performance prediction inference engine components

Main control equipment: deploy intelligent agent development IDE components, algorithm secondary development SDK components, simulation environment integration SDK components, platform management WEB interface components, support external computers to access the system through WEB browsers, and support users to complete input and output operations.

Cloud management equipment: deploy the scheduling layer software at the bottom of the software architecture, including container management components, heterogeneous resource scheduling components, distributed storage components, and virtual network management components to realize the creation, management, and scheduling of heterogeneous resources.

Training and management equipment: deploy the most important intelligent engine layer and algorithm training layer in the cloud platform architecture, including large-scale data generation engine components, distributed continuous learning engine components [20], high-performance prediction and inference engine components, as well as core algorithm libraries, network model libraries, and training method libraries, to manage the deep reinforcement learning process and provide mature algorithm resources for user intelligent agent development.

Hardware server cluster: deploy large-scale data generation engine components, distributed continuous learning engine components, high-performance prediction and inference engine components, as well as simulation and inference environments, receive training management equipment allocation, and complete deep reinforcement learning processes such as sample generation, feature processing, gradient descent, and network parameter updates.

3.2.2. Computing Platform Layer

The computing platform layer abstracts heterogeneous and scattered hardware into an integrated computing resource through comprehensive scheduling of large-scale distributed heterogeneous computing resources, flexibly distributes computing tasks to support the operation of deep reinforcement learning algorithms, and realizes parallel computing of intelligent algorithms.

Building an application support for reinforcement learning based on the computing platform layer, namely the reinforcement learning runtime system, to complete the deployment of reinforcement learning tasks and training calculations, specifically composed of the following four aspects. Building a reinforcement learning task planning system based on heterogeneous resource scheduling components for scheduling reinforcement learning tasks, planning the distribution of each component instance in the task, and establishing communication relationships between them.

On the basis of virtual network management components, constructing a reinforcement learning sample flow and a reinforcement learning model parameter synchronization chain for data transmission of reinforcement learning model iteration.

On the basis of distributed storage components, building a reinforcement learning application deployment system for installing user-defined functions into planned training tasks.

Building a reinforcement learning model pool based on distributed storage components for storing, synchronizing, and distributing reinforcement learning models.

3.2.3. Intelligent Engine Layer

The intelligent engine layer is the key to supporting the training of intelligent agents, which achieves processes such as sample data generation, collection, feature processing, intelligent agent training support, and intelligent agent estimation application.

The intelligent engine layer converts large-scale computing power into large-scale data capabilities to enhance the using and learning of learning algorithms. The intelligent engine layer consists of large-scale data generation engine components, distributed continuous learning engine components, and high-performance prediction and inference engine components. Among them, the large-scale data generation engine component is responsible for producing massive amounts of data, the distributed continuous learning engine component is responsible for consuming massive amounts of data and using relevant algorithms to optimize intelligent agents, and the high-performance prediction and inference engine component is used to quickly respond to and drive the simulation environment to generate data [21].

The operating relationship of the above three main modules is as follows: Large-scale data generation engine components interact with multiple simulation environments in parallel. With each interaction, the simulation environment outputs the situation to the large-scale data generation engine component, and the large-scale data generation engine component returns instructions to the environment; the generated data will be streamed and stored in the training data buffer pool and will be read in batches when needed by the distributed continuous learning engine components.

Each instance of the distributed continuous learning engine component will independently batch obtain samples from the sample pool, perform neural network forward calculation, calculate the gradient according to the loss function defined by the reinforcement learning algorithm, determine the update direction of the network parameters, and then perform reverse calculation to update the parameters.

The new version of the model generated by the distributed continuous learning engine component will be loaded by the high-performance prediction and inference engine component. When the large-scale data generation engine component has interaction needs, it will send environmental observations to the high-performance prediction and inference engine component, perform forward calculations, and return actions to the large-scale data generation engine component.

3.2.4. Algorithm Training Layer

The algorithm training layer consists of the core algorithm library, network model library, and training method library.

(1) Core algorithm library

The platform provides typical deep reinforcement learning algorithms as shown in Table 2.

Table 2. Core algorithm library algorithm.

Serial Number	Algorithm	Referred to as	Task Category	Examples of Applicable Scenarios
1	Deep Q-value Network Learning Algorithm	DQN	Decision problems in discrete spaces	Single-agent small-scale combat scenarios
2	Deep Dual Q-value Network Learning Algorithm	DDQ	Decision problems in discrete spaces	Single-agent small-scale combat scenarios
3	Advantage Q-value Network Learning Algorithm	Dueling DQN	Decision problems in discrete spaces	Single-agent small-scale combat scenarios
4	A Learning Algorithm for Noise Q-value Networks	Noisy DQN	Decision problems in discrete spaces	Single-agent small-scale combat scenarios
5	A Learning Algorithm for Q-valued Networks with Priority Data Queues	Prioritized DQN	Decision problems in discrete spaces	Single-agent small-scale combat scenarios
6	Rain Bow	Rainbow	Decision problems in discrete spaces	Single-agent small-scale combat scenarios
7	Deep Deterministic Strategy Gradient Algorithm	DDPG	Decision Problems in Continuous Spaces	Single-agent small-scale combat scenarios
8	Enhanced version of near-end optimization algorithm	PPO	Decision problems in mixed spaces	Medium-scale tactical confrontation scenario with single agent
9	Asynchronous Near End Optimization Algorithm	A-PPO	Decision problems in mixed spaces	Medium-scale tactical confrontation scenario with single agent
10	Gradient algorithm for multi-agent deep deterministic strategy	MADDPG	Decision problems in multi-agent discrete space	Multi-agent small-scale tactical confrontation scenarios
11	Multi Agent Near End Optimization Algorithm	MAPPO	Decision problems in multi-agent mixed space	Medium-scale tactical confrontation scenarios with multi-agent systems

(2) Network Model Library

It mainly provides three types of network models: basic neural networks, composite neural networks, and intelligent agent networks.

(3) Basic neural network

It provides a single scalable network structure such as full connection network MLP, short-term memory network LSTM, convolutional neural network CNN, and cyclic neural network RNN.

(4) Composite neural network

It provides ResNet, Transformer, Pointer Net, Multi RNN, attention mechanism, etc.

(5) Intelligent agent network

It provides a universal single agent neural network, which consists of multiple basic neural networks and composite neural networks and supports the construction of single decision agent models. It provides a universal multi-agent neural network, which is composed of multiple basic and composite neural networks and supports the construction of multi decision agent models.

It supports the adjustment of the internal structure of the intelligent agent network model.

(6) Training Method Library

In this project, due to the complete difference of observation space, business logic, and strategy space of unmanned platforms, it is a typical asymmetric game problem. Therefore, the main training method applied is large-scale asymmetric asynchronous learning training technology. To address the issue of different evolutionary efficiency and direction of agents in asymmetric heterogeneous adversarial game scenarios, it is necessary to continuously and stably train agents from both sides of the adversarial game and improve their ability level. This technology includes large-scale heterogeneous computing resource comprehensive scheduling technology, distributed reinforcement learning engine technology, agent multiple training and adjustment technology, and asynchronous learning algorithm optimization technology.

(7) Comprehensive scheduling technology of large-scale heterogeneous computing resources

By studying the comprehensive scheduling technology of large-scale heterogeneous computing resources, the rapid response to the elastic computing power demand of upper training tasks is achieved, and specific tasks are scheduled to idle computing nodes to avoid resource use conflicts between different training tasks.

The overall design idea of large-scale heterogeneous computing resource integrated scheduling technology is to separate the application layer's request for resources from the actual call logic of the underlying operating environment and provide a set of standard resource scheduling interfaces for heterogeneous computing resources for the upper technology. Specifically, this technology takes the container technology as the core, abstracts heterogeneous scattered hardware into an integrated computing resource for management and scheduling, quickly stores and uses massive data, and establishes a virtual connection between heterogeneous computing devices.

The integrated scheduling technology of large-scale heterogeneous computing resources uses container technology to uniformly abstract the hardware in computing clusters. Specifically, the entire cluster is divided into four types: main node, work node, network service node, and distributed storage node. Among them, the main node is used for managing and scheduling cluster resources, monitoring the usage of computing nodes in real time by maintaining the cluster status database, responding to resource requests, and deploying specific training tasks to idle computing nodes; as the provider of cluster resources, the work node is responsible for executing upper-level training tasks based on the tasks assigned by the main node and providing feedback on the execution status of the training tasks to the main node; the network service node has implemented an efficient virtual router and developed an IP routing-based data forwarding mechanism for each working node; each distributed storage node is bound to the disk one-to-one and records the status and configuration information of the distributed storage node through the globally accessible persistence volume (PV), which supports access to distributed storage services on any node.

(8) Distributed Reinforcement Learning Engine Technology

The distributed reinforcement learning engine technology adopts a three-layer architecture of sampling training predictor, which converts large-scale computing power into large-scale data processing power, achieving efficient reinforcement learning training of decision models under distributed computing power.

The traditional distributed reinforcement learning engine adopts a sampler-trainer two-layer architecture. Among them, the Sampler module is responsible for continuously generating training samples, while the Trainer module is responsible for continuously updating decision model parameters based on the training samples. The traditional reinforcement learning engine's sampling module and prediction module run on the CPU, while the training module runs separately on the GPU, resulting in a large amount of data communication between the CPU and GPU. Due to the bottleneck of network bandwidth, a large amount of data transmission and parameter synchronization of neural networks will reduce the efficiency of the engine. In addition, as the parameters of the neural network model increase, CPU used for forward inference will meet an obvious decrease in running speed. Finally, the sampling module needs to schedule CPU resources for both simulation and neural network inference, unable to fully improve the utilization of CPU multithreaded resources.

Therefore, adding an additional prediction module (Predictor) using GPU resources to improve the inference speed of the neural network effectively solves the several problems above. Based on the newly designed three-layer architecture, the sampling module runs batch simulation and deduction on the CPU with real-time situation and decision data interacting with the neural network inference running on the GPU. Compared with model data and complete batch data under traditional engine architecture, it significantly reduces data communication traffic and effectively utilizes network bandwidth [22]. On the other hand, the training module and prediction module run simultaneously on the GPU, fully utilizing the high concurrency ability of the GPU, improving the speed of batch inference of the neural network, reducing the delay of model parameter updates, and accelerating the efficiency of model parameter synchronization.

(9) Asynchronous learning algorithm optimization technology

The design idea of asynchronous learning algorithm optimization technology is to establish two independent data channels for two intelligent agents (submarines and anti submarine patrol aircraft), which are, respectively, used for training sample transmission and neural network parameter updates of a single agent. In addition, to ensure the convergence ability of the agent while accelerating the efficiency of data generation and agent parameter updates, streaming data technology and asynchronous parameter correction technology are adopted.

Firstly, through streaming data technology, the efficiency of data generation and intelligent agent parameter updates can be accelerated. In response to the problem of low real-time performance in traditional database technology, streaming data technology can effectively address the high real-time data processing needs. In addition, streaming data technology records the production sequence of transmitted data according to the timeline and records the timestamp of a single confrontation after each round is completed. The optimization algorithm can quickly query corresponding data nodes based on timestamp and data order, achieving the requirement of training data resampling.

Although streaming data technology has accelerated the efficiency of data generation and intelligent agent parameter updates, it has also caused the optimization and non-convergence of neural networks due to outdated data. By using asynchronous parameter correction technology to adjust and truncate the weights of training samples, the convergence of the neural network during the training process is ensured. After recollecting each training sample, the weight of each sample is adjusted and the loss function is calculated according to the neural network strategy distribution that produces the current training sample to make the generation probability of each sample close to the latest version of the neural network strategy distribution, thus ensuring the convergence of neural network parameters.

3.2.5. User

The user layer mainly implements intelligent agent development IDE components, algorithm secondary development SDK components, simulation environment integration SDK components, and platform management WEB interface components.

(1) Intelligent agent development IDE components

The intelligent agent development IDE component provides users with programming interfaces and environments for intelligent agent design and development based on algorithm training layers, provides intelligent agent training, evaluation, deployment, and application functions based on the intelligent engine layer, and provides resource configuration and scheduling functions for intelligent agent training tasks based on the computing platform layer.

(2) Algorithm secondary development SDK components

The algorithm secondary development SDK component is based on the algorithm training layer, providing a programming call interface and development environment for users to self-developed and extended algorithm libraries. For specific interfaces, please refer to Section Intelligent Agent Training "Agent programming development/optimization process".

(3) Simulation Environment Integration SDK Components

The simulation environment integration SDK component provides a unified standard programming call interface and development environment for various simulation environments, achieving dynamic parallel inference of the simulation environment, accelerating the simulation process, data sampling, and instruction input.

(4) Platform management WEB interface components

The platform management WEB interface component provides users with a system management operation interface for the platform, including user management interface, task management interface, monitoring management interface, configuration management interface, etc.

4. Simulation

There are two types of system users, namely super administrators and regular users [23]. The super administrator account has global permissions and is assigned to the league sponsor in this project, while the ordinary user account is assigned to each team.

The main operations of a super administrator include four categories:

Create user: create other super administrators or regular users and define user resource usage quotas.

Resource management: add new hardware resources to the computing cluster, allowing the computing platform layer to virtualize them and increase hardware computing power.

View all users: view all user accounts, resources, permissions, etc.

View the resource status of all users: view the startup task status, hardware resource usage rate, and hardware resource usage records of all users.

The main operations of ordinary users include two types:

View the status of this user: view the status of this user account, resources, permissions, etc.

View the user's resource status: view the user's startup task status, hardware resource usage rate, and hardware resource usage records.

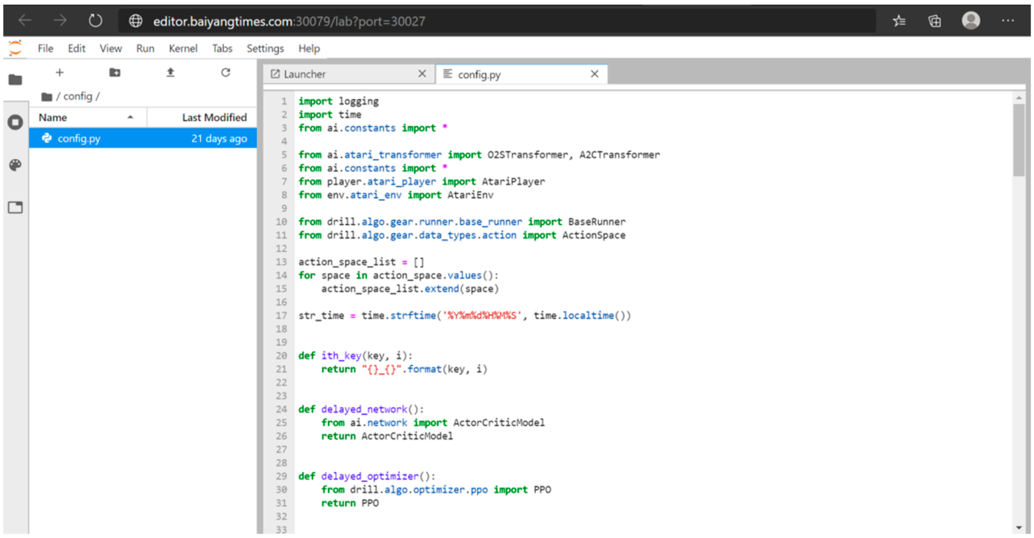
Intelligent Agent Training

The application of intelligent agent training cloud platform for the development of reinforcement learning agents in two typical scenarios of this project is a typical large-scale asymmetric asynchronous learning training process. Among them, "asymmetry" refers to the differences in neural network structures caused by different business logic of unmanned platforms; "asynchronous" refers to the real-time extraction of data by distributed continuous learning engine components based on the acquisition of data in a distributed parallel simulation environment, instead of waiting for the slowest simulation environment data to arrive before being uniformly extracted.

Among them, the development/optimization process of intelligent agent programming is mainly based on the development of IDE components by the agent, calling the algorithm model and training methods provided by the algorithm training layer to form the neural network structure and training methods of the agent. The scheduling process of parallel training tasks is mainly based on the platform management of Web interface components. Configure computing resources are needed for parallel training in the simulation environment, such as setting a reasonable number of CPU cores and logical segmentation of GPU, to improve the utilization of resources and form containers to push to CPU and GPU. Scenario loading, which is based on the scenario starting situation formed according to specific task scenarios and task conditions, is used as the basis for subsequent parallel deduction. The parallel sampling and intelligent agent model updating process in the simulation environment mainly consists of large-scale data generation engine components and high-performance prediction inference engine components, which continuously interact with the simulation environment to generate large-scale training data and inject them into the training data buffer pool [24]. At the same time, the intelligent agent neural network model is continuously learning and updating based on the distributed continuous learning engine component. In the iterative process of "generating data consumption data" mentioned above, users continuously optimize based on the level of the intelligent agent then program and adjust based on the visualization effect display, training logs, and other functions provided by the platform management WEB interface components until they meet the user's adversarial strength requirements and generate intelligent agents that can be used for actual deduction.

(1) Agent programming development/optimization process

The development/optimization process of intelligent agent programming is shown in Figure 4.

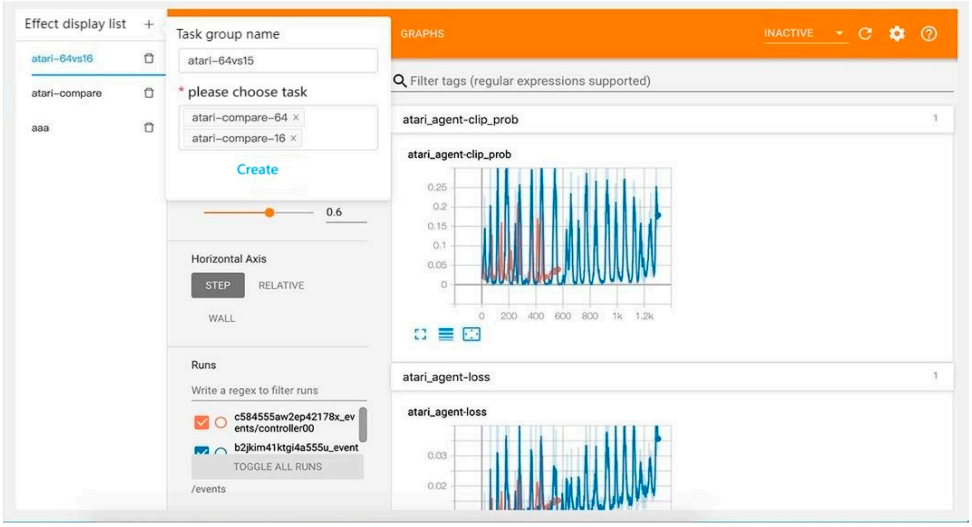


```
import logging
import time
from ai.constants import *

from ai.atari_transformer import O2STransformer, A2CTransformer
from ai.constants import *
from player.atari_player import AtariPlayer
from env.atari_env import AtariEnv

from drill.algo.gear.runner.base_runner import BaseRunner
from drill.algo.gear.data_types.action import ActionSpace

action_space_list = []
for space in action_space.values():
    action_space_list.extend(space)

str_time = time.strftime('%Y%m%d%H%M%S', time.localtime())


def ith_key(key, i):
    return "{}_{}".format(key, i)


def delayed_network():
    from ai.network import ActorCriticModel
    return ActorCriticModel


def delayed_optimizer():
    from drill.algo.optimizer.ppo import PPO
    return PPO


```

Figure 4. Intelligent agent programming development interface.

Developing IDE components based on intelligent agents allows users to develop intelligent agents for specific scenarios in this project. By calling the core algorithm library, network model library, and training method library, users can form neural network structure design and define loss function of neural network optimization and training methods (asymmetric asynchronous learning training in this project) for agents. In addition, SDK components based on algorithmic secondary development can be integrated with user-defined reinforcement learning algorithms. After completing the development of intelligent agent programming, the intelligent agent will pack to form containers based on large-scale data generation engine components, distributed continuous learning engine components, and high-performance prediction and inference engine components.

After completing the development of the intelligent agent, during the subsequent parallel sampling and updating of the intelligent agent model in the simulation environment (see details in the following text), based on the generated data, the platform manages the WEB interface components to help users optimize the intelligent agent through visual effects display, training logs, and other means (Figure 5).



Task group name
atari-64vs15

* please choose task
atari-compare-64 ×
atari-compare-16 ×

Figure 5. Visual display of training effects.

Based on platform management of WEB interface components, users can configure the number of CPUs and GPUs required during parallel training. The heterogeneous resource scheduling component pushes tasks to the CPU and GPU while using container management components to pull packaged containers, achieving containerized deployment of intelligent agent models in large-scale data generation engine components, distributed continuous learning engine components, and high-performance prediction inference engine components, serving the subsequent parallel sampling and intelligent agent model update processes.

(2) Scenario loading process

Load scenario scenarios and task conditions are specified by users to form scenario scenarios and load configuration files containing the aforementioned scenario scenarios to form the initial training state of the simulation environment and intelligent agent training cloud platform, which serves as the basis for subsequent parallel deduction.

(3) Parallel Sampling and Agent Model Updating Process in Simulation Environment

The large-scale data generation engine components running in parallel interact with multiple simulation environments, and high-speed network transmission is achieved through virtual network management components to obtain simulation situation information. After completing the format style conversion from simulation environment data to input data of the intelligent neural network (including useless data reduction, data normalization, one hot encoding, etc.), the neural network features are input to the high-performance prediction and inference engine component, which calculates and outputs the neural network decision output [25]. After completing the mapping of neural network decision output to simulation environment instructions, the large-scale data generation engine component sends specific instructions to the simulation environment. The training data formed during this process, namely the <State, Action, Reward> triplet data stream, are input into the training data buffer pool for use by subsequent distributed continuous learning engine components.

After the formation of training data, the distributed continuous learning engine component asynchronously reads the training data buffer pool, where “asynchrony” refers to the container deployed distributed continuous learning engine component reading data in real time, rather than waiting for the slowest of training data to arrive and uniformly read it. The distributed continuous learning engine component calls the reinforcement learning algorithm, calculates the gradient through the loss function, determines the update direction of the agent neural network, and carries out gradient back propagation and updates the neural network parameters [26]. During the parameter update process of the neural network model, a Ring-All Reduce method is used based on virtual network management components. After the update is completed, synchronize the neural network model parameters with the high-performance prediction inference engine component.

During the training process, certain agents, namely the process versions of neural network models, are stored based on distributed storage components. These intelligent agent models mainly include three categories:

Main Agent: Refers to the intelligent agent that the user ultimately applies. In this project, it is a submarine/anti submarine patrol aircraft intelligent agent.

Adversarial Agent: Adversarial agent is the opponent of the main agent, which enhances the intelligence level of the main agent through self-game. In this project, submarine/anti submarine patrol aircraft are the adversarial agents of each other.

Frozen Agent: Including historical versions of main agents, adversarial agents, and rule agents provided by users.

The internal interface transmission content during parallel sampling and agent model updating in the simulation environment is shown in Table 3.

Table 3. Table of internal interface transmission content.

Serial Number	Sender	Receiving End	Transferring Content
1	Simulation environment	Situation input module	Real-time situational data in simulation environment
2	Situation input module	Simulation environment	Command data of intelligent agents in simulation environments
3	Situation input module	Decision result generation module	Acceptable state information of neural networks
4	Decision result generation module	Situation input module	Acceptable instruction formats for simulation environments
5	Large-scale data generation engine components	Training data buffer pool	<State, Action, Reward> State is the simulation environments, action is the training data buffer pool, reward is the distributed continuous learning engine components reward
6	Training data buffer pool	Distributed continuous learning engine components	<State, Action, Reward> State is the simulation environments, action is the training data buffer pool, reward is the distributed continuous learning engine components reward
7	Distributed continuous learning engine components	Distributed continuous learning engine components	Decision result generation module neural network parameters, updated using Ring-All Reduce method
8	Distributed continuous learning engine components	High-performance prediction inference engine components	Decision result generation module neural network parameters
9	Distributed continuous learning engine components	Distributed storage component (agent model)	Decision result generation module neural network parameters, phased neural network model: intelligent agent

5. Conclusions

Given the limitations of digital environments in meeting the generalized training requirements of combat scenarios, this paper proposes an innovative parametric scenario training approach and draws the following conclusions.

(1) The training model is designed based on a parameterized environment and incorporates a feedback loop utilizing a digital environment evaluation model. This allows for continuous verification of decision-making strategies and assessment of their transferability.

(2) Experimental tests were conducted to investigate the transfer learning approach from parametric scenes to physical scenes. This research draws upon foundational theories such as algorithm migration tasks, virtual self-play, multi-agent virtual self-games, and league training.

(3) An internal interface transmission content model was established to facilitate the transfer of information between different components of the training model.
(4) Utilizing a digital environment as a testing ground significantly enhances the iterative efficiency of virtual-to-real conversion. Moreover, it offers benefits such as reduced deployment costs and improved security in virtual–real migration experiments.

Author Contributions: Conceptualization, N.Z.; methodology, W.S.; software, W.S.; formal analysis, N.Z.; investigation, N.Z.; resources, W.S.; data curation, W.S.; writing—original draft preparation, N.Z.; writing—review and editing, G.Y.; supervision, B.S.; project administration, B.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data is unavailable due to privacy or ethical restrictions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Li, Y.; Lin, L.; Dai, Q.; Zhang, L. Allocating common costs of multinational companies based on arm's length principle and Nash non-cooperative game. *Eur. J. Oper. Res.* **2020**, *283*, 1002–1010. [CrossRef]
2. Yu, Y.; Huang, G.Q. Nash game model for optimizing market strategies, configuration of platform products in a Vendor Managed Inventory (VMI) supply chain for a product family. *Eur. J. Oper. Res.* **2010**, *206*, 361–373. [CrossRef]
3. Yin, H.; Chen, Y.H.; Yu, D.; Lü, H. Nash game oriented optimal design in controlling fuzzy dynamical systems. *IEEE Trans. Fuzzy Syst.* **2018**, *27*, 1659–1673. [CrossRef]
4. Sun, Q.; Yang, G.; Wang, X.; Chen, Y.H. Optimizing constraint obedience for mechanical systems: Robust control and non-cooperative game. *Mech. Syst. Signal Process.* **2021**, *149*, 107207. [CrossRef]
5. Belkov, S.; Evstigneev, I.V.; Hens, T.; Xu, L. Nash equilibrium strategies and survival portfolio rules in evolutionary models of asset markets. *Math. Financ. Econ.* **2020**, *14*, 249–262. [CrossRef]
6. Tang, Z.; Zhang, L. Nash equilibrium and multi criterion aerodynamic optimization. *J. Comput. Phys.* **2016**, *314*, 107–126. [CrossRef]
7. Tang, Z.L.; Chen, Y.B.; Zhang, L.H. Natural laminar flow shape optimization in transonic regime with competitive Nash game strategy. *Appl. Math. Model.* **2017**, *48*, 534–547. [CrossRef]
8. Tang, Z.; Désidéri, J.A.; Périaux, J. Multicriterion Aerodynamic Shape Design Optimization and Inverse Problems Using Control Theory and Nash Games. *J. Optim. Theory Appl.* **2007**, *135*, 599–622. [CrossRef]
9. Leon, E.R.; Pape, A.L.; Costes, M.; Désidéri, J.A.; Alfano, D. Concurrent Aerodynamic Optimization of Rotor Blades Using a Nash Game Method. *J. Am. Helicopter Soc.* **2016**, *61*, 1–13. [CrossRef]
10. Hou, F.; Zhai, Y.; You, X. An equilibrium in group decision and its association with the Nash equilibrium in game theory. *Comput. Ind. Eng.* **2020**, *139*, 106138. [CrossRef]
11. E Oliveira, H.A.; Petraglia, A. Solving generalized Nash equilibrium problems through stochastic global optimization. *Appl. Soft Comput.* **2016**, *39*, 21–35. [CrossRef]
12. Su, W.; Yao, D.; Li, K.; Chen, L. A novel biased proportional navigation guidance law for close approach phase. *Chin. J. Aeronaut.* **2016**, *19*, 228–237. [CrossRef]
13. Zhang, N.; Gai, W.; Zhong, M.; Zhang, J. A fast finite-time convergent guidance law with nonlinear disturbance observer for unmanned aerial vehicles collision avoidance. *Aerosp. Sci. Technol.* **2019**, *86*, 204–214. [CrossRef]
14. Qian, M.S.; Wu, Z.; Jiang, B. Cerebellar model articulation neural network-based distributed fault tolerant tracking control with obstacle avoidance for fixed-wing UAVs. *IEEE Trans. Aerosp. Electron. Syst.* **2023**, *59*, 6841–6852. [CrossRef]
15. Hafezi, R. How artificial intelligence can improve understanding in challenging chaotic environments. *World Futures Rev.* **2020**, *12*, 219–228. [CrossRef]
16. Van Belkom, R. The impact of artificial intelligence on the activities of a futurist. *World Futures Rev.* **2020**, *12*, 156–168. [CrossRef]
17. Díaz-Domínguez, A. How futures studies and foresight could address ethical dilemmas of machine learning and artificial intelligence. *World Futures Rev.* **2020**, *12*, 169–180. [CrossRef]
18. Boysen, A. Mine the gap: Augmenting foresight methodologies with data analytics. *World Futures Rev.* **2020**, *12*, 239–248. [CrossRef]
19. Riva, G.; Riva, E. OS for Ind Robots: Manufacturing robots get smarter thanks to artificial intelligence. *Cyberpsychol. Behav. Soc. Netw.* **2020**, *23*, 357–358. [CrossRef]
20. Bevolo, M.; Amati, F. The potential role of AI in anticipating futures from a design process perspective: From the reflexive description of "esign" to a discussion of influences by the inclusion of AI in the futures research process. *World Futures Rev.* **2020**, *12*, 198–218. [CrossRef]
21. Xu, L.; Tu, P.; Tang, Q. Contract design for cloud logistics (CL) based on blockchain technology (BT). *Complexity* **2020**, *28*, 1–13. [CrossRef]

22. Banadkooki, F.B.; Ehteram, M.; Panahi, F.; Sh Sammen, S.; Othman, F.B.; EL-Shafie, A. Estimation of total dissolved solids (TDS) using new hybrid machine learning models. *J. Hydrol.* **2020**, *587*, 124989. [CrossRef]
23. Kosovic, I.N.; Mastelic, T.; Ivankovic, D. Using artificial intelligence on environmental data from internet of things for estimating solar radiation: Comprehensive analysis. *J. Clean. Prod.* **2020**, *266*, 121489. [CrossRef]
24. Daniyan, I.; Mpofu, K.; Oyesola, O.; Ramatsetse, B.; Adeodu, A. Artificial intelligence for predictive maintenance in the railcar learning factories. *Procedia Manuf.* **2020**, *45*, 13–18. [CrossRef]
25. Maathuis, C.; Pieters, W.; Berg, J.V. Decision support model for effects estimation and proportionality assessment for targeting in cyber operations. *Def. Technol.* **2020**, *17*, 352–374. [CrossRef]
26. Ding, R.-X.; Palomares, I.; Wang, X.Q.; Yang, G.-R.; Liu, B.C.; Dong, Y.C.; Herrera-Viedma, E.; Herrera, F. Large-Scale decision-making: Characterization, taxonomy, challenges and future directions from an artificial intelligence and applications perspective. *Inf. Fusion* **2020**, *59*, 84–102. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.