Knowledge Acquisition of Self-Organizing Systems With Deep Multiagent Reinforcement Learning

Hao Ji
Department of Aerospace and Mechanical Engineering,
University of Southern California,
3650 McClintock Avenue, OHE 400,
Los Angeles, CA 90089-1453
e-mail: haoji@usc.edu

Yan Jin[1]
Department of Aerospace and Mechanical Engineering,
University of Southern California,
3650 McClintock Avenue, OHE 400,
Los Angeles, CA 90089-1453
e-mail: yjin@usc.edu

Self-organizing systems (SOS) can perform complex tasks in unforeseen situations with adaptability. Previous work has introduced field-based approaches and rule-based social structuring for individual agents to not only comprehend the task situations but also take advantage of the social rule-based agent relations to accomplish their tasks without a centralized controller. Although the task fields and social rules can be predefined for relatively simple task situations, when the task complexity increases and the task environment changes, having a priori knowledge about these fields and the rules may not be feasible. In this paper, a multiagent reinforcement learning (RL) based model is proposed as a design approach to solving the rule generation problem with complex SOS tasks. A deep multiagent reinforcement learning algorithm was devised as a mechanism to train SOS agents for knowledge acquisition of the task field and social rules. Learning stability, functional differentiation, and robustness properties of this learning approach were investigated with respect to the changing team sizes and task variations. Through computer simulation studies of a box-pushing problem, the results have shown that there is an optimal range of the number of agents that achieves good learning stability; agents in a team learn to differentiate from other agents with changing team sizes and box dimensions; the robustness of the learned knowledge shows to be stronger to the external noises than with changing task constraints. [DOI: 10.1115/1.4052800]

Keywords: deep Q-learning, complex system, self-organizing system, scalability, robustness, artificial intelligence, computational synthesis, machine learning for engineering applications, model-based systems engineering

1 Introduction

Self-organizing systems can consist of simple agents that work cooperatively to achieve complex system-level behaviors without requiring global guidance. The design of SOS takes a bottom-up approach, and the top-level system complexity can be achieved through local agent interactions [1,2]. Complex system design by applying a self-organizing systems approach has many advantages, such as scalability, adaptability, and reliability [3,4]. Moreover, compared with traditional engineering systems with centralized controllers, self-organizing systems can be more robust to external changes and more resilient to system damages or component malfunctions [5–7].

Various approaches have been proposed to support the design of SOS. The field-based behavior regulation (FBR) approach [8] models the task environment with a field function, and the behavior of the agents is regulated based on the positions of these agents in the field by applying a field transformation function. Generally, an agent is striving to maximize its own interests by moving toward higher (or lower, depending on definition) positions. The advantage of this approach is that the agents' behaviors are simple hence require little knowledge to perform tasks since moving toward a higher or lower position in a given field is the sole behavior. This behavioral simplicity has its limits in solving more complex domain problems because the field representation has its limits in capturing all features of the task domains, and the inter-agent relations are ignored in this approach.

In order to overcome the limitations of the FBR approach, an evolutionary design method [4] and the social structuring approach [5] have been proposed to make the design of SOS parametric and optimizable and to allow an SOS to deal with more complex domain tasks by considering both task fields and social fields modeled by social rules [5]. It has been demonstrated that applying social rules can promote the level of coherence among agents' behaviors by avoiding potential conflicts and utilizing cooperation opportunities. A fundamental issue with this social rule-based approach is that a designer must know a priori what the rules are and how they should be applied, which may not be the case, especially when the tasks are complex and changeable.

Therefore, in this research, a reinforcement learning (RL) approach is taken to capture the self-organizing knowledge for agent behavior regulation in SOS design. More specifically, a multiagent Q-learning algorithm is explored to address three research questions: Q1: *What are the factors that impact the stability of learning dynamics in self-organizing systems?* Q2: *How does a self-organized agent differentiate from others and become specialized in certain tasks, and what affects such differentiation?* Q3: *Will the knowledge captured from RL be robust enough to be applied in a wide range of task situations and team sizes*?

In the RL literature, multiple agents can be trained using either a universal neural network or independent neural networks. Individual agents gather the state information and can be trained either collaboratively as a team or individually based on the reward they receive from the interactions with the environment [9]. Designing self-organizing systems, in this case, faces a choice of training the system either as a team using a single centralized agent RL approach or as separate individuals going through multiagent RL.

[1]Corresponding author.

Contributed by the Computers and Information Division of ASME for publication in the Journal of Computing and Information Science in Engineering. Manuscript received November 25, 2020; final manuscript received October 12, 2021; published online December 9, 2021. Assoc. Editor: Yusheng Liu.

Although the centralized learning of joint actions of agents as a team can solve coordination problems and avoid learning nonstationarity, it does not scale well as the joint action space grows exponentially with the number of agents [10]. Second, learning to differentiate joint actions can be highly difficult. Further, the neural networks obtained from centralized learning are only applicable to situations with the same number of agents for the trained cases because the action space is fixed by the trained cases.

In contrast to the centralized single-agent RL, during the multiagent RL, each agent can be trained using its own independent neural network. Such an approach solves the problem of the curse of dimensionality of action space when applying single-agent RL to multiagent settings. Although theoretical proof of convergence of multiagent independent Q-learning is not mathematically given, there are numerous successful practices in real-world applications [10]. Thus, applying the state-of-the-art independent multiagent RL is a promising approach in tackling the existing problems faced by SOS design.

In the rest of this paper, Sec. 2 provides a review of the relevant work in self-organizing systems and RL. After that, a multiagent independent Q-learning framework is presented as a complex system design approach in Sec. 3, together with the system design implications. In Sec. 4, a box-pushing case study is introduced that applies the proposed Q-learning model. Section 5 provides a detailed analysis and discussion of the simulation results. Finally, in Sec. 6, the conclusions are drawn from the case study, and future work directions are pointed out.

2 Related Work

2.1 Design of Complex Systems. A complex system is a system that consists of many interacting components and whose collective behavior is difficult to model due to interdependencies or interactions between its parts [11]. Much research has been carried out thus far to study how to design complex systems. For example, Arroyo et al. introduced a binary-tree based bio-inspired tool for fault adaptive design in complex engineering systems [12]. Königseder and Shea compared different strategies for rule applications in two computational design synthesis case studies: gearbox synthesis and bicycle frame synthesis and found that the effect of the strategy is task-dependent [13]. Meluso and Austin-Breneman developed an agent-based simulation that models the parameter estimation strategy in large-scale complex engineering systems [14]. McComb et al. developed a computation model to analyze how the characteristics of configuration design problems can be used to choose the best values for team characteristics such as team size and frequency of interaction [15]. Min et al. analyzed the structural complexity of complex engineering systems at various levels of decomposition [16]. Ferguson and Lewis proposed a method for the design of effective reconfigurable systems such as vehicles. They focused on identifying how system design variables change and analyzed the stability of the reconfigurable system using a state-feedback controller [17]. Martin and Ishii developed a design-for-variety approach that can help companies quickly reconfigure their products and address generational product variation to reduce the time from products to market [18]. Previous research on the design of complex systems requires extensive domain knowledge from designers to build models or draw inspiration from nature, which is time-consuming and hardly extendable to different scenarios. Also, generating design rules from global requirements to local interactions remains a challenging task.

2.2 Artificial Self-Organizing Systems. An artificial self-organizing system is a system that is designed by humans and has emergent behavior and adaptability like nature [1]. Much research has been done regarding the design of artificial self-organizing systems. Werfel developed a system of homogenous robots to build a predetermined shape using square bricks [19]. Beckers et al. introduced a robotic gathering task where robots have to patrol around a given area to collect pucks [20]. Khani et al. [5] and Khani and Jin [6] developed a social rule-based regulation approach in enforcing the agents to self-organize and push a box toward the target area [5,6]. Swarms of UAVs can self-organize based on a set of cooperation rules and accomplish tasks such as target detection, collaborative patrolling, and shape formation [21–24]. Chen and Jin used a FBR approach and guides self-organizing agents to perform complex tasks such as approaching long-distance targets while avoiding obstacles [8]. Price and Lamont investigated the use of genetic algorithm in optimizing self-organizing multi-unmanned aerial vehicle (UAV) swarm behavior [25]. The robotic implementations mentioned above have demonstrated the potentials of building self-organizing systems, and the design methods of self-organizing systems [5,6,8] have had their drawbacks, as indicated in Sec. 1.

2.3 Multiagent Reinforcement Learning. Multiagent RL applies to multiagent settings and is based largely on the concept of single-agent RL such as Q-learning, policy gradient, and actor-critic [9,26]. Compared with single-agent RL, multiagent learning is faced with the nonstationary learning environment due to the simultaneous learning of the multiple agents. In the past several years, there has been a move from tabular-based methods to the deep RL approach, resulting from the need to deal with the high-dimensionality of state and action spaces in multiagent environments and to approximate state-action values [27–29]. Multiagent systems can be classified into cooperative, competitive, and mixed cooperative and competitive categories [27]. In the SOS design, we focus on the cooperative agents since they share the same task goals.

One natural approach for multiagent RL is to optimize the policy or value functions of each individual. The most commonly used value function-based multiagent learning is independent Q-learning [30]. It trains each individual's state-action values using Q-learning [30,31] and is served as a common benchmark in the literature. Tampuu et al. [27] extended previous Q-learning to deep neural networks and applied DQN [32] to train two independent agents playing the game Pong [27]. Foerster et al. applied the COMA framework to train multiple agents to play StarCraft games with centralized critics evaluating decentralized actors [28]. In another work by Foerster et al., they analyzed their replay stabilization methods for independent Q-learning based on StarCraft combat scenarios [33].

As a multiagent environment is usually partially observable, Hauskneche and Stone [34] used deep recurrent networks such as LSTM [35] or GRU [36] to speed up learning when agents are learning over long time periods. Lowe et al. developed Multiagent Deep Deterministic Policy Gradient, which uses centralized training with decentralized execution and tested their algorithm in predator-prey, cooperative navigation, and other environments [37]. Brown and Sandholm developed a superhuman AI model called "Pluribus" using a self-play-with-search algorithm and showed that their model was able to defeat top human professionals in six-player-no-limit Texas hold"em poker [38]. Baker et al. demonstrated that through multiagent competition, simple game rules, and standard reinforcement learning algorithms at scale, multiple agents can create a self-supervised autocurriculum, which can generate various rounds of emergent strategy in hide-and-seek games [39]. Wu et al. developed Bayesian delegation, a decentralized algorithm that makes inferences like human observers about the intent of others and successfully tested their algorithm in a cooking video game called "Overcooked" [40].

Most approaches to multiagent RL attempts to achieve optimal system reward or desirable convergence properties. Many training algorithms are based on fully observable states. Training of multiagent reinforcement model is usually conducted on prespecified environments, and the generalizability of the training network to various multiagent team sizes and robustness of training network to change of situations are not analyzed or considered, which are important factors of consideration in SOS design. It is crucial to develop a multiagent learning framework that is scalable to various team sizes, robust to situation change and also to provide guidelines on how design should be implemented and analyzed. Such areas are often omitted in the literature and are the focus of this paper.

3 A Deep Multiagent Reinforcement Learning Model

3.1 Single-Agent Reinforcement Learning. It is important to discuss the single-agent RL before moving into multiagent RL since many concepts and algorithms of multiagent RL are based on the single-agent RL.

Single-agent RL is used to optimize system performance based on training so that the system can automatically learn to solve complex tasks from the raw sensory input and the reward signal. In single-agent RL, learning is based on an important concept called Markov Decision Process (MDP). An MDP can be defined by a tuple of $<S, A, P, R, \gamma>$. S is the state space, which consists of all the agent's possible sense of environment information. A is the action space, including all the actions that can be taken by the agent. P is the transition matrix, which is usually unknown in a model-free learning environment. R is the reward function, and γ is the discount factor, which means the future value of the reward is discounted and worth less than the present value. At any given time t, the agent's goal is to maximize its expected future discounted return, $R_t = \sum_{t'}^{T} \gamma^{t'-t} r_t$, where T is the time when the game ends. Also, agents estimate the action value function $Q(s, a)$ at each time-step using Bellman Eq. (1) below as an update. Eventually, such a value iteration algorithm will converge to the optimal value function [9]

$$Q_{i+1}(s, a) = E[r + \gamma \max_{a'} Q_i(s', a')|s, a] \tag{1}$$

Researchers in the past uses Q-learning as a common training algorithm in single-agent RL [41,42]. Q-learning is based on Q-tables, each state-action value pair is stored in a single Q-table and such training algorithm has been applied in simple tasks with small discrete state and action spaces [41,42]. However, in real-life engineering applications, state space can often be continuous and action space vast, making it difficult or impossible to build a look-up Q-table to store every state-action value pair. In order to overcome such problems in Q-learning, deep neural networks are introduced as functional approximators to replace the Q-table for estimating Q values. Such learning methods are called deep Q-learning [32]. A Q-network with weights θ_i can be trained by minimizing the loss function at each iteration i, illustrated in Eq. (2)

$$L_i(\theta_i) = E[(y_i - Q(s, a; \theta_i))^2] \tag{2}$$

where

$$y_i = E\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})\right] \tag{3}$$

is the target value for iteration i. The gradient can be calculated with Eq. (4):

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s,a,r,s'}[\{r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a, \theta_i)\} \nabla_{\theta_i} Q(s, a, \theta_i)] \tag{4}$$

Various approaches have been introduced to stabilize training and increase sample efficiency for training deep Q-networks. In our multiagent training algorithm, the neural network of every single-agent is built based on the following two approaches.

3.1.1 Experience Replay. During each training episode, the agents' experiences $e_t = (s_t, a_t, r_t, s_{t+1})$, which represents *state*, *action*, *reward,* and *next state*, are stored and appended to an experience replay memory $D = (e_1, e_2, \ldots, e_N)$. N represents the capacity of the experience replay memory. At every training interval, mini-batches are randomly sampled from experience replay memory D and fed into Q-learning updates. At the same time, an agent selects its action based on the *ϵ-greedy* policy, which means the agent selects its action based on the exploration of random actions and exploitation of the best decision given current information. Experience replay, as Minh described in his paper [43], increases data sample efficiency and can break down the correlations between subsequent experiences and is used to stabilize training performance.

3.1.2 Dueling DQN. The Dueling DQN architecture can identify the right action during policy evaluation faster than other algorithms as it separates the Q value into the representation of state value V and action advantages A, which are state-dependent [44]. In every Q value update, the dueling architecture's state value V is updated, which contrasts with the single-stream architecture, where only value for one of the actions is updated, leaving other actions not updated. This more frequent updating allows for a better approximation of the state values and leads to faster training and better training performance.

3.2 Multiagent Reinforcement Learning. As mentioned above, there are generally two approaches in multiagent training. One is to train the agents as a team, treating the entire multiagent system as "one agent." It has good convergence property similar to single-agent RL but can hardly scale up. In order to increase learning efficiency and maintain scalability, a multiagent independent deep Q-learning (IQL) approach is adopted. In this approach, A_i, $i = 1, \ldots, n$ (n: number of agents) are the discrete sets of actions available to the agents, yielding the joint action set $A = A_1 \times \cdots \times A_n$. All agents share the same state space and the same reward function.

During training, each agent has its own dueling neural network and is trained by applying deep Q-learning with experience replay. Agents perceive the state space through their local sensors. Each agent learns its own policy and value function individually to choose its actions based on its own neural network given the reward from the environment through the shared reward function. As each agent's action space size is the same, each agent's trained neural network can be reused and applied in various team sizes, and such multiagent systems can scale to agent teams of different sizes. Because the advantage of the self-organizing system is that system requires little or no communication between agents, using independent deep Q-learning is most suitable for the training of self-organized agents. Other variants of deep multiagent reinforcement learning algorithms, such as deep distributed recurrent Q-learning [45] require adding extra recurrent layers to learn the communication structure between agents, which greatly increases the training time and is not needed for the training of self-organizing agents.

In our multiagent RL mechanism described above, each agent $i (i = 1, 2, \ldots, n)$ engages in learning as if it is in the single-agent RL situation. The only difference is that the next state of the environment, S_{t+1}, is updated in response to the joint action $a_t = \{a_1, a_2, \ldots, a_n\}$, instead of its own action a_i, in addition to the current state S_t. Although the multiagent RL research has focused on general machine learning issues such as applying deep learning techniques and computing effectiveness [26], little work has been done to address the contextual stability or convergence issue of the learning process—i.e., whether the knowledge can be acquired in the form of neural networks through RL; the differentiation issue—i.e., how agents learn to differentiate themselves during learning and what affects such process, and the adaptability issue—i.e., whether the learned neural networks can be effectively applied to changing situations or different team sizes. These issues are the focus of this research.

4 Case Study

In order to test the concepts and explore the multiagent RL algorithm discussed above, a box-pushing case study has been carried out. In choosing this case example, several requirements were considered based on our long-term goal of developing robotic self-organizing assembly systems. First, the task environment requires relatively intense agent interactions, instead of sparse interactions, for efficient learning. For example, the ant foraging task may be less desirable as the interaction between agents during training is only passive, causing it slow and ineffective. Second, the tasks require cooperative work among agents. Although each agent might have different short-term rewards, in the long run, they work for the same maximum reward. Lastly, only homogeneous cases are considered for simplicity at this stage of research, and the action space should be the same for all the agents. This will allow us to add more agents to the system using the same learned neural networks. After considering several options, the box-pushing problem was finally chosen for the case study.

4.1 The Box-Pushing Problem. The box-pushing problem is often categorized as a trajectory planning or piano mover's problem [46]. Many topological and numerical solutions have been developed in the past [46]. In our paper, a self-organizing multiagent deep Q-learning approach is taken to solve the box-pushing problem. During the self-organizing process, each agent acts based on its trained neural network, and collectively all agents can push the box toward a goal without any system-level global control.

In this research, the box-pushing case study was implemented in pygame, a multiagent game package in the PYTHON environment. In the box-pushing case study, each individual agent is trained with its independent deep Q-learning (IQL) neural network as a member of a team. After the training, the resulting neural networks are applied to the testing situations with various team sizes between 1 and 10. Both training and testing results are analyzed to elicit the learning properties and the quality, scalability, and robustness of the learned neural networks.

A graphical illustration of the box-pushing case study is shown in Fig. 1. The game screen has a width x of 600 pixels and a height y of 480 pixels. Numerous agents (the green squares) with limited pushing and sensing capabilities need to self-organize to push and rotate the box (the brown rectangle or the black dotted rectangle, depending on simulation setup) toward the goal (the black dot with a "+" mark). As there is an obstacle (the red dot) on the path and walls (the solid black lines) along the side, the agents cannot just simply push the box but must rotate the box when necessary [5,6]. This adds complexity to the task. The box has sensors deployed at its outside boundary. When the outside perimeter of the box reaches the horizontal x-coordinate of the goal, represented as a white vertical line, the simulation is considered a success.

There are four major tasks of box-pushing, as summarized below. Agents need to move, rotate the box, and keep the box away from potential collisions with walls and obstacles.

T1 = <Move><Box> to <Goal>
T2 = <Move><Box> to <Goal>
T3 = <Move><Box> away from <Walls>
T4 = <Move><Box> away from <Obstacle>

In pygame, the distance is measured by pixels. Each pixel is a single square area in the simulation environment. In this case study, two different boxes are used, namely, *Smal-Box* which is 60 pixels wide and 150 pixels long, shown as a solid brown box in Fig. 1, and *LargeBox*, which is 90 pixels wide and 225 pixels long, shown as the black dotted hollow box in Fig. 1.

In box-pushing, agents have *limited sensing and communication capabilities*. They can receive information from the sensor on the box, which measures the orientation of the box and senses the obstacles at a range of distance. They have limited storage of observation information: their experiences such as state, action, reward, and next state. They possess a neural network that can transform the perceived state information into action. These assumptions are in line with the definition of the "minimalist" robot [47] and are reasonable with the current applications of physical robot hardware [48].

4.2 State Space and Action Space

4.2.1 Task State Space. Based on the task decomposition and constraint analysis mentioned above, the state space of the box-pushing task is defined as shown in Fig. 2. In order to gather relevant environment information, a sensor is deployed in the center of the box, which can sense nearby obstacles. The radius of the sensor range is 150 pixels, and the entire circular sensor coverage is split into eight sectors of equal size, as shown in Fig. 2. In this case study, part of the state space captures the state of each sector at any given time: whether the sector is occupied by an obstacle, represented as 1, or not, represented as 0. Furthermore, we assume the sensor can also detect the orientation of the box's x-axis with respect to the location of the goal [41,42]. Denoting the sectors with s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, and the box orientation with s_9, respectively, the state space can be defined as

$$S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\} \quad (5)$$

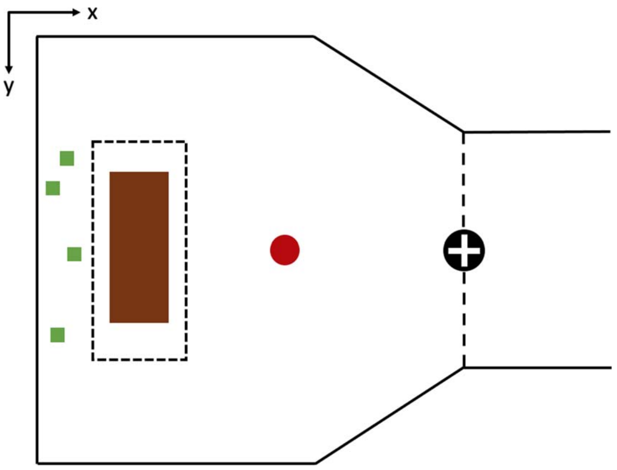

Fig. 1 A graphical illustration of the box-pushing task

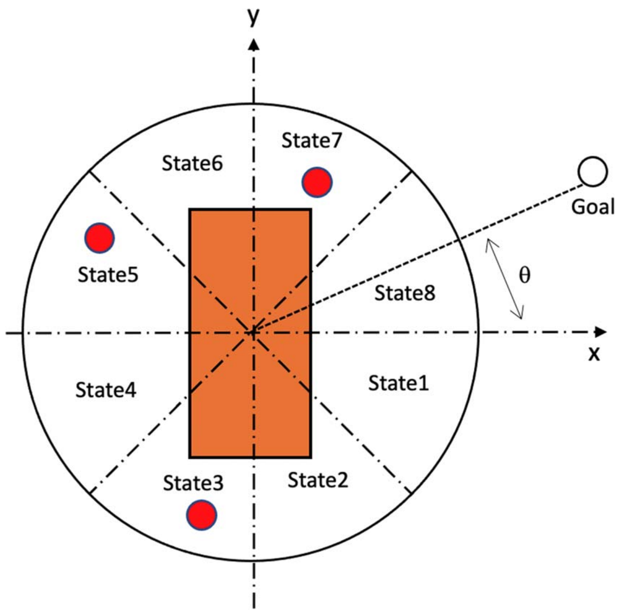

Fig. 2 Box-pushing task state representation

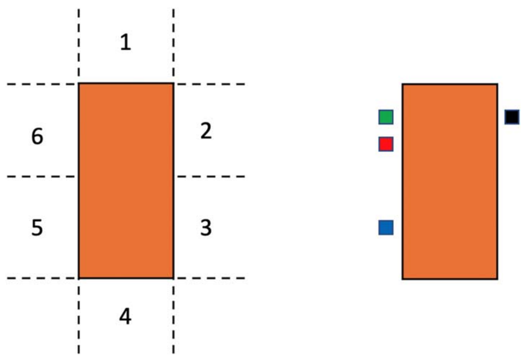

Fig. 3 The six regions of the box neighborhood

For the example of Fig. 2, sectors 3, 5, and 7 are occupied by obstacles. Therefore, the corresponding state attributes s_3, s_5, s_7 are having the value 1, and s_1, s_2, s_4, s_6, s_8 the value 0. In Fig. 2, the box angle θ is about 30 deg. And such degree information can be shaped into the range of $[-1,1]$ by applying $s_9 = \theta - 180/180 = 0.83$. This shaped angle method can facilitate deep Q-network training and is used commonly in practice [41,42].

Given the above, the state representation of Fig. 2 can be expressed as a nine-item tuple <0,0,1,0,1,0,1,0, −0.83>.

During training, each agent is close to the vicinity of the box center, so it can receive the sensor information broadcasted locally among agents. It is assumed that the sensor can also sense the distance from the center of the box to the location of the goal, analogous to real-world radar sensor, and is also like the gradient-based approach in literature where the task field is assumed [5,6]. Agents can also receive such distance information from the sensor.

4.2.2 Box Neighborhood. The box neighborhood is defined as six regions [6,49], as shown in Fig. 3. During each simulation, an individual agent can move to one of the six regions of the box neighborhood and that specific neighborhood is the position of the agent. As the individual agent is relatively small, we assume there can be multiple agents in the same region at the same time. This is in line with the definition of the "minimalist" robot [47].

4.2.3 Box Dynamics. The box dynamics are based on a simplified physical model. The box movement depends on the simulated force and torque. Forces equal to the sum of vector forces of each pushing agent. Every push carries the same amount of force, which acts from an agent toward the box in the normal direction. The sum of two pushes will move the box 10 pixels in a given direction. Torque is assumed to be exerted on the centroid of the box, and two pushes on the diagonal neighborhood of the box (such as positions 2 and 5 each) will rotate the box 10 deg. We assume the box carries a large moment of inertia, and when it hits the obstacle, which is considered rather small, it will continue its movement until its expected end position is reached.

4.2.4 Agent Action Space. The *agent action space* is defined based on the box neighborhood and simulated box dynamics. At each time-step, an agent can choose a place in one of the six regions of the box neighborhoods to push the box. Therefore, the agents share the same actions space of $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, as shown in Fig. 3. For instance, if an agent chooses action a_1, it will *move* to box region "1" and *push* the box from there, the box will move downward along the box's y-axis based on the simulated box dynamics and the same logic applies to other agent actions. It is worth mentioning that it takes multiple moving steps, hence unit times, for an agent to move from one region to another depending on the distance between the starting and ending regions.

4.3 Reward Schema and Training Model. To train multiple agents to self-organize and push the box to the final goal area, which is the group level function, we need to design a proper reward schema to facilitate agent training. Adapted from previous Q-table based box-pushing reward schema [41,42], we designed a new reward schema for agents' box-pushing training. The total reward is composed of four parts: *distance*, *rotation*, *collision*, and *goal*.

4.3.1 Distance Reward. The reward for pushing the box closer to the goal position is represented as R_{dis} and is shown in Eq. (6). The previous distance D_{old} represents the distance, measured in pixels, between the center of the box and the goal position in the previous time-step. D_{new} represents such distance at the current time-step. C_d is a constant, called *distance coefficient* in our simulation, and is set to 2.5. At each simulation time-step, agents calculate the change of distance between the current distance and previous distance based on Eq. (6) and draw its distance reward

$$R_{dis} = (D_{old} - D_{new})^{*} C_d \tag{6}$$

4.3.2 Rotation Reward. The reward for rotation R_{rot} is represented in Eq. (7). where α_1 is the previous time-step angle of the box's x-axis with respect to goal position and α_2, the current angle. The rotation reward is given to discourage the rotation of more than 11 deg, this way, the box can be rotated constantly with small degrees and avoid large rotation momentum, which can result in a collision with obstacles. The rotation reward is relatively small as it is used only for rotation of the box rather than pushing the box toward the goal, which is the task's goal

$$R_{rot} = \mathrm{Cos}(\alpha_2 - \alpha_1) - 0.98 \tag{7}$$

4.3.3 Collision Reward. The collision reward is analogous to the reward schema in common collision avoidance tasks [50] and is represented in Eq. (8) with R_{col}. During each simulation step, if there is no collision for the box with either the obstacle or the wall, $R_{col} = 0$. If a collision occurs, a −900 reward will be given to all the agents as a penalty

$$R_{col} = \begin{cases} -900 & \text{if collision occurs} \\ 0 & \text{if no collision occurs} \end{cases} \tag{8}$$

4.3.4 Goal Reward. The reward for reaching the goal R_{goal} is represented in Eq. (9). At each simulation step, if the box reaches the goal position, each agent will receive a positive 900 reward; if the goal is not reached, the agents do not receive any reward

$$R_{goal} = \begin{cases} 900 & \text{if reaching goal} \\ 0 & \text{if not reaching goal} \end{cases} \tag{9}$$

The total reward is a weighted sum of all these rewards, as shown in Eq. (10)

$$R_{tot} = w_1^{*} R_{dis} + w_2^{*} R_{rot} + w_3^{*} R_{col} + w_4^{*} R_{goal} \tag{10}$$

In our simulations, after repeated testing, the weights were set as $w_1 = 0.6$, $w_2 = 0.1$, $w_3 = 0.1$, $w_4 = 0.2$, with the sum of these weights equal to 1, shown in Eq. (11). The weights are chosen so that during each step in training: $w_1 = 0.6$, which means agents can have a more immediate reward in terms of whether or not they are closer to the goal; $w_2 = 0.1$, which gives a little incentive for agents to rotate the box a small angle; $w_3 = 0.1$, to offer some penalty reward if box collides with an obstacle; $w_4 = 0.2$, as agents' final goal is to reach the target zone, agents should be given more reward if its goal is achieved, than when box collides with an obstacle. The weight for four different rewards is adapted and based on previous research on multiagent box-pushing [41,42]

$$w_1 + w_2 + w_3 + w_4 = 1 \tag{11}$$

During training, as the agents are homogenous and are cooperating to push the box, they should receive the same rewards. This is

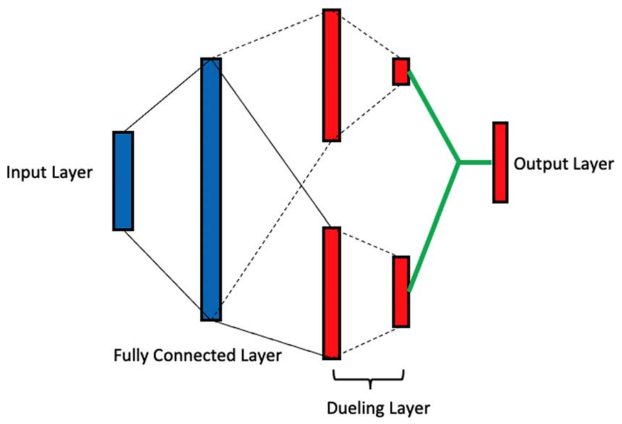

Fig. 4 An illustration of the training neural network

the reason why the reward Eqs. (8) through (10) are defined based only on the box's position and orientation. In this way, each agent's neural network will consider other agents' actions as part of its environment and learn to explore its action space and also its best policy based on an ϵ-*greedy* action selection strategy. Gradually the agent grasps how to differentiate its actions from other agents to collaboratively push the box toward the goal position, which is the characteristic of multiagent independent deep Q-leaning neural networks.

As shown in Fig. 4, the training network consists of an input layer, which gets input information from the raw sensor of the box, a subsequent fully connected layer with 16 hidden neurons, a dueling layer and outputs final state-action values.

4.4 Experiment Design.

4.4 Experiment Design. As aforementioned, the three issues of this research are (1) *the stability of the learning dynamics* for agents to acquire knowledge for self-organizing behavior regulation, (2) *the differentiation of individual behavior* during the self-organizing process, and (3) *the use of neural network knowledge* captured from the RL in the extended situations such as varying team sizes, noisy environment, and change of box dimension. In order to address these issues, a set of multiagent training and testing experiments have been conducted, as described below.

4.4.1 Multiagent Reinforcement Learning-Based Agent Training. As illustrated in Fig. 5, for the multiagent RL based training process, the Deep Q-learning algorithm described above was applied. The two independent variables of the training are the *Number of Agents* involved in the box-pushing task, denoted by *#Tr*, varying between 1 and 10, and the *Box Type* that the agents are trained for, either *Small Box* (60×150) or *Large Box* (90×225). In this paper, only *Small Box* is used for training, and the *Large Box* situation is for future investigation. The dependent variables of the training process are the *Cumulative Reward* signifying the learning behavior of the agents, *Agent Travel Distance* capturing how each agent behaves differently during the process, and a set of *#Tr* neural networks trained for the given *Small Box* type denoted as $NN^{\#Tr}_{Small}$, representing the learning knowledge.

During training, each episode is defined by a complete simulation run, from the starting point to the ending point. The starting point is arranged as shown in Fig. 1, where the agent positions are randomly assigned while the box's initial position and the goal position are both fixed. For the ending point, there are three different situations: (1) the box is pushed into the target position, (2) the box collides with the obstacle or any sidewall, and (3) the simulation reaches the maximum number of 500 training steps (each time an agent chooses its action, it is considered one training step). When any one of the three situations happens, a simulation run is completed and the episode finishes.

One training process in the experiment goes through 8000 episodes for *Small Box* cases and 16,000 for *LargeBox* cases. The numbers are chosen because they are the thresholds for achieving training stabilization at the minimum training cost. In order to maintain statistical stability, each training was run 100 times with different random seeds. The mean value of the 100 cumulative rewards of a corresponding episode is used as the reward value of that episode and plotted as the patterned lines in the cumulative reward plots in Sec. 5. At the same time, the standard errors are plotted as green spreads around the mean values.

As shown in Fig. 5, the three dependent variables of the training process are: *cumulative rewards*, *agent travel distances*, and the resulting *neural networks,* representing the team learning behavior, agent functional differentiation, and the agents' learned knowledge, respectively. Details of these variables are discussed in Sec. 5.

Table 1 summarizes the training parameters used in the simulations. Replay memory size is 1000, and 32 mini-batches are randomly sampled from the replay memory during each training step. The discount factor is 0.99, and the learning rate is 0.001. Total training episodes are 8000 and 16,000 for the *Small Box* and *Large Box, respectively*. The ϵ-greedy action selection algorithm was followed where ϵ is annealed from 1.0 to 0.005 over 100,000 training steps during training. Agents choose actions from entirely random (i.e., $\varepsilon = 1.0$) to nearly greedy ($\varepsilon = 0.005$). The target neural networks are updated at every 200 training steps to stabilize the training.

4.4.2 Testing and Performance Evaluation. The neural networks that are acquired from the MARL based training are evaluated through testing simulations. Figure 6 illustrates the design of the testing experiment, in which *#Tr* indicates the number of agents trained and *#Ts* is the number of agents being tested.

There are four independent variables in the testing experiment. They are *learned neural networks* $NN^{\#Tr}_{Small}$, *agent team size* (1, ..., 10), *box type* (Small, Large), and *noise level* (10%, 30%, 50%). The noise level is defined by the percentage in which the agents choose *random* actions instead of *greedy* ones during testing runs. This variable is introduced to model system malfunctions, such as sensor errors and agent disabilities. By correlating the system response to the noise levels, the robustness of the trained neural networks can be evaluated.

When the number of trained agents *#Tr* is not the same as the number of testing agents *#Ts* (i.e., $\#Tr \neq \#Ts$), there exists a mismatch for deploying the neural networks. To resolve this mismatch, for cases of $\#Tr < \#Ts$, one or more randomly selected networks are repeatedly deployed in the test case. When $\#Tr > \#Ts$, then *#Ts* networks are randomly selected for the testing team agents.

There is one dependent variable *success rate* in the testing experiment design, as shown in Fig. 6, which indicates the *quality of the*

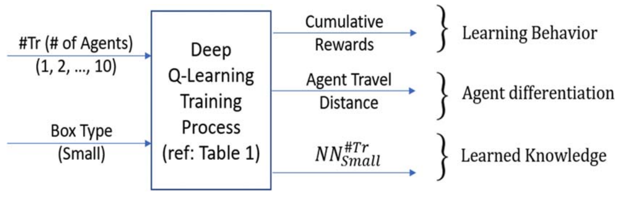

Fig. 5 Agent training experiment design

Table 1 Simulation parameters

Replay memory size	1000
Mini-batch size	32
Discount factor	0.99
Learning rate α	0.001
Total training episodes	8000/16,000
ε	$1.0 \rightarrow 0.005$
Annealing steps	100,000
Target network update frequency	200

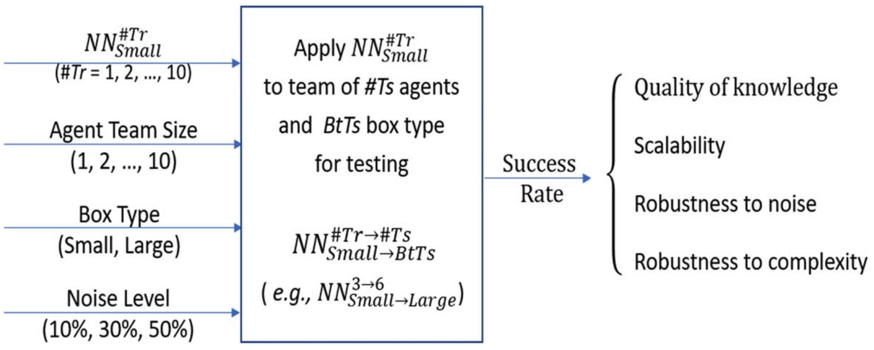

Fig. 6 Testing experiment design and performance evaluation

learned knowledge and *scalability* as it is *trained* ($\#Tr = \#Ts$) and when it is applied in *knowledge transfer* testing situations ($\#Tr \neq \#Ts$ and/or $BoxTypeTr \neq BoxTypeTs$), respectively. Furthermore, correlating the *success rate* with the *noise level* and the *box type* helps assess the *robustness to noise* and *robustness to task complexity*, respectively.

As mentioned above, for each training situation, i.e., given *#Tr* and *BoxType*, there are 100 training runs. At the end of each training run, a *greedy testing simulation* is carried out to evaluate whether the box-pushing is successful. The *training success rate,* in this case, is defined by the percentage of the number of successful runs. Usually, the training success rate is greater than 60% except for the team of one agent, which is about 30%, as will be discussed below.

From the more than 60 successful runs of each training situation (*#Tr, BoxType*), 50 sets of neural networks are randomly selected for testing cases. Again, for statistical stability, each selected set of neural networks is simulated 100 times, and the *success rate* is the percentage of the successful ones among the 100 simulations. This way, after 50×100 testing runs, the average success rate of the 50 sets of neural networks is used as the *success rate* of the testing situation (*#Tr, BoxType, NoiseLevel*).

5 Results and Discussion

5.1 Typical Failure Patterns. Not all training simulation runs are successful. Figure 7 illustrates some typical failure patterns with motion traces during training. Figures 7(*a*)–7(*d*) are the results of running the simulation cases of *six agents* pushing a *Small Box*. During training, the box sometimes experiences excessive momentum, leading to moving toward the red obstacle, as shown in (a). Sometimes agents have insufficient or excessive rotation torque, resulting in the edge of the box hitting the red obstacle, as shown in (b). Agents try many actions but still could not get to the goal; they collide with obstacles or reach a maximum training step size of 500 (each time an agent chooses its action, it is considered one training step size. When maximum training step size is reached,

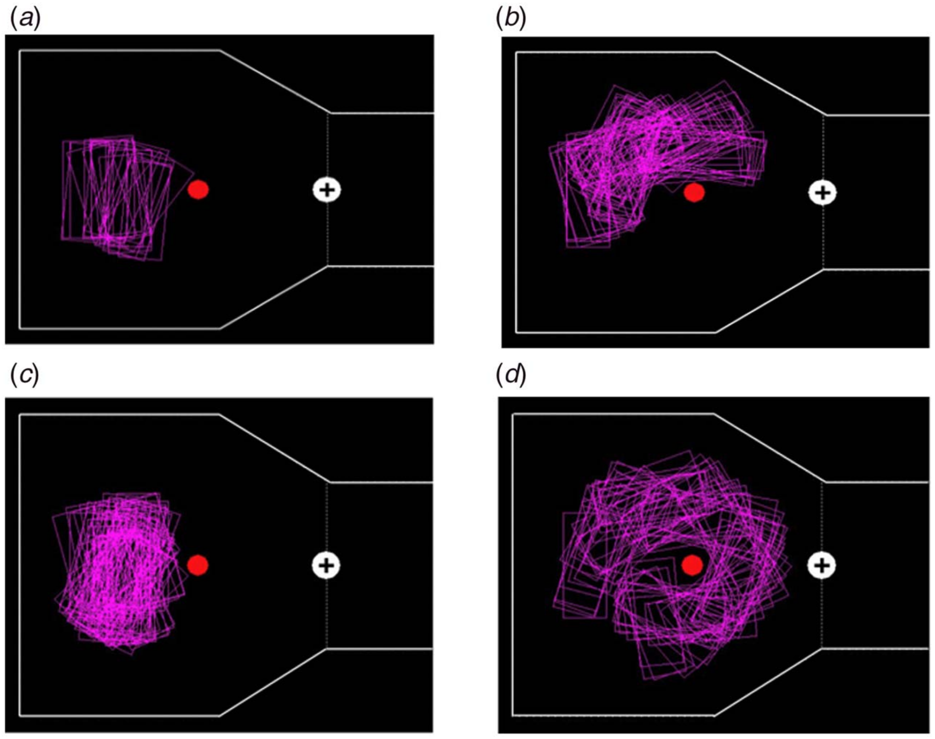

Fig. 7 Typical failure patterns of the box-pushing task: (*a*) failure with excessive pushing momentum, (*b*) failure with insufficient/excessive rotation torque, (*c*) failure with reaching maximum training episode length, and (*d*) failure with pushing box backward

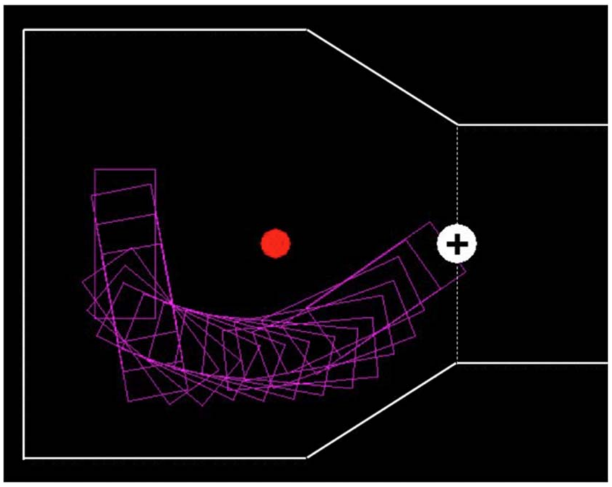

Fig. 8 A successful box-pushing trajectory with motion traces of a six-agent team pushing a small box

the episode terminates), as shown in (c). The boxes are occasionally pushed backward, forming a round circle around the obstacle, as shown in (d). Need to mention that these failure modes are the representation of how agents are exploring and learning from the environment during training. As agents initially adopt complete random actions and gradually move to more greedy actions, these failure modes are not due to inappropriate reward functions as there are always failing cases happening during learning due to random actions. Rather, these failure modes indicate the effectiveness of the training algorithm by showing agents are gradually learning from the feedback they received from exploring the environment.

5.2 Basic Learning Behavior

5.2.1 Box-Pushing Trajectory. Figure 8 shows one successful testing run with *six agents* pushing a *Small Box* (60×150) by following max state-action values with motion trace examples. Although the final optimized trajectory is not strictly perfect, through applying the multiagent deep Q-networks, agents can approximate its actions and push the box toward the goal area.

5.2.2 Training Stability. Figure 9 shows the convergence results of our simulations with different numbers of agents pushing a *small box*. The figures show the average cumulative reward of 100 random training seeds with respect to the training episodes. The standard error of the mean is plotted with the green shaded region.

As shown in Fig. 9, when the number of agents is small, e.g., between 1 and 3, it is hard for the system to reach a stable and high reward level compared with the other team sizes. The standard deviation of these training cases is large. When the agent number is 4, it reaches a reasonable level of reward and the reward becomes relatively stable after 5000 episodes, but its level is around 600, smaller than the 650 achieved by the teams of larger size. This implies that the box-pushing task itself requires the cooperative effort of agents and the size of agents need to be at least five agents to reach a reasonable performance level. This "magic" number *5* appears to be correspondent to the complexity level of the box-pushing task [51], which will be mentioned again below.

When the team size is between 5 and 10, simulations converge to, and stabilize at, the similar maximum reward after a certain number of episodes. It can also be seen that it takes ~500 more episodes incrementally for agents between 5 and 10 to reach maximum reward: 4500 episodes for five agents, 5000 episodes for six agents, 5500 episodes for seven agents, and eventually 7000 episodes for ten agents. This result is reasonable as more agents add more complex behaviors during training and the system needs more time to adapt. Although the complexity of the task does not change (pushing the box to goal position), the complexity of the system itself increases when team size expands: agents need to learn not only the physical environment but also other agents' actions. Thus, a larger number of agents require more training time to reach stable convergence. Increasing the number of agents given that the task complexity corresponds to a system complexity of five agents, having more agents involved only makes the learning less effective and takes longer to converge.

In the training phase of multiagent reinforcement learning, the knowledge learned by the agents is highly dependent on the complexity of the box movements generated by the various agents during box-pushing. For example, in terms of the rotation behavior of the box, when the number of agents is small, the diversity of rotation movements is limited. If there is one agent, the individual agent can only rotate the box a maximum of 5 deg based on the box dynamics. However, when team size increases, the addition of various agents can greatly increase the diversity of box rotation movements. It can be imagined that when there are five agents, there can be various rotation movements ranging from 0 to 25 deg of rotation when various agents are exploring how to push the box. The addition of agents increases the diversity of box movements, thus the complexity of the system during learning. When team size reaches "magic" number 5, the complexity of the system matches well with the task complexity, and thus the system has the best training performance. Further increasing the number of agents does not improve the training anymore as the system complexity already reaches the threshold of task complexity

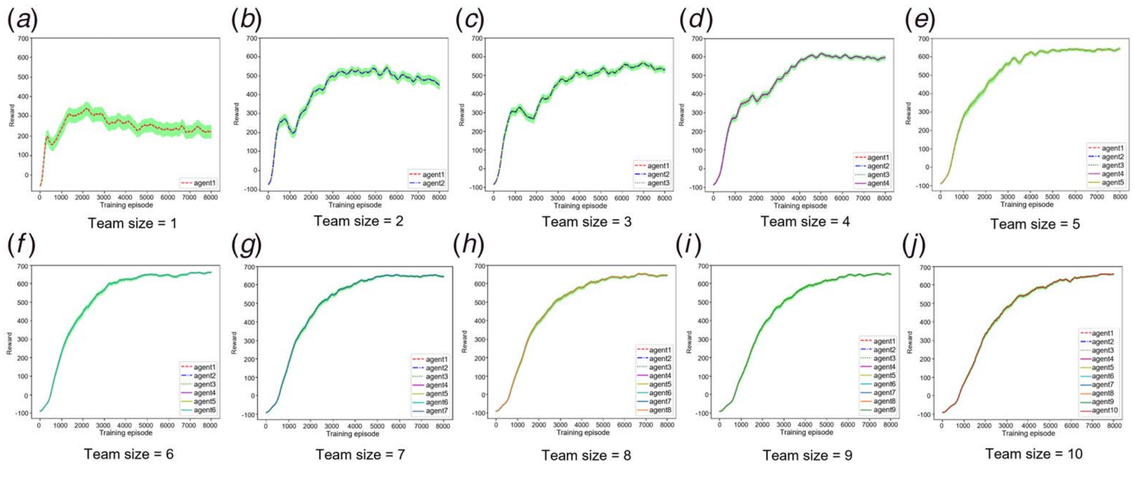

Fig. 9 Cumulative reward versus training episode plots with varying size of teams pushing a small box

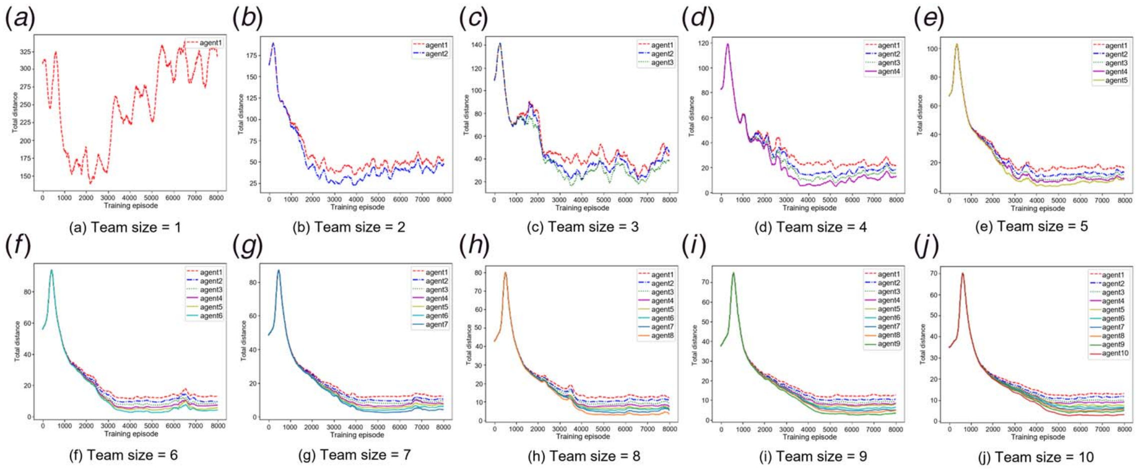
Fig. 10 Total distance traveled by each agent with varying team size for the small box

around "magic" team size, and maximum learning capability of agents is already achieved.

5.3 Behavioral Differentiation. The total distance traveled by each agent in different team size settings is used as a measure of the functional differentiation among individual agents, as shown in Fig. 5, because if two agents hold different total travel distances and stay that way, they must have performed different roles in the box-pushing task. Figure 10 illustrates the simulation results. The vertical axis measures the average distance traveled by the agent during each episode based on 100 random seeds. As there are a total of 8000 episodes, we plotted 8000 distance values during training based on the trained neural networks. Here, the distance is based on a hierarchical measurement [52], meaning the top-level distance is measured. For instance, when agents move from box region 1 to box region 2, as shown in Fig. 3, it is considered as one stepwise distance. When an agent moves from box region 1 to region 3, its distance is two-step distances. If an agent chooses the same action in the subsequent run, its moving distance is 0.

When team size is 1, it is very difficult for 1 agent to figure out how to efficiently push the box. Also, it is hard for teams of sizes between 2 and 4 to reach stable travel distance among agents, indicating that the agents can hardly differentiate among themselves. A little environmental noise will cause the agents to change themselves, and thus their distance traveled is not stable. Beginning from team size 5, the distance traveled by agents tends to reach more stable differentiation. When team size becomes 7 or more, the agents reach a very stable traveled distance. This is similar to human organizations; larger organizations tend to have more stable differentiation among their members to maintain the normal operation of the system, while the small organization members tend to do everything they encounter.

Staring from team size 2, during the initial phase of training, the distance traveled by all agents tends to increase. After some training episodes, it starts to decrease and eventually converges to a stable low value. This is reasonable as initially all agents are learning how to push the box and more randomly exploring the search space. After some episodes, when agents gather enough positive and negative experiences, and as the epsilon value decreases, agents choose more greedy actions; their travel distance tends to decrease. After 8000 episodes of training, the distance traveled by all agents tends to converge to low distance values. The final distance traveled by each individual agent is different as they finally specialize in their own individual behavior to collaboratively push the box to the goal position.

The results in Fig. 10 also show that initially the distance traveled by agents is approximately the same, and the total distance traveled by each agent tends to diverge when it reaches a certain threshold, around 1200 episodes. The results indicate that the real starting in learning differentiation occurs after agents have gathered enough training outcomes and experiences from the environment (reaching goal or collision) and do not depend much on team sizes.

5.4 Quality of Learned Knowledge. In addition to analyzing the learning behavior described above, the quality of the learned knowledge is evaluated based on the success rate. The "success rate" (see Fig. 6) of the learned knowledge of a specific team is defined by the percentage of the successful simulation runs based on the learned knowledge, as discussed in Sec. 4.4. Because the multiagent RL is only partially observable, not all training cases can be successfully trained. By "being successfully trained," it is meant that after training, the trained agents can successfully push the box toward the target by taking actions *greedily* suggested by their own neural networks. Figure 11 illustrates the results of the success rate of the trained teams of different sizes. When the number of agents is small, the success rate of training is low as it is hard to train a small number of agents to push the box. When the number of agents reaches the "magic" number 5, the success rate of training becomes 97%. When the number of agents further increases, the success rate remains almost the same. This means that the quality of the learned knowledge depends on the team size: higher-quality knowledge can be acquired for the team sizes equal to or greater than the "magic" number 5; for smaller teams, the learned knowledge is much less reliable. Identifying the "magic" number—i.e., the matching point between the task complexity and team complexity—is the key to acquiring high-quality knowledge through multiagent RL.

5.5 Scalability and Robustness. The neural networks learned from a given team size are tested in teams of different sizes for scalability evaluation. In addition, random agent actions ranging from 10%, 30%, and 50%, are introduced to evaluate the robustness of the trained networks to environmental changes. The results of the small box are shown in Figs. 12–14. The different patterns of the curves in the figures represent the different sets of neural networks obtained from the training of different numbers of agents. For example, "Tr:3" means the neural networks used by the agents are obtained from the training of 3 agents. The horizontal axis indicates the number of agents that engaged in the testing of the box-pushing task, i.e., *#Ts*. The vertical axis shows the success rate of simulation runs.

5.5.1 Scalability for Knowledge Transfer. As shown in Fig. 12, except for the 1 agent case where the learning dynamic is very different from multiagent settings, when the trained network is applied to the same team size, agents can successfully push the box toward the goal over 90% of testing runs.

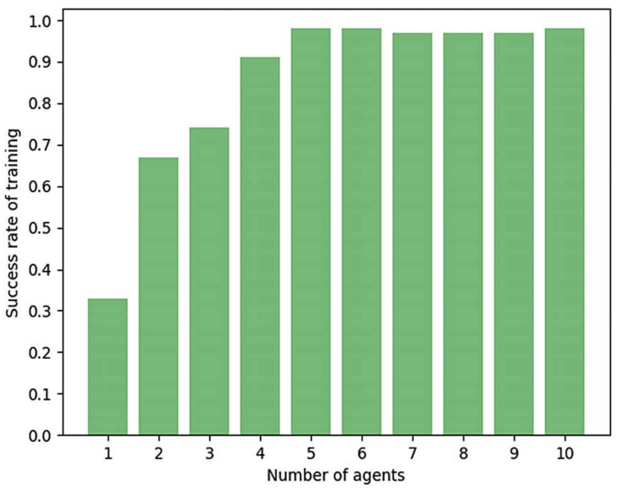

Fig. 11 Success rate of training with respect to different number of agents with small box (60×150)

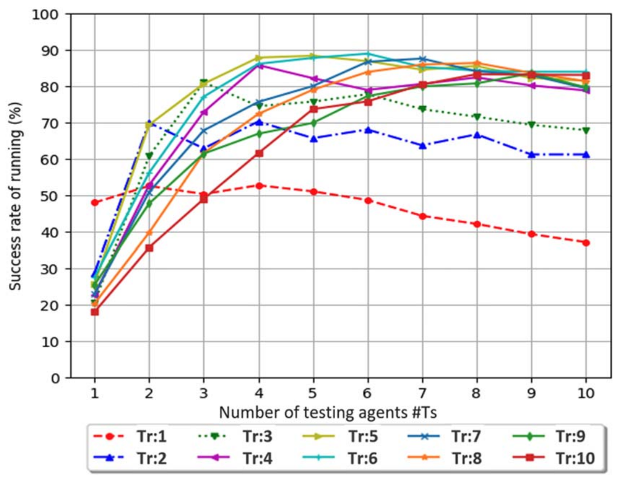

Fig. 13 Success rate of running with 30% individual random actions with varying number of trained agents with a small box

Scaling downward: When the learned neural networks $NN^{\#Tr}_{BoxType}$ are acquired from the trained teams of 6–10 agents, transferring $NN^{\#Tr}_{BoxType}$ downward to the teams of a smaller size *#Ts* results in relatively high success rates which remain stable, within 10%, for the neighboring 3–4 testing teams. When the trained team size *#Tr* is smaller than the "magic" number 5, transferring knowledge downward leads to inferior performance, which can be considered as due to the mismatch between the task complexity and the team complexity [51].

Scaling upward: When the trained agent team size *#Tr* is small and the learned networks are applied to a larger team, i.e., $\#Tr < \#Ts$, the success rates appear to be rather high and stable for trained teams of size $\#Tr \geq 4$. This shows that the learned knowledge from a trained team size of around the "magic" number, 4–5 in this case, has good scalability in the upward direction. For the trained team size *#Tr* of 2 and 3, scaling upward has led to reasonably good performances. It seems to be the case that the "low quality" of the learned knowledge can be compensated by the increased level of system complexity when more team members are added.

5.5.2 *Robustness to Noise.*

Figures 13 and 14 are the results based on 30% and 50% noise levels. The success rates generally drop with the increase of noise, as expected. It is interesting to see the cases where when the networks of smaller *#Tr* are applied to larger *#Ts* teams ($\#Tr < \#Ts$), the success rate is higher than the original teams trained at larger *#Ts*. For example, for the trained team size $\#Tr = 6$, and testing team size $\#Ts = 10$ with 50% noise level shown in Fig. 15, the success rate 60.04% is actually higher than that of directly trained ten agents ($\#Tr = \#Ts = 10$), which is 57.92%.

Overall, the learned neural networks are moderately robust when the system or environment noises lead to less than 30% agent misactions. Beyond that noise level, the networks can be highly unreliable.

5.5.3 *Robustness to Task Change.*

As stated above, the dimensions of the box-pushing environment are fixed, but the box size can vary from Small Box (60×150) to Large Box (90×225). Given the dimension constraints of the environment, pushing the Large Box is a more complex task than pushing the Small Box since it is easier for the Large Box to hit the obstacle or the walls. The robustness to task change of the learned knowledge can be evaluated by applying the networks trained from the Small Box situations to the Large Box testing experiments. As shown in Fig. 15, for all trained agent team sizes, the success rates of the transferred testing runs are low, about or fewer than 50%, indicating that the knowledge learned from multiagent RL is rather specific to the tasks it is trained for. When the tasks change, even only in dimensions, the applicability of the learned knowledge reduces significantly. Proper transfer learning strategies need to be devised so that the learned knowledge can be used as a basis for swift training in the new task environment.

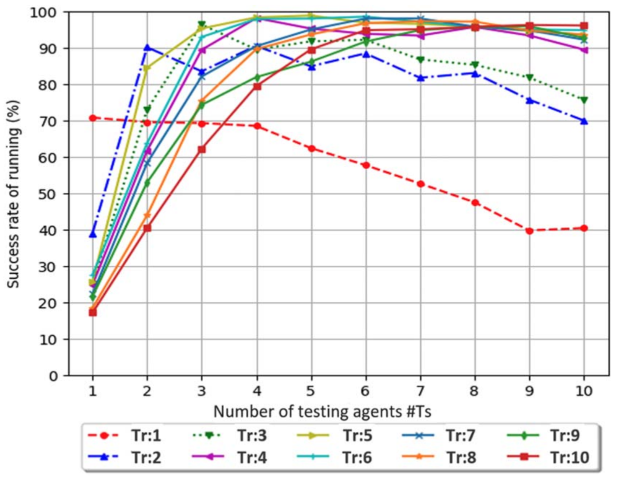

Fig. 12 Success rate of running with 10% individual random actions of varying number of trained agents with a small box

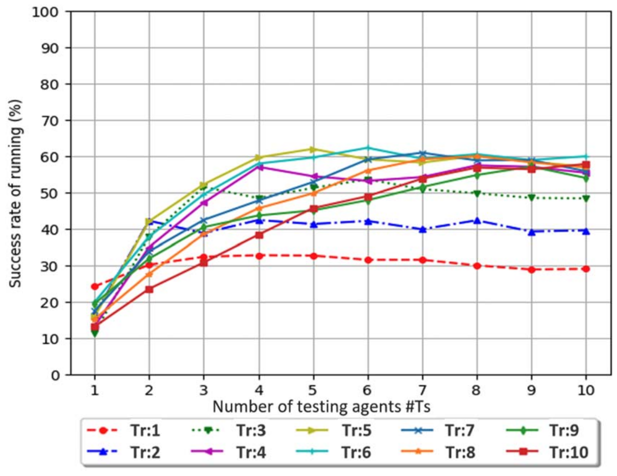

Fig. 14 Success rate of running with 50% individual random actions with varying number of trained agents with a small box

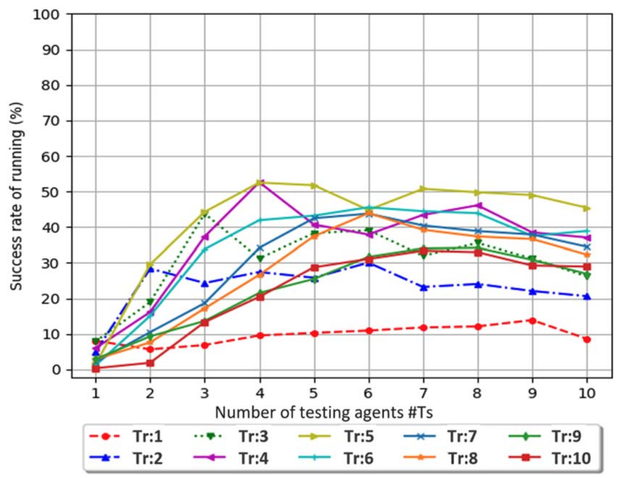

Fig. 15 Success rate with 10% individual random actions with varying team sizes when transferring knowledge from a small box (60 × 150) to a large box (90 × 225)

Although the robustness of the learned knowledge to the task change is low, the scaling pattern does not change from the Small Box cases, as shown in Fig. 15.

5.6 Findings. The results of the case study have shown that there are two forces of an SOS team that interact and govern the behavior of the multiagent teams: the *learned knowledge* and the *team dynamics* manifested as *team size* in this study. In addition, the task complexity level is another important factor that determines the "magic" team size for agent training and hence knowledge acquisition. The following findings and insights are drawn from the case study results.

- Multiagent RL is an effective approach to capture self-organizing knowledge through extensive training. In the box-pushing case, the agents successfully accomplished the task with very high success rates based on the learned neural networks.
- For a given task with specific task complexity, when the number of agents increases, the system complexity increases. When the system complexity matches the task complexity, reaching the "magic" number, the multiagent RL training yields the learned knowledge that has the best team performance and the highest-level of scalability and robustness. The multiagent RL based self-organizing knowledge capture should be performed around the "magic" team size.
- The multiagent RL training process has relatively good stability with a proper range of agent team sizes. In this research, the agents from 4 up to 10 have shown stable learning convergence. Failure cases do happen in the early stage of training. The converged networks have shown highly effective utility.
- Although the agents share the same reward function during training, they do specialize in their own ways as the learning converges at the final stage of training. Increasing the number of agents adds more stability to the differentiation. This is analogous to human organizations where large firms have more stable differentiation among individuals, while in small ones, the members tend to engage in everything in their workplace. Because the agent travel distance was measured in this study, the differentiation reveals the ways of movement of the agents. For more complex tasks, it is possible to identify relevant measures to understand for what and how the agents specialize.
- The influence of noise-induced random actions only has a limited effect until 10%, and a moderate effect until 30%. After that, the impact becomes destructive. As team size increases, the robustness improves and then degrades. For a given task, there appear to be specific "magic" team sizes for training that yields the most robust knowledge against noises.
- The self-organizing knowledge acquired from multiagent RL appears to be negatively sensitive to task changes. It is reasonable since the feature of machine learning in general is to map the specialized skill and knowledge into a nonlinear neural network transformation. There can be two ways to go around this issue. One is to reframe the original task such that the learned knowledge can be applied to the subcategories of the reframed problem. The other way is to add another layer of transfer learning so that the learned knowledge can be utilized to speed up the transfer learning process.

5.7 Implications for Future Self-Organizing Systems Design. While it is worth mentioning that the findings and insights described above are limited to the types of tasks reported in this paper, some general guidelines for future SOS design can be derived as follows.

We have shown that the multiagent RL framework can be an effective design tool for capturing self-organizing knowledge through extensive agents' interactions with the environment during training. After training, such knowledge can be stored in neural networks as integrated rules for agents to follow. As the entire learning process is model-free, it eliminates the need for trial and error processes of rule design in traditional SOS design methods [5].

Also, as was discovered in our box-pushing case study, SOS should be designed around the "magic" team size for best knowledge capture. Though the whole set of experiments were carried out in our simulation study, future SOS designs only need to test a few hypothetical "magic" team sizes using multiagent RL to obtain the system "magic" team size. For example, from the training stability perspective, starting from "magic" team size, training becomes more stable and reaches the maximum cumulative reward. Smaller team sizes have rather unstable training stability. Also, magic team sizes should have more stable behavior differentiation and better quality of the learned knowledge compared with smaller team sizes, while larger team sizes can have even more stable differentiation and similar quality of the learned knowledge. Finally, SOS with the "magic" team size should be most robust in response to environmental noises among all team sizes tested. Designers can choose to evaluate their hypothetical team sizes based on the above guidelines in order to achieve the desired system performance.

6 Conclusions and Future Work

In multiagent RL based SOS design, the learning stability, scalability of the learned knowledge to the varying team sizes, and its robustness to task changes are the three important issues to consider. In addition, the system needs to be robust enough to perform well in the face of noises, such as failures or malfunctions of other agents.

In this paper, a multiagent independent Q-leaning algorithm is devised for the design of SOS, and the issues of learning stability, scalability, and robustness are investigated through a box-pushing case study. The results show that multiagent RL can be a useful tool for designing SOS with needed learning stability. Identifying the optimal number of agents is important to achieve the best learning stability and scalability, and robustness of the learned knowledge. The robustness to task changes of the learned knowledge remains to be a key issue that will be addressed in future research.

In addition to robustness to task changes, future work includes testing the scalability of a multiagent Q-learning algorithm with a wider range of complex tasks. Furthermore, adding individual heterogeneous rewards, controlling, and measuring the specialization of the agents for more complex tasks will be the research for the next step.

Acknowledgment

This paper was based on the work supported in part by MTI Co. Ltd. and Nippon Yusen Kaisha (NYK). The authors are grateful to the sponsors and the MTI team for their discussions and insights on this research.

Conflict of Interest

There are no conflicts of interest.

References

[1] Reynolds, C. W., 1987, "Flocks, Herds and Schools: A Distributed Behavioral Model," Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, Anaheim, CA, July 27–31, pp. 25–34.

[2] Ashby, W. R., 1991, "Facets of systems science ," *Facets of Systems Science*, G. J. Klir, ed., Springer, Boston, MA, pp. 405–417.

[3] Chiang, W., and Jin, Y., 2012, "Design of Cellular Self-Organizing Systems," ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Chicago, IL, Aug. 12–15, American Society of Mechanical Engineers, Vol. 45028, pp. 511–521.

[4] Humann, J., Khani, N., and Jin, Y., 2014, "Evolutionary Computational Synthesis of Self-Organizing Systems," AI EDAM, **28**(3), pp. 259–275.

[5] Khani, N., Humann, J., and Jin, Y., 2016, "Effect of Social Structuring in Self-Organizing Systems," ASME J. Mech. Des., **138**(4), p. 041101.

[6] Khani, N., and Jin, Y., 2015, "Dynamic Structuring in Cellular Self-Organizing Systems," *Design Computing and Cognition'14*, Springer, Cham, pp. 3–20.

[7] Ji, H., and Jin, Y., 2018, "Modeling Trust in Self-Organizing Systems With Heterogeneity," ASME 2018 International Design Engineering Technical Conferences and Computers and Information in Engineering Conferences, Quebec City, Canada, Aug. 26–29.

[8] Chen, C., and Jin, Y., 2011, "A Behavior Based Approach to Cellular Self-Organizing Systems Design," ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Washington, DC, Aug. 28–31, Vol. 54860, pp. 95–107.

[9] Sutton, R. S., and Barto, A. G., 2018, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA.

[10] Rashid, T., Samvelyan, M., Schroeder, C., Farquhar, G., Foerster, J., and Whiteson, S., 2018, "Qmix: Monotonic Value Function Factorisation for Deep Multiagent Reinforcement Learning," International Conference on Machine Learning, Stockholm, Sweden, July 10–15, PMLR, pp. 4295–4304.

[11] Bar-Yam, Y., 2002, *General Features of Complex Systems. Encyclopedia of Life Support Systems (EOLSS)*, Vol. 1, UNESCO, EOLSS Publishers, Oxford, UK.

[12] Arroyo, M., Huisman, N., and Jensen, D. C., 2018, "Exploring Natural Strategies for Bio-Inspired Fault Adaptive Systems Design," ASME J. Mech. Des., **140**(9), p. 091101.

[13] Königseder, C., and Shea, K., 2016, "Comparing Strategies for Topologic and Parametric Rule Application in Automated Computational Design Synthesis," ASME J. Mech. Des., **138**(1), p. 011102.

[14] Meluso, J., and Austin-Breneman, J., 2018, "Gaming the System: An Agent-Based Model of Estimation Strategies and Their Effects on System Performance," ASME J. Mech. Des., **140**(12), p. 121101.

[15] McComb, C., Cagan, J., and Kotovsky, K., 2017, "Optimizing Design Teams Based on Problem Properties: Computational Team Simulations and an Applied Empirical Test," ASME J. Mech. Des., **139**(4), p. 041101.

[16] Min, G., Suh, E. S., and Hölttä-Otto, K., 2016, "System Architecture, Level of Decomposition, and Structural Complexity: Analysis and Observations," ASME J. Mech. Des., **138**(2), p. 021102.

[17] Ferguson, S. M., and Lewis, K., 2006, "Effective Development of Reconfigurable Systems Using Linear State-Feedback Control," AIAA J., **44**(4), pp. 868–878.

[18] Martin, M. V., and Ishii, K., 1997, "Design for Variety: Development of Complexity Indices and Design Charts," ASME Design Engineering Technical Conferences, DFM-4359, Sacramento, CA, Sept. 14–17.

[19] Werfel, J., 2012, "Collective Construction With Robot Swarms," *Morphogenetic Engineering*, Springer, Berlin, Heidelberg, pp. 115–140.

[20] Beckers, R., Holland, O. E., and Deneubourg, J. L., 2000, "Fom Local Actions to Global Tasks: Stigmergy and Collective Robotics," *Prerational Intelligence: Adaptive Behavior and Intelligent Systems Without Symbols and Logic, Volume 1, Volume 2 Prerational Intelligence: Interdisciplinary Perspectives on the Behavior of Natural and Artificial Systems, Volume 3*, Springer, Dordrecht, pp. 1008–1022.

[21] Dasgupta, P., 2008, "A Multiagent Swarming System for Distributed Automatic Target Recognition Using Unmanned Aerial Vehicles," IEEE Trans. Syst. Man Cybern. Part A Syst. Humans, **38**(3), pp. 549–563.

[22] Ruini, F., and Cangelosi, A., 2009, "Extending the Evolutionary Robotics Approach to Flying Machines: An Application to MAV Teams," Neural Networks, **22**(5–6), pp. 812–821.

[23] Lamont, G. B., Slear, J. N., and Melendez, K., 2007, "UAV Swarm Mission Planning and Routing Using Multi-Objective Evolutionary Algorithms," 2007 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making, Honolulu, HI, Apr. 1–5, IEEE, pp. 10–20.

[24] Wei, Y., Madey, G. R., and Blake, M. B., 2013, "Agent-Based Simulation for UAV Swarm Mission Planning and Execution," Proceedings of the Agent-Directed Simulation Symposium, Apr., pp. 1–8.

[25] Price, I. C., and Lamont, G. B., 2006, "GA Directed Self-Organized Search and Attack UAV Swarms," Proceedings of the 2006 Winter Simulation Conference, Monterey, CA, Dec. 3–6, IEEE, pp. 1307–1315.

[26] Busoniu, L., Babuska, R., and De Schutter, B., 2008, "A Comprehensive Survey of Multiagent Reinforcement Learning," IEEE Trans. Syst. Man Cybern. Part C Appl. Rev., **38**(2), pp. 156–172.

[27] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R., 2017, "Multiagent Cooperation and Competition With Deep Reinforcement Learning," PLoS One, **12**(4), p. e0172395.

[28] Foerster, J., Farquhar, G., Afouras, T., Nardelli, N., and Whiteson, S., 2018, "Counterfactual Multiagent Policy Gradients," Thirty Second AAAI Conference on Artificial Intelligence, New Orleans, LA, Feb. 2–7, Vol. 32, No. 1.

[29] Peng, X. B., Berseth, G., Yin, K., and Van De Panne, M., 2017, "Deeploco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning," ACM Trans. Graph., **36**(4), pp. 1–13.

[30] Tan, M., 1993, "Multiagent Reinforcement Learning: Independent vs. Cooperative Agents," Tenth International Conference on Machine Learning, Amherst, MA, July 27–29, pp. 330–337.

[31] Watkins, C. J. C. H., 1989, "Learning From Delayed Rewards," Ph.D. dissertation, Cambridge University, Cambridge, UK.

[32] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., et al., 2015, "Human-level Control Through Deep Reinforcement Learning," Nature, **518**(7540), pp. 529–533.

[33] Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H., Kohli, P., and Whiteson, S., 2017, "Stabilising Experience Replay for Deep Multiagent Reinforcement Learning," International Conference on Machine Learning, Sydney, Australia, Aug. 6–11, PMLR, pp. 1146–1155.

[34] Hausknecht, M., and Stone, P., 2015, "Deep Recurrent Q-Learning for Partially Observable MDPs," arXiv preprint arXiv:1507.06527.

[35] Hochreiter, S., and Schmidhuber, J., 1997, "Long Short-Term Memory," Neural Comput., **9**(8), pp. 1735–1780.

[36] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y., 2014, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," arXiv preprint arXiv:1412.3555.

[37] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I., 2017, "Multiagent Actor-Critic for Mixed Cooperative-Competitive Environments," arXiv preprint arXiv:1706.02275.

[38] Brown, N., and Sandholm, T., 2019, "Superhuman AI for Multiplayer Poker," Science, **365**(6456), pp. 885–890.

[39] Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I., 2019, "Emergent Tool Use From Multiagent Autocurricula," arXiv preprint arXiv:1909.07528.

[40] Wu, S. A., Wang, R. E., Evans, J. A., Tenenbaum, J. B., Parkes, D. C., and Kleiman-Weiner, M., 2021, "Too Many Cooks: Bayesian Inference for Coordinating Multi-Agent Collaboration," Top. Cogn. Sci., **13**(2), pp. 414–432.

[41] Wang, Y., and De Silva, C. W., 2006, "Multi-Robot Box-Pushing: Single-Agent Q-Learning vs. Team Q-Learning," 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, Beijing, China, Oct. 9–15, IEEE, pp. 3694–3699.

[42] Rahimi, M., Gibb, S., Shen, Y., and La, H. M., 2018, "A Comparison of Various Approaches to Reinforcement Learning Algorithms for Multi-Robot Box Pushing," International Conference on Engineering Research and Applications, Dec., Springer, Cham, pp. 16–30.

[43] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M., 2013, "Playing Atari With Deep Reinforcement Learning," arXiv preprint arXiv:1312.5602.

[44] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N., 2016, "Dueling Network Architectures for Deep Reinforcement Learning," International Conference on Machine Learning, June, PMLR, pp. 1995–2003.

[45] Foerster, J. N., Assael, Y. M., de Freitas, N., and Whiteson, S., 2016, "Learning to Communicate to Solve Riddles With Deep Distributed Recurrent Q-Networks," arXiv preprint arXiv:1602.02672.

[46] LaValle, S. M., 2006, *Planning Algorithms*, Cambridge University Press, New York.

[47] Jones, C., and Mataric, M. J., 2003, "Adaptive Division of Labor in Large-Scale Minimalist Multi-Robot Systems," Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453), Las Vegas, NV, Oct. 27–31, IEEE, Vol. 2, pp. 1969–1974.

[48] Groß, R., Bonani, M., Mondada, F., and Dorigo, M., 2006, "Autonomous Self-Assembly in Swarm-Bots," IEEE Trans. Rob., **22**(6), pp. 1115–1130.

[49] Humann, J., Khani, N., and Jin, Y., 2016, "Adaptability Tradeoffs in the Design of Self-Organizing Systems," International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Charlotte, NC, Aug. 21–24, Vol. 50190, p. V007T06A016.

[50] Liu, X., and Jin, Y., 2018, "Design of Transfer Reinforcement Learning Mechanisms for Autonomous Collision Avoidance," International Conference on-Design Computing and Cognition, July, Springer, Cham, pp. 303–319.

[51] Ashby, W. R., 1961, *An Introduction to Cybernetics*, Chapman & Hall Ltd., London, UK.

[52] Makar, R., Mahadevan, S., and Ghavamzadeh, M., 2001, "Hierarchical Multiagent Reinforcement Learning," Fifth International Conference on Autonomous Agents, New York, NY, May 28–June 1, pp. 246–253.