Proceedings of the ASME 2019 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference
IDETC/CIE 2019
Aug 18-21, 2019, Hilton Anaheim, Anaheim, CA

IDETC2019-98268

DESIGNING SELF-ORGANIZING SYSTEMS WITH DEEP MULTI-AGENT REINFORCEMENT LEARNING

Hao Ji
IMPACT Laboratory
Dept. of Aerospace & Mechanical Engineering
University of Southern California
Los Angeles, California, 90089
haoji@usc.edu

Yan Jin*
IMPACT Laboratory
Dept. of Aerospace & Mechanical Engineering
University of Southern California
Los Angeles, California, 90089
yjin@usc.edu
(*corresponding author)

ABSTRACT

Self-organizing systems (SOS) are able to 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 in order to accomplish their overall 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 task environment changes, having *a priori* knowledge about these fields and the rules may not be feasible. In this paper, we propose a multi-agent reinforcement learning based model as a design approach to solving the rule generation problem with complex SOS tasks. A deep multi-agent reinforcement learning algorithm was devised as a mechanism to train SOS agents for acquisition of the task field and social rule knowledge, and the scalability property of this learning approach was investigated with respect to the changing team sizes and environmental noises. Through a set of simulation studies on a box-pushing problem, the results have shown that the SOS design based on deep multi-agent reinforcement learning can be generalizable with different individual settings when the training starts with larger number of agents, but if a SOS is trained with smaller team sizes, the learned neural network does not scale up to larger teams. Design of SOS with a deep reinforcement learning model should keep this in mind and training should be carried out with larger team sizes.

Keywords: deep Q-learning, complex system, self-organizing system, scalability, robustness

1 INTRODUCTION

Self-organizing systems can consist of simple agents that work cooperatively to achieve complex system level behaviors without requiring global guidance. 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 approach has many advantages, such as scalability, adaptability, and reliability [3,4]. Moreover, compared to 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]. A swarm of robots is an example of self-organizing systems. In such systems, robots usually have compact sizes, limited functionality and adopt simple rules of interaction. Such systems often consist of many homogenous robots [8]. The collaborative behavior of the swarm robots can emerge, and such emergent phenomenon has been applied to situations such as search and rescue, distributed sensing, unmanned aerial vehicle patrolling, traffic control, and box-pushing [5,9,10].

Various approaches have been proposed to support the design of SOS. The field-based behavior regulation (FBR) approach [11] 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 requires little knowledge to perform tasks since moving toward a higher or lower position 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.

To overcome the limit of the FBR approach, an evolutionary design method [4] and the social structuring approach [5] have been proposed to make design of SOS parametric and optimizable, and to allow a 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 more 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 highly complex and changeable.

One of the long-term goals of our research is to develop mechanisms for self-organizing robotic agents to autonomously carry out physical structure assembly, as in space structure construction, and disassembly, as in disaster rescue situations, without centralized control or external guidance. It is anticipated that the task situations for these task domains can become highly complex and unpredictable, making it a challenge to predefine task fields and social rules. Therefore, in this research, we take a *reinforcement learning* approach to capture the self-organizing knowledge for agent behavior regulation in SOS design. More specifically, a multiagent Q-learning algorithm is explored to address two research questions: *What are the factors that impact on the stability of learning dynamics in self-organizing systems*? *Will the knowledge captured from reinforcement learning be robust enough to be applied in a wide range of task situations*?

In the reinforcement learning 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 [12]. The problem of designing self-organizing systems comes down to training the system either as a team using a centralized single agent reinforcement learning approach or as separate individuals going through multi-agent reinforcement learning. Although the centralized learning of joint actions of agents as a team can solve coordination problems and avoid learning non-stationarity, it does not scale well as the joint action space grows exponentially with the number of agents [13]. Secondly, learning to differentiate joint actions can be highly difficult. Further, the neural networks obtained from the centralized learning are only applicable to the situations with the same number of agents as the trained cases, because the action space is fixed by the trained cases.

In contrast to the centralized single agent reinforcement learning, during the multi-agent reinforcement learning, each agent can be trained using its own independent neural network. Such approach solves the problem of curse of dimensionality of action space when applying single agent reinforcement learning to multi-agent settings. Although theoretical proof of convergence of multi-agent independent Q-learning is not mathematically given, there are numerous successful practices in real-world applications [13]. Thus, applying the state-of-the-art independent multi-agent reinforcement learning is a promising approach in tackling the existing problems faced by SOS design.

In the rest of this paper, we first review the relevant work in self-organizing systems and reinforcement learning in Section 2. After that, we present a multi-agent independent Q-learning framework as a complex system design approach in Section 3 and illustrate the system design implications. In Section 4, a box-pushing case study is introduced that applies our proposed Q-learning model. Section 5 provides a detailed analysis of the simulation results. Finally, in Section 6, the conclusions are drawn from the case study and future work directions are pointed out.

2 RELATED WORK

2.1 Artificial Self-organizing Systems

An artificial self-organizing system is a system that is designed by human and has emergent behavior and adaptability like nature [1]. Much research has been done regarding the design of an artificial self-organizing system. Werfel developed a system of homogenous robots to build a pre-determined shape using square bricks [14]. Beckers et al. introduced a robotic gathering task where robots have to patrol around a given area to collect pucks [15]. As robots prefer to drop pucks in high-density areas, the collective positive feedback loop contributes to a dense group of available pucks [2,15]. Khani et al 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 formation [16-19]. Chen and Jin used a field-based regulation (FBR) approach and guides self-organizing agents to perform complex tasks such as approaching long-distance targets while avoiding obstacles [11]. Price investigated into the use of genetic algorithm (GA) in optimizing Self-organizing multi-UAV swarm behavior. He tested the effectiveness of GA algorithm in both homogenous and heterogeneous UAV in accomplishing the 'destroying retaliating target' task [20]. The robotic implementations mentioned above have demonstrated the potentials of building self-organizing systems, and the design methods of self-organizing systems [5,6,11] have had their drawbacks as indicated in Section 1.

2.2 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning applies to multiagent settings and is based largely on the concept of single agent reinforcement learning such as Q-learning, policy gradient and actor-critic [12, 21]. Compared to single agent reinforcement learning, multi-agent learning is faced with the non-stationary learning environment due to the simultaneous learning of the multiple agents.

In the past several years, there has seen a move from tabular based methods to the deep reinforcement learning approach, resulting from the need to deal with the high-dimensionality of state and action spaces in multi-agent environments and to approximate state-action values [22-24]. Multi-agent systems can be classified into cooperative, competitive, and mixed

cooperative and competitive categories [22]. Cooperative agents receive the same rewards, competitive agents (often in two-agent settings) have the opposite rewards, and the mix cooperative and competitive settings assume agents are not only cooperating but also have individual preferences. In the SOS design, we focus on the cooperative agents since they share the same task goals.

One natural approach for multi-agent reinforcement learning is to optimize the policy or value functions of each individual. The most commonly used value-function based multi-agent learning is independent Q-learning [25]. It trains each individual's state-action values using Q-learning [25-26] and is served as a common benchmark in the literature. Tampuu [22] extended previous Q-learning to deep neural networks and applied DQN [27] to train two independent agents playing the game Pong. His simulation shows us how the cooperative and competitive phenomenon can emerge based on the individual's different reward schemes [22].

Foerster applied COMA framework to train multiple agents. He used a centralized critic to evaluate decentralized actors and estimated a counterfactual advantage function based on each agent and allocated credit among agents [23]. He trained multiple agents to learn to cooperatively play StarCraft games. In another work by Foerster, he analyzed his replay stabilization methods for independent Q-learning based on StarCraft combat scenarios [28].

As multi-agent environment is usually partially observable, Hausknecht & Stone [29] used deep recurrent networks such as LSTM [30] or GRU [31] to speed up learning when agents are learning over long time periods. Lowe et al developed Multi-agent Deep Deterministic Policy Gradient (MADDPG), which uses centralized training with decentralized execution [32]. They proposed an extension of the actor-critic policy gradient method and augmented critic with additional information about the policies of other agents and then tested their algorithm in predator-prey, cooperative navigation, and other environments. Their training algorithm shows good convergence properties [32]. Drogoul & Zucker developed a framework for multi-agent system design called 'Andromeda', which combines machine learning approach with agent oriented, role-based approach named 'Cassiopeia' [33,34]. The idea is to let learning occur within different layers of 'Cassiopeia' framework such as individual roles, relational roles and organizational roles so that the design of multi-agent system can be more systematic and modular [33,34]. However, as real multi-agent environment is rather complex, agent's actions are affected by not only its own roles but also by other agents and the group. Separating learning into different layers of abstraction may not be feasible.

Most approaches to multi-agent reinforcement learning focus on achieving optimal system reward or desirable convergence properties. Many training algorithms are based on fully observable states. Training of multi-agent reinforcement model is usually conducted on prespecified environments and the generalizability of the training network to various multi-agent team sizes is not analyzed or considered, which is an important factor of consideration in SOS design. It is crucial to develop a multi-agent learning framework that is scalable to various team sizes, 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 MULTI-AGENT REINFORCEMENT LEARNING MODEL

3.1 Single Agent Reinforcement Learning

It is important to discuss single agent reinforcement learning before moving into multiagent reinforcement learning because many concepts and algorithms of multi-agent reinforcement learning are based on the single agent reinforcement learning.

Single agent reinforcement learning 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 reinforcement learning, 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 could be taken by the agent. P is the transition matrix, which is usually not given/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 equation (1) below as an update. E represents the expected value. Eventually, such value iteration algorithm will converge to optimal value function.

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

Researchers in the past uses Q-learning as a common training algorithm in single agent reinforcement learning [35-36]. 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 [35-36]. 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. To overcome such problems in Q-learning, deep neural networks are introduced as functional approximator to replace the Q-table for estimating Q values. Such learning methods are called deep Q-learning [27]. A Q-network with weights θ_i can be trained by minimizing the loss function at each iteration i, illustrated in equation (2),

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

where

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

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

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

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

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 exploration of random actions and exploitation of best decision given current information. Experience replay, as Minh described in his paper [37], increases data sample efficiency and can break down the correlations between subsequent experiences and is used to stabilize training performance.

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 [38]. 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 Multi-Agent Reinforcement Learning

As mentioned above, there are generally two approaches in multi-agent 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 reinforcement learning, but can hardly scale up. To increase learning efficiency and maintain scalability, we adopt a multi-agent independent deep Q-learning approach. 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 $A = A_1 \times \bullet \quad \bullet \quad \bullet \times A_n$. 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 shared reward function from the environment. 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 multi-agent system can scale well to agent teams of different sizes.

In our multiagent reinforcement learning mechanism described above, each agent i ($i = 1, 2, \ldots, n$) engages in learning as if it is in the single agent reinforcement learning 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. Through this research, we aim to explore the stability or convergence issue of the learning process—i.e., whether the knowledge can be acquired in the form of neural networks through reinforcement learning, and the adaptability issue—i.e., whether the learned neural networks can be effectively applied to the situations of similar tasks but different agent team sizes.

3.3 Simulation based System Design

There have been several methods for guiding the design of self-organizing systems [5-6,39]. Based on the previous work [39], a simulation-based system design method is proposed as shown in Figure1. Like other design or system engineering methodologies [5-6,39], it starts with breaking down the tasks into subtasks, analyzing system constraints and then represents the state space of the agents. Functional design defines both individual and group level functions for agents to achieve. As the agent-level behavior is the focus of SOS design, the major design factors will include (1) the agent-level actions and (2) reward schema.

Agent's state, action and reward schema

In self-organizing systems, agents have only its local view of the environment, due to their limited sensor capability and motor constraints. An MDP with such property is called a *partially observable* MDP. An Agent's state representation of the design process should reflect such characteristics of self-organizing systems. For homogeneous self-organizing systems, where agents share the same functionality and capabilities, the agents share the same actions from which they can choose. However, at any given time, different agents may perform different actions, resulting in the combined impact on the overall transition of the state. For heterogeneous systems, on the other hand, agents may have individualized action sets to choose from. This research explores the homogeneous situations and the heterogeneous cases will be dealt with as future work.

Training of individual agents is based on how much reward each agent receives. Designing and allocating reward is very important in self-organizing system design and has been proven to be important in the past multi-agent deep Q-learning algorithms as well [22]. Good reward schema or functions can lead to optimal agent level and system level performance, whereas the bad reward structure leads to nonconvergent learning or undesirable performance.

Simulation/Optimization

Since the dynamics of a complex environment is hard to be modeled and captured analytically, simulation becomes an important step for social rule knowledge capturing. The simulation should be combined with optimization algorithms for searching the optimal policy and value functions given the agent's sensor information. Multi-agent deep Q-learning

networks can be integrated with simulation to perform such simulation optimization tasks. The hope is that the output of the trained neural networks can be applied to various team sizes for system implementation.

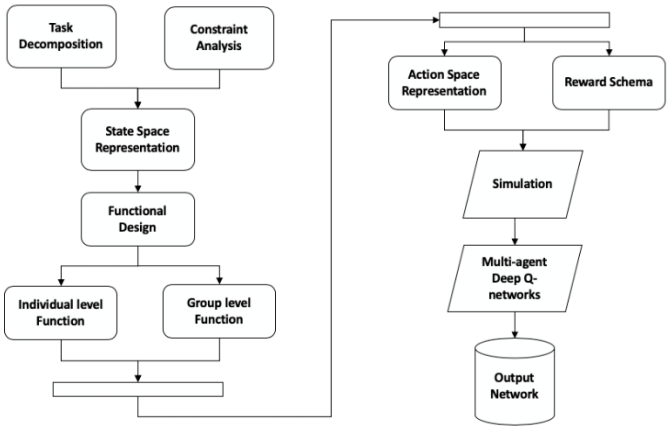

Figure 1. Steps for Simulation based Design of Self-Organizing Systems

4 CASE STUDY

To test the concepts and explore the multiagent reinforcement learning 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, we consider only the homogeneous cases, 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 [40]. Many topological and numerical solutions have been developed in the past [40]. In our paper, we adopt a self-organizing multi-agent deep Q-learning approach 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 towards a goal without system level global control.

In this research, the box-pushing case study was implemented in pygame, a multi-agent game package in the Python environment. In the box-pushing case study, we trained each individual with independent deep Q-learning (IQL) neural networks and tested successfully trained network parameters with various team sizes between 3-6 and analyzed its scalability characteristics.

A graphical illustration of the box-pushing case study is shown in Figure 2. 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 in order to push and rotate the box (the brown rectangle) towards the goal (the white dot with a "+" mark). As there is an obstacle (the red dot) on the path and walls (the white solid lines) along the side, the agents cannot just simply push the box but have to 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 horizontal x-coordinate of the goal, which is represented as a white vertical dotted line, the simulation is deemed success.

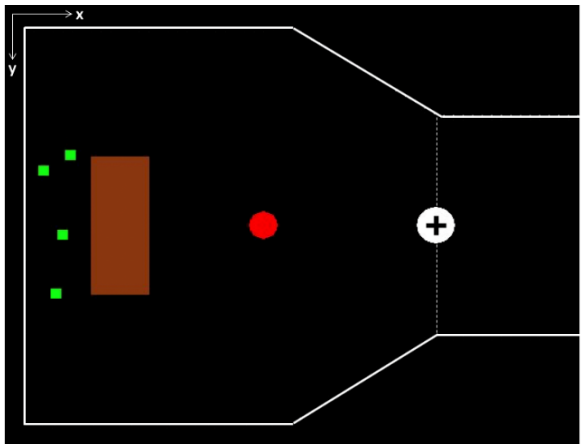

Figure 2. Graphical illustration of Box-pushing task

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 walls and obstacles.

T1 = <Move><Box> to <Goal>
T2 = <Rotate><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. As an example, the box in our simulation is 60 pixels wide and 150 pixels long.

In box-pushing, agents have *limited sensing and communication capabilities*. They can receive information from the sensor of the box, which measures orientation of the box and senses obstacles at a range of distance. They have limited storage of observation information: trained neural network parameters and 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 [41] and are reasonable with the current applications of physical robot hardware [42].

4.2 State and Action

Based on the task decomposition and constraint analysis mentioned above, the state space of the box-pushing task is defined as shown in Figure 3. To gather relevant environment information, a sensor is deployed in the center of the box, which can sense nearby obstacles. The radius of sensor range is 150 pixels and the entire circular sensor coverage is split into 8 sectors of equal size with each sector corresponding to a bit representation of state information. For example in Figure 3, there are three red obstacles within the sensor detection range, and the corresponding state s_3, s_5, s_7 are having value 1, indicating the presence of the obstacle. If there is no obstacle in a sector, the sector's state value will be 0. Like the past literature, we assume sensors can also detect the orientation of the box from the box's x-axis with respect to the goal position, illustrated with angle θ [35-36]. This is a reasonable assumption based on real-world sensor capability. In Figure 3, the current angle θ is around 30 degrees. And such degree information can be shaped into the range of [-1,1] by applying equation (5). Angle θ' serves as the final input state s_9. This shaped angle method can facilitate deep Q-network training and is used commonly in practice [35-36].

$$\theta' = \frac{(\theta - 180)}{180} = -0.83 \quad (5)$$

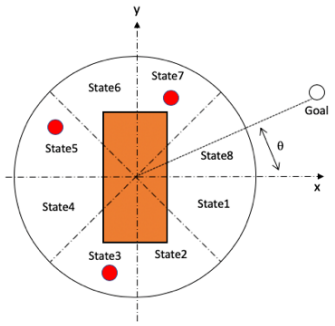

Figure 3. Box State Representation

Given the above, the state representation of Figure 3 can be expressed as a 9-digit tuple <0,0,1,0,1,0,1,0, -0.83>.

As during training, each agent is close to the vicinity of the box center, it can receive the sensor information broadcasted locally among agents. Sensor can also sense the distance from the center of the box to the goal area, 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.

Box neighborhood: The box neighborhood is defined as six regions [6,39], as shown in Figure 4. During each simulation, 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 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 [41].

Box dynamics: The box dynamics is based on a simplified physical model. The box movement depends on the simulated force and torque. Forces equal the sum of vector forces of each pushing agent. Every push carries the same amount of force, which acts from an agent towards the box. 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 equals to the sum of moment arm of all vector forces of the pushing agents. 2 pushes with a moment arm of 75 pixels each will rotate the box 20 degrees. 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.

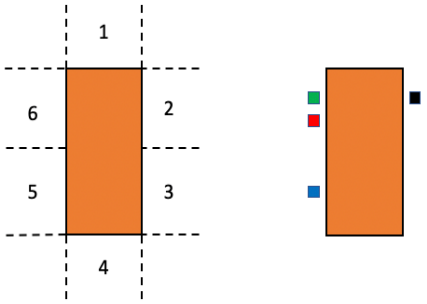

Figure 4. The six regions of box neighborhood

Agent action space: The *agent action space* is defined based on the box neighborhood and simulated box dynamics. As 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 Figure 4. For instance, if an agent chooses action a_1, it will move to box region "1" and push from there, the box will move downwards along the box's y-axis based on the simulated box dynamics and the same logic applies to other agent actions.

4.3 Reward Schema and Training model

In order 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 [35-36], we designed a new reward schema for agents' box-pushing training. The total reward is composed of four parts: *distance*, *rotation*, *collision*, and *goal*.

Distance Reward: The reward for pushing the box closer to the goal position is represented as R_{dis}, and is shown in equation (6). The previous distance D_{old} represents how far the center of the box is away from the goal position (measured in pixels) and can be evaluated by the box sensor and stored into an agent's memory. D_{new} represents the distance to the goal position from the center of the box 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 Equation (6) and draw its distance reward.

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

Rotation Reward: The reward for rotation R_{rot} is represented in equation (7).

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

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 degrees, this way, 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 towards the goal, which is the ultimate goal.

Collision Reward: The collision reward is analogous to the reward schema in common collision avoidance tasks [43] and is represented in equation (8) with R_{col},

$$R_{col} = \begin{cases} -900 \; if \; collision \; occurs \\ 0 \; if \; no \; collision \; occurs \end{cases} \quad (8)$$

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.

Goal Reward: The reward for reaching the goal R_{goal} is represented in equation (9),

$$R_{goal} = \begin{cases} 900 \; if \; reaching \; goal \\ 0 \; if \; not \; reaching \; goal \end{cases} \quad (9)$$

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.

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

$$R_{tot} = w_1 * R_{dis} + w_2 * R_{rot} + w_3 * R_{col} + w_4 * R_{goal} \quad (10)$$

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. The weights are chosen so that during each step in training: $w_1 = 0.6$, which means agents can have 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 multi-agent box-pushing [35-36].

$$w1 + w2 + w3 + w4 = 1 \quad (11)$$

During training, as the agents are homogenous and are cooperating to push the box, they should receive the same rewards. This is the reason why the reward equations (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 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 towards the goal position, which is the characteristics of multi-agent independent deep Q-leaning neural networks.

A shown in Figure 5, 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.

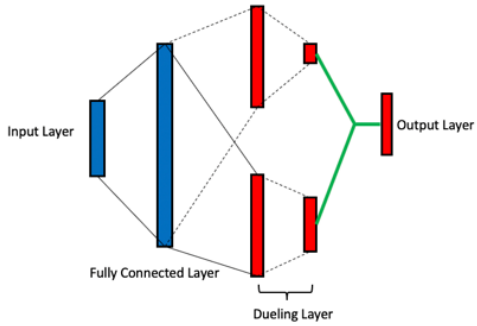

Figure 5. Training network illustration

4.4 Issues and Experiment Setup

As aforementioned, the two issues of this research are (1) *the stability of the learning dynamics* for agents to acquire knowledge for self-organizing behavior regulation, and (2) *the use of neural network knowledge* captured from the reinforcement learning in the extended situations such as varying team sizes and noisy environment.

We vary the number of agents during training and plot the cumulative reward each agent gets as the number of training episodes increases. Through the cumulative reward plot, we can see how fast the cumulative reward converges and whether the cumulative reward is stable or not. The experiment setup for the stability study is shown in Figure 6.

With the box-push case study, we change the number of agents from 3 to 6 and train each agent with independent deep Q-learning networks. Each simulation was run 10 times with different random seed. Then we tested our training neural network with various team sizes from 3-6 by feeding the trained neural network parameters to the new neural network and tested how scalable our trained neural network can be in various teams' sizes.

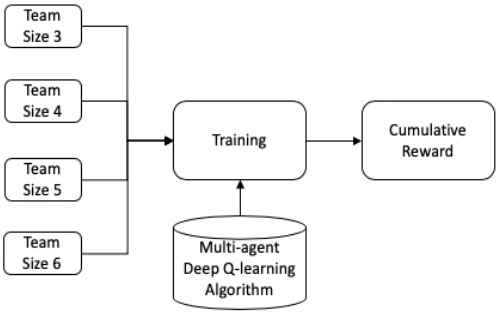

Figure 6. Experiment Setup for Stability of Learning Dynamics of Agents

During testing, we added 10%, 30%, 50% random individual actions. Such randomness does not follow maximum state-action value. This way, the robustness of our trained neural network can be evaluated with individual action noises. Figure 7 is a detailed illustration of the training experiment setup. Need to mention that when training with smaller team size, for instance 3, and

apply learned neural networks to larger team size, such as team size 4, we randomly choose one of the individual networks to the added new team members. When training with larger team size, for instance 5, and apply trained neural networks to a smaller size, such as size 3, we randomly choose three different individual neural network parameters and apply them to smaller teams.

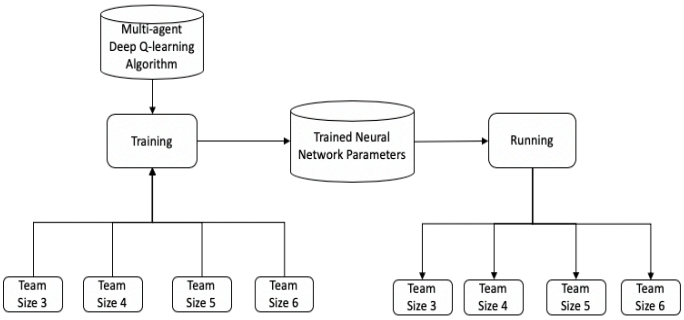

Figure 7. Simulation environment Setup

***Simulation Parameters*:** Table 1 summarizes the training parameters used in our simulation. Replay memory size is 2000, and we sample 32 mini-batches randomly from memory size during each training step. The discount factor is 0.99 and learning rate is 0.001. Total training episode is 5000 (one episode means one simulation run which starts from initial position to ending position). It is chosen because training cost decreases to 0 and stabilizes at this threshold. We follow ε-greedy action selection algorithm where ε is annealed from 1.0 to 0.005 during 100,000 annealing steps. After each annealing step, ε decreases a little bit until total annealing steps are reached and ε is 0.005. Agents choose actions from entirely random to nearly greedy. The target network is updated at every 5 simulation steps to stabilize training.

Table 1. Simulation parameters

Replay memory size	2000
Mini-batch size	32
Discount factor	0.99
Learning rate α	0.001
Total training episodes	5,000
ε	1.0 → 0.005
Annealing steps	100,000
Target network update frequency	5

5 RESULTS AND DISCUSSION

5.1 Typical Failure Modes

Figure 8 illustrates some typical failures with motion trace during training. Figures 8 (a) to (d) are the results of running the simulation cases of 6 pushing agents. During training, the box sometimes experiences excessive momentum, leading to moving towards 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, the episode terminates), as shown in (c). Boxes are occasionally pushed backward, forming a round circle around the obstacle, as shown in (d).

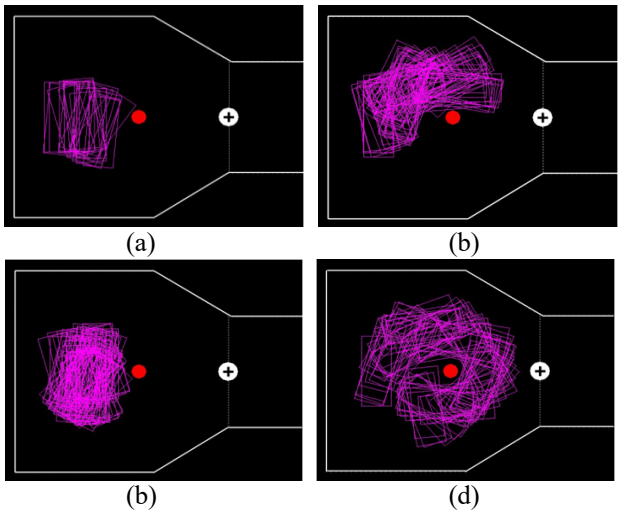

Figure 8. (a) failure with excessive pushing momentum (b) failure with insufficient/excessive rotation torque (c) failure with reaching maximum training episode length (d) failure with pushing box backward

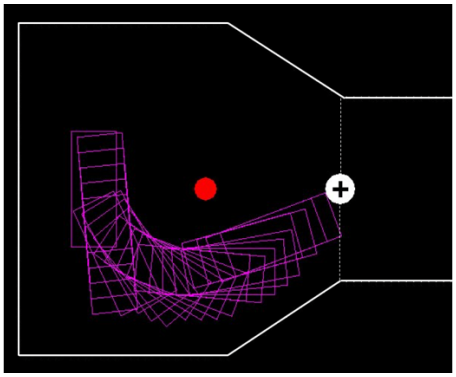

Figure 9 Successful Box-pushing trajectory with motion traces

5.2 Simulation Results

Box-pushing Trajectory

Figure 9 shows one successful training run with 4 agents by following max state-action values with motion trace examples. Although the final optimized trajectory is not strict perfect, through multi-agent deep Q-networks, agents can approximate its actions and push the box towards the goal area.

Training stability

Figure 10 shows the convergence results of our simulation with a different number of agents. The results are based on the running average of 10 training results with different random

seeds. Only the cumulative reward of the training instances where agents successfully pushed the box is plotted.

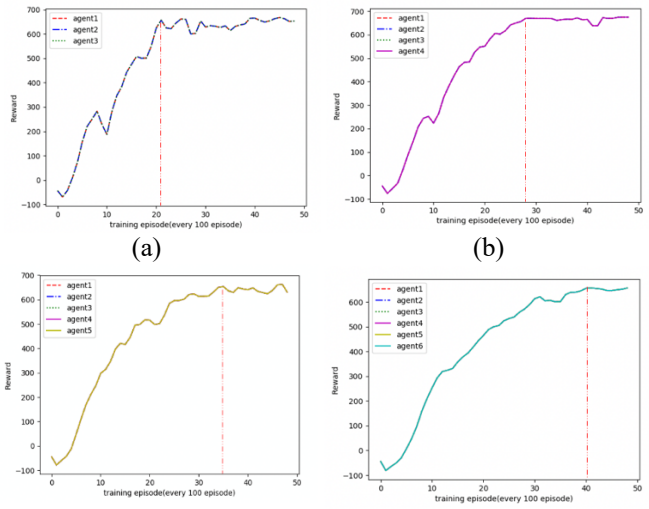

Figure 10. Cumulative Reward Plot vs Training Episode (every 100 episode) with varying team size (a) team size:3 (b) team size:4 (c) team size:5 (d) team size:6

As shown in Figure 10, in all training, the cumulative reward of each agent converges to close to 700 and once the cumulative reward reaches the threshold, it stays the same without much oscillation. Also, as agents number increases, it takes a longer time to reach the maximum cumulative reward. In Figure 10 (a), with team size 3, it takes about 2000 episodes for agents to get an average of around 700 rewards. As agent number increases, it takes about 2800 episodes for 4 agents and 3500 for 5 agents to reach the maximum cumulative reward. In subplot (d), with team size 6, we can see that it takes almost 4000 episodes for a large team to acquire a stable cumulative reward. The increase in convergence time might be explained by that when the team becomes larger, the learning dynamics of the environment are more complex: agents learn not only the physical environment but also other agents' dynamics. Thus, a larger number of agents require more training time to reach stable convergence.

Distance traveled by individual agents

We measure the distance traveled by each agent in different team size settings, shown in Figure 11. The vertical total distance plot is the average of 100 training episodes. As there are a total of 5000 episodes, we plot 50 distance values during training based on the trained neural networks. Here, the distance is based on a hierarchical measurement [44], which means the top-level distance is measured. For instance, when agents move from box region 1 to box region 2, as shown in Figure 4, it is considered as one step-wise 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.

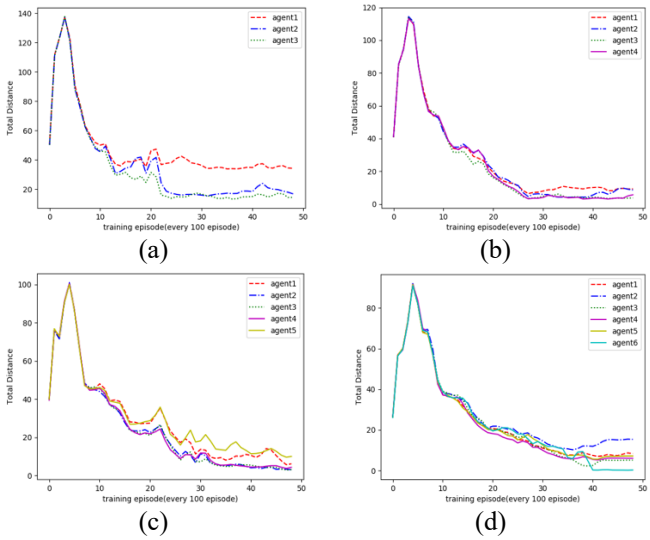

Figure 11. (a) Total distance traveled by each agent with team size 3 (b) Total distance traveled by each agent with team size 4 (c) Total distance traveled by each agent with team size 5 (d) Total distance traveled by each agent with team size 6

During the initial phase of training, the distance traveled by all agents tends to increase. After some training episodes, it starts to decrease. 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 experience, and as the epsilon value decreases, agents choose more greedy actions, their travel distance tends to decrease.

After 5000 episodes of training, the distance traveled by all agents tends to converge to low distance values. The final optimal distance by each individual agent is different as they finally tend to specialize in their own individual behavior to collaboratively push the box to the goal position.

We found that as agent team size increases, the maximum travel distance by each individual agent decreases. This has some implications in real robots learning: if the maximum energy consumption is the limiting factor for each robot, robot training should start with a larger team size to avoid failure. We also found that initially the distance traveled by agents are approximately the same and total distance traveled by each agent tend to diverge when it reaches a certain threshold, around 1000 episode. We speculate that the real speed-up in learning occurs after this threshold as the percentage of random actions drops below a certain value and at the same time agents have gathered enough training experiences.

Scalability of training algorithm to various team sizes

Two important issues of SOS are the scalability to different team sizes and the robustness to various noises. We stored our trained neural networks of different team sizes and applied them to other team sizes to assess their scalability. Further, we introduced a certain percentage of random agent actions ranging from 10%, 30% and 50%, to evaluate the robustness of the trained networks. The results are shown in Table 2-4. Training in

the table means how many agents are trained in the simulation, running means transferring the trained neural network to various other team sizes. When the box is pushed successfully to the goal without hitting obstacle/wall, such running episode is considered ***success***. We run each 10 trained neural networks with a specific team size to various other team sizes. Each simulation is run 1000 times and we average the trained results.

Interesting trends can be seen through all experiments. As shown in Table 2, the diagonal cell represents how the trained network performs with the same team size with 10% random actions. As such trained network is applied to the same team size, agents can successfully push the box towards the goal with ~90% of runs. If we transfer the individual learned neural network to teams of smaller size, the success rate remains *high*, although training-4 to running-3 has a *medium* success rate.

Additionally, we found that when we train teams of smaller size and apply the trained network to larger team sizes, success rate is very low as shown in Table 2. Except for the transferring learning from 3 to 4 agents, where 5.7% success rate is reached, all other test cases get around 0% success rate. This means training multi-agent system is very good at scaling downwards, but not scaling upwards. Such phenomenon can be called 'lower triangle phenomenon'. Table 3 and Table 4 are based on 30% and 50% random actions. The total percentage success rate drops with the increase of noise, as expected.

What is also intriguing is that sometimes when applying the trained network to smaller team sizes, the success rate can be even higher than when applied to its own team sizes. For instance, in Table 2, when we train 6 agents and run it with the same team size, we get 94.6% success rate, but applying such trained network to smaller team sizes such as 3-5 agents, the success rate is higher. This might be because 6 agents' case is a more difficult team setting to learn. While agents learn from more complex situations and apply to simpler settings, they can achieve better success rates. However, this situation does not work for all experiments. For example, in Table 2, if we train agents with team size 4 and apply to a team of size 3, the success drops, so when training with relatively small team size, the knowledge captured might not be transferred well to a smaller size. Only when team sizes reach beyond a threshold, such trained knowledge becomes more transferable to smaller teams.

Table 2. Success Rate with 10% individual random actions

Training \ Running	3 Agents	4 Agents	5 Agents	6 Agents
3 Agents	94.2%	5.7%	0.0%	0.1%
4 Agents	65.0%	99.9%	0.0%	0.1%
5 Agents	80.6%	90.6%	88.8%	0.0%
6 Agents	99.2%	97.9%	96.0%	94.6%

Table 3. Success Rate with 30% individual random actions

Training \ Running	3 Agnets	4 Agents	5 Agents	6 Agents
3 Agents	79.1%	3.3%	0.0%	0.3%
4 Agents	69.0%	95.4%	0.0%	0.2%
5 Agents	78.4%	80.8%	80.6%	0.0%
6 Agnets	89.8%	89.0%	86.3%	80.9%

Table 4. Success Rate with 50% individual random actions

Training \ Running	3 Agents	4 Agents	5 Agents	6 Agents
3 Agents	56.3%	1.2%	0.0%	0.0%
4 Agents	56.6%	80.8%	0.0%	0.0%
5 Agents	69.2%	61.1%	64.9%	0.0%
6 Agents	67.0%	70.0%	68.5%	59.7%

Summary

The following summaries are drawn from the case study results.

- Multiagent reinforcement learning (MARL) is an effective approach to capture self-organizing knowledge through extensive training. In our box-pushing case, the agents successfully accomplished the task with very high success rates based on the learned neural networks.
- The MARL learning process has relatively good stability with a limited number of agents. In our simulation results, the agents up to 6 have shown very stable learning convergence without specific issues. 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. This is a very interesting feature, and we plan to explore more to understand how this feature may be applied to evolve complex mechanisms to deal with highly complex tasks.
- The MARL based self-organizing knowledge capture in the form of neural networks can scale downward but not upward in terms of team size. When the range of scaling down is large, e.g., from 6 to 4 or 5 to 3, the scaling loss is ignorable. For small starting team size, the scaling down loss can be up to 40%. Scaling upward in any case is not advisable.
- The influence of random action noise only has a limited effect until 10%. After that, the impact is more noticeable. The teams with more agents are more robust against random actions. For a given task, there appears to be a specific team size, 4 for the box-pushing task, that is most robust against random actions.

It is worth mentioning that the above conclusions are limited to the experiments reported in this paper. Further investigations are needed to generalize the claims.

6 CONCLUSIONS AND FUTURE WORK

In multiagent reinforcement learning based SOS design, learning stability and scalability of the system with various team sizes are two 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, we applied the multi-agent independent Q-leaning algorithms in the design of SOS and investigated the learning stability, scalability, and robustness characteristics of such design approach through a box-pushing case study. Results show that multi-agent reinforcement learning can be a useful tool in design of SOS and can achieve good learning stability. Also,

training of the agents should occur with team sizes at least as large as the intended end-use case.

Future work includes testing the scalability of multi-agent Q-learning algorithm with a wider range of complex tasks and team sizes. Also, adding individual heterogeneous reward and testing the different weights for reward functions will also be the focus of research for the next step.

7 ACKNOWLEDGMENTS

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

REFERENCES

[1] Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. ACM SIGGRAPH computer graphics, 21(4), 25-34.

[2] Ashby, W. R. (1991). Requisite variety and its implications for the control of complex systems. In Facets of systems science (pp. 405-417). Springer US.

[3] Chiang, Winston, and Yan Jin. "Design of Cellular Self-Organizing Systems." IDETC/CIE. 2012.

[4] Humann, J., Khani, N., & Jin, Y. (2014). Evolutionary computational synthesis of self-organizing systems. AI EDAM, 28(3), 259-275.

[5] Khani, N., Humann, J., & Jin, Y. (2016). Effect of Social Structuring in Self-Organizing Systems. Journal of Mechanical Design, 138(4), 041101.

[6] Khani, N., & Jin, Y. (2015). Dynamic structuring in cellular self-organizing systems. In Design Computing and Cognition'14(pp. 3-20). Springer, Cham.

[7] Ji, Hao, and Yan Jin. "Modeling Trust in Self-Organizing Systems With Heterogeneity." ASME 2018 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. American Society of Mechanical Engineers, 2018.

[8] Kennedy, J. (2006). Swarm intelligence. In Handbook of nature-inspired and innovative computing (pp. 187-219). Springer US.

[9] Pippin, C., & Christensen, H. (2014, May). Trust modeling in multi-robot patrolling. In Robotics and Automation (ICRA), 2014 IEEE International Conference on (pp. 59-66). IEEE.

[10] Pippin, Charles Everett. Trust and reputation for formation and evolution of multi-robot teams. Diss. Georgia Institute of Technology, 2013.

[11] Chen, C., & Jin, Y. (2011). A behavior based approach to cellular self-organizing systems design. ASME Paper No. DETC2011-48833.

[12] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[13] Rashid, Tabish, et al. "QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning." arXiv preprint arXiv:1803.11485 (2018).

[14] Werfel, J. (2012). Collective construction with robot swarms. In Morphogenetic Engineering (pp. 115-140). Springer Berlin Heidelberg.

[15] Beckers, R., Holland, O. E., & Deneubourg, J. L. (1994, July). From local actions to global tasks: Stigmergy and collective robotics. In Artificial life IV (Vol. 181, p. 189).

[16] Dasgupta, P. (2008). A multiagent swarming system for distributed automatic target recognition using unmanned aerial vehicles. IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, 38(3), 549-563.

[17] Ruini, F., & Cangelosi, A. (2009). Extending the Evolutionary Robotics approach to flying machines: An application to MAV teams. Neural Networks, 22(5), 812-821.

[18] Lamont, G. B., Slear, J. N., & Melendez, K. (2007, April). UAV swarm mission planning and routing using multi-objective evolutionary algorithms. In Computational Intelligence in Multicriteria Decision Making, IEEE Symposium on (pp. 10-20). IEEE.

[19] Wei, Y., Madey, G. R., & Blake, M. B. (2013, April). Agent-based simulation for UAV swarm mission planning and execution. In Proceedings of the Agent-Directed Simulation Symposium (p. 2). Society for Computer Simulation International.

[20] Price, I. C., & Lamont, G. B. (2006, December). GA directed self-organized search and attack UAV swarms. In Proceedings of the 38th conference on Winter simulation (pp. 1307-1315). Winter Simulation Conference.

[21] Bu, Lucian, Robert Babu, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning." IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 38.2 (2008): 156-172.

[22] Tampuu, Ardi, et al. "Multiagent cooperation and competition with deep reinforcement learning." PloS one 12.4 (2017): e0172395.

[23] Foerster, Jakob N., et al. "Counterfactual multi-agent policy gradients." Thirty-Second AAAI Conference on Artificial Intelligence. 2018.

[24] Peng, Xue Bin, et al. "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning." ACM Transactions on Graphics (TOG) 36.4 (2017): 41.

[25] Tan, Ming. "Multi-agent reinforcement learning: Independent vs. cooperative agents." Proceedings of the tenth international conference on machine learning. 1993.

[26] Watkins, Christopher John Cornish Hellaby. Learning from delayed rewards. Diss. King's College, Cambridge, 1989.

[27] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[28] Foerster, Jakob, et al. "Stabilising experience replay for deep multi-agent reinforcement learning." Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

[29] Hausknecht, Matthew, and Peter Stone. "Deep recurrent q-learning for partially observable mdps." 2015 AAAI Fall Symposium Series. 2015.

[30] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[31] Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014).

[32] Lowe, Ryan, et al. "Multi-agent actor-critic for mixed cooperative-competitive environments." Advances in Neural Information Processing Systems. 2017.

[33] Drogoul, Alexis, and Jean-Daniel Zucker. "Methodological issues for designing multi-agent systems with machine learning techniques: Capitalizing experiences from the robocup challenge." Rapport technique LIP6 41 (1998).

[34] Collinot, Anne, and Alexis Drogoul. "Using the cassiopeia method to design a robot soccer team." Applied Artificial Intelligence 12.2-3 (1998): 127-147.

[35] Wang, Ying, and Clarence W. De Silva. "Multi-robot box-pushing: Single-agent q-learning vs. team q-learning." 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2006.

[36] Rahimi, Mehdi, et al. "A Comparison of Various Approaches to Reinforcement Learning Algorithms for Multi-robot Box Pushing." International Conference on Engineering Research and Applications. Springer, Cham, 2018.
[37] Mnih, Volodymyr, et al. "Playing Atari with deep reinforcement learning." *arXiv preprint arXiv: 1312.5602* (2013)
[38] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXivpreprint arXiv:1511.06581 (2015).
[39] Humann, J., Khani, N., & Jin, Y. (2016). Adaptability Tradeoffs in the Design of Self-Organizing Systems. In ASME 2016 IDETC (pp.V007T06A016). American Society of Mechanical Engineers
[40] LaValle, Steven M. Planning algorithms. Cambridge University Press, 2006.
[41] Jones, Chris, and Maja J. Mataric. "Adaptive division of labor in large-scale minimalist multi-robot systems." Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on. Vol. 2. IEEE, 2003.
[42] Groß, Roderich, et al. "Autonomous self-assembly in swarm-bots." IEEE transactions on robotics 22.6 (2006): 1115-1130.
[43] Liu, Xiongqing, and Yan Jin. "Design of Transfer Reinforcement Learning Mechanisms for Autonomous Collision Avoidance." International Conference on-Design Computing and Cognition. Springer, Cham, 2018.
[44] Makar, Rajbala, Sridhar Mahadevan, and Mohammad Ghavamzadeh. "Hierarchical multi-agent reinforcement learning." Proceedings of the fifth international conference on Autonomous agents. ACM, 2001.