Learning Coordination in Adversarial Multi-Agent DQN with dec-POMDPs

Elhadji Amadou Oury Diallo and Toshiharu Sugawara
Department of Computer Science and Communications Engineering
Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
diallo.oury@fuji.waseda.jp, sugawara@waseda.jp

Abstract

We examine whether a team of agents can learn geometric and strategic group formations by using deep reinforcement learning in adversarial multi-agent systems. This is a significant point underlying the control and coordination of multiple autonomous and intelligent agents. While there are many possible approaches to solve this problem, we are interested in fully end-to-end learning method where agents do not have any prior knowledge of the environment and its dynamics. We propose a scalable and distributed DQN framework to train adversarial multi-agent systems. We show that a large number of agents can learn to cooperatively move, attack and defend themselves in various geometric formations and battle tactics like encirclement, guerrilla warfare, frontal attack, flanking maneuver, and so on. We finally showed that by using only local views from the environment, agents create an emergent and collective flocking behaviors.

1 Introduction

Multi-agent systems (MAS) [26] are systems that contain many agents in the same environment. Although they can be used to collectively solve a problem that is too complex and costly to solve by a single agent [2, 3, 17]. We can often learn some interesting coordinated behaviors from nature and society such as flocking [19, 16, 5, 10] and group formation [4, 6, 7]. For instance, in an adversarial environment, agents learn how to minimize their encounters with opponents by strategically forming a group and by combining their knowledge of the environment to maximize the probability of avoiding opponents and obstacles.

In this paper, we address how deep reinforcement learning agents find strategic group formation by combining the local views of individual agents in an adversarial environment. When we apply reinforcement learning techniques in MAS, it is still unknown what will occur; for example, whether their learning converges and becomes stable and they can learn useful policies that facilitate achieving their goal. We also explore how agents adapt to the changes of opponent strategies.

2 Problem and proposed method

2.1 Adversarial MAS environment

A decentralized partially observable Markov decision process (Dec-POMDP) [11, 18] is defined as a tuple \(\langle D, S, A, T, R, O, O, h, I\rangle\). \(D = \{1, ..., n\}\) is a set of \(n\) agents. \(S\) is a finite set of states \(s\) in which the environment can be. \(A\) is the finite set of joint actions of agents, \(A = A_1^{\top} \times ... \times A_n^{\top}\). \(T\) is a probabilistic transition function and \(R\) is the immediate reward function. \(O\) is the finite set of joint observations with \(O = O_1^{\top} \times ... \times O_n^{\top}\). \(O\) is the observation probability function, \(h\) is the horizon of the problem, and \(I\) is the initial state distribution at time \(t = 0\). When a joint action \(a_t = (a_1^t, ..., a_n^t)\)
is executed, the environment transits from $s_t$ to $s_{t+1}$ with a probability $p(s_{t+1}|s_t, a_t) \in T$ and agent $i$ receives a reward $r^i_t = R(s_{t+1}|s_t, a^i_t)$. The goal of the agents is to find a deterministic joint policy $\pi = (\pi^1, \ldots, \pi^n)$ so as to maximize the sum of their individual reward $r_t = \sum_{i=1}^n r^i_t$.

In this paper, we used a well-known adversarial multi-agent domain, the so-called Battle game [21], in which two teams of many agents are fighting against each other. Agents of the same team cooperate with their teammates to find some strategies in order to defeat the opponent team. The main goal is to kill as many opponents as possible by invading the neighbor territories and using some warfare strategies. Our environment operates on two teams of $n$ agents and runs from some initial positioning of the agents inside a square in two-dimensional space for each team. An agent can take up to 7 actions: going up, going down, going left, going right, turning left, turning right and shooting. An agent is dead after being attacked three times.

### 2.2 Proposed learning framework

We proposed a distributed and parallel training framework (Fig. 1) for adversarial multi-agent systems with a large number of homogeneous agents. Each agent has its own local view $o^i_t$, its own action space $a^i_t$ and its own reward $r^i_t$. At every time step $t$, an individual agent observes its local environment, takes an action, and receives a local scalar reward. There is no need for communication between agents of the same team.

The neural net of each team receives a tensorized observation at times $t$ and $t-1$, and the actions taken at time $t-1$. Each agent senses distances to neighboring agents within a radius $k$ from its current position. The observation is of shape $(n, 2k + 1, 2k + 1)$ where $n$ is the number of agents. Agents asynchronously infer their actions from the output of the network which contains the joint Q-values of every possible action for each team. Also, agents simultaneously take a joint action $a^i_t$ before the network received its next observation $o_{t+1}$.

### 3 Results

Both teams learn the same strategies, can kill approximately the same number of opponents, and rapidly co-adapt themselves based on the opponents’ behaviors by finding some good defense strategies. Note that all networks have approximately the same values as the number of agents increases. The details are described in Appendix B. We can conclude that our proposed learning framework is scalable in non-stationary and adversarial learning environments with a large number of agents.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Random Team 1</th>
<th>Random Team 2</th>
<th>DQN Team 1</th>
<th>DQN Team 2</th>
<th>DDQN Team 1</th>
<th>DDQN Team 2</th>
<th>DuelDDQN Team 1</th>
<th>DuelDDQN Team 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-4034 ± 13</td>
<td>-4027 ± 12</td>
<td>281 ± 42</td>
<td>360 ± 60</td>
<td>393 ± 51</td>
<td>335 ± 39</td>
<td>223 ± 128</td>
<td>332 ± 114</td>
</tr>
<tr>
<td>200</td>
<td>-7990 ± 24</td>
<td>-7987 ± 21</td>
<td>649 ± 145</td>
<td>506 ± 121</td>
<td>716 ± 85</td>
<td>658 ± 92</td>
<td>498 ± 99</td>
<td>607 ± 102</td>
</tr>
<tr>
<td>300</td>
<td>-11792 ± 27</td>
<td>-11785 ± 27</td>
<td>842 ± 140</td>
<td>824 ± 167</td>
<td>1010 ± 88</td>
<td>1106 ± 164</td>
<td>915 ± 123</td>
<td>914 ± 87</td>
</tr>
<tr>
<td>400</td>
<td>-16274 ± 39</td>
<td>-16267 ± 33</td>
<td>1318 ± 140</td>
<td>1412 ± 175</td>
<td>1514 ± 111</td>
<td>1643 ± 138</td>
<td>1440 ± 113</td>
<td>1450 ± 141</td>
</tr>
</tbody>
</table>

Table 1: Average reward and number of alive agents at the end of each episode during the test.
APPENDIX A: METHODS

Reward scheme

Each agent receives a reward of $-0.01$ after each step. An agent gets $+5$ when it kills an opponent, and receives $-5$ when the agent is killed by an opponent. An agent receives a small negative reward of $-0.01$ whenever it attacks an empty position. This reduces the number of attempts an agent can attack empty positions, and thus making the training slightly faster. By reward shaping [13 20], an agent receives $+1$ for attacking an opponent, and $-1$ if it is being attacked. The global reward of each team is the sum of all the rewards received by agents of the same team at any time step $t$, $r_t = \sum_{i=1}^{n} r_i^t$.

Exploration vs exploitation

Exploration vs. exploitation is one of the main dilemmas in RL. The optimal policy for an agent is to always select the action found to be most effective based on history (exploitation). On the other hand, to improve or learn a new policy, the agent must explore new state it has never observed before (exploration) by taking non-optimal actions. We use $\varepsilon$-greedy [22] to balance between exploration and exploitation with $0 \leq \varepsilon(t) \leq 1$. At every timestep, the agent acts randomly with probability $\varepsilon$ and acts according to current policy with probability $1 - \varepsilon(t)$. In practice, it is common to start with $\varepsilon = 1$ and to progressively decay $\varepsilon$ as the training is going until we reach the desired $\varepsilon$. In general, we could use a fixed $\varepsilon$ value. However, this is not always an optimal choice, because after many episodes of training, we would have a more accurate estimation of the policies and we could reduce the amount of exploration. The initial and final value of $\varepsilon$ are, respectively, $1$ and $0.01$. In this case, our agents will be able to extensively explore during the first episodes and use the optimal policy without much exploration in the end of the training as agents get more knowledge about the environment.

DQN

In RL, agent $i$ learns an optimal control policy $\pi_i$. At each step $t$, $i$ observes the current state $s_i (s_i = (o_i, o_{t-1}, a_{t}))$, chooses an action $a_i$ using $\pi_i$, receives a reward $r$. Then, the environment transits to a new state $s_{i+1}$. The goal of the agent is to maximize the cumulative discounted future reward at time $t_0$ as $R_t = \sum_{t=t_0}^{T} \gamma^{t-t_0}r_t$, where $T$ is the terminal time step and $\gamma \in [0, 1]$ is the discount factor that weights the importance of rewards. The action-value of a given policy $\pi$ at state $s$, where the utility is defined as the expected and discounted future reward, $Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a]$. The optimal $Q^*$ is defined as $Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$.

Q-learning [23] is an approach that iteratively estimate the Q-function using the Bellman optimality $Q^*(s, a) = \mathbb{E}[r + \gamma \max_a Q^*(s_{t+1}, a)|s, a]$. We can use a parameterized value function $Q(s, a; \theta)$ when the task is complex. Then, we update the parameters by using the following formula: $\theta_{t+1} = \theta_t + \alpha(y^Q_t(s_t, a_t; \theta_t) - \nabla_{\theta_t} Q(s_t, a_t; \theta_t))$, where $\alpha$ is a scalar step size and $y^Q_t$ is the target value function.

\begin{align*}
y^Q_t &= R_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) \quad (1a) \\
&= R_{t+1} + \gamma \max_a Q(s_{t+1}, \text{arg max}_a Q(s_{t+1}, a; \theta_t); \theta_t) \quad (1b)
\end{align*}

DQN [13 14] is an extension of Q-learning that uses a stack of convolutional neural network [8 11] and fully connected layers to estimate $Q(s, a; \theta)$, where $\theta$ are the weights of the network. We use a separate network to estimate the target Q-values, $y^Q_{\text{DQN}}$. The target network has the same architecture as the DQN network but with frozen parameters $\theta^\text{target}$ that are updated after every $\tau$ from the online network. This leads to more stable training because it keeps the target $\theta^\text{target}$ fixed for a while. Now, we can can rewrite Eq. II as:

\begin{align*}
y^Q_{\text{DQN}} &= R_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) \quad (2a) \\
&= R_{t+1} + \gamma Q(s_{t+1}, \text{arg max}_a Q(s_{t+1}, a; \theta^\text{target}; \theta^\text{target}); \theta^\text{target}) \quad (2b)
\end{align*}

And we finally update the neural network weights by:

\begin{align*}
\theta_{t+1} &= \theta_t + \alpha(y^Q_{\text{DQN}} - Q(s_t, a_t; \theta_t)) \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \quad (3)
\end{align*}
Double DQN

The traditional Q-learning algorithm overestimates the action-state values. This is due to the use of the max function when updating the Q-values. One solution is Double DQN that consists of using two independent value functions that are randomly updated, resulting in two sets of weights, \( \theta \) and \( \theta' \). Then we use one set of weights to determine the greedy policy and the other to determine its value. Given the fact that we already have two different value functions in DQN, \( y_{DDQN}^t \) becomes:

\[
y_{DDQN}^t = R_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t); \theta^\text{target}
\]

We stored the experience \((s, a, r, s_{t+1})\) in a replay buffer \(D\) and sampled mini-batches from it to train the network by using the standard squared error loss. In the beginning, the buffer is filled with experiences played by random agents.

\[
L(\theta_t) = E_{s,a,r,s_{t+1}} \sim D \left[ \left( y_{DDQN}^t - Q(s, a, \theta_t) \right)^2 \right]
\]

We might note that Double DQN is shown to sometimes underestimate rather than overestimate the expected Q-values. However, the chances of both estimators underestimate or overestimate at the same action is lesser.

Dueling Network Architectures

In many reinforcement learning problems, it is unnecessary to estimate the action value for each action as some actions have little to no value for a given state. Dueling networks decompose the Q-Network into a value stream \( V(s; \theta, \beta) \) and an advantage stream \( A(s; a; \theta, \alpha) \). The value stream expresses how good it is for an agent to be in any given state while the advantage stream compares how much better taking a certain action would be compared to the others. The relationship between value and advantage streams is expressed in the following formula:

\[
A(s; a) = Q(s, a) - V(s) .
\]

This decomposition helps to generalize the learning for the state values. The output layer of this architecture is a combination of the value stream output and the advantage streams output. The aggregator is given by

\[
Q(s, a; \theta, \zeta, \beta) = V(s; \theta, \beta) + \left( A(s, a; \theta, \zeta) - \frac{1}{|A|} \sum_{a} A(s, a; \theta, \zeta) \right),
\]

where \( \beta \) is a parameter of \( V \), \( \zeta \) is a parameter of \( A \), and \( |A| \) represents the length of the advantage stream.

APPENDIX B: RESULTS

Experimental settings

We use the following parameters for all our experiments: learning rate \( \alpha = 0.00025 \), discount factor \( \gamma = 0.99 \). We update the target network every 10,000 timesteps. To stabilize learning, we feed the network with medium size mini-batches of 250 samples. The DQN network architectures are shown in Fig. 2. The experimental results in the following sections describe the average values of twelve (12) experimental runs with different random seeds.

Detailed experimental results

Fig. 3 represents the training performance of different networks with different number of agents per team with a narrow local field of view \( (k = 3) \). After more or less than 3,000 training episodes, each network’s training loss converges to a very low loss value. The environment is non-stationary because both teams are simultaneously learning. Hence, the best way to know if teams are learning strategic formations is probably to check if both networks converge to approximately the same values. In this case, the difference between the two errors is negligible. By using double DQN,
we make sure that the networks do not overestimate the Q-values, and consequently, the action-state values will converge to the actual Q-values. The DQN networks have lower losses because they have less parameters to tune and the network structure is much simpler than the others. Dueling network architectures were useful in our environment due to the fact that the networks generate all possible actions per team. It makes sense to calculate to decompose the Q-values into a state value and an advantage stream because the action space is huge. Finally, we can see that dueling DDQN agents have the lowest discrepancy between the losses of the two teams.

Fig. 3 shows the average reward per episode for each team during test time. We see that both networks tend to have approximately the same reward. Our reward scheme could make a group of agents learn how to efficiently form various strategic group formations based on their opponents’ strategies. The cumulative reward for random agents is always negative because agents are not learning any useful strategies.
It seems like most of the following strategies shown in Fig. 5 always appeared in all experiments. As we train agents, we might observe slightly variant tactics which do have the same roots. For instance, it is not unusual to observe an improved attack and/or defense tactics appear again. First, the team with the largest number of agents at time $t$ uses some tactics to eliminate as many opponents as possible in a very short time before the opponents' counter-attack. Then, both teams reach a kind of equilibrium in which they use similar group formation strategies. And finally, agents periodically update their strategies to effectively defend themselves against the opponents. Agents often prefer short-term and aggressive strategies to long-term and safe ones.

See Fig. 6 for a density estimation of the number of visits by each team.
Conclusion

We demonstrated that agents keep forming groups whenever new ones are necessary. Also, the formations are dissolved whenever it is beneficial to do so. The networks have learned how to effectively position and move agents for the emergence of group formation during and after training. This proves that grouping provides greater protection against opponents. We confirmed that local behavior of an individual can conspire to determine very complex global behaviors of multi-agent systems.

While we believe that our results show how agents optimally choose their strategies to form alliances, we also recognize that further investigations are needed. These include investigating what will happen if we have obstacles inside the environment and if agents don’t have the same speed or field of views. Our near future work is to address the previous points.

Acknowledgments

This work is partly supported by JSPS KAKENHI Grant Number 17KT0044.

References


