Reinforced Inter-Agent Learning for Multi-Agent Particle Environment

Annalena Bebenroth
Cognitive Science
Universität Osnabrück
Osnabrück, Germany
abebenroth@uni-osnabrueck.de

Elif Kizilkaya
Department of Computer Engineering
Bogazici University
Istanbul, Turkey
elif.kizilkaya@boun.edu.tr

***Abstract*— We investigated the idea of reinforced inter-agent learning (RIAL) [1] for a multi-agent particle environment (MPE). RIAL introduces communication across agents via communication protocols that are learned while training and that have no impact on the environment. It combines the idea of communication with the usage of Deep Recurrent Q-Learning [2]. We implement RIAL for a simple task, with the intention to speed up the training and earn faster success, because we hypothesise that communicating while learning a task is an advantage that is shown in the original RIAL paper. The results show that we must have made an error in calculating and updating the models, because there is no obvious learning process. We have some suggestions were we made the mistake and analyse this in the end in more detail.**

I. INTRODUCTION

When people come together to play games either against each other or towards a shared goal, it will most likely not be quiet, except the game requires this. This assumption is based on the fact, that communication is a main part of the human behaviour, because "it is impossible to not communicate" [3].

When humans communicate while playing a game, why do not imitate this with artificial agents playing a game together? To achieve this, some form of communication has to be introduced in the training of the agents, to let them use this in a game. One way to do this is to let the agents learn communication protocols beside the actions. This idea was first introduced in [1] and the authors propose two algorithms, Reinforced Inter-Agent Learning (RIAL) and Differentiable Inter-Agent Learning (DIAL), for it.

Another problem that occurs in some multi-agent settings is, that the agents are not omniscient and observe only parts of the full state, so the observations differ between the agents. With partial observability comes the possible lack of knowledge about the history, because the markov property, which holds that the future states are only dependent on the current state, cannot be hold, when only parts of the whole state are observed [2].

When thinking of a cooperative game where a specific sequence of actions have to be taken in order to receive a reward, the history of what happened before is important to determine the best next action to take. To tackle this, [2] introduces the usage of recurrent layers like Long Short Term Memory (LSTM) layers in the model architecture of a deep reinforcement learning model like a Deep Q-Network, called Deep Recurrent Q-learning (DRQN) . RIAL adapts the idea of DRQN to a multi-agent setting and combines it with the learning of communication protocols.

To shortly summarise, RIAL is a deep reinforcement learning approach for multiple agents that have actions which have impact on the environment and messages send between them that have no impact and which underlying communication protocols have to be learnt from scratch. The agents only have access to their own observations and compensate this by recurrent neural networks for the action and message policies, that uses an internal state to keep information from further steps. While the original RIAL assigns each agent its own models for each action and message policy, an extension towards parameter sharing allows all agents to use the same model, when they receive an extra input indicating which agent called it. RIAL is trained like DQN, therefore for each action and message one agent makes in one step, the optimal Q-values are calculated by a target model and the policies are optimised toward the maximum discounted return.

The described problem of partial observability in multi-agent settings is also part and parcel of the paper [4]. Among their interesting solution in terms of an actor-critic model, the team from OpenAI introduces different variations of a Multi-Agent Particle Environment (MPE) [5].

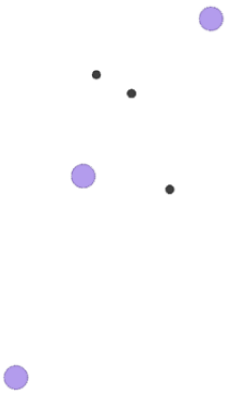


Fig. 1. An Example of how Simple Spread works for 3 agents (blue dots) and their goals, the black landmarks

One of the environments, called Simple Spread [6] consists

of a given number of agents and the same number of landmarks with the task for the agents to cover all landmarks 1. Simple Spread has a discrete action space, which only contains actions concerning movement and no given template for communication. This makes the environment suitable for RIAL and makes it the environmental setting for our implementation of RIAL.

II. Related Work

A. Deep Q-Learning

Deep Q -Learning is a method that takes Q-learning one step further and uses a neural network instead of a q-table. It tries to approximate the q-values for each action based on a state.

The RIAL technique we use for the Simple Spread environment makes use of Deep Q-Learning. [2] Although we have not come across a study using Deep Q-Learning in the Simple spread environment before, deep Q learning has been applied to other multi-agent environments.

One of them was applied to Cartpole-v1, LunarLander-v2 and Maze Traversal openAI Gym environments. [7] In the study, a simplified DQN was used to reduce complexity in multi-agent systems. In this study, where actions are agent-specific and shared state and rewards are used, a performance that exceeds the baseline has been observed. [7]

In another study, Q learning, N-DQN and DQN results were compared with each other in maze finding and ping-pong environments. [8]

B. Recurrent Neural Networks for Reinforcement Learning

As mentioned at the beginning, Deep Recurrent Q Learning (DRQN) comes into play in games where the agent needs to remember data from more distant past, unlike DQN. The agent is trained using the last 4 states in practice when using DQN. [3]

In a study conducted at Stanford University, DRQN which utilized recurrent neural network (RNN) and DQN performances were compared. It was seen that DRQN performed better in the study using the Q*bert game, which is difficult to learn for DQN. At the same time, adding attention to DRQN has been found to prevent performance in some games. [9]

The Recurrent Neural Network has also been used to estimate the number of Covid 19 patients. [10] The model developed using RNN, especially with Modified Long-Short Term Memory, was aimed to predict the number of newly infected patients, loss and cures in a few days. [10]

C. Communication in Multi-Agent Reinforcement Learning

Systems with multiple agents are more complex because the strategies followed by other agents may be different. Therefore, the uncertainty is high, and it becomes difficult for agents to work together. At the same time, learned policies may be overfitting to other agents. Establishing a communication channel between agents can offer a solution to this. [11] There are various approaches for this in multi-agent systems. DIAL [2], CommNet [12], BICNet [13], and Master-Slave [14] and ATOC [11] are examples.

Differentiable Inter-Agent Learning (DIAL) is a learnable communication model using backpropagation and deep Q network. In each step, the message produced by one agent is given to the other agent as an input. Gradients are transferred between agents via the communication channel. In this way, a more effective communication channel is developed and an end-to-end trainable across agents' system emerges. [2] [11]

CommNet is a large single feed forward neural network. This network maps the inputs and actions of the agents. Agents have a communication channel through which they can share information, and they send their hidden states to this channel as a message at each step. These messages are averaged and sent as input to next layer. [12]

BICNet is a predefined communication channel based on actor-critic, used in environments with continuous actions. [13]

In the master-slave model, centralized and decentralized perspectives are combined. The master represents centralized learning and organizes agents. The slave has a decentralized perspective and depends on information from the master. [14]

The master-slave technique is also a predefined communication model. Such models restrict communication and reduce collaboration between agents. [11]

ATOC is an attentional communication model proposed by the research done in 2018. In environments with too many agents, agents may not be able to distinguish which information is useful, which may harm collaboration. At the same time, these systems with too many agents have high cost and complexity. ATOC was created to deal with these problems. [11]

III. METHODS

In order to give a comprehensive walk through our implementation of RIAL for the Simple Spread environment, it is important to notice that some decisions are based on multiple factors and it was not easy to create a structure that does not require all the details of the algorithm to explain beforehand, but to mention it side by side to our way to implement it.

Moreover we build our model with the keras API from tensorflow and use tensorflow for calculating the gradients.

A. Environment and Task

The agents in the Simple Spread environment [6] aim to cover all landmarks in the environment through moving left, right, up, down or nothing at all. Each agent chooses an action one after one before it is passed to the environment and a reward as well as a new state is given.

The rewards are not sparse and only if a goal is reached, but dependent on the distances from the agents to the landmarks. To be exact, the reward is the sum of the distances from the agents closest to each landmark and additionally they receive -1 for each collusion of agents.

For iterating over the agents and taking steps in the environment like it assumes it, the PettingZoo API offers the function $agent_iter()$ [7], which yields the current agent and

stops when all agents are done or the given maximum of steps is reached.

B. Data Storage and Processing

One main modification from DQN [17] to RIAL is the absence of experience replay. On the one hand this counteracts the dynamics of learning with multiple agents, so previous steps may not be helpful for further learning [1]. On the other hand, this leads to higher memory costs for training, because for each training step new samples have to be produced from the current models.
Due to the fact that the PettingZoo API only offers a function that iterates over the agents until an episode is finished, we decided to generate the amount of episodes needed for a training batch sequentially instead of parallel. Therefore we use a batch memory that stores the necessary information from every timestep for every generated episode. For one step of one agent this means storing the action taken, the message send, the state observed before and after executing the action, the reward received, the information if the agent is done, the index of the agent as well as the outcoming hidden states from the recurrent layers.
Usually in DQN [17] only one step is made, included in the memory and then a batch sampled from it is used for an update. We hypothesise that first generating whole episodes and then going through them for updating does not worse the training of the models, because of the usage of the recurrent layers. We will go into detail in the training part. Sampling one trajectory from the start to the agent's done or reaching the maximum amount of steps looks (shortened from the original code to cover only the case of sampling one episode for training) like:

Algorithm 1 Sample one trajectory

Reset environment and init score s and batch memory
Init features action a, message m, hidden states h_a, h_m
for agent in agent_iter() or while not done **do**
    Get agent index ind
    Get agent's observation o
    if not first step of episode **then**
        Get last a, m, h_a, h_m from memory
    end if
    Choose message $\hat{m} = \max_m Q_m(o, a, m, h_m, ind; \theta)$ and receive $\hat{h}_a$ from model
    Choose action $\hat{a} = \max_a Q_a(o, a, m, h_a, ind; \theta)$ and receive $\hat{h}_m$ from model
    Take $\hat{a}$ and get new $\hat{o}$, reward r and done d
    s += r
    Append $[o, \hat{a}, \hat{m}, r, \hat{o}, \hat{h}_a, \hat{h}_m, d, ind]$ to batch memory
end for

One important part is the case of the first step, when there is no history before. In this case we have to initialise the hidden states for each model and set the message and actions received to no action/message. The initialisation of the hidden states happens in the agent in the function to choose in an epsilon-greedy manner an action or message and looks like:

```
hidden = [self.action_model.rnn1.get_initial_state(
    batch_size=1, dtype=float).numpy(), self.
    action_model.rnn2.get_initial_state(batch_size
    =1, dtype=float).numpy()]
```

C. Forward Pass

In order to generate a whole episode the agents need to know how to choose their actions and messages. In the following we will explain the process for choosing an action, because it is the same procedure for the message. The agents choose their next action based on an ϵ-greedy policy [17], meaning that with a low probability ϵ the agents choose a random action from their action space to support exploration and otherwise take the action with the highest Q-value from the policy.
In contrast to DQN where the policy model only needs the state to calculate the q-values for all possible actions, in the case of a discrete action space, the combination of DRQN, multi-agent, communication and parameter sharing extend the necessary data needed to calculate all q-values. Additionally to the q-values, the model outputs the current hidden states from the action and message model, which have to be saved for further processing. To keep the idea of a memory we keep the hidden states in the memory, so we can easily access the information from last timestep. Another possibility to realise the storing of hidden state could be to keep it as variable for each agent, which would make more sense if we have not used parameter sharing, but individual models for each agent.

D. Model Architecture

Like mentioned before, the models for estimating the q-values for each action and message, have a lot of different input, which we want to explain deeper in addition to how it is processed in the models 2.
Though the important part of each model is the recurrent part, it needs a meaningful input that contains all necessary information. Therefore the different information is extracted from the input vector, given that the Tensorflow call function only can have one input, and are each processed different to have the same shape and to be summed together.
The observations should be processed by a problem-specific Multi-Layer Perceptron (MLP). We decided to use two Dense layers, that extend the observation shape of 18 sequentially to 64 and 128. The decision is based on the fact that the observation consists of information about the agent's velocity, position, related distances to the landmarks and related distances from the other agents [6], and we want the fully connected layer to keep all information and at the same time extend the dimension to the wanted size of 128.
Moreover we decided to use an embedding layer for processing each action and agent index, that are represented by one integer. While the paper [1] states that the single numbers are extended to a shape of 128 through lookup tables, we implemented this by an embedding layer, that represents the discrete variable as a continuous one with the desired dimension. The message, that is represented by

a discrete variable is processed through one Dense layer expanding the dimension to 128 and afterwards normalised throughout the batch. This is given by the paper and we adopt this, because the parts handling the messages are the ones important for introducing and learning communication. After

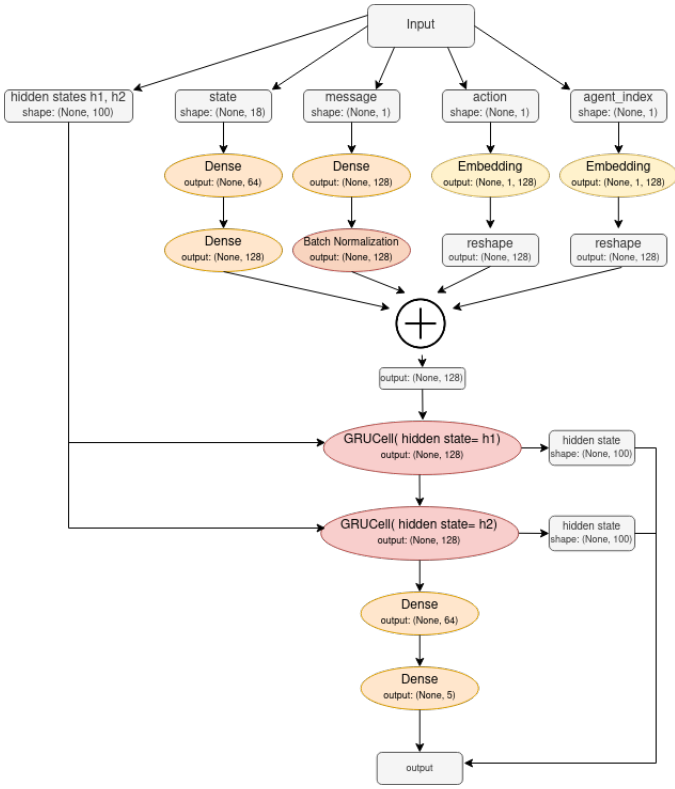


Fig. 2. Model Architecture of RIAL. The variables and their shapes are represented in grey, in contrast to keras layers that are colored in red, orange and yellow for a better overview. The graph flow shows one forward step of the model.

these preprocessing steps, the different parts all have the same last dimension size 128 and can be summed together element-wise to serve as input for the recurrent layers. More precisely the recurrent part of the model consists of two GRUCells [18], that are used instead of a RNN layer, because the cell processes only one timestep and outputs the new hidden state, whereas a RNN layer processes a sequence of timesteps in the keras API. Both the forward and the backward pass of the RIAL algorithm process only one timestep, else we always have to keep the whole history and with the cells, we only have to know the last hidden state that keeps all the information about the former steps. That's the reason why we output the two hidden states of both cells aside the q-values for either all actions or all messages.

E. Training

Algorithm 1 shows the pseudocode for our implementation of the training. Each step can be described in more detail, but first of all the variables used should be defined. K and N are the hyperparameters describing the amount of episodes to train and after how many steps the target models are updated. T is the length of an episode and t indicates the current timestep, and o indicates the observation, a the action, m the message, h the hidden state, ind the agent's index and θ the model's parameter, as well as Q for either the action or message model. The models are each updated individually, but after the same scheme, so it is shortened in the pseudocode.

Algorithm 2 RIAL

Init Neural Nets Q^a_θ, Q^m_θ, Q^a_{target}, Q^m_{target}
for K episodes **do**
  for N times **do**
    Sample batch of trajectories and store them in memory
    for each timestep, $t = T-1, T-2, \ldots, 1$ **do**
      for each action and message model **do**

$$y_t = \begin{cases} r_t, \text{ if } o_t \text{ terminal, else} \\ r_t + \gamma \max_a Q(o_{t+1}, a_t, m_t, h_t, ind; \theta) \end{cases}$$

$$\Delta Q_t = y_t - Q(o_t, a_{t-1}, m_{t-1}, h_{t-1}, ind; \theta)$$
$$\nabla\theta = \nabla\theta + \frac{\partial}{\partial\theta}(\Delta Q_t)^2$$
$$\theta = \theta + \alpha\nabla\theta$$

      end for
    end for
    Reset memory
    Q^a_{target}, $Q^m_{target} \leftarrow Q^a_\theta$, Q^m_θ
  end for
end for

All the necessary variables for one update step can be extracted from the memory by using list comprehension and converting it afterwards to the needed numpy array to get the wanted variable from the whole batch. This also allows us to easily get the actions, messages and hidden states from the last time step through choosing the $t-1$ time step of each batch:

```
import numpy as np
# assert a memory with shape [batch size, time
    steps, features]
# with features saved as list [state, action,
    message, reward, next_state, hidden_message,
    hidden_action, dones]
actions = np.asarray([[b[t][1]] for b in memory])
last_actions = np.asarray([[b[t-1][1]] for b in
    memory])
```

Furthermore the time steps are counted backwards going from the end to the beginning to simulate the unrolling of a recurrent neural net, because the models only inhibit recurrent cells that process one time step.
The action and message models are updated the same way with using tensorflow's *GradientTape()* instead of the more common way of using keras' *model.fit()*. We choose this way, because we do not calculate the loss between the outcomes of the policy model and the target model, but have to calculate the target from the outcome of the target model (line 7 in the code snippet below) and further need to filter the outcome of the policy model to the chosen actions from the target network (line 11 and 13). This filter is realised through first creating a mask of the same shape as the outcome of the policy model with *tensorflow.one_hot()* that has one's at the

index of the chosen actions for each batch sample.
Through simply multiplying this mask with the q-values for every possible action in every batch sample, so the outcome of the model, only the one's we want stay, while the others become zero. To reduce the dimensions to only the desired action for each batch sample, we can use *tensorflow.reduce_sum()* that sums up the values for each batch, which returns the desired q-value of one action, because the others are zero.
The loss is then given by the Mean Squared Error between the target and the estimated q-values and tensorflow offers a function to calculate the corresponding gradients for all trainable variables and applying them to the variables using the chosen optimizer RMSProp that moves the variables slowly towards the calculated gradient. The math to the following code snippet describing the backward pass for one model can be seen in lines 8 to 11 from **Algorithm 1**:

```
1  import tensorflow as tf
2  # assert this inside an agent (self), who inhibits
       all the models and needed variables
3  with tf.GradientTape() as tape:
4
5      pred, _, _ = self.t_action_model((next_states,
       actions, messages, agent_inds, hidden_action))
6
7      targets = rewards + self.gamma*(tf.math.
       reduce_max(pred, axis=1))*(1-dones.squeeze())
8
9      q_vals, _, _ = self.action_model((states,
       last_actions, last_messages, agent_inds,
       last_hidden_action))
10
11     q_inds = tf.one_hot(actions.squeeze(), self.
       action_space)
12
13     exp_q = tf.reduce_sum(tf.multiply(q_vals,
       q_inds), axis=1)
14
15     a_loss = self.mse_loss(targets, exp_q)
16
17 gradients = tape.gradient(a_loss, self.action_model
       .trainable_variables)
18
19 self.action_model.optimizer.apply_gradients(zip(
       gradients, self.action_model.
       trainable_variables))
```

IV. RESULTS

We trained the Simple Spread environment with the following parameters: 20 epochs, 250 episodes, 32 batch size, 2 agents, 25 max episode length, update target network for 50 episodes, epsilon by 0.05, discount factor by 1, learning rate by 0.0005, and momentum by 0.95.

In the beginning, episodes start with a very low reward. We can attribute the reason for this low reward to the fact that the environment started training randomly. In the first steps, we see a big jump in the scores and then these scores reach some level by looking at the average of the latest 50 episodes. The single episode reward bounces between -55 and -70. If we look at the paper [1], we observe a nearly rapid learning in one of the experiments conducted using RIAL in Figure 2.

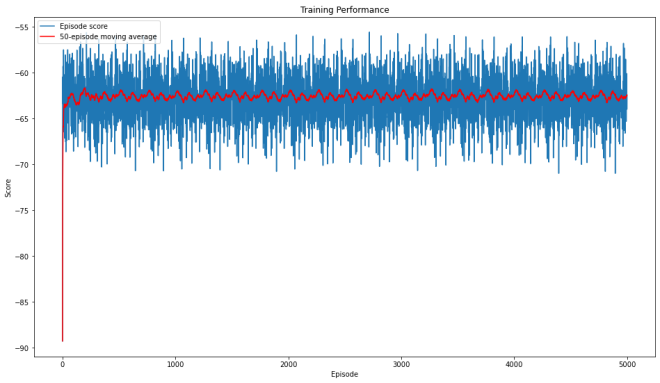
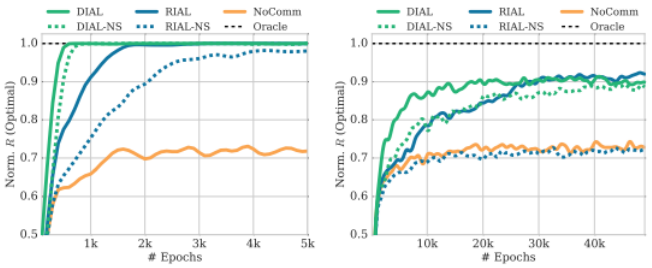


Fig. 3. Simple Spread: The red line shows the average score of the latest 50 episodes. The blue line shows the single episode score.



(a) Evaluation of $n = 3$ (b) Evaluation of $n = 4$

Fig. 4. (a-b) For the Switch Riddle environment: (Performance comparison of DIAL and RIAL, with and without (-NS) parameter sharing, and NoComm-baseline, for 3 and 4 agents respectively. [1]

In this study, 3 agents were used and fluctuations in learning increased when the number of agents was increased. From this point of view, we can hypothesize that the number of agents may be related to this, depending on the environment. Using 2 agents in our environment may have caused us to see a big jump in the graph at the very beginning.
After this big jump, the average scores seem stable and we can say the noise may cause the graph to look like that. In the Deep Reinforcement Learning lecture notes for Deep Q-Network [15], we see a similarity between the third graph and our graph below. This makes us think that double Q-learning may cause an overestimation in our policy since RIAL utilizes double q-learning.

We found a previous study for the Simple Spread environment on the Internet. The data of this study are as follows:

In this study, CommNet, BICNet, and Multi-Agent Deep Deterministic Policy Gradient (Maddpg) approaches were applied to the Simple Spread environment. [19] The graphic was taken from the work done and was not run in the same environment as our work. However, if we assume that this data is correct, we can see that we do not have a correct learning style when we compare it with our study.
When we compare the rewards, we realize that while the

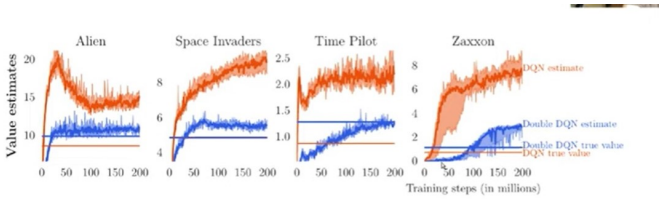


Fig. 5. DRL20222 Lecture06: The Overestimation Bias, 13:36
In those graphs, we can see that the DQN overestimates true values.

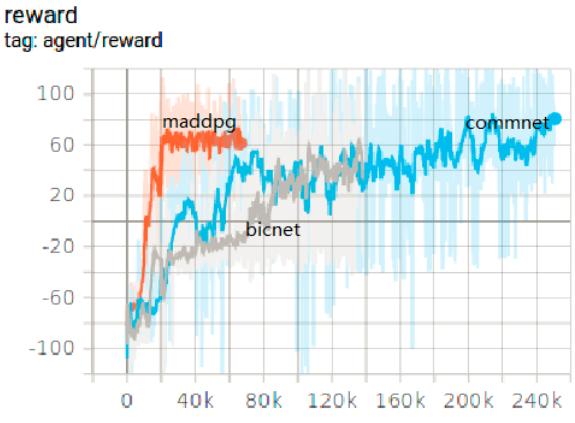


Fig. 6. Training curve taken from the work [19]

average score in our study remained almost stable at some point between -60 and -65, it saw +60 values in this study. However, we can also attribute the reason why we encounter such a result here to the parameters we use. When we look at Figure 6, we see that 240K episodes were run, but we only used the 20 epochs per 250 episodes as parameter. In this graph, we can see that the rewards are moving in negative values around 5K episodes.

Another possible reason could be our variation from the original paper in terms of sampling the whole episodes before updating. The main difference is, that the agents can use their updated policies not before the next episode. The fluctuations in the graph 6 therefore may exist, because the policy is changed after every episode, even though the updates should happen in tiny steps to not overfit.

V. DISCUSSION

Since we could not achieve a smooth learning line in our study, we came up with some ideas about the reasons for this and thought about how we can improve our existing resources.

In the current code, training was costly in terms of memory, as a new sample was produced from the model at each step. As a solution, agents can store the hidden states of the recurrent neural network in a variable instead of memory, and then use their own instead of using other agents' hidden states as they currently are.

We are updating the models after one step of one agent but in the environment description, it is said that all agents have to take a step before the environment is updated. It may not give a better result one hundred percent but adapting the code better to the multi-agent particle environment could enhance our results.

REFERENCES

[1] Foerster, Jakob, et al. "Learning to communicate with deep multi-agent reinforcement learning." Advances in neural information processing systems 29 (2016).

[2] Hausknecht, Matthew, and Peter Stone. "Deep recurrent q-learning for partially observable mdps." 2015 aaai fall symposium series. 2015.

[3] Watzlawick, Beavin, H. BEAVIN JANET, and D. Don. "JACKSON: Menschliche Kommunikation." Formen, Störungen, Paradoxien 10, 1974, p.53.

[4] Lowe, Ryan, et al. "Multi-agent actor-critic for mixed cooperative-competitive environments." Advances in neural information processing systems 30 (2017).

[5] OpenAi, Multi-Agent Particle Environment https://github.com/openai/multiagent-particle-envs, accessed September 2022

[6] PettingZoo Documentation, Simple Spread, https://pettingzoo.farama.org/environments/mpe/simple_spread/, accessed September 2022

[7] PettingZoo Documentation, "Basic Usage", https://pettingzoo.farama.org/content/basic_usage/, accessed September 2022

[8] Hafiz, A. M., Bhat, G. M. "Deep Q-Network Based Multi-agent Reinforcement Learning with Binary Action Agents".

[9] Kim, K. Multi-Agent Deep Q Network to Enhance the Reinforcement Learning for Delayed Reward System. Appl. Sci. 2022, 12, 3520. https://doi.org/10.3390/ app12073520.

[10] Chen, C., Ying, V., amp; Laird, D. (n.d.). Deep Q-Learning with Recurrent Neural Networks. https://doi.org/ http://cs229.stanford.edu/proj2016/report/ ChenYingLaird-DeepQLearningWithRecurrentNeuralNetwords-repo pdf.

[11] Lakshmana, K. R., Firoz, K., Sadia, D., S., B. S., Amir, M., amp; Ebuka, I. (2021). Recurrent Neural Network and Reinforcement Learning Model for COVID-19 Prediction . Frontiers in Public Health, 9. https://doi.org/10.3389/fpubh.2021.744100.

[12] Jiang, J., amp; Lu, Z. (2018). Learning Attentional Communication for Multi-Agent Cooperation. https://doi.org/10.48550/ARXIV.1805.07733.

[13] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. Learning multiagent communication with backprop agation. In Advances in Neural Information Processing Systems (NIPS), 2016.

[14] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Mul tiagent bidirectionally-coordinated nets for learning to play starcraft combat games. arXiv preprint arXiv:1703.10069, 2017.

[15] Xiangyu Kong, Bo Xin, Fangchen Liu, and Yizhou Wang. Revisiting the master-slave architecture in multi-agent deep reinforcement learning. arXiv preprint arXiv:1712.07305, 2017.

[16] Schmid, Leon. "DRL2022 Lecture06 The Oversestimation Bias." YouTube, 6 May 2022, youtu.be/REOQmn4L0pk.

[17] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[18] Cho, Kyunghyun, et al. "On the properties of neural machine translation: Encoder-decoder approaches." arXiv preprint arXiv:1409.1259 (2014).

[19] isp1tze. (n.d.). Isp1tze/maproj: Multi-agent Project (commnet, bicnet, maddpg) in pytorch for multi-agent particle environment. GitHub. Retrieved September 28, 2022, from https://github.com/isp1tze/MAProj.