Report for AI3007: Group 9

1st Nguyen Ngo Viet Trung
MSSV 22022598

2nd Vu Minh Tien
MSSV 22022645

3rd Pham Quang Vinh
MSSV 22022648

***Index Terms*—MAgent2, Double Deep Q Learning, QMix, Multi-Agent, Value Decomposition Network**

I. Introduction

The link to the project's source code: **SOURCE CODE HERE**

In fulfillment of the Reinforcement Learning Course final project, this report records and evaluates our progress in training multiple reinforcement learning algorithms and testing them against three benchmark models, namely the Random, Pretrained-1 and Full-pretrained models, given by our lecturer, all within a simulation named Battlev4 of Magent2. The primary objective is to achieve the highest winrate possible, given a few constraints in training time.

Magent2 is an engine for high-performance environments, where a number of agents (multi-agents) share and interact in a common environment. Each agent is driven by its own interests, and acts to maximize its own reward. In some environments, these interests are opposed to the interests of other agents, resulting in complex group dynamics. Our environment of interest in Magent2 is the Battle (or Battlev4) environment, a large-scale battle, where agents can either move or attack each turn.

For this project, we implemented several well-known reinforcement learning algorithms that we find suitable for the environment's characteristics, all within the constraints of limited training time. Specifically, we deployed QMix, Double Deep Q Learning and Value-Decomposition Networks. Out of all solutions, our Double Deep Q Learning model achieved absolute win rate against all benchmark pretrained models, including the Random model, the Pretrained-1 model and the Full-Pretrained model, as already mentioned.

The outline of sections in this report is as follows: In section 2, we will introduce 3 methods that use and test to solve this problem. At the same time we will present some configuration and transformations to better adapt these algorithms to MAgent2. In the third section, we will provide detail information about implementation : functions and setup of each method. In the fourth section, we will experiment with these methods under various changes in parameters, Q-Network architecture,.. to determine the best method. Finally, we will present the result of these experiments and draw some conclusions based on that.

II. Methods

A. The Magent2 Battlev4 environment - A brief introduction

According to official documentation, Magent2 is an engine for high-performance multi-agent environments with very large numbers of agents, along with a set of reference environments. Among them, this report provides the implementations for Battlev4, a large-scale team battle.

In this environment, agents are rewarded for their individual performance and not for the performance of their neighbors, so coordination is difficult. Agents slowly regain HP over time, so it is best to kill an opposing agent quickly. Specifically, agents have 10 HP, are damaged 2 HP by each attack, and recover 0.1 HP every turn.

In addition to Battlev4's characteristics, the project introduces another constraint – model training time is capped at a maximum of 2 hours. This condition is critical, and we have shaped our approach to address this constraint while solving the challenges presented by the environment.

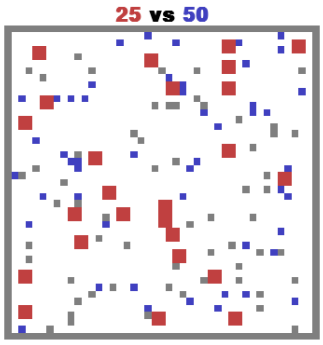

Fig. 1. The Battlev4 environment of Magent2

B. Our approach

Although Battlev4 is a multi-agent environment, the limited training time makes many multi-agent algorithms inefficient – particularly those that train multiple models, each tailored for an individual agent. Given that the final submitted result must consist of a single model, dividing the already limited time to train multiple models means that all agents, except the one used in the final submission, become redundant. A key characteristic of this environment is that all agents share the same objective – to defeat the enemy team. Based on these insights, we opted to train a single, generalized model that can be applied to all agents on our side in the Battlev4 environment.

To reinforce our approach, we proposed the use of several algorithms, including Value Decomposition Network, two variants of Deep Q Learning – Double Deep Q Learning and QMix. These algorithms utilize neural networks alongside Q-value updating functions, addressing the limitations of traditional reinforcement learning. Additionally, they offer the flexibility needed to operate effectively in complex environments like Battlev4 in Magent2.

C. Deep Q Learning

Deep Q Learning combines a deep neural network with conventional Q learning. Rather than storing Q tables, the algorithm utilizes a neural network, where the inputs represent encoded state information, and the outputs correspond to the Q values. The network's weights are denoted by θ, and the model updates through direct interaction with the environment, incorporating state information such as observations, actions, rewards, and next observations.

The ultimate objective is to minimize the temporal difference (TD) error:

$$L(\theta) = \sum_i (y_{dqn} - Q(s, a, \theta))^2$$

y_{dqn} is updated according to the following Q function: $y_{dqn} = r + \max'_u Q(s', a'; \theta^-)$, a is action, s is observation matrix and θ^- is parameters of target networks.

D. Double Deep Q Learning

1) Algorithm overview: Double Deep Q Learning (DDQN) is a variant and an improvement of standard Deep Q Learning. By decoupling the action selection and evaluation processes, DDQN enhances the stability and accuracy of Q-value estimates, addressing one of the key issues faced by conventional Q Learning – overestimation, which hinders the performance of the conventional algorithm

In DDQN, two networks are utilized, namely the policy network and the target network. While the former is responsible for selecting the action that maximizes the Q value for the next state, the latter directly evaluates and compute the said value. This approach has proven to be effective, reducing overestimation.

The DDQN update rule can be presented as follows:

$$Y_i^{DDQN} = R_{t+1} + \gamma Q_{target}(S', argmax_a Q_{online}(S', a, \theta_t), \theta)$$

2) Implementation method in Magent2: Although Battlev4 is a multi-agent environment, all agents share a unified objective, making Double Deep Q Learning (DDQN) – a single-agent algorithm – a suitable choice for implementation. In this approach, the input consists of observations from all allied agents, each with a 13x13x5 field of view. This consistent observation space supports our goal of training a single, generalized model applicable to all agents. To implement DDQN, we designed a Q network architecture that serves both the policy and target models. The network was trained using data collected from all allied agents, resulting in an approach that is efficient in terms of training time and model convergence speed.

E. Value Decomposition Network

1) Algorithm overview: Value Decomposition Network (VDN) is designed to solve the problem of "optimizing a joint policy" for multiple agents operating under a single team reward. The algorithm achieves this by decomposing the joint Q value into the sum of individual Q values for each agent. This decomposition allows each agent to independently learn and optimize its own policy while still contributing to the overall team reward.

In VDN, each agent possesses an independent DQN network. These DQN networks take their own individual observations as input and output the corresponding Q value for the chosen action. Before the training loop, agents share their Q values and target Q values with each other. While training, these Q values and target Q values are summed to form a total Q value (or Q_{tot}) and a total target Q value (Q_{tot}^{total}). These two values serve as the foundation for optimizing each agent's DQN network.

Below are a few functions for each O value that VDN makes use of:

- Total Q value: Adds all the Q values to get the total Q value, $Q_{tot}(s, a) = \sum_{i=1}^{n} Q_i(s^i, a^i)$
- Total target Q value: Adds all the target Q values to get total target Q value $Q_{tot}^{target}(s', a') = \sum_{i=1}^{n} max_{a'} Q_i^{target}(s'_i, a'_i)$
- Q learning: Every iteration get a better total Q value estimation, passing gradient to each Q function to update it. $\mathcal{L}(\theta, \mathcal{D}) = E_{r \sim \mathcal{D}}[Q_{tot}(s, a) - r + \gamma Q_i^{target}(s'_i, a'_i))(1 - done_i)]$

Where $\mathcal{D}$ represents the replay buffer, which is shared among all agents. The variable r denotes the reward. The value of $done_i$ is set to 1 if the episode has ended and set to 0 (False) otherwise.

2) Implementation method in Magent2: : To integrate into Magent2, the algorithm is implemented as follows:

- Q_a network: We employed a CNN-based architecture, similar to other models.
- Independent Q_a networks for agents: Each agent in the Battlev4 environment is initialized with its own Q_a network, which was updated independently. This ensures that each agent has a separate Q network for determining its actions.
- Line Buffer: We experimented with two versions – first one is where each agent had its own line buffer, and the second is where all agents shared a common line buffer. Through experimentation, we noticed that sharing a line buffer among all agents received better results and less resource-intensive, compared to assigning individual line buffers to each agent.
- Summing Q values: The Q values of all agents are added together to obtain a mixed Q value, denoted as Q_{tot}. We then optimize the Q networks of each agent based on Q_{tot}, the team reward r, and Q_{tot}^{target}.

F. QMix

QMix is a multi-agent algorithm that blends the principles of Independent Q-Learning (IQL) and Centralized Q-Learning. As an enhancement to its predecessor, Value Decomposition Network (VDN), QMix introduces a more flexible approach to computing and optimizing Q values. Unlike VDN, which relies on the sum of individual Q values, QMix focuses on satisfying a single condition, which is matching the argmax of Q_{tot} with the argmax of Q_a for each agent.

As a result, QMix enables each agent to determine its actions based on its Q_a network, while still ensuring coordinated interactions among allies through the Q_{tot} condition

QMix is implemented with three core components: The Agent network, Mixing network and the Hypernetworks.

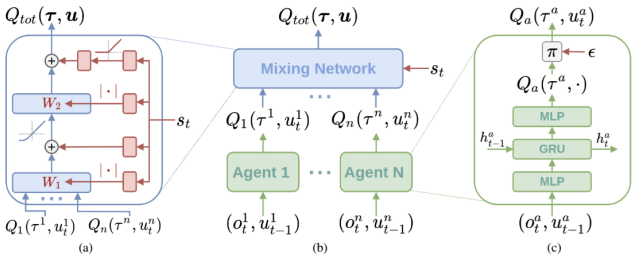

Fig. 2. (a) Mixing network structure. The hypernetworks (in red) generate the weights and biases for the mixing network layers (in blue). (b) The overall QMix architecture. (c) Agent network structure.

- Agent Networks: Each agent possesses a neural network to represent the Q function. This network is similar to that of Q Learning, taking as input the agent's current field of view and previous action. According to [**rashid2020monotonic**], these Q networks have a similar architecture to the RNN.
- Mixing Network: A feed-forward network that receives each agent's Q value (the output of the Agent network) and outputs Q_{tot}. This network is guaranteed to be monotonic by limiting its weights to prevent negative values.
- Hypernetworks: Hypernetworks generate the weights for the Mixing Network. These weights are derived from state information (collected from all agents). The weights are produced by a linear layer followed by a ReLU activation function, to ensure non-negative values. By learning parameters as a function of state input S, Q_{tot} becomes dependent on S without the need to pass S directly into the Mixing Network.

QMix is trained to optimize the Mean Square Loss, similarly to Deep Q learning:

$$L(\theta) = \sum_i (y_{tot} - Q_{tot}(s, a, \theta))^2$$

where b is the batch size of transitions sampled from the replay buffer and $y_{tot} = r + \gamma max_{a'} Q(s, a, \theta)$, a is action, s is observation matrix and θ is parameters of DQN.

Fig. 3. Project Structure

1) Implementation method in Magent2: To integrate into Magent2, the algorithm is implemented as follows:

- Q_a network structure: Instead of using RNN like [**rashid2020monotonic**], less complex CNN-based architecture was leveraged. Said networks were also applied to Qmix training and received high performance, indicating the structure's suitability.
- Additional usage of the num_agent parameter: Instead of including all agents in the algorithm, group said agents into groups of 9, 28, 27, 81, can help reducing the shape of the input matrixes in computations, storing, and increasing the quantity of data batch-wise, aiding agents in cooperation in narrower spaces.
- Using one single Q network for all agents: Due to training time limitations, using one generalized model for all agents leads to better convergence, as the quantity of data is large, and the unified field of view from all agents leading to the feasibility of this modification.
- Due to the grouping of agents, the concatenated field of view from each said group is used instead of the global 45x45x5 environment state. At the same time, in the Mixing network, the implementation of CNN architecture can assist in state feature extraction, as well as decreasing the state size before passing to the Hypernetworks.

III. Implementation

1) Design: In order to implement and train our proposed algorithms on the Battlev4 environment, we have created a few customized modules, tailored to handle a few roles in 3):

- **Networks**: This module contains the neural network architecture used for Q function and Mixing network training in our algorithms. We referenced to the pretrained and final-pretrained model structure, while also making our own changes to make way for the integration of QMix
- **Agents**: To enhance flexibility in training and inference, we implemented this module to represent both the red and blue team. The Agent modules' functionalities include taking action based on their respective given policies, auto-updating/improving their policy from the training

process, loading pretrained parameters, etc, ... (implementation details will be presented in ...)

- **Trainer**: This module is treated as a medium, possessing configurations and interactions between agents and environment. Depending on the algorithm in context, the Trainer provides corresponding training functions, network weights update, connection to Memory module for environment's settings, metrics logging for training process.
- **Memory**: The presence of a Memory Buffer is needed in QMix and Double Deep Q learning to store agent/environment interactions. However, due to the two said algorithm's difference in buffer data type storing, we designed two separate classes: The ReplayBuffer for DQL and the StateMemory for QMix. Implementation details are provided in later sections.
- **Inference**: Similarly to the Trainer, Inference is used solely for evaluating the performance of different policies against one another (extended from the instructor's eval() code). Additionally, it also recorded the gameplay results in video format for further analysis.

2) Memory: As mentioned above, QMix, Double Deep Q, and VDN algorithms all require the use of buffers to store environmental data during training. However, for Double Deep Q, these data simply consist of the observation matrices of each agent, along with the actions and rewards they receive. Meanwhile, QMix requires grouping this information according to the number of agents per group (based on the parameter num_agent). Based on these characteristics, we have designed the following two classes:

- **ReplayBuffer** : This class is implemented for Double Deep Q learning and VDN. Similarly to the initial algorithm descriptions, this class is responsible for storing pairs of values (observation, action, reward, next_observation, done) for each individual agent. A queue structure with a fixed size is used for storage. To optimize the training process, this class inherits the Dataset class in torch.utils.data and includes push and sample methods to insert and retrieve data.
- **StateMemory**: Unlike ReplayBuffer, this class requires that data (observation, action, reward, next_observation, done) be stored in groups of a fixed size. The agents in each group must belong to the same team, and dead agents need to be handled to guarantee uniform size. To solve this, we implemented a push method similar to ReplayBuffer, but each data point (s, a, r, s', d) is stored in a separate deque. The agents are iterated in the order provided by MAgent2 (e.g., red_0 - red_80, blue_0 - blue_80) and stored in the deques based on the modulo of their order with the number of agents. After completing a full iteration, for dead agents, the ensemble() method encodes them with an observation value of -100 and an action value of -1.

3) Agents: For each different algorithm, an agent class is implemented to interact with the environment, take actions, and update during training. As illustrated in the figure, we have created three Agent classes corresponding to the three algorithms, as well as three additional Agent classes that make decisions based on preexisting policies. Specifically:

- We implemented RandomAgent, PretrainedAgent, and FinalAgent, which perform actions based on the respective following strategies: random actions, pretrained policies, and final policies provided by the instructor. These classes allow for loading the pretrained weights of the Q network and include a $get\_action()$ method that selects actions based on the respective policy (using epsilon-greedy).
- We also implemented VDNAgent, QMixAgent, and DQNAgent. Similar to the three classes above, these classes include a $get_actions()$ method, but they also feature an additional $train()$ method to update parameters during the training phase. All three classes utilize the epsilon decay method to gradually reduce epsilon throughout the training process.

4) Networks: We implemented five different networks shared across the three algorithms. The Pretrained and Final networks follow CNN architectures published by our instructor. Additionally, we customized a Q network with a similar structure to the Pretrained model but with fewer parameters.

In the QMix algorithm, the Q_a network is designed to closely resemble the Pretrained network architecture. Regarding the Q_{tot} network, we implemented it based on the design described in [**rashid2020monotonic**], but with the addition of a CNN to reduce the size of the observation matrix for agent groups retrieved from StateMemory.

IV. Experiment Results

1) Evaluation metrics: : To assess the performance of the model during the training phase, we use the following metrics:

- Gap-reward: The difference between the total reward obtained by the blue team (training side) and the red team (opponent side). A larger gap indicates that the blue agents receive higher rewards and perform more optimally compared to the red agents.
- Blue's kill count and red's kill count : The total number of agents that the blue and red teams can eliminate during each episode. This is evaluated based on rewards being greater than 4.5 (aligned with the instructor's evaluation method).
- Time: The time taken to complete an episode by agents.

2) General Settings: During training, we applied consistent settings across different experiments. Specifically:

- Batch_size was fixed at 64.
- Memory capacity was set to a maximum of 10,000 elements.
- The target network for each algorithm was updated every 2 episodes.
- To enhance the agents' offensive capability when competing against random opponents, we increased the attack_opponent_reward parameter in the training environment to 0.5.

- Most algorithms were tested over 70 episodes, with the blue agents (training side) competing against random opponents.
- The initial epsilon started at 1 and gradually decayed to 0.05, with a 4% reduction per episode.

3) Experiment 1: : To evaluate which of the three algorithms performed better, we applied the same general settings described above. All three tested algorithms utilized Q networks with architectures identical to the pretrained network published by the instructor. For QMix where grouping is used, the agents were grouped together, encompassing all 81 agents on the team. Meanwhile, VDN used 81 individual Q networks, each with its own replay buffer, and was trained exclusively on data from the blue team. In contrast, the other two algorithms were trained using data from both the blue and red teams.

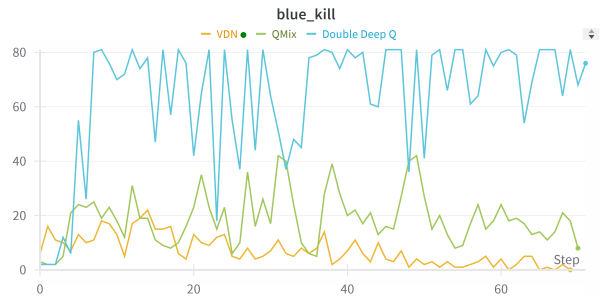

Fig. 4. The number of agents eliminated by the blue team in each episode.

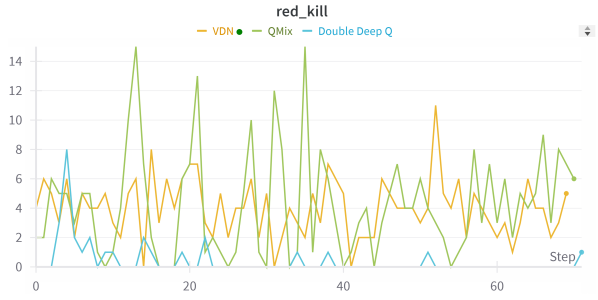

Fig. 5. The number of agents eliminated by the red team in each episode.

The results in Figure IV-3 show that the number of agents eliminated by Double Deep Q is significantly higher compared to the other two models. On average, Double Deep Q eliminated between 45 and 81 agents, converging quickly—within just 9 episodes. Simultaneously, the number of agents the red team was able to eliminate steadily decreased to near zero (fluctuating between 0 and 2). This highlights the superior performance of Double Deep Q relative to the other algorithms.

VDN performed the worst; after 70 episodes, the model showed minimal learning progress, with the number of agents eliminated by the blue team gradually dropping to zero. QMix also demonstrated underwhelming performance, as the number of agents eliminated by the blue team was almost equal to the number eliminated by the red team.

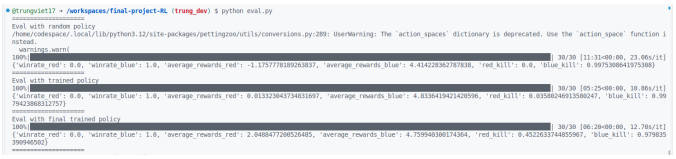

Fig. 6. Evalutation results for 3 benchmark model: Random, Pretrained-1, Full-pretrained

At the end of the experiment above, Double Deep Q demonstrated significantly greater effectiveness compared to QMix and VDN algorithms. To further test it's performance, we conducted evaluations against the three benchmark models provided by the instructor (Random, Pretrained-1, and FUll-pretrained). The game duration for all three models was relatively short, ranging from 7 to 10 seconds. Additionally, after evaluating 30 games against each of the models, the results were as follows:

Benchmark model	*red_winrate*	*blue_winrate*	*avg_blue_reward*	*avg_blue_kill*
Random	0.0	1.0	4.41	0.9975
Pretrained-1	0.0	1.0	4.83	0.9979
Full-pretrained	0.0	1.0	4.75	0.979

TABLE I
EVALUATION RESULTS

It can be seen that, when evaluated using CPU, the model took 10 minutes to complete 30 games against the Random model and approximately 5 minutes for 30 games against the pretrained and final models.

We also recorded two additional metrics – the average number of agents killed by the red and blue teams in each game. As shown in Figure 6, the average rewards for blue team (our team) were consistently much higher than those of the two pretrained agents. Additionally, the number of agents eliminated by the blue team was significantly greater (99%) – in nearly all cases, the blue team managed to eliminate all enemy agents before the stopping condition of 300 steps was reached.

The results also indicate that the model achieved a win rate of 1 across all three evaluation models.

Furthermore, we analyzed the behavior of the agents and observed that most blue agents tended to move towards the left side and consistently attacked whenever a red agent was within sight. The number of red agents steadily decreased, and after advancing to the left, the blue agents often clustered in the bottom-left corner of the map. Especially, any red agents that remained alive and moved outside the current observation range of the blue agents, were unable to be located and eliminated. This led to a significant increase in the duration of games against the Random model.

4) Experiment 2: Following the first experiment, it became evident that Double Deep Q algorithm with the pretrained network performed exceptionally well and met the initial objectives. However, in this experiment, we replaced the Q networks (using our modified network) and altered the training method by employing self-play. Using the same configurations

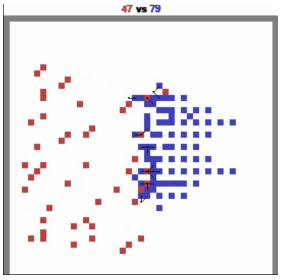

Fig. 7. Agents' innitial movement path

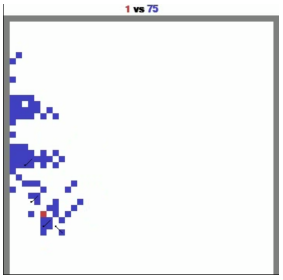

Fig. 8. Agents' post opponent elimination movement path

and evaluation methods as in the first experiment, the results obtained are shown in Figures 10 and 9.

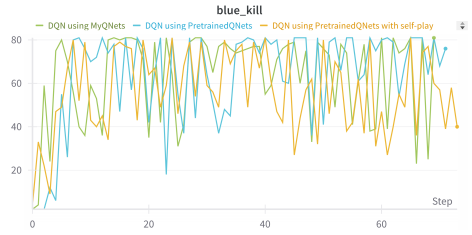

Fig. 9. Blue agents' kill count in each episode in 3 DQN approaches

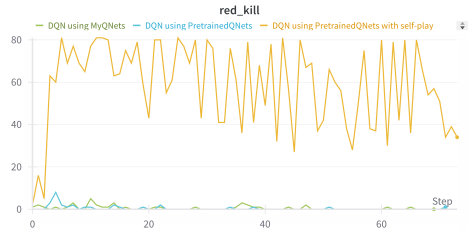

Fig. 10. Red agents' kill count in each episode in 3 DQN approaches

The results show that our network architecture produced nearly identical performance to the pretrained network architecture. The algorithm still achieved relatively strong results after approximately 9 episodes. By utilizing self-play during training, the algorithm also demonstrated somewhat reasonable effectiveness, as both red and blue teams eliminated a comparable number of agents over multiple episodes. From this experiment, it can be observed that the algorithm performs well across various network architectures and different learning methods.

V. Conclusion

In this report, we implemented three algorithms to oppose multiple predefined model given by the lecturer, all inside the Battlev4 environment of Magent2. Through a series of experiments, we found that the Double Deep Q model performed the best, achieving absolute dominance over the three benchmark models (Random, Pretrained-1 and Full-pretrained). Additionally, this model demonstrated the highest efficiency and fastest convergence compared to QMix and VDN.

However, there still exists some limitations to this model. Particularly, its lack of adaptability and flexibility in movement after advancing entirely to the opponent's side of the map. This drawback is the primary reason for the long runtime when competing against the Random model. Moreover, multi-agent algorithms in this work, such as QMix and VDN, have not been fully optimized. The main reasons for this lack of optimization include inefficient memory storage, usage, and module design. Additionally, the Q network architecture may not be fully suitable, and issues related to handling dead agents and grouping them have not been entirely resolved.

References

[1] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, Yong Yu "MAgent: A Many-Agent Reinforcement Learning Platform for Artificial Collective Intelligence".
Doi: https://doi.org/10.48550/arXiv.1712.00600

[2] Definition of multi-agent reinforcement learning from Wikipedia. Link: Multi-agent

[3] Rashid, Tabish and Samvelyan, Mikayel and De Witt, Christian Schroeder and Farquhar, Gregory and Foerster, Jakob and Whiteson, Shimon. "Monotonic value function factorisation for deep multi-agent reinforcement learning." Journal: Journal of Machine Learning Research.
Doi: https://doi.org/10.48550/arXiv.2003.08839

[4] Peter Sunehag, Guy Lever, Audrunas Gruslys, et al, . "Value-Decomposition Networks For Cooperative Multi-Agent Learning" Journal: AAMAS '18: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems Doi: https://doi.org/10.48550/arXiv.1706.05296