MAGNet: Multi-agent Graph Network for Deep Multi-agent Reinforcement Learning

Aleksandra Malysheva
JetBrains Research
St Petersburg, Russia
National Research University
Higher School of Economics
St Petersburg, Russia
malyshevasasha777@gmail.com

Daniel Kudenko
University of York
York, United Kingdom
JetBrains Research
St Petersburg, Russia
daniel.kudenko@york.ac.uk

Aleksei Shpilman
JetBrains Research
St Petersburg, Russia
National Research University
Higher School of Economics
St Petersburg, Russia
aleksei@shpilman.com

ABSTRACT

Over recent years, deep reinforcement learning has shown strong successes in complex single-agent tasks, and more recently this approach has also been applied to multi-agent domains. In this paper, we propose a novel approach, called MAGNet, to multi-agent reinforcement learning that utilizes a relevance graph representation of the environment obtained by a self-attention mechanism, and a message-generation technique inspired by the NerveNet architecture. We applied our MAGnet approach to the synthetic predator-prey multi-agent environment and the Pommerman game and the results show that it significantly outperforms state-of-the-art MARL solutions, including Multi-agent Deep Q-Networks (MADQN), Multi-agent Deep Deterministic Policy Gradient (MADDPG), and QMIX.

KEYWORDS

multi-agent system; relevance graphs; deep-learning

1 INTRODUCTION

A common difficulty of reinforcement learning in a multi-agent environment (MARL) is that in order to achieve successful coordination, agents require information about the relevance of environment objects to themselves and other agents. For example, in the game of Pommerman [9] it is important to know how relevant bombs placed in the environment are for teammates, e.g. whether or not the bombs can threaten them. While such information can be hand-crafted into the state representation for well-understood environments, in lesser-known environments it is preferable to derive it as part of the learning process.

In this paper, we propose a novel method, named MAGNet (Multi-Agent Graph Network), to learn such relevance information in form of a relevance graph and incorporate this into the reinforcement learning process. Furthermore, we propose the use of message generation techniques along this graph. Those techniques were inspired by the NerveNet architecture [18], which has been introduced in the context of robot locomotion, where it has been applied to a graph of connected robot limbs. MAGNet uses a similar approach, but bases the message generation on the learned relevance graph (see Section 3. The final decision making stage accumulates messages and calculates the best action for every agent.

The contribution of this work is a novel technique to learn object and agent relevance information in a multi-agent environment, and incorporate this information in deep multi-agent reinforcement learning. We applied MAGNet to the synthetic predator-prey game, commonly used to evaluate multi-agent systems [11] and the popular Pommerman [9] multi-agent environment, and achieved significantly better performance than state-of-the-art MARL techniques including MADQN [3], MADDPG [7] and QMIX [14]. Additionally, we empirically demonstrate the effectiveness of utilized self-attention [17], graph sharing and message generating modules with an ablation study.

2 DEEP MULTI-AGENT REINFORCEMENT LEARNING

In this section we describe the state-of-the-art deep reinforcement learning techniques that were applied to multi-agent domains. The algorithms introduced below (MADQN, MADDPG, and QMIX) were also used as evaluation baselines in our experiments.

Reinforcement learning is a paradigm which allows agents to learn by reward and punishment from interactions with the environment [15]. The numeric feedback received from the environment is used to improve the agent's actions.

The majority of work in the area of reinforcement learning applies a Markov Decision Process (MDP) as a mathematical model [13]. An MDP is a tuple (S, A, T, R), where S is the state space, A is the action space, $T(s, a, s') = Pr(s'|s, a)$ is the probability that action a in state s will lead to state s', and $R(s, a, s')$ is the immediate reward r received when action a taken in state s results in a transition to state s'. The problem of solving an MDP is to find a policy (i.e., mapping from states to actions) which maximises the accumulated reward. When the environment dynamics (transition probabilities and reward function) are available, this task can be solved using policy iteration [1].

We can define multi-agent extension of MDPs through partially observable MDPs (POMDPs)[12], where state is defined by a set of agents' observations $o_1, \ldots, o_N$, and action is defined by a set of individual agent's actions $a_1, \ldots, a_N$. The problem of solving an multi-agent POMDP is to find policies $\pi_i : O_i \times A_i \longrightarrow [0, 1]$, which produce the next state according to the state transition function and maximize the individual accumulated reward. If we consider a game with N agents the aim is to find policies $\pi = \pi_1, \ldots, \pi_N$ that maximize the expected reward $J_i = E_{o \sim p^\pi a \sim \pi_i}[R]$ for every agent i, where p^π is the distribution of states visited with policy π.

We do not describe the NervNet [18] approach in detail, since it was not intended to solve multi-agent reinforcement learning tasks, but rather a complex single agent. We present the adaptation of NerveNet message passing system in Section 3.2.

2.1 Multi-agent Deep Q-Networks

Q-learning is a value iteration method that tries to predict future rewards from the current state and an action. This algorithm applies so called temporal-difference updates to propagate information about values of state-action pairs, $Q(s, a)$. After each transition, $(s, a) \rightarrow (s', r)$, in the environment, it updates state-action values by the following formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max Q(s', a') - Q(s, a)] \quad (1)$$

where α is the learning rate and γ is the discount factor. It modifies the value of taking action a in state s, taking into account the received reward r.

Deep Q-learning utilizes a neural network to predict Q-values of state-action pairs. [10]. This so-called deep Q-network is trained to minimize the difference between predicted and actual Q-values:

$$y = r + \gamma \max_{a} Q^{past}(s', a') \quad (2)$$

$$L(\theta) = \mathbb{E}_{s \sim \rho^{\pi}, a \sim p(s)}[y_i - Q(s, a|\theta)]^2 \quad (3)$$

where y is the best action according to the previous deep Q-network, θ is the parameter vector of the current Q-function approximation and $a \sim p(s)$ denotes all actions that are permitted in state s.

The Multi-agent Deep Q-Networks (MADQN, [3]) approach modifies this process for multi-agent systems by performing training in two repeated steps. First, they train agents one at a time, while keeping the policies of other agents fixed. When the agent is finished training, it distributes its policy to all of its allies as an additional environmental variable. This approach shows the best results among other modifications of DGN algorithm.

2.2 Multi-agent Deep Deterministic Policy Gradient

When dealing with continuous action spaces, the methods described above can not be applied. To overcome this limitation, the actor-critic approach to reinforcement learning was proposed [16]. In this approach an actor algorithm tries to output the best action vector and a critic tries to predict the value function for this action.

Specifically, in the Deep Deterministic Policy Gradient (DDPG [6]) algorithm two neural networks are used: $\mu(s)$ is the actor network that returns the action vector. $Q(s, a)$ is the critic network, that returns the Q value, i.e. the value estimate of the action of a in state s.

The gradient for the critic network can be calculated in the same way as the gradient for Deep Q-Networks described above (Equation 3). Knowing the critic gradient $\nabla_a Q$ we can then compute the gradient for the actor as follows:

$$\nabla_{\theta^{\mu}} J = \mathbb{E}_{s \sim \rho^{\pi}}[\nabla_a Q(s, a|\theta^q)|s = s_t, a = \mu(s|\theta^{\mu})] \quad (4)$$

where θ^q and θ^{μ} are parameters of critic and actor neural networks respectively, and $\rho^{\pi}(s)$ is the probability of reaching state s with policy π.

The authors of [8] proposed an extension of this method by creating multiple actors, each with its own critic, where each critic takes in the respective agent's observations and actions of all agents. This then constitutes the following value function for actor i:

$$J(\theta_i) = E_{s \sim p^{\pi}, a \sim \pi^{\theta}}[\nabla_{\theta_i} log\pi_i(a_i|o_i) Q_i^{\pi}(o_i, a_1, \ldots, a_N)] \quad (5)$$

This Multi-agent Deep Deterministic Policy Gradient method showed the best results among widely used deep reinforcement learning techniques in continuous state and action space.

2.3 QMIX

Another recent promising approach to deep multi-agent reinforcement learning is the the QMIX [14] method. It utilizes individual Q-functions for every agent and joint Q-function for a team of agents. The QMIX architecture consists of three types of neural networks:

- Agent networks evaluate individual Q-functions for agents taking in the current observation and the previous action.
- A Mixing network takes as input individual Q-functions from agent networks and a current state and calculates a joint Q-function.
- Hyper networks add an additional layer of complexity to the mixing network. Instead of passing the current state to the mixing network directly, hyper networks use it as input and calculate weight multipliers at each level of the mixing network. We refer the reader to the original paper for a more complete explanation [14].

While authors demonstrate that this approach outperforms both MADQN and MADDPG methods, it is yet to be tested by time. Nevertheless, we included it in the number of our baselines.

3 MAGNET APPROACH AND ARCHITECTURE

Figure 1 shows the overall network architecture of our MAGNet approach. The whole process can be divided into a relevance graph generation stage (shown in the left part) and a decision making stages (shown in the right part). We see them as a regression and classification problem respectively. In this architecture, the concatenation of the current state and previous action forms the input of the models, and the output is the next action. The details of the two processes are described below.

3.1 Relevance graph generation stage

In the first part of our MAGNet approach, a neural network is trained to produce a relevance graph: a matrix $|A| \times (|A| + |O|)$, where $|A|$ is the number of agents and $|O|$ is the maximum number of environment objects. The relevance graph represents the relationship between agents and between agents and environment objects. The higher the absolute weight of an edge between an agent a and another agent b or object o is, the more important b or o are for the achievement of agent a's task. The graph is generated by MAGNet from the current and previous state together with the respective actions.

Figure 6B shows an example of such a graph for two agents in the game of Pommerman. The displayed graph only shows those edges

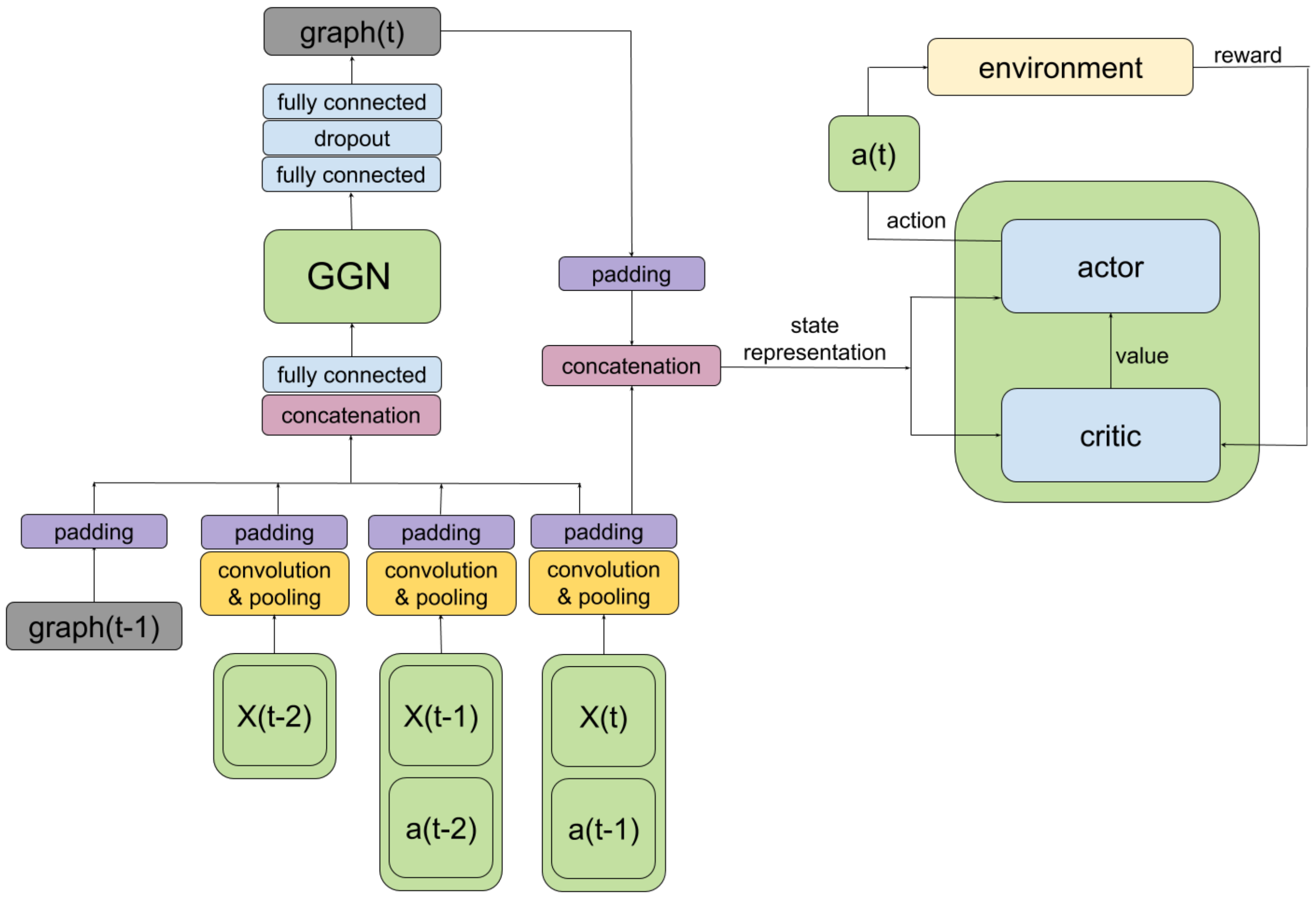


Figure 1: The overall network architecture of MAGNet. Left section shows the graph generation stage. Right part shows the decision making stage. $X(t)$ denotes the state of the environment at step t. $a(t)$ denotes the action taken by the agent at step t. GGN refers to Graph Generation Network see Section 3.1.

which have a non-zero weight (thus there are objects to which agent 1 is not connected in the graph).

To generate this relevance graph, we train a neural network via back-propagation to output a graph representation matrix. The input to the network are the current and the two previous states (denoted by $X(t)$, $X(t-1)$, and $X(t-2)$ in Figure 1), the two previous actions (denoted by $a(t-1)$ and $a(t-2)$), and the relevance graph produced at the previous time step (denoted by $graph(t-1)$). For the first learning step (i.e. $t = 0$), the input consists out of three copies of the initial state, no actions, and a random relevance graph. The inputs are passed into a convolution and pooling layer, followed by a padding layer, and then concatenated and passed into fully connected layer and finally into the graph generation network (GGN). In this work we implement GGN as either a multilayer perceptron (MLP) or a self-attention network, which uses an attention mechanism to catch long and short term time-dependencies. We present the results of both implementations in Table 1. The self-attention network is an analogue to a recurrent network such as LSTM, but takes much less time to compute [17]. The result of the GGN is fed into a two-layer fully connected network with dropout, which produces the relevance graph matrix.

The loss function for the back-propagation training is composed of two parts:

$$L = ||W^t - W^{t-1}||_2^2 + \sum_{\xi_{(v,u)} \in \Xi^t} (w^t_{(v,u)} - s(\xi_{(v,u)}))^2 \qquad (6)$$

The first component is based on the squared difference between weights of edges in the current graph W^t and the one generated in the previous state W^{t-1}. The second iterates through events Ξ^t at time t and calculates the square difference between the weight of edge (v, u) between objects that are involved in that event $\xi_{(v,u)}$ and the event weight $s(\xi_{(v,u)})$. In the base MAGNet configuration we use a simple heuristic rule for event weights: 1 for a positive event (i.e injuring a prey in predator-prey game) and -1 for a negative one (i.e., being blown-up in the game of Pommerman). We do explore the benefits of using a more complex approach and adding domain specific heuristics in Section 4.9.

We can train the neural network for graph generation without training the agents network if some pre-trained agent is provided. Both Pommerman and predator-prey environments have these default agents. If fact, we found out that the better way to train MAGNet is to first pre-train the graph generation and then add the agent networks (see also Section 4.8).

We can train individual relevance graphs for every agent or one shared graph (GS) that is the same for all agents on the team. We performed experiments to determine which way is better (see Table 1).

3.2 Decision making stage

The agent AI responsible for decision making is also represented as a neural network whose inputs are accumulated messages (generated by a method inspired by NerveNet [18] and described below) and the current state of the environment. The output of the network is an action to be executed.

The graph G generated at the last step is $G = (V, E)$ where edges represent relevance between agents and objects. Every vertex v has a type $b(v)$. Types of verteces for both test environments are described in Section 4.3.

The final (action) vector is computed in 4 stages through message passing system, similar to the NerveNet system [18]. Stages 2 and 3 are repeated for a specified number of message propagation steps at every step of the game.

(1) **Initialization of information vector.** Each vertex v has an initialization network $MLP^{b(v)}_{init}$ associated with it according to it's type $b(v)$ that takes as input the current individual observation O_v and outputs initial information vector μ^0_v for each vertex.

$$\mu^0_v = MLP^{b(v)}_{init}(O_v) \tag{7}$$

(2) **Message generation.** Message generation performs in iterative steps. At message generation step $\tau + 1$ (not to be confused with enviromental time t) message networks $MLP^{c(v,u)}_{mess}$ compute output messages for every edge $(v, u) \in E$ based on type of the edge $c(v, u)$.

$$m^\tau_{(v,u)} = MLP^{c(v,u)}_{m}(\mu^\tau_v) \tag{8}$$

(3) **Message processing.** Information vector $m^{\tau+1}_v$ at message propagation step τ is updated by update network $LSTM^{b(v)}_{up}$ associated with it according to it's type $b(v)$, that takes as input a sum of all message vectors from connected to v edges multiplied by the edge relevance $w_{(v,*)}$ and information at previous step m^τ_v.

$$\mu^{\tau+1}_v = LSTM^{b(v)}_{up}(\mu^\tau_v, \sum m^\tau_{(v,*)} w_{(v,*)}) \tag{9}$$

(4) **Choice of action.** All vertices that are associated with agents have a decision network $MLP^{b(v)}_{choise}$ which takes as an input its final information vector m^τ_v and compute the mean of the action of the Gaussian policy.

$$a_v = MLP^{b(v)}_{choise}(\mu^\tau_v) \tag{10}$$

Since message passing system outputs an action, we view it as an actor in the DDPG actor-critic approach [6], and train it accordingly.

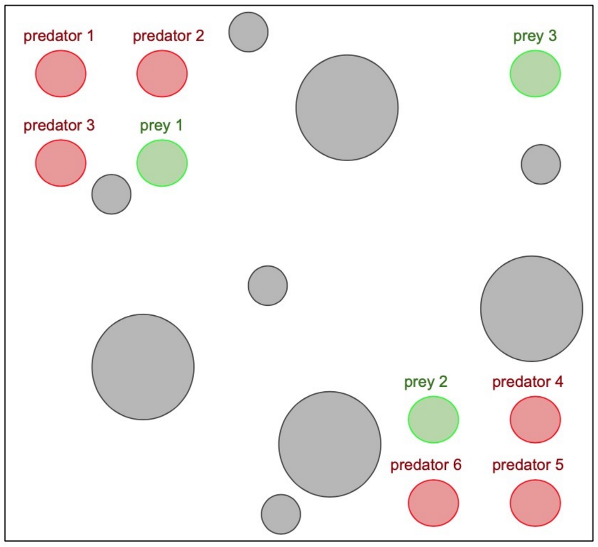


Figure 2: Synthetic predator-prey game. In order to win the game, predators (red) must catch all three prey (green) that are moving at faster speed. Game lasts 500 iterations. Random obstacles (grey) are placed in the environment at the start of the game.

4 EXPERIMENTS

4.1 Environments

In this paper, we use two popular multi-agent benchmark environments for testing, the synthetic multi-agent predator-prey game [11], and the Pommerman game [9].

In the predator-prey environment, the aim of the predators is to kill faster moving prey in 500 iterations. The predator agents must learn to cooperate in order to surround and kill the prey. Every prey has a health of 10. Predator coming close to the prey lowers the prey's health by 1 point. Lowering the prey health to 10 kills the prey. If even one prey survives after 500 iterations, the prey team wins. Random obstacles are placed in the environment at the start of the game (seen as grey circles in Figure 2). The starting positions of predators and prey can be seen in Figure 2.

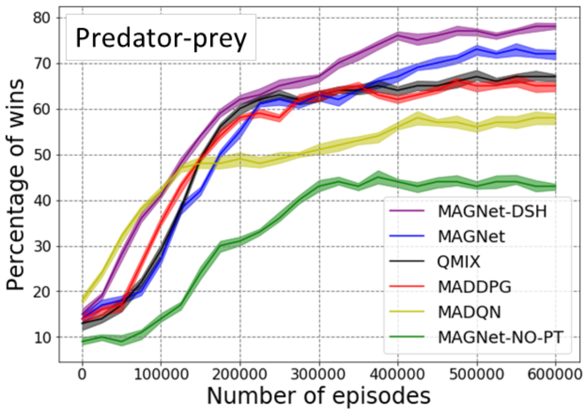
The Pommerman game is a popular environment which can be played by up to 4 players. The multi-agent variant has 2 teams of 2 players each. This game has been used in recent competitions for multi-agent algorithms, and therefore is especially suitable for a comparison to state-of-the-art techniques.

In Pommerman, the environment is a grid-world where each agent can move in one of four directions, lay a bomb, or do nothing. A grid square is either clear (which means that an agent can enter it), wooden, or rigid. Wooden grid squares can not be entered, but can be destroyed by a bomb (i.e. turned into clear squares). Rigid squares are indestructible and impassable. When a wooden square is destroyed, there is a probability of items appearing, e.g., an extra bomb, a bomb range increase, or a kick ability. Once a bomb has been placed in a grid square it explodes after 10 time steps. The explosion destroys any wooden square within range 1 and kills any agent within range 4. If both agents of one team die, the team loses the game and the opposing team wins. The map of the environment is randomly generated for every episode.

The game has two different modes: free for all and team match. Our experiments were carried out in the team match mode in order to evaluate the ability of MAGnet to exploit the discovered relationships between agents (e.g. being on the same team).

We represent states in both environments as $D \times D \times M$ tensor S, where $D \times D$ are the dimensions of the field and M is the maximum possible number of objects. $S[i, j, k] = 1$ if object k is present in $[i, j]$ space and is 0 otherwise. Predator-prey state is $64 \times 64 \times 20$ tensor, and Pommerman state is $11 \times 11 \times 30$.

4.2 Evaluation Baselines



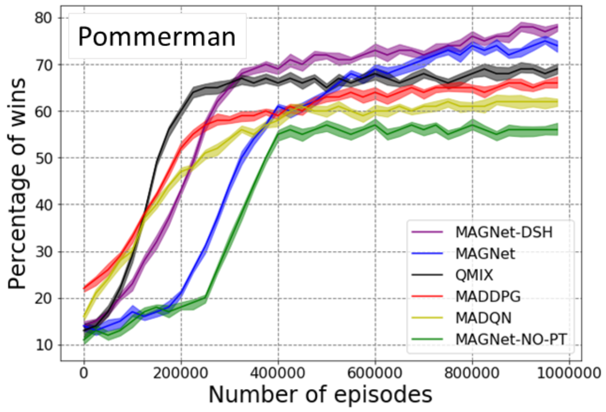


Figure 3: MAGNet variants compared to state-of-the-art MARL techniques in the predator-prey (top) and Pommerman (bottom) environments. MAGNet-NO-PT refers to MAGNet with not pretraining for graph generating network (Section 4.8). MAGNet-DSH refers to MAGNet with domain specific heuristics (Section 4.9). Every algorithm trained by playing against a default environment agent for a number of games (episodes) and a respective win percentage is shown. Default agents are provided by the environments. Shaded areas show the 95% confidence interval from 20 launches.

In our experiments, we compare the proposed method with state-of-the-art reinforcement learning algorithms in two environments. One is the predator-prey game [11] and the other is the Pommerman game simulated in team match mode. Figure 3 shows a comparison with MADQN [3], MADDPG [7] and QMIX [14] algorithms. Each of thees algorithms trained by playing a number of games (i.e. episodes) against the heuristic AI, and the respective win rates are shown. All graphs display a 95% confidence interval over 20 launches to illustrate the statistical significance of our results. The parameters chosen for **MADQN** the baselines through parameter exploration were set as follows.

The network for predator-prey environment consists of seven convolutional layers with 64 5x5 filters in each layer followed by five fully connected layers with 512 neurons each with residual connections [4] and batch normalization [5] that takes an input an 128x128x6 environment state tensor and one-hot encoded action vector (a padded 1x5 vector) and outputs a Q-function for that state-action pair.

The network for Pommerman consists of five convolutional layers with 64 3x3 filters in each layer followed by three fully connected layers with 128 neurons each with residual connections and batch normalization that takes an input an 11x11x4 environment state tensor and one-hot encoded action vector (a padded 1x6 vector) that are provided by the Pommerman environment and outputs a Q-function for that state-action pair.

For our implementation of **MADDPG** we used a multilayer perceptron (MLP) with 3 fully connected layers with 512-128-64 neurons for both actor and critic for predator-prey game and 5 fully connected layer with 128 neurons in each layer and for the critic and a 3 layer network with 128 neurons in each layer for the actor in the game of Pommerman.

Parameter exploration for **QMIX** led to the following settings for both environments. All agent networks are DQNs with a recurrent layer of a Gated Recurrent Unit (GRU [2]) with a 64-dimensional hidden state. The mixing network consists of a single hidden layer of 32 neurons. Hyper networks consists of a single hidden layer with 32 neurons with a ReLU. As in the original paper, we set learning rate linearly from 1.0 to 0.05 over first 50k time steps and than keep it constant. As we can seen from Figure 3, our MAGnet approach significantly outperforms current state-of-the-art algorithms.

4.3 MagNet network training

In both environments we first trained the graph generating network on 50,000 episodes with the same parameters and with the default AI as the decision making agents. Both predator-prey and Pommerman environments provide these default agents. After this initial training, the default AI was replaced with the learning decision making AI described in section 3. All learning graphs show the training episodes starting with this replacement.

Table 1 shows results for different MAGNet variants in terms of achieved win percentage against a default agent after 600,000 episodes in the predator-prey game and a 1,000,000 episodes in the game of Pommerman. The MAGNet variants are differing in the complexity of the approach, starting from the simplest version which takes the learned relevance graph as a direct addition to the input, to the version incorporating message generation, graph sharing, and self-attention. The table clearly shows the benefit of each extension.

Table 1: Influence of different modules on the performance of the MAGnet model. Modules are self-attention (SA), graph sharing (GS), and message generation (MG). Environments are predator-prey game (PP) and the Pommerman game (PM).

MAGnet modules			Win % PP	Win % PM
SA	GS	MG		
+	+	+	74.2 ± 1.2	76.3 ± 0.7
+	+	-	61.3 ± 0.9	56.7 ± 1.8
+	-	+	63.2 ± 1.3	62.4 ± 1.7
+	-	-	43.3 ± 1.5	54.5 ± 2.6
-	+	+	69.3 ± 1.5	67.1 ± 1.9
-	+	-	39.3 ± 2.0	52.0 ± 1.7
-	-	+	41.5 ± 1.4	45.2 ± 3.6
-	-	-	25.1 ± 2.3	32.7 ± 5.9

Each of the three extensions with their hyper-parameters are described below:

- Self-attention (**SA**). We can train Graph Generating Network (GGN) as a simple multi-layer perceptron (number of layers and neurons was varied, and a network with 3 fully connected layers 512-128-128 neurons achieved the best result) or as a self-attention transformer network (**SA**) layer [17] with default parameters.
- Graph Sharing (**GS**): relevance graphs were trained individually for agents, or in form of a shared graph for all agents on one team.
- Message Generation (**MG**): the message generation module was implemented as either a MLP or a message generation (MG) architecture, as described in Section 3.2.

4.4 Self-attention and graph sharing in training a relevance graph

We also analyzed the influence of self-attention module and graph sharing on graph generation loss function during the pretraining stage.

Figures 4 and 5 show the graph generation module loss values 6 for predator-prey and Pommerman environments respectively with and without a self-attention module and with or without graph sharing. As we can see from these figures, both self-attention and graph sharing significantly improve graph generation in terms of speed of convergence and final loss value. Furthermore, their actions are somewhat independent which is seen in that using them together gives additional improvement.

4.5 Relevance graph visualization

Figure 6 shows examples of relevance graphs with the corresponding environment state. Red vertices denote friendly team agents, purple vertices denote the agents on the opposing team, and the other vertices denote environment objects such as walls (green) and bombs (black). The lengths of edges represent their absolute

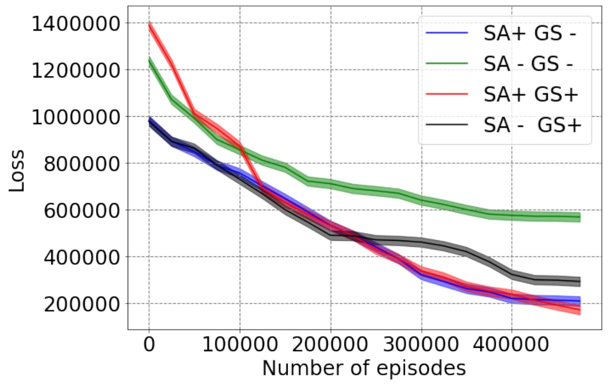


Figure 4: Loss value in training the graph generator with and without a self-attention module (SA+/-) and with or without graph sharing (GS+/-) in predator-prey environment.

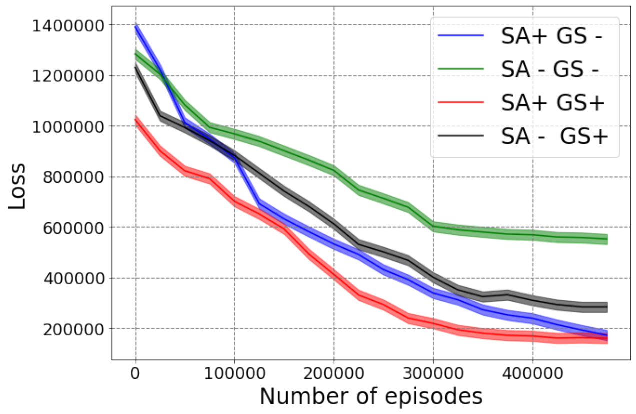


Figure 5: Loss value in training the graph generator with and without a self-attention module (SA+/-) and with or without graph sharing (GS+/-) in Pommerman environment.

weights (shorter edge equals higher weight, i.e. higher relevance). The graph in Figure 6B is shared, while Figure 6C shows individual graphs for both agents on the team.

As can be seen when comparing the individual and shared graphs, in the shared case agent 1 and agent 2 have different strategies related to the opponent agents (agents 3 and 4). Agent 4 is of relevance to agent 1 but not to agent 2. Similarly, agent 3 is of relevance to agent 2, but not to agent 1. In contrast, when considering the individual graphs, both agents 3 and 4 have the same relevance to agents 1 and 2. Furthermore, it can be seen from all graphs that different environment objects are relevant to different agents.

4.6 MAGNet parameters

We define vertex types $b(v)$ and edge types $c(e)$ in relevance graph as follows:

$b(v) \in \{0, 1, 2, 3\}$ in case of predator-prey game that corresponds to: "predator on team 1 (1, 2, 3)", "predator on team 2 (4, 5, 6)", "prey", "wall". Every edge has a type as well: $c(e) \in \{0, 1, 2\}$, that

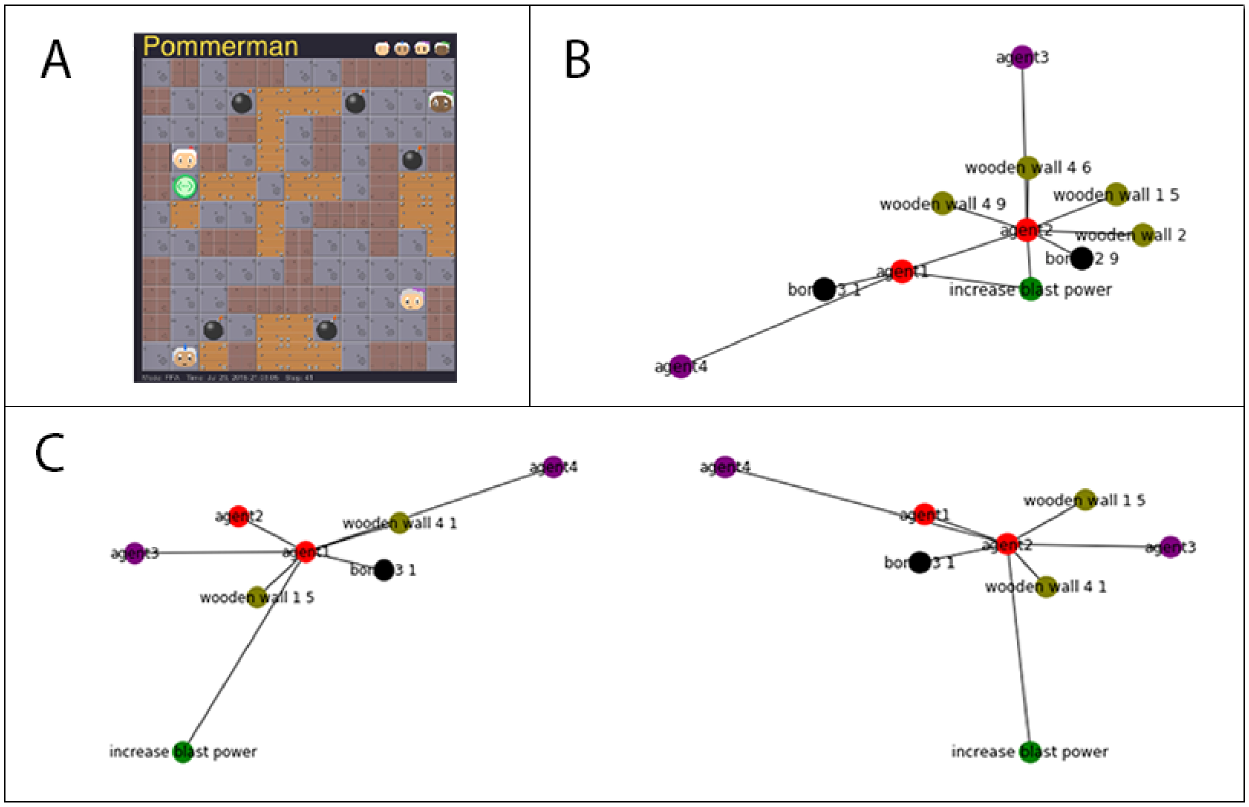


Figure 6: Visualization of the relevance graph for Pommerman game. (A) Corresponding game state. (B) Shared graph. (C) Agent-individual graphs. Vertex color corresponds to the type of the object. Olive — wooden wall, black — bomb, green — blast power bonus, red — trained agents, i.e., our team, purple — default agents, i.e., opposing team.

corresponds to “edge between predators within one team”, “edge between predators from different teams” and “edge between the predator and the object in the environment or prey”.

$b(v) \in \{0, 1, 2, 3, 4, 5, 6\}$ in case of Pommerman game that corresponds to: "ally", "enemy", "placed bomb" (about to explode), "increase kick ability", "increase blast power", "extra bomb" (can be picked up). Every edge has a type as well: $c(e) \in \{0, 1\}$, that corresponds to “edge between the agents” and “edge between the agent and the object in the environment”.

We tested the MLP and message generation network with a range of hyper-parameters. In case of predator-prey game, the MLP with 3 fully connected layers with 512-512-128 neurons, while for the message generation network 5 layers with 512-512-128-128-32 neurons was found to produce best result. For the Pommerman environment, the MLP with 3 fully connected layers 1024-256-64 neurons achieved the best result, while for the message generation network worked best with 2 layers with 128-32 neurons. In both cases 5 message passing iterations showed the best result.

Dropout layers were individually optimized by grid search in [0, 0.2, 0.4] space. We tested two convolution sizes: [3x3] and [5x5]. [5x5] convolutions showed the best result. Rectified Linear Unit (ReLU) transformation was used for all connections.

4.7 Heuristic graph only

For comparison and justification of the necessity of the graph generation stage, we carried out experiments using only heuristic data graph. In this experiment, instead of the graph generated in the graph generation stage, we set the graph weights according to empirical rule ($w^t_{(v,u)} = s(\xi_{(v,u)}), \xi_{(v,u)} \in \Xi^t$. This approach showed very poor results. In fact, the network in this case fails to train at all. This is most likely due to the fact that heuristic graph matrix is almost always filled with zeros and that in turn breaks the message passing system.

4.8 No pre-training

With regards to pre-training of the graph generating network we need to answer the following questions. First, we need to determine whether or not it is feasible to train the network without an external agent for pre-training. In other words, can we simultaneously train both the graph generating network and the decision making networks from the start. Second, we need to demonstrate whether pre-training of a graph network improves the result.

To do that, we performed experiments without the pre-training of the graph network. Figure 3 show the results of those experiments (line MAGNet-NO-PT). As can be seen, the network indeed can learn without pre-training, but pre-training significantly improves the results. This may be due to decision making error influencing the graph generator network in a negative way.

4.9 Domain specific heuristics

We also performed experiments to see whether or not additional knowledge about the environment can improve the results of our method. To incorporate this knowledge, we change the weights $s(\xi)$ in the Equation 6 from $-1/1$ for a negative/positive event to weights evaluated by human experts according to the environment.

For example, in the Pommerman environment we set $s(\xi)$ corresponding to our team agent killing an agent from the opposite team to 100, and the $s(\xi)$ corresponding to an agent picking up a bomb to 25. In the predator-prey environment, if a predator kills a prey, we set the event's weight to to 100. If a predator only wounds the prey, weight for that event is set to 50.

As we can see in Figure 3 (line MAGNet-DSH), the model that uses this domain knowledge about the environment trains faster and performs better. It is however important to note that a MAGNet network with a simple heuristic of -1/1 for negative and positive events still outperforms current state-of-the-art methods. For future research we consider creating a method for automatic assignment of the event weights.

5 CONCLUSION

In this paper we presented a novel method, MAGNet, for deep multi-agent reinforcement learning incorporating information on the relevance of other agents and environment objects to the RL agent. We also extended this basic approach with various optimizations, namely self-attention, shared relevance graphs, and message generation inspired by NerveNet. The MAGNet variants were evaluated on the popular predator-prey and Pommerman game environments, and compared to state-of-the-art MARL techniques. Our results show that MAGNet significantly outperforms all competitors.

REFERENCES

[1] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. 1995. *Dynamic programming and optimal control*. Vol. 1. Athena scientific Belmont, MA.

[2] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2015. Gated feedback recurrent neural networks. In *International Conference on Machine Learning*. 2067–2075.

[3] Maxim Egorov. 2016. Multi-agent deep reinforcement learning. (2016).

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[5] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[6] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[7] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*. 6379–6390.

[8] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*. 6379–6390.

[9] Tambet Matiisen. 2018. Pommerman baselines. https://github.com/tambetm/pommerman-baselines. (2018).

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[11] Igor Mordatch and Pieter Abbeel. 2018. Emergence of Grounded Compositional Language in Multi-Agent Populations. In *AAAI Conference on Artificial Intelligence*.

[12] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI*, Vol. 3. 1025–1032.

[13] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[14] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2018. QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485* (2018).

[15] Richard Stuart Sutton. 1984. *Temporal Credit Assignment in Reinforcement Learning*. Ph.D. Dissertation. University of Massachusetts Amherst. AAI8410337.

[16] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*. 1057–1063.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[18] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. 2018. Nervenet: Learning structured policy with graph neural networks. *Proceedings of the International Conference on Learning Representations* (2018).