Improving Multiagent Actor-Critic Architectures, with Opponent Approximation and Dropout for Control

Gabor Paczolay, Istvan Harmati

Department of Control Engineering, Budapest University of Technology and Economics, Magyar tudósok krt. 2, H-1117 Budapest, Hungary
paczolay@iit.bme.hu; harmati@iit.bme.hu

Abstract: In the domain of reinforcement learning, solution proposals to multiagent problems are evolving. We propose a new algorithm, MADDPGX, to handle the problem of higher uncertainty created by other agents' actions by an enemy actor approximator, and we investigate the most efficient techniques of estimations. This approximation works using a neural network, which has the input of the state and the output as the action (probably preferred by the enemy agent). We also experimented with dropout, a tool commonly used for neural networks, but has not been used efficiently for reinforcement learning until now. We have also found that in multiagent actor-critic scenarios, it can improve overall performance. Generally, our contribution is the use of action approximation of adversaries and the dropout usage in actor-critic systems, with a conclusion that the newly proposed methods will perform better in zero-sum multi-agent robot system scenarios. The experiments were conducted in a multiagent predator-prey environment.

Keywords: reinforcement learning; multiagent learning; dropout; MADDPG; MADDPGX

1 Introduction

Reinforcement learning is a fast emerging field in the domain of artificial intelligence. Due to the increase of the computing powers, it is becoming a reality to solve more complex control problems efficiently, which would otherwise require a very fragile and precise mathematical model. However, having multiple agents in the environment makes it much harder to solve the task as each agent introduces a high uncertainty related to their specific policy, because as each agent performs their own action, the learning algorithms generally rely on that specific choice, and it is harder to deal with the fact that those agents will later perform another action in the same situation due to their learning having advanced. This problem is addressed by the sub-domain of multiagent reinforcement learning, which considers the multitude of the agents and either tries to pay attention to the opponents' actions (this branch is called as a competitive multi-agent scenario) or tries to figure out an action ensemble which leads to the most reward (this is called as a cooperative multi-agent scenario). Our proposed algorithm, that is presented in this paper was tested on both the cooperative and the competitive elements of the environment that we use, and as it will later be clear due to the results, it behaves the best when it is used on the competitive domain due to the lower uncertainties in this scenario.

Whenever we, humans, try to figure out the best action that would contradict our opponent's hostile behavior, we can ask ourselves: "What would my opponent do in this situation?" Then, we take this estimation and form our next action, based on it, deciding the action which yields us the best returns. This method is what we use in our proposed algorithm: We estimate the opponents' state-action assignment (in actor-critic algorithms, it is the actor) and train our actor such that it takes this estimation as part of its input. The critic is also trained based on the assumptions made by the approximating model.

Later in this paper we examine the effects of applying dropout in the field of reinforcement learning. It is actively used in general deep learning due to its effect of decreasing the possibility of overfitting, but up until now, its usage was opposed for reinforcement learning, especially because higher variance is not required in single-agent reinforcement learning. The dropout effects were not tested in the domain of multiagent reinforcement learning, we address this problem in our paper. Our results show that dropout has its certain place in this field as well, and we try to give as precise information on its possible applications.

Littman [12] was the first to use Minimax-Q, a zero-sum multiagent reinforcement learning algorithm and he applied it to a simple robotics soccer game environment, later Hu and Wellmann [11] brought the Nash-Q algorithm to the world and they utilized in a small grid-world environment to show the algorithm's achievements. Bowling [4] sped up the training process while ensuring convergence by varying the learning rate. Later, he applied his proposal, the Win or Learn Fast methodology to an algorithm based on an actor-critic system to have better multiagent performance [5].

Reinforcement learning's huge leap forward happened when convergence of deep neural networks was improved and one could use them in these scenarios. Mnih et al. [15] invented the Deep Q-Network (DQN) algorithm and its invention was made to play Atari games with success by multiple frame feeds and using experience replay for better chance for convergence. Then, researchers combined deep reinforcement learning and multi-agent systems, its most simple form is called independent multi-agent reinforcement learning. Foerster et al. [8] experimented with stabilizing experience replay, a buffer of states, actions, rewards and next states to improve convergence, for independent Q-learning (IQL) by utilizing so-called fingerprints. Researchers have also made multiple advancements in the field of centralized learning and decentralized execution as well, for example, when Foerster et al. [7] created Counterfactual Multi-Agent Policy Gradients, where the problem of multi-agent credit assignment was solved by training agent policies by comparing its actions to other actions it could have taken. Sunehag et al. [21] used Value Decomposition Networks with a common reward and Q function decomposition. Lowe et al. [14] made improvements to the Deep Deterministic Policy Gradient algorithm by a changing the critic to contain all actions of all of the agents, this change, thus the algorithm would be able to learn multi-agent environments with better efficiency. Shihui et al. [19] made advancements to the previously described MADDPG algorithm by altering it to achieve better performance in environments with zero-sum payoff by using a method based on the Minimax-Q learning algorithm. Davies [6] applied a Model Opponent Learning algorithm to the previously mentioned MADDPG method. Casgrain et al. [3] modified Mnih's Deep Q-network algorithm by using methods based on Nash equilibria, which made it able to solve multi-agent environments.

To inspect the performance of their algorithms, researchers have also made several benchmarks, even for multiagent environments. Vinyals et al. [24] took the game of Starcraft II and made it to be a learning environment for multi-agent scenarios. Samvelyan et al. [18] also utilized Starcraft as a multiagent benchmark environment, but in this case, the aim was micromanagement, controlling each unit separately. Liu et al. [13] created a multiagent, continuous simulated physics-based soccer environment. Bard et al. [2] made huge advancements of multi-agent learning with the Hanabi game benchmark, where the agents have to cooperate with each other in partially observable scenarios.

Other kinds of control and learning algorithms may also be considered, too. Babqi et al. [1] compared MPC and PI control for power electronic devices. Hakan et al. [9] created a test platform for vertical drones. Preitl et al. [17] utilized quadratic programming in fuzzy systems. Precup et al. [16] used generic 2DOF linear and fuzzy controllers for integral processes. Hemza et al. [10] utilized fixed point iteration in single variable second order systems, while Zamfirache et al. [25] used an actor-critic RL solution for servo systems. Our method is more robust than the methods listed in the paragraph.

Our work also builds upon the MADDPG algorithm and takes a similar route to the Model Opponent Learning algorithm, but relies more upon the action approximation of the actors using a neural network where the inputs are the states and the outputs are the agent's most probable preferred actions. This is our main contribution in this paper, and it can improve the performance of agents in competitive environments. Also, we dive deeper into the possible architectures and algorithm realizations of the action approximation. In addition, we approach the usage of dropout in ways that were not used before with as precise information on its usage as we can. This contribution utilizes a method previously, solely, used in other domains of deep learning and proves to be useful in deep reinforcement learning as well.

Our new methodologies have their certain positions in engineering applications as well. The opponent approximator method is usable for zero-sum multi-agent robot systems, such as collision avoidance in aircraft control, where the other planes can be considered as "enemies", and the objective is to minimize the time of arrival. It is also in Unmanned Aerial Vehicles of other types, for guarding an area against intruders. Our dropout contribution is also utile under these conditions, but it can also be useful in any multi-agent scenarios, control problems included. The new ideas of this paper are the MADDPGX algorithm and the dropout utilization.

In this work, first we take a look at the theoretical background. Later, we show the used benchmarks for our tests, then we explain our experiments and the results obtained by them. At the end, we conclude our work and give suggestions for future research possibilities.

2 Theoretical Background

2.1 Markov Decision Processes

Markov Decision Process is a discrete-time process for decision making modeling. Figure 2 shows the basics of this mathematical framework. It has the following elements: states of the whole environment, selectable actions by the agent, transition probabilities between the states with respect to the actions and rewards given to the agents. [1]. Every timestep the process has the same method: starting at a specific state (s), it has an available action space. From that, the agent selects and action (a), and based on the state-action pair, it will receive a reward (r), then it arrives in a new state (s'). A stochastic process is called Markovian if:

$$P(\boldsymbol{a^t} = \boldsymbol{a}|\boldsymbol{s^t}, \boldsymbol{a^{t-1}}, \dots, \boldsymbol{s^0}, \boldsymbol{a^0}) = P(\boldsymbol{a^t} = \boldsymbol{a}|\boldsymbol{s^t}) \tag{1}$$

which can be described such as state transitions depend only on the last state and the currently selected action. Due to this, only these two are important in the decision of the following state.

The policy, a state-action assignment, is very important in Markov Decision Processes. Agents are trying to seek for an optimal one which can maximize the return, the sum of discounted expected rewards. Discount in this case means that agents prefer an immediate reward against one in the future, thus, a coefficient determines how better a reward is with respect to the same amount of reward in the next state. To find a general solution for the policy, one has to seek for a fixed point of the Bellman equation via iterative search. The Bellman equation has the following form:

$$v(\boldsymbol{s}, \pi^*) = max_a(r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \textstyle\sum_{s'} \boldsymbol{p}(\boldsymbol{s'}|\boldsymbol{s}, \boldsymbol{a}) v(\boldsymbol{s'}, \pi^*)) \tag{2}$$

where $r(s, a)$ is the reward gained from selecting action a in state s, γ is the coefficient deciding how much more important are rewards of the present in comparison with the future rewards, and $\boldsymbol{p}(s'|s, a)$ is the transition probability function. It is concluded from this equation that if the agent is familiar with the dynamics of the environment, it can find the optimal values.

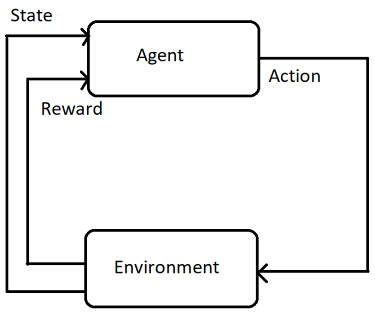

Figure 1
Markov Decision Process

2.2 Reinforcement Learning

Reinforcement learning is a subproblem of Markov Decision Process whenever the either rewards or the state transition probabilities are not known. In this case, one has to seek to create a proper model of the environment by trying specific actions and learning from the rewards and the errors.

There are two main kinds of reinforcement learning: **value-based** and **policy-based** reinforcement learning.

In the scenario of value-based reinforcement learning, agents are rendering values to the states or to the actions that are selectable from specific states. The aforementioned values coincide with the expected reward whenever the agent selects a certain action in a specific state.

The most generally utilized kind of value-based reinforcement learning is called **Q-learning**. It is an algorithm based on action-values, so these values are rendered to all of the state-action pairs of the environment. These values, called Q-values, correspond to the value equal to the amount of reward one could get by taking a certain action in a certain state. The equation for the update of the Q-values is the following:

$$Q(\boldsymbol{s}', \boldsymbol{a}) \leftarrow (1-\alpha) \cdot Q(\boldsymbol{s}, \boldsymbol{a}) + \alpha \cdot (r + \gamma \cdot \max_{\boldsymbol{a}'} Q(s', a')) \quad (3)$$

where α corresponds to the learning rate and γ corresponds to the discount for the reward [17]. is an off-policy TD (temporal difference) control algorithm. The policy is to choose the action that would currently maximize the Q-function in the present state.

In policy-based reinforcement learning, the actions are a parametric function of the state. Of this type, the most common technique is **policy gradient** [18], when the policy is given by the parameters θ, and the agent tries to reach the maximum expected reward for a specific trajectory $r(\tau)$. This gives us that the cost function based on the parameters is this equation:

$$J(\theta) = E_{\pi_\theta}[r(\tau)] \tag{4}$$

The tuning of the parameters is performed with respect to the gradient of the cost function:

$$\boldsymbol{\theta}_{k+1]} = \boldsymbol{\theta}_t + \alpha \Delta J(\boldsymbol{\theta_t}) \tag{5}$$

Policy-based methods have their own advantages and disadvantages. They are able to map environments with great or continuous action spaces efficiently. Value-based methods cannot map huge action spaces due to the increasing value-table size. Policy-based methods are also efficient for scenarios with randomness. On the other hand, they are more prone to get stuck in a local maximum instead of finding the optimal policy.

2.3 Multi-Agent Systems, Markov Games

To fully understand the Markov games, one has to talk about the stochastic framework of matrix games. In that, first each agent takes an action, then they get their current reward. This reward is based on all of the agents' action. These scenarios can be described in a matrix form, where one agent selects the row based on its action, and the other selects the column, and the intersection contains the reward for each agent. These games can only contain a single state.

A multi-state augmentation of matrix games is called Markov game, or with another wording, Stochastic game. Another approach can be that Markov games are a multi-agent extension of Markov Decision Processes. In these games, all of the states contain a specific matrix called payoff matrix, which has the same form as the matrix in Matrix games. Thus, the reward is decided by the mutual action of the agents, and this can be also said about the next state. A game is said to be Markovian if it adheres to the following:

$$P(a_i^t = \boldsymbol{a}_i | \boldsymbol{s^t}, \boldsymbol{a_i^{t-1}}, \dots, \boldsymbol{s^0}, \boldsymbol{a_i^0}) = P(\boldsymbol{a_i^t} = \boldsymbol{a}_i | \boldsymbol{s^t}) \tag{6}$$

which means that the upcoming state is solely dependent on the previous state and the present actions selected by all of the agents.

2.4 Deep Reinforcement Learning

Deep reinforcement leaning is a subclass of reinforcement learning which is aided by a neural network.

An artificial neural network is a subset of machine learning. In this case, the function approximation is performed by a network of (even huge amounts of) artificial neurons. Artificial neurons resemble biological neurons, and their behavior is determined by the following equation:

$$y = Act(\sum \boldsymbol{wx} + b) \tag{7}$$

In this equation, x corresponds to the input vector. w is the weight vector, which is taken with a by-element product of the previously mentioned input vector. b is called bias, as it is a variable constant added to the other inputs. This can also be described as a regular weight connected to the input of the constant 1. $Act()$ corresponds to the activation function, which ensures that the system becomes nonlinear by introducing nonlinearity, thus letting otherwise linear systems predict nonlinear relations. The tuning of the parameters, w and b, are performed by backpropagation, where the partial derivative errors with respect to the inputs are calculated starting from the final error, and then this error is propagated backwards through previous layers up until the input vector.

One must talk about the difference between traditional reinforcement learning and deep reinforcement learning. The latter has a considerable number of advantages, such as abandoning the state table by approximating the states with neural networks, which is more robust than general linear approximators. This allows us to map scenarios with huge or even continuous state spaces without worrying about the memory need of the whole state space. On the other hand, deep reinforcement learning converges in less situations, thus a multitude of improvements have been made to ensure convergence of the learning in more scenarios.

2.5 Actor-Critic

An amalgamation of value-based and policy-based reinforcement learning is called an actor-critic algorithm. This algorithm contains two distinct neural networks: The first is called Critic, which resembles value-based reinforcement learning, by approximating a value function, and the second is called the Actor, which, as in policy-based reinforcement learning, renders an action to the present state. The latter network is tuned, based on the direction suggested by the Critic. The actor follows an approximate policy gradient as:

$$\begin{gathered} \nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a)Q_w(\boldsymbol{s},\boldsymbol{a})] \\ \Delta\theta = \alpha\nabla_\theta log\pi_\theta(s,a)Q_w(\boldsymbol{s},\boldsymbol{a}) \end{gathered} \tag{8}$$

The latter equation is the more critical from a practical point of view, as it gives us the direction of the parameter updates. In that equation, α corresponds to the learning rate, a scalar which determines the amount of parameter change. The other parts show that the direction is given by the gradient of the log-policy times the value function.

The policy gradient approximation reduces efficiency in one part due to the bias introduced, and this bias can make our learning fail. The value function approximation has to be chosen with great care to avoid this bias.

In comparison with regular deep reinforcement algorithms, actor-critic algorithms achieve better performance. By the utilization of the critic network, the system can avoid being stuck in a local extremum, and by the usage of the actor network, better convergence can be achieved in addition to mapping systems with huge or even continuous action spaces.

2.6 MADDPG Algorithm

MADDPG, Multiagent Deep Deterministic Policy Gradients is a multiagent extension to the DDPG (Deep Deterministic Policy Gradients) algorithm, which is an actor-critic algorithm for continuous action spaces.

First of all, both MADDPG and DDPG use an experience replay buffer to recall previous state-action-reward-next state tuples. It stores and recalls the tuples (x^j, a^j, r^j, x'^j) . By its utilization, the system will utilize previous experience more efficiently as it will learn the experience multiple times, as well as it will converge with a higher success rate due to access not only to the latest experiences.

Let's take a closer look at the training of the actor and critic networks. The critic is updated by minimizing the loss as here:

$$\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^\mu(x^j, a_1^j, \ldots, a_N^j)\right)^2 \tag{9}$$

Where,

$$y^j = r_i^j + \gamma Q_i^{\mu}(x'^j, a_{1'}, \ldots a_{N'})|_{a_{k'}=\mu_{k'}(o_k^j)} \tag{10}$$

This latter equation shows that for the next actions, we use the target actors to compute them. Meanwhile, the actor is updated using the following sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i}Q_i^\mu(x^j, a_1^j, \ldots, a_i, \ldots, a_N^j)|_{a_i=\mu_i(o_i^j)} \tag{11}$$

In this equation, we see that we take the gradient with respect to the Actor's parameters with the help of a central critic.

2.7 Dropout

Dropout is mostly used to reduce overfitting in deep neural networks. Overfitting is a case when the function approximator (in this case, the neural network) renders the parameters to the training data well, but misses generalization, thus the approximator cannot be utilized for any useful task (apart from the occasions where input values are a subset of the training values). [19] [21] If it is applied to a neural network (or rather, to a layer), then the network's (or layer's) neurons are only present with a p probability during the training process. In other words, at each training stage, individual neurons are either eliminated from the net with a probability $1 - p$ or kept with a probability p, such that only a reduced network is left. In the testing/inference process, however, all neurons are present. This training method makes the training process noisy, forcing nodes to probabilistically take on more or less responsibility for the inputs. As the network is not used fully during the training process, instead, a subset of the layers is used, a wider network is required for layers with dropout that for ones without, for the same level of representation. The most used, and usually the most efficient probability rate for the dropout is 0.5.

3 Experiments

As a benchmark, we used the Multiagent Particle Environments (MPE) library. It is a multiagent environment ensemble with several environments, which are either communication-based or are about circle-shaped agents moving in a continuous 2D world, trying to accomplish specific tasks. It is written in Python, and its interface resembles (and is built upon) the quasi-standard interface of OpenAI's Gym environments, which makes connecting agents easier. The differences between its interface and Gym's are due to the fact that Gym does not support multiagent environments up until the writing of the paper, thus the multitude of observations and actions are listed in a unique but easily comprehensible way. From this environment ensemble of MPE, we used the "simple-tag" environment. This is a predator-prey (or pursuit-evasion) environment with 3 predators and one prey, and the latter is faster (and also has better acceleration). There are also obstacles on the plane that cannot be crossed. The agent movement behavior is described as follows:

$$F_i = m_i \cdot a_i * u + z \tag{12}$$

where m_i is the agent mass, a_i is the agent acceleration (if it exists in the scenario, otherwise it is strictly 1), u is the agent action and z is the noise (if exists). Then, from the forces, a velocity is calculated:

$$v_i = v_i + \frac{F_i \cdot (1-d)}{m_i} \cdot dt \tag{13}$$

where d is the damping. The here unnecessary product and division by the mass is applied due to the possible addition of environmental forces in other scenarios. Finally, the position is calculated such as:

$$x_i = x_i + v_i \cdot dt \tag{14}$$

The predators have to catch the prey agent, and they get a positive reward for catching the agent, while being caught, the prey gets a negative reward. For evading the problem of sparse rewards, which means that the received rewards are present only on a small subset of the environment steps, reward shaping was turned on. In this case, the prey gets bigger rewards for being as distant from the predators as possible, and the predators get negative reward based on the minimum distance to the prey agent (thus the reward is relatively bigger if one agent is closer to the prey). The exact reward function for the prey is as follows:

$$r = c \cdot (-10) + 0.1\sqrt{\sum (x_{prey} - x_{pred})^2} \tag{15}$$

where r is the reward, c is the collision boolean, and x are the positions. The reward function for the predator is the following:

$$r = c \cdot 10 - 0.1 \cdot min_{a \in prey}\sqrt{\sum (x_a - x_{pred})^2} \tag{16}$$

In our environment settings, as there is only one prey, the minimizing part disappears and the latter parts of both equations become the same. The only difference between the two-reward function is the c part: while for the preys it means a boolean (0 or 1), for the predators it counts all collision between the predators and the prey, thus reaching the prey with multiple agents yields more reward.

The episodes are terminated after 25 steps, this number seemed to be well balanced regarding the training being meaningful and the episode length not being too long for the training process and other calculations. With this length we also evaded that the episodes would stall, with the prey agent getting stuck caught by predators for long times, modifying the rewards by a big amount. A sample picture of the environment can be seen on Figure 3, where the red circles are the predators, the green one is the prey, and the bigger black circles are the obstacles that cannot be traversed. The goal of the environment for the predator is to minimize the time in which it arrives the closest to the prey, and for the prey the goal is to maximize the distance between itself and the predators.

First, we tried to improve the training by approximating other agents' behavior. In this case, the agent observations are augmented by the most probable action that some other agents (the enemies or all other agents) would take. The most probable action for each opponent is approximated by a neural network, which takes the observation as input and outputs a value (the action) with a dimension equal to its action space. The training of this neural network consists of applying the agents' selected action to the present observation.

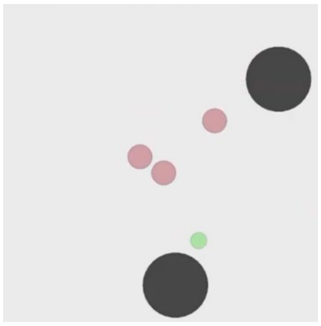

Figure 3
Predator-Prey environment of the Multiagent Particle Environments library

This training can happen online, at each timestep, or from the experience replay in conjunction with the training of the actors and critics. As the actor now requires the most probable actions as well, these have to be computed for the training process. The critic training was not modified from the MADDPG training for this algorithm.

Algorithm 1 shows the MADDPGX algorithm. As it can be seen, it is mostly based on the MADDPG algorithm, with some differences. The most important difference is that the selected actions are not calculated as $a = \mu_i(x)$ but $a = \mu_i(x, m)$, where m is the ensemble of the approximated actions and is calculated for all i as $m_i = N_{e_i}(x)$. The other difference between our algorithm and the original is also related to the former, it is that the actor is updated with the calculation of the previous $N_{e_i}(x)$ values. Inbetween, the N_{e_i} networks are also updated such that they would approximate the enemy actions based on the corresponding states. The training of the neural networks happens independently. The control law is given just as in MADDPG, but with the opponents added:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j, N_{e_i}(o_i^j))\nabla_{a_i}Q_i^{\mu}(x^j, a_1^j, \dots, a_i, \dots, a_N^j)|_{a_i=\mu_i(o_i^j,, N_{e_i}(o_i^j))} \quad (31)$$

The computational complexity of our algorithm is:

$$O(4n^4)$$

Where, n is the number of neurons in the network. The stability of the controller is the same as the stability of MADDPG, so it can be described as stable but not in all situations. The system converges, but the optimality is not guaranteed for MARL situations, as in all other similar scenarios.

The algorithm was created with Python and PyTorch, and the experiments were run on a Google Colab instance (due to the varying speed of the instances, the average running times could not be usefully extracted for the experiments, thus this information is rather omitted). The networks consisted of three layers: two hidden layers and one output layer. The hidden layers' dimension was 64 neurons. This level of network complexity seemed to be enough for learning the environments for all of the algorithms. The activation function of the inner neurons was ReLU (Rectified Linear Unit), while for the output, a tanh activation function was applied. As noted, the actors' dimension was equal to the observation space dimension plus the tracked agents' action space dimension, and the output dimension was equal to the agent's action space dimension. The action approximators' input dimension is equal to the tracked agent's observation space and its output is the tracked agent's action space. The critics were the same as in the MADDPG algorithm, with the input dimension being equal to the sum of all agents' action and observation space dimension, and the output is 1 (the Q value). An Adam optimizer was used as all of the neural network optimizers. The critic loss was a mean squared error loss and the actor loss was the same as in the base MADDPG algorithm, with the exception that for the actor loss, the approximations are needed to be made. The action approximator loss was dependent on the type of the action space: for discrete action spaces, cross-entropy loss was used, while for continuous action spaces, a mean squared error was used. In both cases, the losses consider the difference between the approximated and the taken action. Throughout all experiments, the learning rate as 0.01, the batch size was 1024 and τ (the target network coefficient) was 0.01. γ was set to be 0.95.These values are selected to provide a good balance between convergence and learning speed.

In a later experiment, we checked how applying dropout would affect the performance of the agents. As it applies more variance, the basic idea to check it was the expectation of finding more rewarding Nash equilibria. We experimented with the application of dropout on the actor and the critic network separately, and have seen which one is capable of improving the performance score. For the experiments, the dropout layers of 0.5 probability were applied after the first and second fully connected layers of the networks.

```
**Initialize Models**: $\mu_i$, $C_i$, $N_{e_i}$
**for** episode = 1 to $M$ **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial state $\mathbf{x}$
    **for** $t = 1\ to\ max - episode - length$ **do**
        For each agent $i$, take the subset of opponent agents $e_i$
        calculate $m_i = N_{e_i}(o_i)$
            $m_i$ is the most probable action of the enemies
            $N_{e_i}(o_i)$ is the actor approximator with the observation as input and most probable action as
            output
        for each agent $i$, select action $a_i = \mu_{\theta_i}(o_i, m_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
        Execute actions $a = (a_1, \dots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
        Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
        $\mathbf{x} \leftarrow \mathbf{x}'$
        **for** agent $i = 1\ to\ N$ **do**
            Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
            Set $y^j = r_i^j + \gamma\, Q_i^{\mu'}(\mathbf{x}'^j, a_{1'}, \dots, a_{N'})|_{a_{k'}=\mu'_k(o_k^j)}$
            Update $N_{e_i}$ networks:
            Loss is MSE for continuous actions, Cross-Entropy for discrete
            For all $e_i$, training input is $x^j$, training output is $a^j$
            Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\mu}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
            Update actor using the sampled policy gradient:
            $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j, N_{e_i}(o_i^j))\nabla_{a_i} Q_i^{\mu}(x^j, a_1^j, \ldots, a_i, \ldots, a_N^j)|_{a_i=\mu_i(o_i^j, N_{e_i}(o_i^j))}$
        end for
        Update target network parameters for each agent $i$:
        $\theta_{i'} \leftarrow \tau\theta_i + (1-\tau)\theta_{i'}$
    end for
end for
```

Algorithm 1
MADDPGX

Figures 4 and 5 show some examples of the environment working. In all of them, we can see the fleeing agent (blue) is trying to maneuver away from the red agents. The examples were taken after the 23000th episode to give the agents enough time to learn the environment.

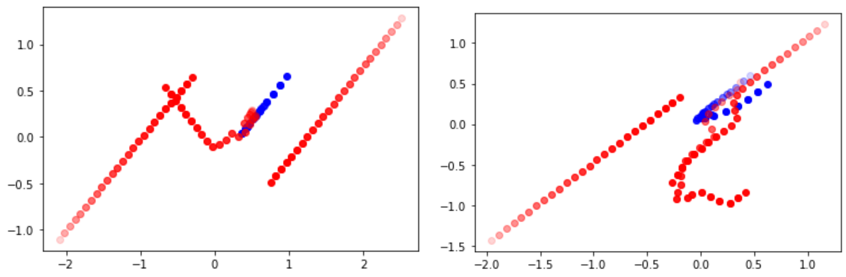

Figure 4

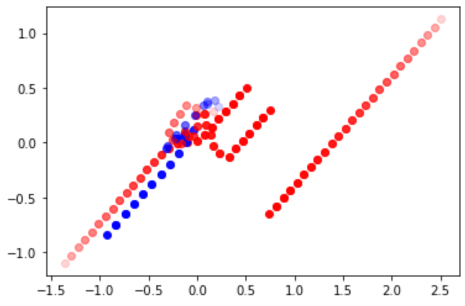

Figure 5

In all of the three examples, we see that both the predator and the prey agents have successfully learned the environment, as the predators are chasing the prey, and the prey is trying to flee from the predators. Figure 5 shows that two predators are trying to surround the prey, and a third one is chasing it directly. After the latter being close, the prey finds a way to free from the agents. Figure 6 shows that now two agents are chasing the prey directly, and the prey had to change its direction towards its starting point to evade the agents. In the third example, Figure 7, the prey and the two predator agents are "fighting", but then the prey agent successfully escapes.

4 Results

All experiments were run for 25000 episodes, this seemed to be enough for learning the environment and the opponent and the results did not change significantly by further increasing the episode number. Then, we extracted the mean rewards for all agents and episodes. For the easier digestion of the huge dataset (or in other words, to extract useful data out of the mean episode rewards), we introduced two baselines to be compared: One is the number of episodes where the prey agent's mean reward was above zero, and the other is the number of episodes where the prey agent's mean reward is higher than the sum of the other three agents' mean reward. These baselines were chosen arbitrarily, but they still represent the performance of the algorithms adequately.

Table 0 shows how many times the prey agent got a mean reward above zero of the 25000 episodes. Table 1 shows the number of occurrences when the prey mean reward was higher than the reward sum of the three predators.

First, let's check how the action approximation performs compared to the MADDPG vs MADDPG contests. Out of 8 different situations, 3 ended with univocal dominance for our algorithm, 4 was contested (in a sense that either the predator or the prey side was better with our algorithm, but the other side was worse after that) and one ended with definitely worse results than the original algorithms. This result clearly shows that our addition to the MADDPG algorithm improved the performance of the agents.

Now, let's check the different versions of the action approximator algorithm. For these experiments, the results can be seen in Table 4 and 5, where the former expresses the number of episodes where the prey's reward was greater than zero, and in the latter table, the predator sum is compared, just as in Table 1. It can be seen that from all the 6 comparisons, at 4 times the one where the approximator is updated online, right after the reception of the opponents' selected actions, supersedes the performance of the agent where the action approximator is updated from the experience replay, together with the learning of the actor and the critic in the general MADDPG algorithm. It can also be seen that for some reason, when the action approximator is used not only for enemies, but for friendly agents as well, the performance of the system severely drops. Thus, our algorithm is better to be used only for modeling the enemies, not for modeling all other agents and trying to find a friendly Nash equilibrium using the approximators. This can be a result of divergence due to the higher variance of the system caused by the approximators, as it is much harder to find an equilibrium when all of the models are approximating each other.

The effect of the dropout can also be examined. First, let's look at the case when dropout is added to the Actor network, its results can be seen in Table 0 and 1. Of all the 16 comparisons, according to Table 0, in 14 cases the network with dropout superseded the one without it, meanwhile according to Table 1, 13 cases were better with dropout than without it. This shows that applying dropout to the actor network can generally yield better performance in multi-agent scenarios.

Table 1
Number of episodes where the prey's mean reward was above zero. The first line shows in which agent(s) our algorithms were used, and which subtype was implemented. Pred means predator, and A means Actor.

No.	Pred Dropout A	Pred	Normal	Prey	Prey Dropout A
1	31	43	70	11502	12854
2	11	11	20	2426	3352
3	5590	21685	49	26	27
4	15690	17882	21	27	23
5	18052	20519	20575	20	33
6	38	44	2314	22	40
7	86	17	436	45	51
8	28	39	1646	7039	9959

Table 2
Number of episodes where the prey's mean reward was above the sum of the predators' mean reward. The first line shows in which agent(s) our algorithms were used, and which subtype was implemented. Pred means predator, and A means Actor.

No.	Pred Dropout A	Pred	Normal	Prey	Prey Dropout A
1	2276	2792	3405	12729	14463
2	1652	1651	1675	4128	4415
3	8307	22458	2903	1871	1820
4	17582	19461	1928	1545	1039
5	19024	21368	20827	1356	2298
6	2198	2567	4444	1894	2580
7	2939	1569	5003	2298	2299
8	2196	2603	2861	10600	11082

We applied dropout to the critic network as well, and the results can be seen in Tables 2 and 3. Of the 16 cases, only in 7 cases this method happened to be better than the system without it. Also, when the network with critic dropout was better, usually the score did not improve much. However, in one case (No. 3 while our algorithm is on the predator side) the score improved significantly, so there can be some cases when applying dropout to the critic can be beneficial. The utility of the dropout possibilities in the critic network could be further examined.

The results show that our contributions are an improvement upon the previously available methods.

Table 3

Number of episodes where the prey's mean reward was above zero. The first line shows in which agent(s) our algorithms were used, and which subtype was implemented. Pred means predator, and C means Critic.

No.	**Pred Dropout C**	**Pred**	**Normal**	**Prey**	**Prey Dropout C**
1	4088	43	70	11502	60
2	32	11	20	2426	14
3	46	21685	49	26	35
4	20065	17882	21	27	43
5	21036	20519	20575	20	34
6	38	44	2314	22	39
7	18920	17	436	45	47
8	4285	39	1646	7039	32

Table 4

Number of episodes where the prey's mean reward was above the sum of the predators' reward. The first line shows in which agent(s) our algorithms were used, and which subtype was implemented. Pred means predator, and C means Critic

No.	**Pred Dropout C**	**Pred**	**Normal**	**Prey**	**Prey Dropout C**
1	7639	2792	3405	12729	1602
2	2661	1651	1675	4128	1142
3	2267	22458	2903	1871	1918
4	21040	19461	1928	1545	1971
5	21812	21368	20827	1356	2221
6	2542	2567	4444	1894	2406
7	19913	1569	5003	2298	2344
8	7684	2603	2861	10600	1380

Table 5
Number of episodes where the prey's mean reward was above zero. The first line shows in which agent(s) our algorithms were used, and which subtype was implemented. Pred means predator, V means that the version where the update is done in each step is implemented, and all means that the approximation for cooperative agents is also implemented

No.	Pred V All	Pred V	Pred	Normal	Prey	Prey V
1	17936	46	43	70	11502	19441
2	18858	11	32	20	2426	38
3		18799	21685	49	26	27

Table 6
Number of episodes where the prey's mean reward was above the sum of the predators' mean reward. The first line shows in which agent(s) our algorithms were used, and which subtype was implemented. Pred means predator, V means that the version where the update is done in each step is implemented, and all means that the approximation for cooperative agents is also implemented.

No.	Pred V All	Pred V	Pred	Normal	Prey	Prey V
1	19606	2614	2792	3405	12729	20741
2	20594	1652	2661	1675	4128	2775
3		19878	22458	2903	1871	1907

Conclusions and Future Work

According to our results, our proposals have clear benefits compared to the systems without it. Approximation of the actors of other agents visibly improves the performance, however, it is only beneficial in the system when it is used for competitive elements of the environments. Instead, it is rather advised to be omitted when using it for cooperative elements, or in other words, it is better to only approximate the enemies' behavior rather than using it for the approximation of the friendly agents' actors. Dropout also has clear benefits, but only when it is used for the actor, for the critic it is only really beneficial in some rare cases.

Regarding the actor approximation, the proposed idea is favorable to other algorithms due to the fact that with an insignificant loss on data efficiency, one can receive much better performance on the long run. As a comparison of the proposed systems and other types of control algorithms, actor-critic reinforcement learning algorithms are more versatile and robust than hand-made and other data-driven solutions.

There are still some challenges to explore in the future that we leave for upcoming researches. Further testing of dropout in the world of multi-agent reinforcement learning still awaits us to do. It is surely worth more effort to check whether the dropout on the critic is unusable for all of the possible situations, or as our results stated, which are the situations where the critic dropout is also utile. In addition, further testing of the dropout rates can be interesting. Also, the efficiency of the algorithm can be examined when the enemies' observations are not available, even in a Partially Observable Markov Decision Process. This could be done, for example, with the help of Long Short-Term Memory (LSTM) systems, where the neural network preserves a state between timesteps, thus it is able to have a memory. Also, a variable learning rate could be utilized for further improvement of the learning systems, such as Bowling's Win or Learn Fast (WoLF) [3] method. Its usage for deep reinforcement learning could be investigated.

Acknowledgements

The research reported in this paper is part of project no. BME-NVA-02, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021 funding scheme. Support by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory.

References

[1] A. J. Babqi, B. Alamri: A comprehensive comparison between finite control set model predictive control and classical proportional-integral control for grid-tied power electronics devices. Acta Polytechnica Hungarica, Volume 18, Issue 7, 07-2021

[2] N. Bard, J. N. Foerster, S. Chandar, N. Burch, M. Lanctot, H. F. Song, E. Parisotto, V. Dumoulin, S. Moitra, E. Hughes, I. Dunning, S. Mourad, H. Larochelle, M. G. Bellemare, M. Bowling: The hanabi challenge: A new frontier for ai research, Artificial Intelligence, 2019

[3] P. Casgrain, B. Ning, S. Jaimungal: Deep q-learning for nash equilibria: Nash-dqn, arXiv preprint arXiv:1904.10554, 2019

[4] M. Bowling and M. Veloso: Multiagent learning using a variable learning rate, Artificial Intelligence, (136):215-250, 2002

[5] M. H. Bowling, M. M. Veloso: Simultaneous adversarial multi-robot learning, In IJCAI, pp. 699-704, 2003

[6] I. Davies, Z. Tian, J. Wang: Learning to model opponent learning, arXiv preprint arXiv:2006.03923, 2020

[7] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, S. Whiteson: Counterfactual multi-agent policy gradients, Proceedings of the AAAI Conference on Artificial Intelligence, 32(1), 2017

[8] J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. S. Torr, P. Kohli, S. Whiteson: Stabilising experience replay for deep multi-agent reinforcement learning, International conference on machine learning, pp. 1146-1155, 2017

[9] Ü. Hakan, Ö. Irfan, Y. Uğur, K. Metin: Test Platform and Graphical User Interface Design for Vertical Take-Off and Landing Drones. Romanian Journal of Information Science and Technology, 25, pp. 350-367, 2022

[10] R. Hemza, J. Tar:. Multiple Components Fixed Point Iteration in the Adaptive Control of Single Variable 2nd Order Systems. Acta Polytechnica Hungarica, Volume 18, pp. 69-86, 09-2021

[11] J. Hu, M. Wellman. Nash q-learning for general-sum stochastic games. Journal of Machine Learning Research, 4:1039-1069, 01 2003

[12] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In Proceedings of the Eleventh International Conference on Machine Learning, pp. 157-163, 1994

[13] S. Liu, G. Lever, J. Merel, S. Tunyasuvunakool, N. Heess, T. Graepel: Emergent coordination through competition, arXiv preprint arXiv:1902.07151, 2019

[14] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, I. Mordatch: Multi-agent actor-critic for mixed cooperative-competitive environments, In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, Advances in Neural Information Processing Systems 30, pp. 6379-6390, Curran Associates, Inc., 2017

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller: Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602, 2013

[16] R-E Precup, S. Preitl, E. M. Petriu, J. K. Tar, M. L. Tomescu, C. Pozna: Generic two-degree-of-freedom linear and fuzzy controllers for integral processes, Journal of the Franklin Institute, Volume 346, Issue 10, pp. 980-1003, 2009

[17] Z. Preitl,. R. E. Precup, J. Tar, M. Takács: Use of Multi-parametric Quadratic Programming in Fuzzy Control Systems. Acta Polytechnica Hungarica, Volume 3, 2006

[18] M. Samvelyan, T. Rashid, C. Schroeder de Witt, G. Farquhar, N. Nardelli, T. G. J. Rudner, C. Hung, P. H. S. Torr, J. Foerster, S. Whiteson: The starcraft multi-agent challenge, arXiv preprint arXiv:1902.04043, 2019

[19] Shihui, Y. Wu, X. Cui, H. Dong, Fang, Fei, S. Russell: Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient, In Proceedings of the AAAI Conference on Artificial Intelligence, Volume 33, pp. 4213-4220, 2019

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting, The journal of machine learning research, 15(1):1929-1958, 2014

[21] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, T. Graepel: Value-decomposition networks for cooperative multi-agent learning, arXiv preprint arXiv:1706.05296, 2017

[22] R. S. Sutton, D. McAllester, Singh: Policy gradient methods for reinforcement learning with function approximation, In Proceedings of the 12th International Conference on Neural Information Processing Systems, page 1057-1063, Cambridge, MA, USA, 1999. MIT Press

[23] R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction, Volume 9, 1998

[24] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Kättler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, R. Tsing: Starcraft ii: A new challenge for reinforcement learning, arXiv preprint arXiv:1708.04782, 2017

[25] I. A. Zamfirache, R.-E. Precup, R. C. Roman, E. M. Petriu: Neural Network-based control using Actor-Critic Reinforcement Learning and Grey Wolf Optimizer with experimental servo system validation, Expert Systems with Applications, Volume 225, 2023