ACTA IMEKO
ISSN: 2221-870X
September 2021, Volume 10, Number 3, 28 - 35

A2CM: a new multi-agent algorithm

Gabor Paczolay[1], Istvan Harmati[1]

[1] *Budapest University of Technology and Economics, Magyar Tudósok Körútja 2, 1117 Budapest, Hungary*

ABSTRACT
Reinforcement learning is currently one of the most researched fields of artificial intelligence. New algorithms are being developed that use neural networks to compute the selected action, especially for deep reinforcement learning. One subcategory of reinforcement learning is multi-agent reinforcement learning, in which multiple agents are present in the world. As it involves the simulation of an environment, it can be applied to robotics as well. In our paper, we use our modified version of the advantage actor–critic (A2C) algorithm, which is suitable for multi-agent scenarios. We test this modified algorithm on our testbed, a cooperative–competitive pursuit–evasion environment, and later we address the problem of collision avoidance.

Section: RESEARCH PAPER

Keywords: Reinforcement learning, multiagent learning

Citation: Gabor Paczolay, Istvan Harmati, A2CM: a new multi-agent algorithm, Acta IMEKO, vol. 10, no. 3, article 6, September 2021, identifier: IMEKO-ACTA-10 (2021)-03-06

Section Editor: Bálint Kiss, Budapest University of Technology and Economics, Hungary

Received January 15, 2021; **In final form** August 13, 2021; **Published** September 2021

Corresponding author: Gabor Paczolay, e-mail: paczolay.gabor@gmail.com

1. INTRODUCTION

Reinforcement learning is one of the most researched fields within the scope of artificial intelligence. Newer algorithms are continually being developed to achieve successful learning in more situations or with fewer samples.

In reinforcement learning, a new challenge arises when we take other agents into consideration. This research field is called 'multi-agent learning'. Dealing with other agents – whether they are cooperative, competitive or a mixture of both – brings the learning model closer to a real-world scenario. In real life, no agent acts alone; even random counteracts can be treated as 'counteracts of nature'.

In our work, we optimised the synchronous actor–critic algorithm to perform better in cooperative multi-agent scenarios (those in which agents help each other).

Littman [1] utilised the minimax-Q algorithm, a zero-sum multiagent reinforcement learning algorithm, and applied it to a simplified version of a robotic soccer game. Hu and Wellmann [2] created the Nash-Q algorithm and used it on a small gridworld example to demonstrate the results. Bowling [3] varied the learning rate of the training process to speed it up while ensuring convergence. Later, he applied the win or learn fast methodology to an actor–critic algorithm to improve its multi-agent capabilities [4].

Reinforcement learning advanced significantly when neural networks gained popularity and convergence was improved. Mnih et al. [5] successfully applied deep reinforcement learning to playing Atari games by feeding multiple frames at once and utilising experience replay to ensure convergence. Later, deep reinforcement learning was applied to multi-agent systems, such as independent multi-agent reinforcement learning. Foerster et al. [6] stabilised experience replay for independent Q-learning using fingerprints. Omidshafiei et al. [7] utilised decentralised hysteretic deep recurrent Q-networks for partially observable multi-task multi-agent reinforcement learning problems.

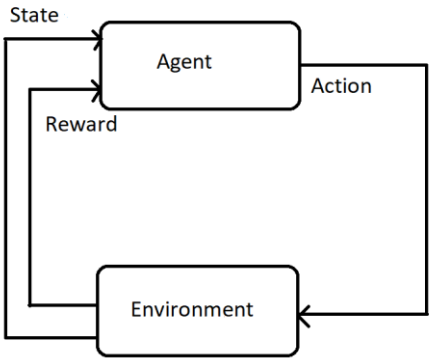

Figure 1. Markov decision process.

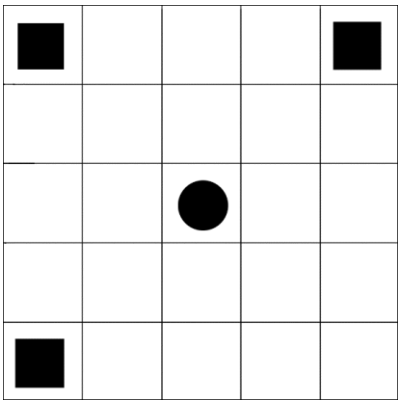

Figure 2. The simulation environment. The squares represent the controlled agents, while the circle represents the fleeing enemy. The goal is to catch the enemy by moving horizontally or vertically.

Multiple advancements have also been made in the field of centralised learning and decentralised execution. Foerster et al. [8] created counterfactual multi-agent policy gradients to solve the issue of multi-agent credit assignment. Peng et al. [9] created multiagent bidirectionally-coordinated nets with actor–critic hierarchy and recurrent neural networks for communication. Sunehag et al. [10] utilised value-decomposition networks with common rewards and Q-function decomposition. Rashid et al. [11] utilised QMIX with value function factorisation, Q-function decomposition and a feed-forward neural network with better performance than the former value-decomposition one. Lowe et al. [12] improved the deep deterministic policy gradient by altering the critic to contain all actions of all agents, thus making the algorithm capable of processing more multi-agent scenarios. Shihui et al. [13] improved upon the previous MADDPG algorithm, increasing its performance in zero-sum competitive scenarios by utilising a method based on minimax-Q learning. Casgrain et al. [14] upgraded the deep Q-network algorithm utilising methods based on Nash equilibria, making it capable of solving multi-agent environments.

Benchmarks have also been created to analyse the performance of various algorithms in multi-agent environments. Vinyals et al. [15] modified the StarCraft II game to make it a learning environment. Samvelyan et al. [16] also pointed to StarCraft as a multi-agent benchmark but with a focus on micromanagement. Liu et al. [17] introduced a multi-agent soccer environment with continuous simulated physics. Bard et al. [18] reached a new frontier with the cooperative Hanabi game benchmark.

Cooperative multiagent reinforcement learning and the proposed algoirthm are usable in many scenarios in robotics. As our algorithm is decentralised, it can be installed into the robots themselves without any central command center. It might be useful in exploration or localisation tasks in which the use of multiple agents would significantly speed up the process. Our testbed can be considered a simplified version of a localisation task, as the pursuer robots are trying to approach and measure a non-cooperative moving object. For proper use in robotics, a well-prepared simulation of the robots and the environment is required, in which thousands of episodes can be run for learning.

In our work, we modified the already existing advantage actor–critic (A2C) algorithm to make it better suited for multi-agent scenarios by creating a single-critic version of the algorithm. Then, we tested this modified A2CM algorithm on our cooperative–competitive pursuit–evasion testbed.

Initialise Model:
Initialise N+1 hidden and N+1 output (1 value + N action) layers (4 different networks in one model, 1 critic + 3 actor) number of updates batch size
for number of updates **do**
for batch size **do**
Calculate next actions a based on the previous state
Take actions a, get terminal state boolean and new rewards
Store the actions, the terminal state booleans, the calculated values, the rewards and the states
end for
Calculate returns based on (13)
Calculate advantages based on (12)
Update critic neural network based on the observed states and the corresponding returns: loss is the mean squared error between the returns and calculated values
Update actor neural networks based on the observed states, the taken actions and the advantages: loss is policy loss(weighted sparse categorical cross-entropy loss) − entropy loss(cross-entropy over itself)
end for

Algorithm 1: A2CM.

In the following section, we explain the theoretical background for our work. Then, the experiments themselves and the testbed are introduced. We continue by presenting the results and end with our conclusions and suggestions for future work on the topic.

2. THEORETICAL BACKGROUND

2.1. Markov decision processes

A Markov decision process is a mathematical framework for modeling decision making, as shown in Figure 1. In a Markov decision process there are states, selectable actions, transition probabilities and rewards [1]. At each timestep, the process starts at a state s and selects an action a from the available action space. It gets a corresponding reward r and then finds itself in a state s' given by the probability of $P(s,s')$. A process is said to be Markovian if

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

which means that a state's transition is based only on the previous state and the current action. Thus, only the last state and action are considered when deciding on the next state.

In a Markov decision process, the agents are trying to find a policy that maximises the sum of discounted expected rewards. The standard solution for this uses an iterative search method that searches for a fixed point of the Bellman equation:

$$v(s,\pi^*) = \max_a \left(r(s,a) + \gamma \sum_{s'} p(s'|s,a) v(s',\pi^*) \right). \quad (2)$$

2.2. Reinforcement learning

When the state transition probabilities or the rewards are unknown, the problem of the Markov decision process becomes a problem of reinforcement learning. In this group of problems, the agent tries to make a model of the world around itself via trial and error.

One type of reinforcement learning is **value-based reinforcement learning.** In this case, the agent tries to learn a

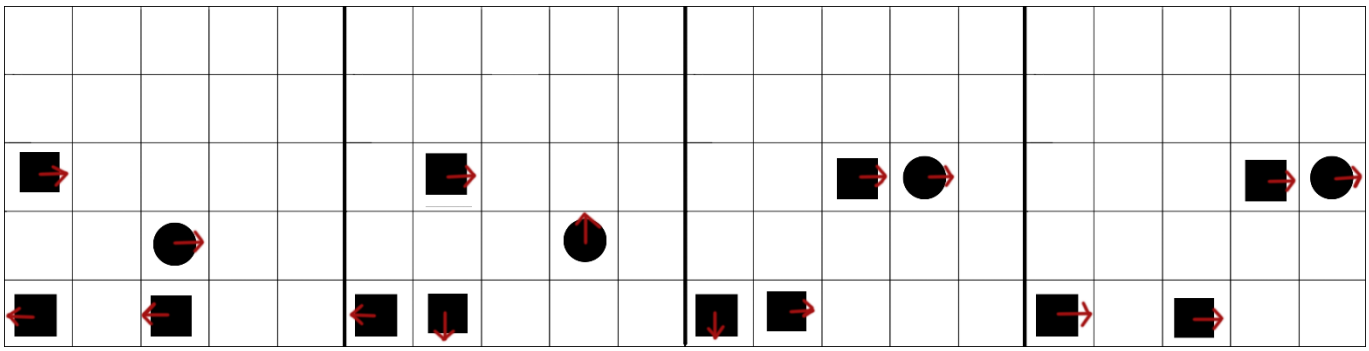

Figure 3. An example of catching the randomly moving opponent.

value function that renders a value to states or to actions from states. These values correspond to a reward achieved by reaching a state or taking a specific action from a state.

The most commonly used type of value-based reinforcement learning is **Q-learning** [2], in which the so-called Q-values are estimated for each of the state–action pairs of the world. These Q-values represent the value of choosing a specific action in a state, meaning the highest reward the agent could possibly get by taking that action. The equation for Q-learning for updating the Q-values of a state is:

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

where α is the learning rate and γ is the discount for the reward. The agent always selects an action that maximises the Q-function for the state that the agent is in.

Another type of reinforcement learning is **policy-based reinforcement learning**. In this case, actions are derived as a function of the state itself. The most common policy-based reinforcement learning method is the **policy gradient** approach [19]. In this case, the agent tries to maximise the expected reward by following the policy π_θ parametrised by θ based on the total reward for a given trajectory $r(\tau)$. Thus, the cost function of the parameters θ is the following:

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

The parameters are then tuned based on the gradient of the cost function:

$$\theta_{k+1]} = \theta_t + \alpha \Delta J(\theta_t). \quad (5)$$

The advantages of policy-based methods include the ability to map environments with huge or even continuous action spaces and solve environments with stochasticity. However, when using these methods, there is also a much greater possibility of getting stuck in a local maximum rather than following the optimal policy.

Apart from the aforementioned model-free reinforcement learning methods, there is also **model-based reinforcement learning**. In this case, a model is built (or just tuned) to perform the reinforcement learning. This is more sample-efficient than model-free methods and thus requires fewer samples to perform equally, but it is very dependent on the particular model. It can be combined with model-free methods to achieve better results, as in [20].

2.3. Multi-agent systems and Markov games

A matrix game is a stochastic framework in which each player selects an action and gets an immediate reward based on their action and those of the other agents [1]. They are called 'matrix games' because the game can be written as a matrix, with the first two players selecting actions in the rows and columns of the matrix. Unlike Markov decision processes, these games have no states.

Markov games, or stochastic games, are extensions of Markov decision processes with multiple agents. They can also be thought of as extensions of matrix games with multiple states. In a Markov game, each state has a payoff matrix for all of the states. The next state is determined by the joint actions of the agents. A game is Markovian if

$$P(a_i^t = a_i | s^t, a_i^{t-1}, \ldots, s^0, a_i^0) = P(a_i^t = a_i | s^t), \quad (6)$$

so the next state depends only on the current state and the current actions taken by all agents.

2.4. Deep reinforcement learning

A reinforcement learning algorithm is called 'deep' if it is assisted by a neural network.

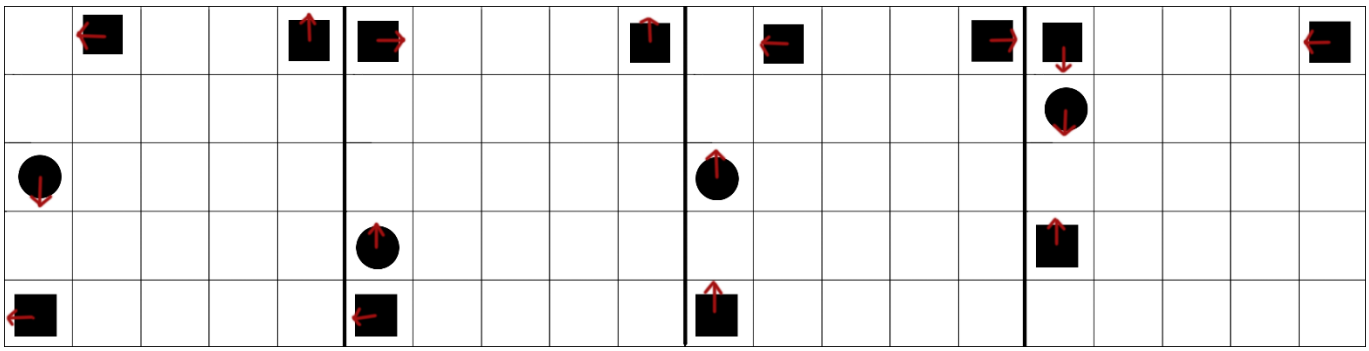

Figure 4. An example of catching the fleeing opponent.

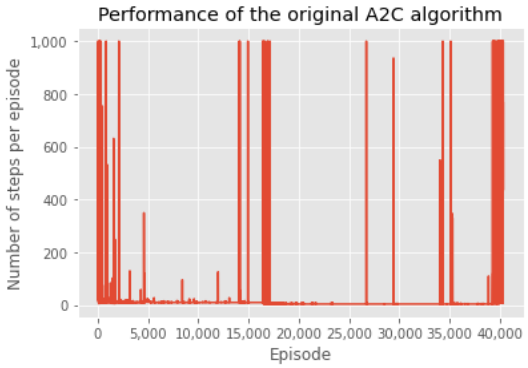

Figure 5. The performance of the original A2C algorithm on our benchmark.

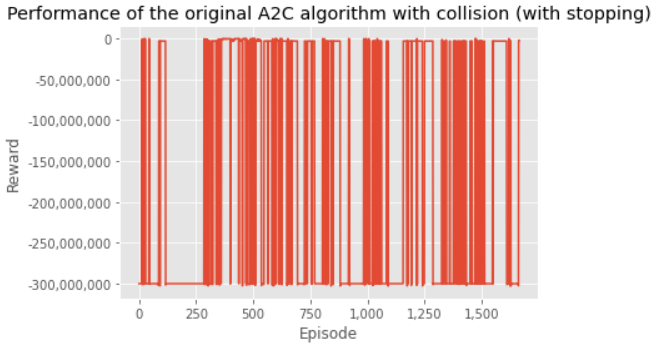

Figure 7. Performance of the original A2C algorithm on our benchmark with collision (with terminating at collision).

A neural network is a function approximator built from (sometimes billions of) artificial neurons. An artificial neuron, which is based on the real neurons of the brain, has the following equation:

$$y = \mathrm{Act}\left(\sum w\,x + b\right), \tag{7}$$

where x is the input vector, w is the weight vector, b is the bias and $\mathrm{Act}()$ is the activation function to introduce nonlinearity in an otherwise linear system. The parameters (w and b) are tuned with backpropagation, calculating the partial derivative error of all parameters propagated from the final error to the input vector.

The selection of the activation function is important in deep learning due to the vanishing gradients: when many layers are stacked upon each other, higher layers' gradients are too small during backpropagation, and thus, those layers are difficult to train. A basic activation function can be a sigmoid or logistic activation function:

$$y = \frac{1}{1 + \mathrm{e}^{-x}}. \tag{8}$$

A common activation function in deep learning is rectified linear unit (ReLU) [21], which has gradients that are less vanishing and therefore better to train. It has the following equation:

$$\begin{aligned} y &= x \text{ if } x > 0 \\ y &= 0 \text{ if } x <= 0\,. \end{aligned} \tag{9}$$

For multi-class classification, another activation function is used: the softmax activation function. When used as the last layer, the probabilities of all of the output neurons add up to exactly 1. Thus, in reinforcement learning, it is utile to use it as the probability distribution of the possible actions. It has the following equation:

$$y = \frac{\mathrm{e}^{x_i}}{\sum_j \mathrm{e}^{y_j}}. \tag{10}$$

Deep reinforcement learning algorithms have several advantages compared to traditional reinforcement learning algorithms. First of all, they are not based on a state table, as the states are approximated (which is much more robust than using linear function approximators). This allows many more states to be mapped and even allows for continuous states. However, they are more prone to diverging, and thus, many optimisations have been created on deep reinforcement learning algorithms to provide better convergence on the problems.

2.5. Actor–critic

An actor–critic system combines value-based and policy-based reinforcement learning. In these systems, there are two distinct parametrised networks: the critic, which estimates a value function (as in value-based reinforcement learning), and an actor, which updates the policy network based on the direction suggested by the critic (as in policy-based reinforcement learning). Actor–critic algorithms follow an approximate policy gradient:

$$\begin{aligned} &\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\; Q_w(s,a)] \\ &\Delta\theta = \alpha\, \nabla_\theta \log \pi_\theta(s,a)\; Q_w(s,a)\,. \end{aligned} \tag{11}$$

Approximating the policy gradient introduces bias to the system. A biased policy gradient may not find the right solution, but if we choose the value function approximation carefully, then we can avoid introducing any bias.

Actor–critic systems generally perform better than regular reinforcement learning algorithms. The critic network ensures that the system does not get stuck in a local maximum; meanwhile, the actor network enables the mapping of environments with huge action spaces and provides better convergence [19].

2.6. The A2C algorithm

A2C stands for synchronous advantage actor–critic. It is a one-environment-at-a-time derivation of the asynchronous advantage actor–critic (A3C) algorithm [22], which processes multiple agent-environments simultaneously. In that algorithm, multiple workers update a global value function, thus exploring the state space effectively. However, the synchronous advantage

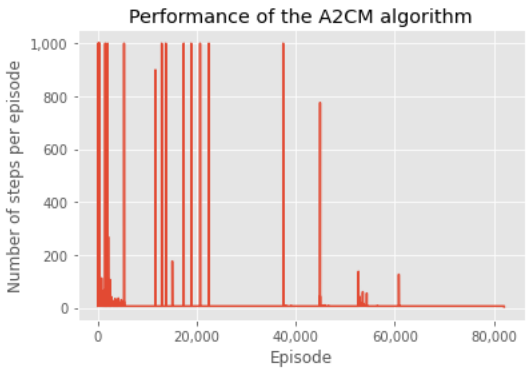

Figure 6. Performance of the modified A2C algorithm on our benchmark.

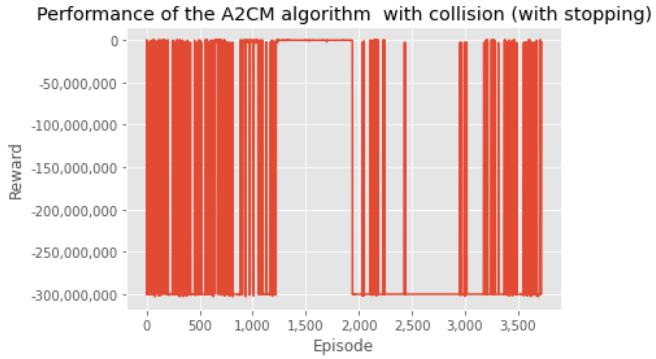

Figure 8. Performance of the A2CM algorithm on our benchmark with collision (with terminating at collision).

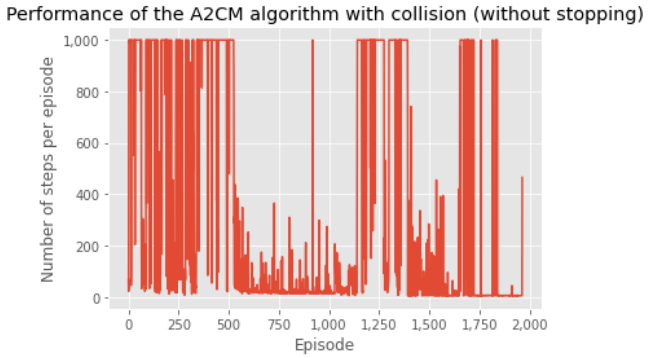

Figure 10. Number of steps per episode of the A2CM algorithm on our benchmark with collision (without terminating at collision).

actor–critic provides better performance than the asynchronous model.

Advantage function is a method that significantly reduces the variance of the policy gradient by subtracting the cumulative reward using a baseline to make smaller gradients; thus, it provides much better convergence than regular Q-values. It has the following equation:

$$A(s,a) = Q(s,a) - V(s)\,. \tag{12}$$

Returns are calculated using the equation:

$$G_t = r_t + \gamma * r_{t+1} * (1 - T_t)\,, \tag{13}$$

where G is the return, r_t is the reward at time t, γ is the discount factor and T_t indicates whether the step at time t is a terminal state.

3. EXPERIMENTS AND RESULTS

The testbed is a 5 × 5 grid with three cooperating agents (the squares) in three of the four corners of the environment. In the middle, there is a fourth agent (the circle). The former three agents have the objective of catching the fourth agent, which moves randomly. This testbed is analogous to pursuit–evasion (or predator–prey) scenarios that are also significant in robotics. The agents can move in four directions: up, down, left or right. When one of the three agents catches the fourth one, the episode ends. A penalty is introduced to the cooperative agents every timestep; thus, the return of an episode is maximised by ending the episode as soon as possible (i.e. catching the fleeing agent as quickly as possible). Each episode must end in 1,000 timesteps to avoid getting stuck.

In the modification of the A2C algorithm, we followed the theory of centralised learning and decentralised execution. This

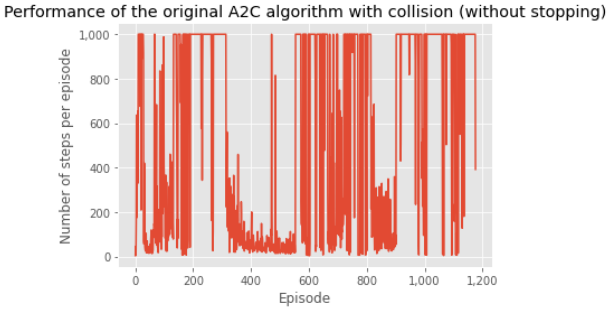

Figure 9. Number of steps per episode of the original A2C algorithm on our benchmark with collision (without terminating at collision).

means that the execution is decentralised, but the learning phase can be assisted by additional information from other agents. In our case, we used the information that the agents are cooperative; thus, they acquire the same rewards (and returns). As noted before, decentralised execution is most helpful in real-world scenarios in which communication difficulties make a centralised task-solving achitecture impossible. Such scenarios are often encountered in robotics.

In our experiment, many A2C models with one actor and one critic were substituted for one model with one critic and multiple actors. The pseudocode of the algorithm can be seen in Algorithm 3. All neural network layers were subclasses of the TensorFlow model class, which provides utile functions for training and prediction – even for batch tasks – by providing only the forward steps of the network. The optimiser was RMSprop, with a learning rate of $7 \cdot 10^{-3}$.

The value estimator critic contained a neural network of 128 hidden unit layers with ReLU activation function and one output layer with one unit. Its loss function was a simple mean squared error between the returns and the value.

The actors contained a hidden layer with 128 hidden units and an output layer with four units (the number of actions in the action space). The loss function contained two distinct parts: policy and entropy loss. The policy loss was a weighted sparse categorical cross-entropy loss, where the weights were given by the advantages. This method increased the convergence of the algorithm. Entropy loss is a method for increasing exploration by encouraging actions that are not in the local minimum. This is very important for tasks with sparse rewards due to the fact that the agent does not receive feedback often. This loss was calculated as a cross-entropy over itself, and it was subtracted from the policy loss because it should be maximised, not minimised. The entropy loss was tuned by a constant, which was taken as $1 \cdot 10^{-4}$.

Episode rewards were taken to be a list where a value of 0 was appended to the end of the list at each episode's end. During the episodes, only the last value of the list was incremented by the episode reward of the given step. For the training, a batch-sized container was created for the actions, rewards, terminal state booleans, state values and observed states. Then, a two-level loop was started: the outer one was run for the number of required updates (set by us), while the inner loop was run as many times as the batch size. The state observations, the taken actions (which were selected by a probability distribution based on the actor neural network results), the state values, the rewards, the terminal state booleans and the last observed state were stored in the aforementioned containers. Next, the returns and advantages

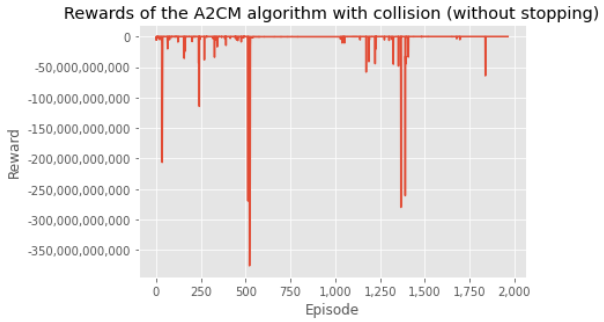

Figure 11. Rewards per episode of the A2CM algorithm on our benchmark with collision (without terminating at collision).

were calculated on the batch using the collected data, and then a batch training was performed on those data. There was no need to calculate the gradients themselves due to the use of the Keras API.

During our experiment, the system was run 5,000 times in batches of 128, thus running the environments over a total of 640,000 steps. Gamma was taken to be 0.99.

Figure 3 and Figure 4 show the ends of some remarkable episodes of catching the opponent. Figure 6 and Figure 7 show the results of our experiments. It is important to note the x-coordinates in Figure 5 and Figure 6: for the same number of steps, the original was run for 40,340 episodes, while the modified algorithm managed to complete 82,119 episodes. This means that the A2CM algorithm spent half as many steps in an episode and was able to catch the fleeing opponent in, on average, half of the time required by the agent based on the original algorithm. These figures also show that the original algorithm did not find an optimal solution without diverging later, and even between divergences, the solutions were not as stable. Our agent, on the other hand, found a solution with no divergences later and only small divergencess after the first half of the episodes. The A2CM algorithm found a solution with which it can catch the opponent in 6 steps, and it maintained this knowledge for 20,000 episodes, with one positive spike where it found the solution to the problem in just 3 steps.

The run times are worth considering, as well. The regular A2C algorithm took 14,567.45 seconds to run, while the modified one ran for 14,458.28 seconds. It is worth noting that, due to the fact that almost twice as many episodes were completed, the environment had to be reset twice as often, so the modified algorithm is even faster than the normal one.

Later, the difference between the algorithms were tested with collision turned on, bringing the problem set even closer to real-world robotics scenarios. In this case, the agents received a penalty if they collided with each other. This method makes the environment much harder to learn, as failure will probably only result from chasing the enemy agent. It also makes the training process harder, as the steps leading to success are not as easy to determine; a collision that occurs before the enemy is caught will make similar attempts less likely to be selected as actions.

When considering the training process of the environment with collision detection turned on, it is important to pay attention to the reward ratio between the negative rewards for each step and the negative reward for collision. The larger the reward for collision, the better the agents will evade collision; otherwise, they will be optimised to finish the episode as fast as possible. For this reason, the negative reward for each step was selected as −1,000, and the negative reward for the collision was −150,000,000, providing a ratio that is large enough to encourage the agents to follow a collision-evasion policy.

In the first experiment on the environment with collision detection, we tried to set the algorithms such that a collision would terminate the episode. This scenario is analogous to certain scenarios in robotics in which collisions can cause malfunctions in the robots themselves and should be evaded even via high-level control. Apart from turning on the collision, all other conditions and parameters of the training process were the same. Figure 7 and Figure 8 show the cumulated rewards per episode for the original A2C and our A2CM algorithm, respectively. It can be seen that, while neither was able to solve the environment over the timespan of the training, there was a time span of ca. 700 episodes in which our algorithm was able to catch the enemy without colliding. The original algorithm lacked any of these longer periods. The training of the original algorithm in this case took 14,173.42 seconds, while the training of the A2CM took 14,659.00 seconds. It is worth noting that the original algorithm completed 1,665 episodes, and the A2CM completed 3,723; the different numbers of reinitialisations should be considered when comparing the training times.

To make the environment easier to train on, the second experiment with collisions was conducted such that the episodes only terminated if the opponent was caught. This way, the episodes were longer and always terminated successfnotedully and therefore might provide better training information than the setting of the previous experiment. This scenario is analogous to problems in robotics in which the presence of two robots in the same area is discouraged, such as area scanning scenarios or sub-tasks in which two robots should not scan the same area at once. Just as in the previous experiment, all other parameters were left as they were in the training of the system without collision. Figure 9 and Figure 10 show the number of steps required to finish each episode for the original A2C and the modified A2CM algorithms, respectively, while Figure 11 and Figure 12 show the cumulated (negative) rewards per episode (higher is better) for the A2C and the A2CM algorithms, respectively. It can be seen that, while the original A2C algorithm did not show any clear sign of successful training, there is some indication of success for the A2CM algorithm. Approaching the end of the training process, the number of steps were kept low, and, as per Figure 12, collisions were also evaded, with the exception of some episodes. The original algorithm completed 1,177 episodes, while the modified one completed 1,964, which can also be seen as a sign of the superiority of the A2CM algorithm. Regarding the training times, the original algorithm was trained for 13,981.22 seconds, while the modified one was trained for 19,519.85 seconds. In this case, it is clear that our algorithm used significantly more training time.

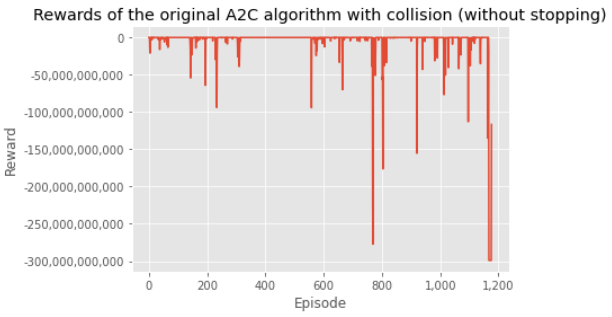

Figure 12. Rewards per episode of the original A2C algorithm on our benchmark with collision (without terminating at collision).

4. CONCLUSION

Looking at the previous section, we can conclude that our modification of the original A2C algorithm, the A2CM algorithm, was able to perform much better than the original on our testbed without collision. To some extent, it outperformed the original A2C algorithm even in enviroments with collision; thus, it is recommendable for tasks in robotics. However, the algorithm has the caveat of being usable only when the agents are fully cooperative and do not have special, predefined roles.

There are still many ways to improve upon the current state of our algorithm. One possibile improvement would be to introduce a variable learning rate, such as win or learn fast [3], in a deep reinforcement learning algorithm. Another possible improvement is to include the fleeing agent in the algorithm so that the algorithm encompasses the full cooperative–competitive nature of the environment. In addition, other activation functions could be tried to check their behavior; for example, exponential linear units [23] might have better convergence at the price of slightly more training time. The algorithm could be extended using recurrent neural networks so that it could handle partially observable Markov decision processes in which the full state is unknown.

ACKNOWLEDGEMENT

The research reported in this paper and carried out at the Budapest University of Technology and Economics was supported by the TKP2020, Institutional Excellence Program of the National Research Development and Innovation Office in the field of Artificial Intelligence (BME IE-MI-SC TKP2020).

The research was supported by the EFOP-3.6.2-16-2016-00014 project, which was financed by the Hungarian Ministry of Human Capacities.

REFERENCES

[1] M. L. Littman, Markov games as a framework for multi-agent reinforcement learning, Proceedings of the Eleventh International Conference on Machine Learning, New Brunswick, USA, 10 – 13 July 1994, pp. 157-163.
DOI 10.1016/b978-1-55860-335-6.50027-1

[2] J. Hu, M. Wellman, Nash q-learning for general-sum stochastic games, Journal of Machine Learning Research 4 (2003), pp. 1039-1069. Online [Accessed 6 September 2021]
https://www.jmlr.org/papers/volume4/hu03a/hu03a.pdf

[3] M. Bowling, M. Veloso, Multiagent learning using a variable learning rate, Artificial Intelligence 136 (2002), pp. 215-250. DOI: 10.1016/S0004-3702(02)00121-2

[4] M. H. Bowling, M. M. Veloso, Simultaneous adversarial multi-robot learning, IJCAI (2003) pp. 699-704.
DOI: 10.5555/1630659.1630761

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing Atari with deep reinforcement learning, arXiv (2013), 9 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1312.5602

[6] 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, PMLR 70 (2017) pp. 1146-1155.
DOI: 10.5555/3305381.3305500

[7] S. Omidshafiei, J. Pazis, C. Amato, J. P. How, J. Vian, Deep decentralized multi-task multi-agent reinforcement learning under partial observability, PMLR 70 (2017) pp. 2681-2690.
DOI: 10.5555/3305890.3305958

[8] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, S. Whiteson, Counterfactual multi-agent policy gradients, Proceedings of the AAAI Conference on Artificial Intelligence, New Orleans, USA, 2 – 7 February 2018, pp. 1146-1155, arXiv (2017), 12 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1705.08926

[9] P. Peng, Y. Wen, Y. Yang, Q. Yuan, Z. Tang, H. Long, J. Wang, Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play StarCraft combat games, arXiv (2017), 10 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1703.10069

[10] 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 (2017), 17 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1706.05296

[11] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, S. Whiteson, QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning, Proceedings of Machine Learning Research, Stockholm, Sweden, 10 – 15 July 2018, pp. 4295-4304. arXiv (2018), 14 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1803.11485

[12] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, I. Mordatch, Multi-agent actor-critic for mixed cooperative-competitive environments, Advances in Neural Information Processing Systems 30 (2017), pp. 6379-6390.
DOI: 10.5555/3295222.3295385

[13] S. Li, Y. Wu, X. Cui, H. Dong, F. Fang, S. Russell, Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient, Proceedings of the 33rd AAAI Conference on Artificial Intelligence, Honolulu, Hawaii, USA, 27 January – 1 February 2019, pp. 4213-4220.
DOI: 10.1609/aaai.v33i01.33014213

[14] P. Casgrain, B. Ning, S. Jaimungal, Deep Q-learning for Nash equilibria: Nash-DQN, arXiv (2019), 16 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1904.10554

[15] 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 (2017), 20 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1708.04782

[16] M. Samvelyan, T. Rashid, C. S. 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 (2019), 14 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1902.04043

[17] S. Liu, G. Lever, J. Merel, S. Tunyasuvunakool, N. Heess, T. Graepel, Emergent coordination through competition, arXiv (2019), 19 pp. Online [Accessed 14 September 2021]
https://arxiv.org/abs/1902.07151

[18] 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 280 (2020), 103216.
DOI: 10.1016/j.artint.2019.103216

[19] R. S. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, Proceedings of the 12th International Conference on Neural Information Processing Systems, Denver, USA, 29 November – 4 December 2000, pp. 1057-1063.
DOI: 10.5555/3009657.3009806

[20] A. Nagabandi, G. Kahn, R. S. Fearing, S. Levine, Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning, 2018 IEEE International Conference on Robotics and Automation, Brisbane, Australia, 21 – 26 May 2018,

pp. 7559-7566.
DOI: 10.1109/ICRA.2018.8463189

[21] A. F. Agarap, Deep learning using rectified linear units (ReLU), arXiv (2018), 7 pp. Online [Accessed 14 September 2021] https://arxiv.org/abs/1803.08375

[22] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, Proceedings of Machine Learning Research, New York, USA, 20 – 22 June 2016, pp. 1928-1937. DOI: 10.5555/3045390.3045594

[23] D. A. Clevert, T. Unterthiner, S. Hochreiter, Fast and accurate deep network learning by exponential linear units (ELUs), arXiv (2015), 14 pp. Online [Accessed 14 September 2021] https://arxiv.org/abs/1511.07289