EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2018*

Deep reinforcement learning i distribuerad optimering

MARCUS LINDSTRÖM

JAHANGIR JAZAYERI

KTH
SKOLAN FÖR TEKNIKVETENSKAP

DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Deep Reinforcement Learning in Distributed Optimization

MARCUS LINDSTRÖM

JAHANGIR JAZAYERI

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Sammanfattning

Reinforcement learning har nyligen blivit ett alltmer framgångsrikt område inom maskininlärning där stora prestationer utförts. Att datorer lyckats vinna över mänskliga experter på Atari-spel och datorprogrammet AlphaGo som lyckats uppnå högsta rank i spelet Go är några av framgångarna som på senaste tiden uppmärksammats.

Det här projektet ämnar att implementera en så kallad Policy Gradient Method (PGM) i en multiagentmiljö. PGM inom Reinforcement learning anses vara applicerbart på en större bredd av problem jämfört med exempelvis Deep Q-learning men den har istället en tendens att konvergera till lokala optimum. I denna rapport undersöker vi om en optimal policy går att hitta vid användandet av PGM i ett multiagentsramverk.

Numeriska simulationer med implementering av tidigare nämnda metoder i en miljö med upp till fyra agenter och rörliga hinder visade en konvergens till en lösning samt effektiviteten i metoden. En relativt liten andel kollisioner tog plats när de tränade agenterna skulle utvärderas. Resultaten skiljde sig åt då hyperparametrar som inlärningshastighet och mängden neuroner i det neurala nätverket hade ändrats.

Slutsatsen är att en snabbt konvergens på åtminstone ett lokalt optima var framgångsrikt i vår ansats.

Abstract

Reinforcement learning has recently become a promising area of machine learning with significant achievements in the subject. Recent successes include surpassing human experts on Atari games and also AlphaGo becoming the first computer ranked on the highest professional level in the game Go, to mention a few.

This project aims to apply Policy Gradient Methods (PGM) in a multi agent environment. PGM are widely regarded as being applicable to more problems than for instance Deep Q-Learning but have a tendency to converge upon local optimums. In this report we aim to explore if an optimal policy is achievable with PGM in a multi-agent framework.

Numerical simulations implementing the aforementioned method in an environment with up to 4 agents and moving obstacles showed a convergence and the efficiency of the approach. A relatively small amount of collisions took place once the learnt agents were tested. These result differed when changing some parameters such as learning rates and number of neurons in the neural network.

The conclusion was that a very fast convergence upon at least a local optimal policy was achieved in this setting.

Deep Reinforcement Learning in Distributed Optimization

Marcus Lindström and Jahangir Jazayeri

***Abstract*—Reinforcement learning has recently become a promising area of machine learning with significant achievements in the subject. Recent successes include surpassing human experts on Atari games and also AlphaGo becoming the first computer ranked on the highest professional level in the game Go, to mention a few.**

This project aims to apply Policy Gradient Methods (PGM) in a multi agent environment. PGM are widely regarded as being applicable to more problems than for instance Deep Q-Learning but have a tendency to converge upon local optimums. In this report we aim to explore if an optimal policy is achievable with PGM in a multi-agent framework.

Numerical simulations implementing the aforementioned method in an environment with up to 4 agents and moving obstacles showed a convergence and the efficiency of the approach. A relatively small amount of collisions took place once the learnt agents were tested. These result differed when changing some parameters such as learning rates and number of neurons in the neural network.

The conclusion was that a very fast convergence upon at least a local optimal policy was achieved in this setting.

I. Introduction

In the late 1950's, "Machine Learning" was coined as a field that is concerned with the study of algorithms that learn from data and make predictions. While the subject suffered periods of low interest (primarily during the 70's), it has now seen an increase in popularity much due to deep learning methods becoming feasible.

Google's DeepMind team caused lots of media attention when their AlphaGo AI won against the reigning world champion go player [1], and has since further developed this into Alpha Go Zero; an AI based on "Tabula Rasa" learning. Meaning it learns without any human intervention or data, but exclusively from self play to the point where it wins in about 90% [2] of games against the earlier champion beating version.

This report investigates methods for solving a specific problem within a paradigm of machine learning called Reinforcement Learning (RL), an area defined by the problem of how AI's ought to make actions within an environment in order to maximize cumulative reward. The problem that we address in this report consists of implementing model free deep learning algorithms for optimizing the cooperative behaviour of multiple agents. Our special case of this consist of a $\mathbb{R}^2$ grid in which agents move along common paths, having knowledge of other agents absolute positions and speeds while transporting items between unique sets of starting and ending positions.

II. Preliminaries and background

This section covers our approach to finding an optimal control policy in the case of multi agent reinforcement learning (MARL). First we outline the theory and methodology needed for finding solutions in the case of single agents. Then move on to how these can be manipulated and combined with *deep learning* algorithms for tackling the more general case.

A. Single agent reinforcement learning

Definition 1 below introduces the finite Markov decision process (MDP) which is the formal model used in single agent RL:

Definition 1. *A finite Markov decision process is a tuple* $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ *where* $\mathcal{S}$ *is the finite set of environment states,* $\mathcal{A}$ *is the finite set of agent actions,* $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ *is the state transition probability function,* $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ *is the reward function, and* γ *is a discount factor* $\gamma \in [0, 1]$.

For deterministic systems, the transition function is replaced and takes on another form: $\bar{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, meaning also that the reward function is determined completely by the current action and state.

At each time step t the environment is described by $s_t \in \mathcal{S}$, the agent observes the state and takes an action $a_t \in \mathcal{A}$ which transitions the state to s_{t+1} with probability $T(s_{t+1} \mid s_t, a_t)$. The transition is evaluated by the reward $r_{t+1} \in \mathbb{R}$ according to $r_{t+1} = R(s_{t+1} \mid s_t, a_t)$.

The agent follows a policy $\pi(s) : \mathcal{S} \to \mathcal{A}$, i.e., a mapping from perceived environment states to actions, and the goal is suitably to find the optimal policy π^* which maximizes its total expected reward, this is where the learning of the agent takes place, from modifying π over time. By taking the above into consideration we introduce the *state value function for policy* π [3]:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right], \forall s \in \mathcal{S}, \tag{1}$$

which describes for each state s its value for the agent to start in if following policy π (the use of the discount factor γ can be interpreted as minding less about future reward). The optimal value function $v^*(s)$ is closely related to the optimal policy π^*, and as can be seen in equation (2a) and (2b), finding $v^*(s)$ means finding $\pi^*(s)$:

$$v^*(s) = \max_{\pi} v_\pi(s) = v_{\pi^*}(s), \tag{2a}$$

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \sum_{s' \in \mathcal{S}} T(s' \mid s, a) v^*(s'). \tag{2b}$$

The aforementioned theory has so far not exclusively been revolving around model free RL per se, but it does apply to it. In the case of having complete knowledge of the MDP, one can actually find the optimal control policy without ever letting an agent "step into" the environment with use of planning algorithms. The RL problem is not so simple, instead the agent does not know beforehand how the environment will change in response to its actions or how much reward it's going to get. Instead, the agent has to learn from experience and adapt to its environment by taking actions within it and observing the consecutive states it ends up in.

B. Neural networks

Deep learning methods are based on using *neural networks* in order to find (or learn) a set of weights that characterize the importance of different attributes of data. In the case of RL, these weights are used in order for the agent to make 'good' decisions and thus defines the policy described in section II.

A neural network consists of artificial neurons that maps input to an output, mimicking the biological ones in the human brain. Consider the simplest case of one neuron taking as input for example, X_1, X_2 and X_3 see figure 1.

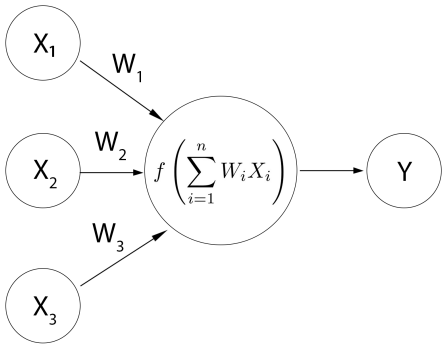

Fig. 1. Illustration of a single neuron taking three inputs and mapping them to a single output through an activation function f.

The weight W_i can be thought of as the importance of attribute X_i for said neuron. The activation function denoted $f : \mathbb{R} \mapsto \mathbb{R}$ takes as input the sum of all weights times the attributes and maps this to an output, it can be modelled differently depending on the scenario but a few worth mentioning are the *sigmoid*, *tanh* and *rectified linear* functions given in descending order by equations (3a), (3b) and (3c) below and visualized in figure 2.

$$f(z) = \frac{1}{1 + e^{-z}}, \tag{3a}$$

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{3b}$$

$$f(z) = \max(0, z). \tag{3c}$$

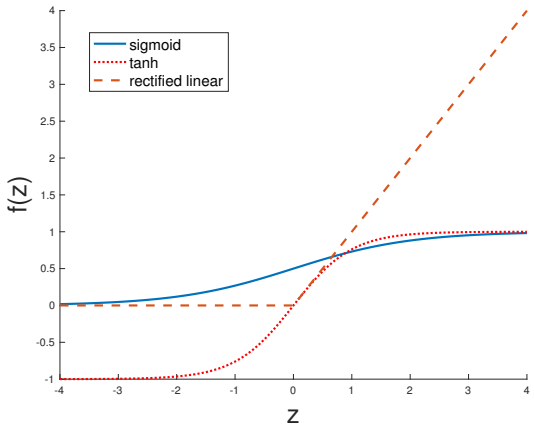

Fig. 2. Different types of activation functions used in neural networks.

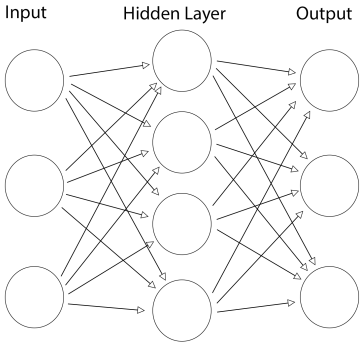

Fig. 3. A neural network with one hidden layer.

A more complex network comprises many connected neurons in different layers where one neurons output may be the input of another, see figure 3. From the outside, only the input and output layers are observed thus the middle layer is referred to as 'hidden' and an arbitrary amount of these may be used. If we denote the amount of layers by η_l, which is equal to three in our case, and denote layer l by L_l we can collect the weights in matrices W^l where element $W^l_{(ij)}$ corresponds to the weight connecting neuron i in layer l with neuron j in layer $l+1$. Furthermore, let a^l_i denote the output of neuron i in layer l, and z^l_i the sum of weighted inputs to neuron i in layer l, (for $l = 1$ we simply let $a^1_i = x_i$ where the x_i's are the input values to the network). Also note that when modelling a neural network, one can use varying kinds of activation functions for each layer, thus we let the activation function for layer l be denoted f_l.

The output of our example network is referred to as a hypothesis $h_W(x)$ and with the terminology laid out above we can describe our networks computations with the following

equations, (we let the activation function f_l take vectors as inputs and compute element-wise outputs):

$$a^{(1)} = x, \tag{4a}$$
$$z^{(2)} = W^{(1)}a^{(1)}, \tag{4b}$$
$$a^{(2)} = f_2(z^{(2)}), \tag{4c}$$
$$z^{(3)} = W^{(2)}a^{(2)}, \tag{4d}$$
$$h_W(x) = a^{(3)} = f_3(z^{(3)}). \tag{4e}$$

There are other kinds of architectures for building neural networks which we will not cover here, our example network is referred to as *Feedforward*, characterized by connections between neurons not forming cycles or loops.

Initially the weights W_{ij}^l are chosen randomly, otherwise it would result in $a_1^{(2)} = a_2^{(2)} = a_3^{(2)}$... etc. because of symmetry. To update them smart, an algorithm called Gradient Descent can be utilized which lightly speaking means that one follows the opposite direction of a loss function $J(W)$'s (indicating how 'bad' an output is), gradient with respect to the weights according to the following:

$$W_{ij}^l = W_{ij}^l - \alpha \frac{\partial}{\partial W_{ij}^l} J(W). \tag{5}$$

The partial derivatives $\frac{\partial}{\partial W_{ij}^l} J(W)$ are calculated numerically with help of a method called *Backpropagation* [4] and α is chosen as to indicate how sensitive each update of the network should be to the gradient. Note that the loss function is not restricted to being convex which means the algorithm is susceptible to converging upon local optima rather than global ones, this is in practice though usually not an issue.

C. Deep reinforcement learning

In order to utilize neural networks when optimizing a control policy for a reinforcement agent one can use different approaches. One example is to approximate the value function mentioned earlier which indirectly gives the policy, in this paper though we use a direct optimal policy approximation through what are referred to as *Policy Gradient Methods*. The network takes as input a representation of the state and directly maps it onto a probability distribution of actions (or a policy), this is then used in order to determine what action to take.

The loss function introduced in the previous part has to be modelled in such a way so as to represent how actions and rewards are related, a positive reward has to be the result of a good action. We introduce the *Cross Entropy* function $J(W)$:

$$J(W) = -\sum r log\pi(a \mid s; W). \tag{6}$$

We see that in order to minimize $J(W)$, the policy has to favour actions that give positive reward given a certain state.

III. Methods

In this section, we discuss the practical issues of our implementation. We show also how the rewards affected each agent with respect to what state it transitioned to after taking an action. Furthermore, we represent how the environment was defined and represented.

A. Environment and rewards

The environment consisted of a 10 by 10 grid with three horizontally moving obstacles where each agent had its own respective goal (see figure 4). The rewards were adjusted so that the agents always received a small negative reward from each transition. Moreover, a larger penalty from either taking an action that puts it on top of another agent or obstacle and a big positive reward from getting to its goal (see table 1).

The state representation that the environment supplied agents with consisted of a tuple with components representing relative distances and directions to obstacles, agents and its own goal and also rewards from each. It also contained information regarding what direction each moving obstacle was headed.

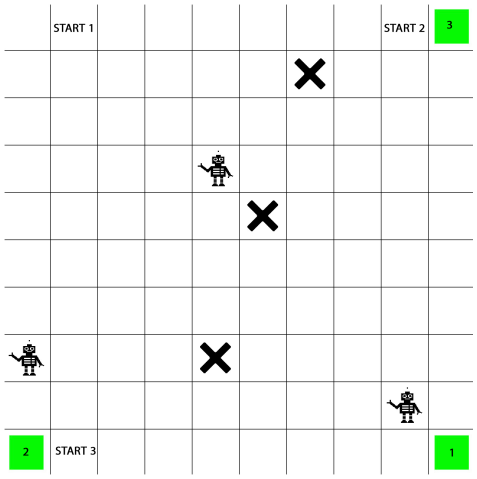

Fig. 4. The visualized environment with three agents in it, crosses represent horizontally moving obstacles and squares are stationary goals.

TABLE I
DISTRIBUTION OF REWARDS THAT AGENTS COULD RECEIVE FROM TAKING AN ACTION THAT TRANSITIONS IT TO A DIFFERENT STATE.

State	Reward
Obstacle	-1
Another agent	-1
Goal	2
All states	-0.1

B. Algorithm details

Initially, the goals, agents and obstacles were placed at preset locations. This was followed by the obstacles updating their position with one step horizontally and only changing direction if encountering a wall. All agents could then take one step and in response the obstacles took another and so on. If one agent was to find its way to goal it froze in place until all other agents also found theirs thus ending one episode.

All the implementations were done in *Python3* using an edited version of a publicly available *GitHub* code [5].

IV. Main Results

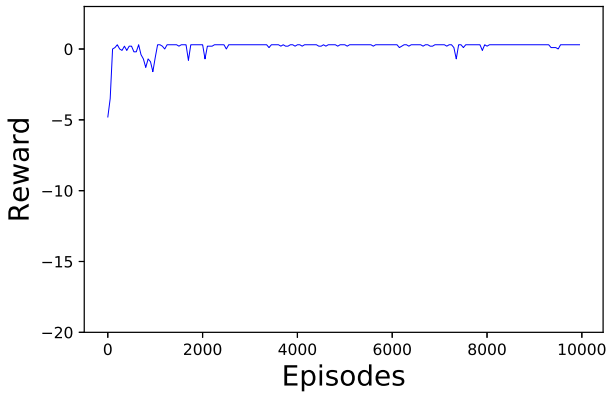

Fig. 5. Results from 10000 episodes one agent, two hidden layer with 24 neurons. The maximum reward was 0.3.

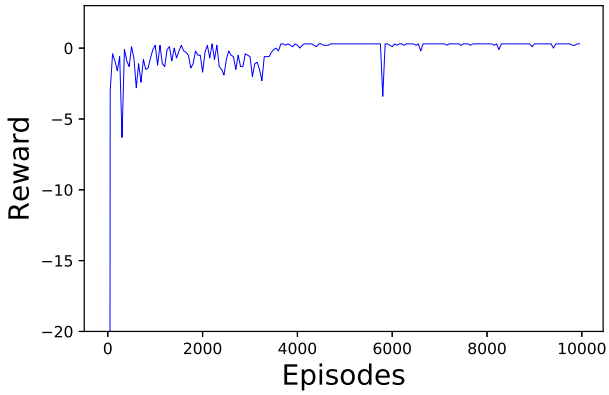

Fig. 6. Results from 10000 episodes one agent, two hidden layer with 24 neurons. The maximum reward was 0.3.

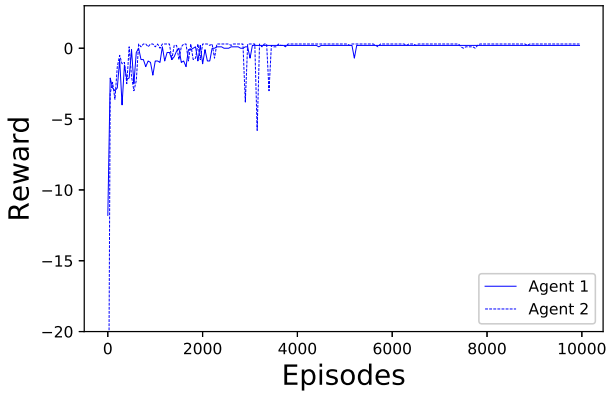

Fig. 7. Results from 10000 episodes two agents, two hidden layer with 24 neurons. The maximum reward for the agents were [0.2, 0.3].

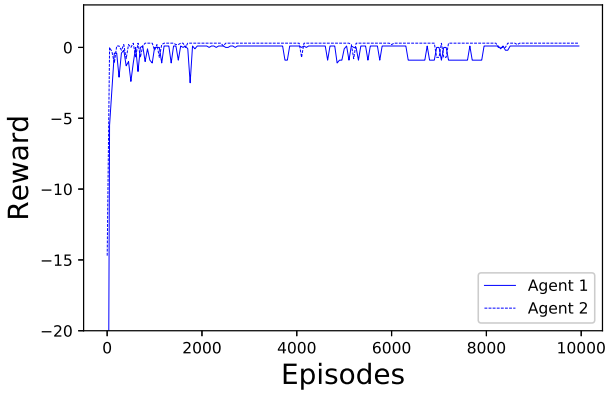

Fig. 8. Results from 10000 episodes two agents, two hidden layer with 32 neurons. The maximum reward for the agents were [0.2, 0.3].

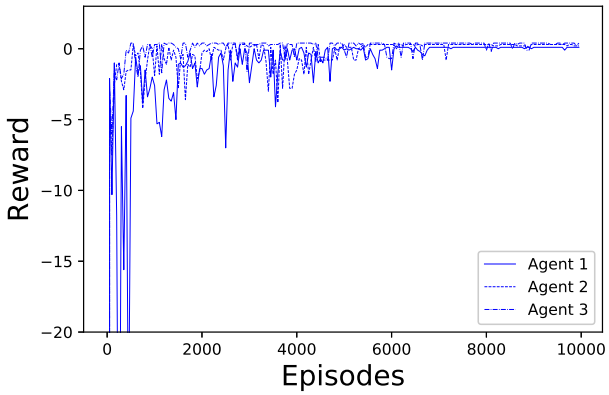

Fig. 9. Results from 10000 episodes thee agents, two hidden layer with 32 neurons. The maximum reward for the agents were [0.2, 0.3, 0.4].

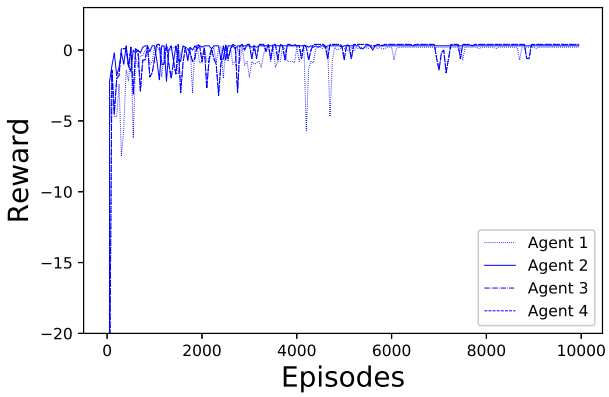

Fig. 10. Results from 10000 episodes four agents, two hidden layer with 32 neurons. The maximum reward for the agents were [0.2, 0.3, 0.4, 0.4].

TABLE II
AVERAGE PERFORMANCE/COMPUTATIONAL TIME FOR A LEARNING SEQUENCE OF ONE TO FOUR AGENTS.

No Agents / nodes	Global steps	Time (s)
1 / 24	180,000	110
1 / 32	180,000	125
2 / 24	380,000	250
2 / 32	390,000	260
3 / 32	730,000	390
4 / 32	900,000	500

TABLE III
AVERAGE NUMBER OF COLLISIONS PER 2000 EPISODES EVALUATED OVER TEN RUNS WITH LEARNED POLICIES.

No. agents	Obstacle collisions	Agent collisions
1	0	-
2	1.4	0
3	10.1	0.1
4	23.7	4.4

V. DISCUSSION

This section covers a direct analysis of the results given in the previous section along with concerns regarding generality and modelling of our algorithm. We also set the stage for possible future research on the subject.

A. Data Analysis

All of the measurements performed and the data gathered for the results were computed on a desktop with 4 cores, 2,2 GHz CPU and 16GB of RAM. The different times and performance measurements are with regards to that computational power. During this section of the report references to certain terms will be made, we begin with describing these. An episode means that all of the agent have reached their respective goals and a learning sequence is defined as completing 10000 episodes.

B. Learning sequences

Table II functions as an quick overview for the time spent on a learning sequence. The nodes defined in the table corresponds to the number of neural network nodes in the two hidden layers. Global steps represents the number of actions taken by all agents and the time was clocked from start to finish. The values of Table II are not from a single learning sequence but are instead composed of rounded averaged values from three sessions.

Fig. 5 until 10 shows the gathered results of a complete learning sequence. They are portrayed by graphs of rewards as a function of episode. The interval for all six figures have been set to be x-axis $= [0, 10^4]$ and y-axis $= [-20, 3]$. In all of the figures discussed here there have been large negative rewards especially during the first episodes that were not able to fit.

When increasing the number of agents to three or above the acquired policy could often lead to a special situation. The majority of the agents converged upon a reward near their respective maximum. The remaining agents would instead choose a policy leading to a negative reward caused for example by a collision. This behaviour is partly a consequence of the reward design in the algorithm. In MARL the design of a suitable reward signal is a central problem and we used a local one based on individual behaviour which is prone to educate selfish agents. They ignore the others needs in order to maximize their own reward and leaves them in a more difficult state [6]. A typical scenario in the learning sequence can look as in figure 11 where it is clear that when one agent chooses greedily others may suffer.

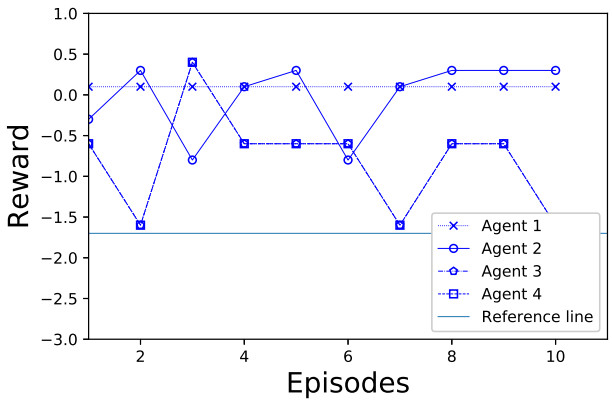

Fig. 11. Four agent environment, the reference line is at -1.7 on the reward axis and corresponds to the agents going the fastest way to goal with two collisions on the way.

C. Evaluation of Learned Policies

After having learned policies over the course of 20.000 episodes, these were evaluated based on their collision frequencies presented in table III. A collision is defined as one agent taking an action that puts it on top of either an obstacle or another agent, therefore; agents who stand still and collide do not constitute a collision.

Not surprisingly, we see an increase in the number of collisions when more agents are being used, although not substantial. These numbers could likely be lowered with help of more learning and fine tuning of α (see eq. (5)).

When visualizing the system, it is apparent that these policies constitute agents moving along paths with as much space between them as possible, thus lowering the chance of having agents collide. This as opposed to making more room between agents and obstacles, the reason being likely that the behaviours of these are easier to learn since their movements are deterministic.

D. Generality and Model Analysis

This project used the direct approach of attempting to map states to actions i.e. Policy Gradient Methods. As has been touched upon in the report, the method is susceptible to converging upon local optima rather than global ones. This was noticed as the agents sometimes would stay at certain positions and not continue to explore the grid. However, if an agent was to find its goal, it did not take much time for it to

afterwards find its optimal policy, though not necessarily after having explored the whole environment.

The learned policies do seem to generalize in up scaled environments but fail to be meaningful in different settings, i.e. repositioning of goals and starting positions. This however does not have to be a problem depending on the situation in which the algorithm were to be implemented.

E. Future Work

- A rather large bias that was introduced in our algorithm was how the network output was used in order to take actions, this could have been modelled differently e.g. instead taking its argmax and using an *epsilon greedy* [7] approach.
- It would be interesting to compare convergence rates between our approach and others, e.g. *deep Q-value networks* [8].
- Learned policies in less complex environments e.g. with less agents could be utilized in order to learn more complex policies.
- Since the problem has a direct graphical interpretation, an approach using *Deep Convolutional Neural Networks* [9] would be worth inspecting.
- Modelling the environment with more bias towards making collisions more frequent e.g. choke points could be worth exploring to evaluate behaviour of a more cooperative nature.
- In this project a constant learning rate was used to generate policies, taking into account that this parameter has a big impact on the result; it would be wise to consider ways to choose this with more care, e.g. a decaying scheme.
- Implementing a mixed reward signal i.e. a combination of a local and global one, e.g. methods mentioned in [6]. This could be compared to the approach considered in this report.

VI. Conclusions

- Finding optimal policies for multi agent systems does work up to a limited amount of agents when using policy gradient methods. Depending on the nature and complexity of the system, simplifications may have to be made.
- The initial neural network weights do affect the convergence rate greatly when networks are learning.
- Learned policies are sensitive to how rewards are defined during learning, if chosen poorly; unwanted behaviour does arise in the result.
- Learned policies do not generalize to new environments, though they may be utilized in similar settings to make the learning more effective.
- The learning rate i.e. α is hard to choose so as to not end up in a local maximum but still find the optimal policy.

Acknowledgment

The authors would like to thank Alexandros Nikou, Mina Ferizbegovic and Takuya Iwaki for their excellent feedback and help with organizing our work.

References

[1] (2017, Oct.) Alphago zero: Learning from scratch. DeepMind, London, UK. [Online]. Available: https://deepmind.com/blog/alphago-zero-learning-scratch/

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354 EP –, 10 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[3] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[4] J. Li, J.-h. Cheng, J.-y. Shi, and F. Huang, "Brief introduction of back propagation (bp) neural network algorithm and its improvement," in *Advances in Computer Science and Information Engineering*, D. Jin and S. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 553–558.

[5] (2018, Apr.) Reinforcement learning github. [Online]. Available: https://github.com/rlcode/reinforcement-learning

[6] H. Mao, Z. Gong, and Z. Xiao, "Reward design in cooperative multi-agent reinforcement learning for packet routing," 2018. [Online]. Available: https://openreview.net/forum?id=r15kjpHa-

[7] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *CoRR*, vol. abs/1402.6028, 2014. [Online]. Available: http://arxiv.org/abs/1402.6028

[8] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Z. Leibo, and A. Gruslys, "Learning from demonstrations for real world reinforcement learning," *CoRR*, vol. abs/1704.03732, 2017. [Online]. Available: http://arxiv.org/abs/1704.03732

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf