Post-print of: Z. Kowalczuk, J. Cybulski and M. Czubenko, "JamesBot - an intelligent agent playing StarCraft II," 2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR), Międzyzdroje, Poland, 2019, pp. 105-110, doi: 10.1109/MMAR.2019.8864611.

JamesBot – an intelligent agent playing StarCraft II

Zdzisław Kowalczuk
Faculty of Electronics
Telecommunications and Informatics
Gdansk Univesity of Technology
Gdansk, Poland
Email: kova@pg.edu.pl

Jan Cybulski
Faculty of Electronics
Telecommunications and Informatics
Gdansk Univesity of Technology
Gdansk, Poland

Michał Czubenko
Faculty of Electronics
Telecommunications and Informatics
Gdansk Univesity of Technology
Gdansk, Poland

Abstract—The most popular method for optimizing a certain strategy based on a reward is Reinforcement Learning (RL). Lately, a big challenge for this technique are computer games such as StarCraft II which is a real-time strategy game, created by Blizzard. The main idea of this game is to fight between agents and control objects on the battlefield in order to defeat the enemy. This work concerns creating an autonomous bot using reinforced learning, in particular, the Q-Learning algorithm for playing StarCraft. JamesBot consists of three parts. State Manager processes relevant information from the environment. Decision Manager consists of a table implementation of the Q-Learning algorithm, which assigns actions to states, and the epsilon-greedy strategy, which determines the behavior of the bot. In turn, Action Manager is responsible for executing commands. Testing bots involves fighting the default (simple) agent built into the game. Although JamesBot played better than the default (random) agent, it failed to gain the ability to defeat the opponent. The obtained results, however, are quite promising in terms of the possibilities of further development.

Index Terms—machine learning, reinforcement learning, Q-Learning, StarCraft II.

I. Introduction

„Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal" [1]. The main goal of the reinforced learning (RL) algorithms is to find the optimal strategy for a certain environment. Algorithms of this type allow for indirect programming of agents - teaching them on the basis of actions and observations. Reinforcement learning is successfully used in solving many practical problems, such as: drone control, face evolution in the image, or stock trading [2]–[4].

StarCraft II: Wings of Liberty, also briefly referred to as SC2 (along with two additional extensions), is a computer game released by Blizzard Entertainment in 2010, belonging to the genre called Real-Time Strategy (RTS). In addition to strategic and tactical elements, it also contains agility mechanisms. The game itself is very popular all over the world, mainly due to the interesting competition and a long history. There are even international tournaments in Starcraft II, in which a great number of players from all over the world take part (although the winners are mostly Koreans).

Currently, the largest SC2 championships are held during the Intel Extreme Masters 2019 in Katowice.

II. State of the art

Along with the progress of artificial intelligence, in addition to human players, an important part of the environment associated with StarCraft become virtual intelligent agents (called bots). Every year there are competitions in which several dozen bots take part in order to choose the best ones [5].

Until now, the best results have been achieved by scripting bots that implement the strategy planned by the programmer. However, this method is less and less used due to new developments in the context of learning with reinforcement [6]. The examples listed below characterize the development trends of intelligent bots playing SC2 based on reinforcement learning.

A. Baseline Reinforcement Learning Agents

In [7] the company DeepMind described its experience in the study of intelligent agents. To this end, they created their own environment StarCraft II Learning Environment [8]. In their tests they used many available bot architectures, e.g. Atari-net Agent – an agent who played on the Atari 2600 console, among others [9]. The research led to the conclusion that „the current best artificial StarCraft bots, based on the built-in AI or research on previous environments, can be defeated by even amateur players" [7]. However, since then, a new, more breakthrough architecture of bots has been created.

B. TStarBots

One of the more successful implementations is the bot called TStarBot1 [10], which also uses RL algorithms. However, instead of providing an agent with a whole set of states and actions (as in classic RL techniques), they are stored in the form of properly prepared macros. Macros representing actions usually consist of several commands. Thanks to this, the action space is much smaller, although each of the macros/actions still has its significant impact on the game. The states in this space can be defined in a vector or scalar way.

TStarBot1 uses techniques called Dueling Double Deep Q-Learning and Proximal Policy Optimization, which implement reinforced learning. In tests, the bot reached 100% win with a weak opponent built into the game and 81% with the strongest.

C. AlphaStar

The most advanced example of a StarCraft bot is AlphaStar, also created by DeepMind. Alphastar is „the first Artificial Intelligence to defeat a top professional player, in a series of test matches held on 19 December. AlphaStar decisively beat Team Liquid's Grzegorz 'MaNa' Komincz, one of the world's strongest professional StarCraft players, 5-0" [11]. This bot is a milestone and a huge breakthrough in the domain of artificial intelligence.

Earlier, despite the successes of AI against other bots, human opponents were beyond their reach. The bot's architecture is based on deep neural networks and reinforcement learning. AlphaStar has extensive experience in playing SC2 obtained by repeating human games and self-struggle, which can be estimated at 200 years in terms of duration of conducted test games.

III. Q-Learning Algorithm

The agent environment can be described in the form of the Markov Decision Process (MDP), containing sets of states, actions, transitions between states and their expected rewards [12]. Such a description fulfills the requirements of the Markov process, which says that the future state depends only on the current state [13]. In this case, the main purpose of RL is to find the optimal strategy for a given MDP.

An important element of this process is the reward function $Q_\pi(S, A) = Q(S, A)$ which tells us about the expected reward value. This function depends on the current state S and the action A selected by the agent that relies on the specified strategy π.

The Q-Learning algorithm is one way to specify the reward function $Q(S, A)$.

After initializing the state (S_0) and the action (A_0), an example of a single iteration of such an algorithm may look like this:

1) at the time t, the agent in the state S_t executes the selected action A_t based on the adopted strategy π,
2) at the next moment $t + 1$, the agent receives a new environment state S_{t+1} and reward R_{t+1} for the previously performed action,
3) then the reward function $Q(S, A)$ is updated according to the following rules:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \Delta Q), \qquad (1)$$

where α is a learning rate, ΔQ describes the increase in the function Q by the specified discount factor, or prediction rate, γ [1]:

$$\Delta Q = \gamma \max_A(Q(S_{t+1}, A)) - Q(S_t, A_t), \qquad (2)$$

4) return to step 1 to implement the updated strategy expressed by the new value of $Q(S, A)$.

With the play of episodes that are a finite sequence in the MDP, the value of $Q(S, A)$ should coincide with the desired value, i.e. the agent's strategy should approach the optimal one.

IV. JamesBot architecture

The computer environment in which the bot was implemented consists of the Python programming language with the PySC2 library, i.e. the previously mentioned StarCraft II Learning Environment.

The bot was designed in form of isolated elements/subsystems that fulfill separate functions:

1) State Manager (SM) processes input data,
2) Decision Manager (DM) selects the action based on the state S and the function $Q(S, A)$, which evolves during learning,
3) Action Manager (AM) processes the decisions taken in DM for environmentally understandable actions PySC2.

This hierarchy clearly indicates the separate stages of the operation of the entire system. In the beginning, the environment (SC2) performs its algorithm (the default operation of the game) and then transfers control to the agent. In addition, two conditions are checked. First, whether the environment has reached its final state, i.e. all bot operations have already been completed. The second condition: checking whether the bot performs the selected action – meaning the need to pass the control to AM, and if not – transferring control to the MS.

In the situation where the bot does not perform any action, the observation obtained from the environment goes to the SM module. Then the information processed by this subsystem goes to the DM, which implements the RL algorithm, i.e. it updates the function $Q(S, A)$ and selects a new action.

The optimally selected action is further carried out by AM, and the orders (in the form intelligible to the API) are sent to the environment. Connections between subsystems and the flow of control logic are illustrated in Fig. 1.

A. State Manager (SM)

The task of SM is to retrieve information from the environment and its further processing. It is a subsystem intended for the analysis of observations sent by PySC2. To reduce the analyzed data space, the number of environmental variables was limited. The following variables affect the bot's status:

- number of barracks, supply depots, command centers, workers (SCV),
- the value of supplies consumed by combat units,
- positions of own and enemy objects.

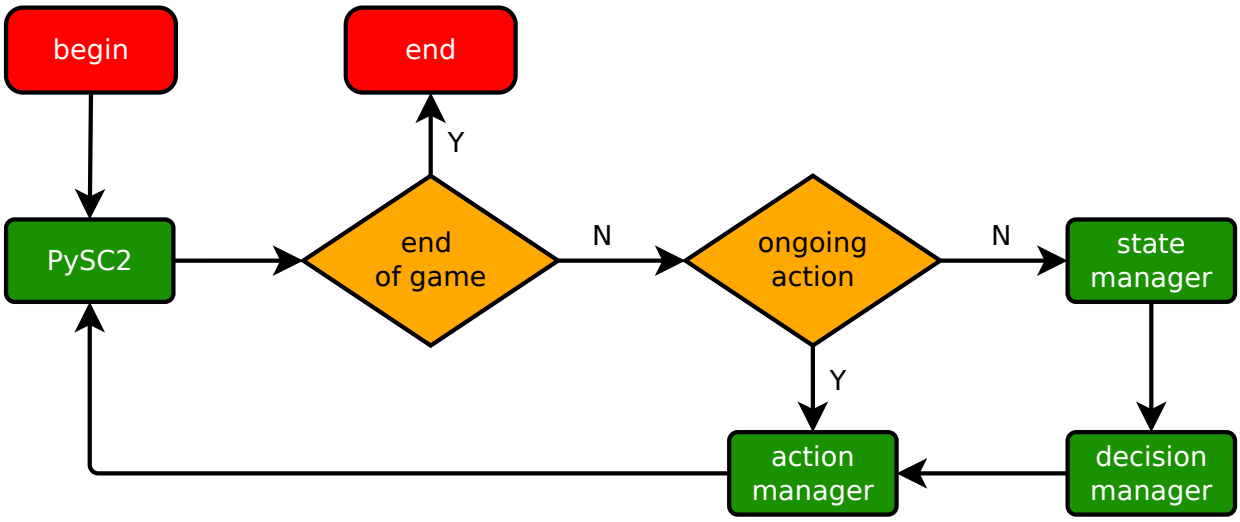


Fig. 1: Block diagram describing the flow of control logic between the agent subsystems: the red blocks indicate the beginning and end of the activity, the green blocks – the bot subsystems, and the orange diamonds are predictors (conditional blocks).

On the basis of information received from the environment, the SM module determines which activities are impossible to perform (for example due to resource requirements). In this way, the list of available actions is dynamically modified during the operation of the bot. Note that in a similar way (with gain controlled/programmed by a scheduling variable) when modeling the brain, its resources can be controlled by a suitably selected variable or state that represents the concept of emotion [14]–[18].

B. Decision Manager (DM)

Decision Manager is based on the Q-Learning algorithm and the ϵ-greedy strategy. Implemented here is a table version of the Q-Learning algorithm, in which each action-state pair is assigned the appropriate reward value in a matrix representing the function $Q(S, A)$.

This approach has two advantages: simplicity of implementation and speed of operation. On the other hand, the representation used here is quite demanding in terms of memory and has an obvious limitation when there are continuous-time elements. The ϵ-greedy strategy is based on the principle that a random action is selected with a certain low probability ϵ, otherwise (that is with the probalibity $1 - \epsilon$) – the best action is taken. Such a mechanism ensures a higher level of stochastic exploration of the environment.

C. Action Manager (AM)

Like SM, Action Manager aims to separate the decision element from the game environment. The main task of such a separator (interface) is to serve the implementation of actions selected by the bot. The main function of AM is sending the order directly to the game environment.

From the point of view of the decision-making subsystem (DM), possible actions are abstract (at the meta level), e.g. 'build barracks' or 'train a unit'. In fact, such activities are more complex and consist of many orders and decisions. For example, questions arise: which worker should build barracks, where to place the building, etc.

Therefore, the AM module, limiting the decision manager's workspace, processes the abstract actions provided to him in order to obtain detailed instructions understandable for the SC2 game environment. Among the predesigned and implemented actions, the following can be distinguished:

- building barracks and supply depots,
- training combat units (Marines) and workers (SCV),
- attacking the selected positions.

In addition, the possibility of skipping or resigning from the action execution has also been implemented.

V. Tests

A number of tests were carried out for the analysis of the JamesBot's learning process and effectiveness. Each of them consisted of a number of multiplayer games. It was decided on the typical kind of game that is used in classic e-games – „one on one", i.e. the bot against its opponent. In terms of concept and difficulty, this is the simplest form of gameplay.

In this game the battlefield was a map, known as Simple64. The simple bot built into SC2 representing a random race was chosen as the agent's opponent. On the other hand, the JamesBot itself represented a previously determined race, Terrans – meaning people using advanced robotics achievements. Additionally, for comparative purposes, a random bot was also implemented.

A. Basic assumptions

To further simplify the computing concept of JamesBot, limitations such as the number of buildings, warehouses, barracks, SCVs and other parameters, like APM (Actions Per Minute), were introduced. The limitation of APM is primarily aimed at maintaining the reality of human behavior. The applied types of constraints used and their values can be found in tab. I. The last unknown, requiring determination are the parameters of reinforcement learning. As the bot uses the Q-Learning algorithm along with the ϵ-greedy strategy, the parameters such as: the learning factor α, the discount coefficient γ, the coefficient of randomness ϵ and the reward value R remain to be specified. The values of the above-mentioned parameters are summarized in tab. II.

TABLE I: JamesBot limitation parameters applied in the tests.

Bot	SCV	buildings	barracks	supply	APM
Random	20	6	2	4	62
JamesBot		8		6	

TABLE II: The RL and DM parameters of JamesBot used in tests.

α	γ	ϵ	R_{step}	R_{win}	R_{draw}	R_{loss}
0.01	0.9	0.1	-1	50000	0	-50000

B. Results

The random bot's gameplay consisted of choosing each action with equal probability regardless of what was happening on the battlefield (actions were the same like in the case of JamesBot). The graphs are presented in Fig. 2. The random bot was not able to achieve the winnings, and due to the simplicity of its strategy, it managed to get about 20% of draws to 80% of losers (losses). The test games ended after about 200 episodes, when the effectiveness of the bot stopped changing.

The performance quality indicators of the developed autonomous agent JamesBot are shown in Fig. 3. Clearly, they go above the level of effectiveness of the random bot. The increasing rewards of JamesBot can be observed in Fig. 3a. Between 1200 and 1400 episode it can be observed that the bot kept its effectiveness at the level near 45%. For the last 100 games JamesBot has achieved 33% winnings, 21% draws and 46% losses. Its learning process ended after a period of 2000 episodes.

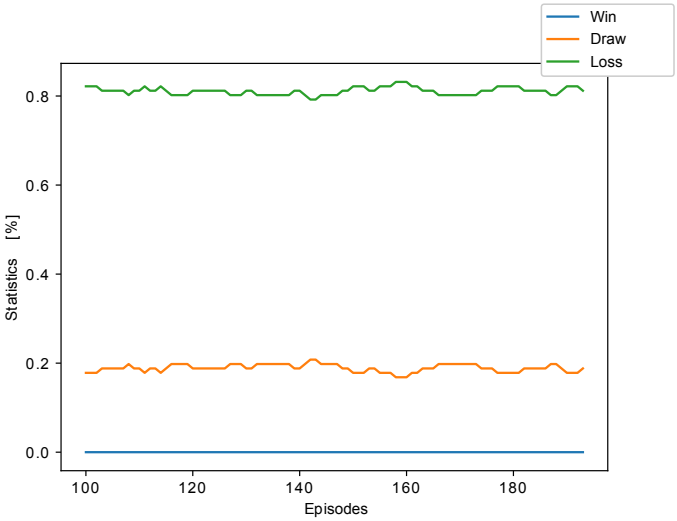


Fig. 2: Statistics of random bot; winnings (blue), draws (orange), losses (green).

In order to determine the overall assessment of the effectiveness of the bot, the following winning indicator (achievement of victory) was introduced:

$$J = \frac{3 * W + 1 * D + 0 * L}{3} \tag{3}$$

where W – wins [%], D - draws [%], a L - losses [%]. The list of the measured partial parameters of the quality of the JamesBot along with the winning indicator of bots (J) is presented in tab. III.

TABLE III: List of performance parameters derived from the last 100 games of the two bots.

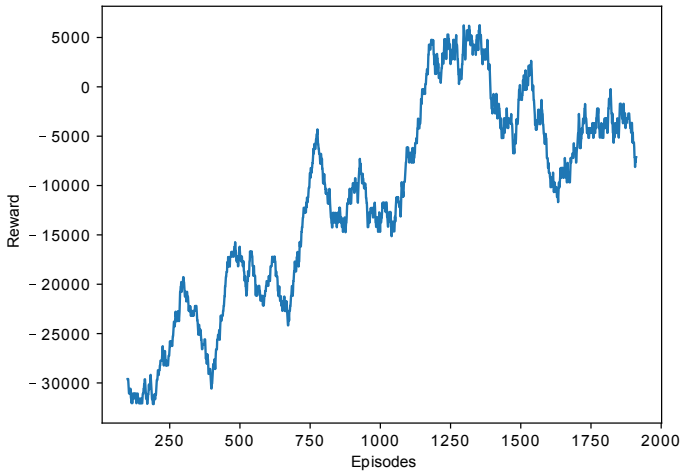
Bot	Wins [%]	Draws [%]	Losses [%]	J [%]
Random	0	20	80	6.7
JamesBot	33	21	46	40

VI. Conclusions

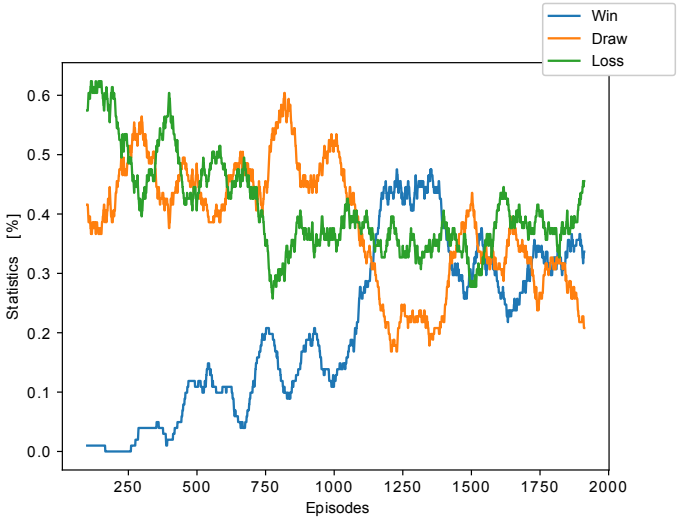
Thanks to the use of the Q-learning technique, the developed and implemented JamesBot was able to gain an advantage over a fairly simple random bot and a good absolute result in terms of achieved performance representing winning skill (40%).

It should be noted that the presented solution of the agent decision system is not too complicated, therefore its actions can be easily interpreted by a human (in contrast to applications based on deep neural networks).

With the current state of the art, the Q-learning method is quite easy to implement in various programming languages. It has a low demand for resources (it does not require high computing power as opposed to neural networks), with the exception of large memory usage.



(a) The reward from the last 100 games.



(b) Wins (blue), draws (orange), losses (green) from the last 100 games.

Fig. 3: JamesBot results.

What's more, it's easy to interpret the State-Action Matrix from which the agent derives action. The elements of this matrix contain values corresponding to state-action pairs, where the higher the value, the more appropriate is the given action for a given state (from the perspective of the agent's strategy). Thanks to this, you can, for example, determine the trajectory of this value to see the evolution of the decision making process by the agent. Such a result can not be obtained using neural networks.

The presented method does not work very well (in terms of repeatability, for example) due to the high degree of randomness in the applied bot solution on the one hand and the relatively high complexity of the considered game problem on the other hand.

The main intention of this work was at least to beat a completely random bot. This goal has been achieved. In addition, it turned out that the use of Q-Learning to solve even such a complex game problem seems promising.

Analyzing the possible sources of the agent's imperfections, we can mention three elements that could affect unsatisfactory results. The first source can be the implementation of the Q-Learning algorithm. The state-action matrix method works well in simple environments where the number of states and actions is small [1].

In the applied algorithm (in the context of the SC2 game), a better solution might be to use a neural network estimating the $Q(S, A)$ function values. This method is called Deep Q-Network (DQN) [19]. We can also benefit from a more sophisticated approach, such as described in [20], which uses multiple networks DQN.

In addition, frequent updates of the function $Q(S, A)$ can cause a phenomenon when unnecessary/inappropriate actions are also rewarded and the size of the reward does not approach the optimal value. A possible solution to this problem may be rarer updates of the reward function – only in the most important moments of the game, such as winning a battle or reaching the SCV limit.

The cause of the imperfections can also be attributed to an insufficient set of actions, states and rewards. A larger number of actions to choose from would help improve the system's functionality. You can also use well-chosen information from the battlefield to build a better strategy. The principle of sporadic rewarding (with small values) may allow the bot to learn tactics more freely. Obtaining significant rewards at the end of each episode slows the bot's learning. System rewards, depending on the condition of the army, the size of resources and overall success on the battlefield, should be more beneficial from the point of view of the ultimate goal of the game.

References

[1] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. MIT press, 2018.

[2] H. X. Pham, H. M. La, D. Feil-Seifer, and L. V. Nguyen, "Autonomous uav navigation using reinforcement learning," arXiv preprint arXiv:1801.05086, 2018.

[3] C. N. Duong, K. Luu, K. G. Quach, N. Nguyen, E. Patterson, T. D. Bui, and N. Le, "Automatic face aging in videos via deep reinforcement learning," arXiv preprint arXiv:1811.11082, 2018.

[4] Z. Xiong, X.-Y. Liu, S. Zhong, A. Walid et al., "Practical deep reinforcement learning approach for stock trading," arXiv preprint arXiv:1811.07522, 2018.

[5] D. Churchill, "Aiide starcraft ai competition," 2018. [Online]. Available: "url–http://www.cs.mun.ca/˜dchurchill/starcraftaicomp/"

[6] M. Čertickỳ and D. Churchill, "The current state of starcraft ai competitions and bots," in 13th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference, Little Cottonwood Canyon, 2017.

[7] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, 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, and R. Tsing, "Starcraft ii: A new challenge for reinforcement learning," arXiv preprint arXiv:1708.04782, 2017.

[8] DeepMind, "Starcraft ii learning environment," 2018. [Online]. Available: https://github.com/deepmind/pysc2

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," arXiv preprint arXiv:1312.5602, 2013.
[10] P. Sun, X. Sun, L. Han, J. Xiong, Q. Wang, B. Li, Y. Zheng, J. Liu, Y. Liu, H. Liu, and T. Zhang, "Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game," arXiv preprint arXiv:1809.07193, 2018.
[11] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II," 2019. [Online]. Available: "url–https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/"
[12] D. Silver, "Reinforcement learning lecture 2: Markov decision processes," University College London, 2015. [Online]. Available: "url–http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching˙files/MDP.pdf"
[13] S. N. Ethier and T. G. Kurtz, Markov Processes: Characterization and Convergence. John Wiley & Sons, 2009, vol. 282.
[14] Z. Kowalczuk and M. Czubenko, "Emotions embodied in the SVC of an autonomous driver system," IFAC-PapersOnLine, vol. 50, no. 1, pp. 3744–3749, 2017.
[15] M. Czubenko, A. Ordys, and Z. Kowalczuk, "Autonomous driver based on intelligent system of decision-making," Cognitive Computation, vol. 7, no. 5, pp. 569–581, 2015.
[16] Z. Kowalczuk and M. Czubenko, "Computational approaches to modeling artificial emotion–an overview of the proposed solutions," Frontiers in Robotics and AI, vol. 3, no. 21, pp. 1–12, 2016.
[17] ——, "Intelligent decision-making system for autonomous robots," International Journal of Applied Mathematics and Computer Science, vol. 21, no. 4, pp. 621–635, 2011.
[18] ——, "An intelligent decision-making system for autonomous units based on the mind model," in 23rd International Conference on Methods & Models in Automation & Robotics (MMAR). Mędzyzdroje, Poland: IEEE, 2018, pp. 1–6.
[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, p. 529, 2015.
[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in International Conference on Machine Learning, 2016, pp. 1928–1937.