2021 29th Mediterranean Conference on Control and Automation (MED)
June 22-25, 2021. Bari, Puglia, Italy

2021 29th Mediterranean Conference on Control and Automation (MED) | 978-1-6654-2258-1/20/$31.00 ©2021 IEEE | DOI: 10.1109/MED51440.2021.9480232

A Deep Q-Learning based approach applied to the Snake game

Alessandro Sebastianelli, Massimo Tipaldi, Silvia Liberata Ullo, Luigi Glielmo *

***Abstract*— In recent years, one of the highest challenges in the field of artificial intelligence has been the creation of systems capable of learning how to play classic games. This paper presents a Deep Q-Learning based approach for playing the Snake game. All the elements of the related Reinforcement Learning framework are defined. Numerical simulations for both the training and the testing phases are presented. A particular focus is given to the associated Neural Network hyperparameters tuning, which is a crucial step in the agent design process and for the achievement of a desired target level of performance.**

I. Introduction

During the last years, Deep Reinforcement Learning (DRL) has been widely used in different fields, such as robotics, self-driving cars, industry automation, remote sensing, and games [1], [2], [3].

Reinforcement Learning (RL) is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. It formulates the environment as a Markov decision process (MDP), without having the exact knowledge of its underlying dynamics. RL based approaches solve large multi-stage decision and control problems, involving one or more decision makers (or agents). They deal with problems that could be addressed in principle by Dynamic Programming (DP), but are solved in practice by using methods able to handle the so-called *curse of modelling and dimensionality* [4], [5], [6].

DRL enables RL to handle problems with high-dimensional state and action spaces. In particular, thanks to the usage of deep neural networks, DRL solutions allow representing high-dimensional state and action spaces via compact low-dimensional (and feature-driven) function approximations [1], [4].

The combination of the Q-Learning algorithm [5] and deep neural networks has given rise to the Deep Q-Learning, where neural networks are used to approximate Q-value functions and policies [1], [7], [8]. The resulting Deep Q-learning Networks (DQNs) are trained and tested by using real (or simulated) data, and can be used as agent to operate in the real environment. Such approaches can also be useful for applications with strict safety requirements, which can impose constraints to the experience-driven learning methods [9], [10]. In such cases, it is appropriate to train the designed DQN via numerical simulations, before operating in the actual environment.

This work proposes a Deep Q-Learning based approach for the Snake game [11], [12]. As shown in [1], DQNs have been used to learn how to play different video games directly from image pixels. DQN-based solutions can surpass all the previous solutions when applied to different video games [13], [14], [15]. The main difference between our approach and the ones available in the literature regards the RL system formulation of the Snake game. More specifically, the architectures shown in, e.g., [16], [17] are based on CNNs (Convolutional Neural Networks) since the agent has to be trained by means of image pixels. Sequences of such images can be processed to extract spatio-temporal features of the snake environment. On the other hand, our DQN architecture uses sensor measurement data, thus complex CNNs are not needed. Such data are used to formulate and compute the RL system elements (e.g, the current state and the reward received from the environment).

The paper is organised as follows. Section II introduces the reader to RL and DQN. Section III describes the proposed RL system for the Snake game, including the definition of the reward function. Section IV describes the Deep Q-Learning based agent. Section V presents the numerical simulations and results. It also provides an analysis about the tuning of the DQN hyperparameters. Section VI concludes the paper.

II. Background on RL and Deep Q-Learning

The Machine Learning branch called Reinforcement Learning (RL) offers a wide range of computational approaches in which an agent learns an optimal control strategy (or policy) by the direct interaction with its environment [18]. The RL agent iteratively observes a representation of the environment's state, executes an action, and receives a scalar reward signal from the environment. Usually, the agent's objective is to learn a proper policy by maximising the expected cumulative reward it receives over time. The reward function definition incorporates the specification of the control problem to be solved and has to indicate what we want to accomplish.

A Markov Decision Process (MDP) is a classical formulation of sequential decision making problems, and thus it is the straightforward framing of RL problems. An MDP can be defined as a tuple $\langle \mathcal{S}, \mathcal{A}, R, P, \gamma, p_0 \rangle$, where $\mathcal{S}$ is the state space (with $s \in \mathcal{S}$ its generic state), $\mathcal{A}$ is the action space (with $a \in \mathcal{A}$ its generic action), $R(\cdot|s,a)$ is the reward probability distribution, $P(\cdot|s,a)$ is the state transition probability distribution, $0 < \gamma < 1$ is the discount factor, and p_0 is the initial state probability distribution.

The agent's policy $\pi(\cdot|s) := \mathcal{S} \rightarrow \Delta_{\mathcal{A}}$ is a mapping from states to probability distributions over the whole action space $\mathcal{A}$, completely determining the control strategy. Note that this definition includes the case of a deterministic policy

* The authors are with the Department of Engineering, University of Sannio, Benevento, Italy. *E-mail*:{sebastianelli, mtipaldi, ullo, luigi.glielmo}@unisannio.it

$\pi(\cdot) := \mathcal{S} \rightarrow \mathcal{A}$. More specifically, the interaction between the agent and the environment proceeds as follows: at each time instant $t = 1, 2, \ldots$, the agent receives a representation of the environment state $s_t \in \mathcal{S}$, stochastically selects an action $a_t \sim \pi(\cdot|s_t)$ (where the symbol $\sim$ stands for 'belonging to'). Then, it receives a stochastic reward $r_{t+1} \sim R(\cdot|s_t, a_t)$, and enters into a new state $s_{t+1} \sim P(\cdot|s_t, a_t)$. Note that the subscripts t and $t+1$ are used to indicate the system state values and the selected actions over a specific episode. Moreover, we denote with S_t and R_t the discrete time indexed random variable associated to $P(\cdot|s, a)$ and $R(\cdot|s, a)$, respectively.

The agent aims at finding an optimal policy, i.e., a policy that maximises the expected cumulative reward (or in short, return) over a predefined time horizon N. As explained in [18], episodic tasks (with a finite time horizon N) and continuing tasks (with $N \rightarrow \infty$) can be represented by the same notation considering the episode termination to be the entering of a special absorbing state with a reward equal to zero. As a consequence, we can use the following definition for the total return

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \quad (1)$$

We define $Q^\pi(s, a)$ as the Q-value function of a given policy π. It corresponds to the expected return upon taking an action a in state s, and afterwards following the policy π, i.e.,

$$Q^\pi(s, a) = \mathbf{E}_\pi[G_t \mid S_t = s, A_t = a] \quad (2)$$

The optimal policy π^* is the one maximising $Q^\pi(s, a)$ and satisfies the following Bellman equation [5], [18]

$$Q^*(s, a) = \mathbf{E}\Bigg[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q^*(S_{t+1}, a') \Bigg| S_t = s, A_t = a\Bigg]. \quad (3)$$

The Q-learning algorithm allows the computation of the optimal Q-value function $Q^*(s, a)$ by approximating the Bellman equation (3) by sampling and simulation [5]. In particular, by generating an long sequence of state-control pairs (s_t, a_t), the estimated Q-factor $Q(s_t, a_t)$ is updated according to the following rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)\right], \quad (4)$$

where α is the learning rate and r_{t+1} is the reward signal received by the agent when performing the action a_t in the state s_t. The selection of the action a_t is an important aspect. In this regard, the exploration-exploitation trade-off can be applied. In particular, the agent can exploit what it has already learned or it can explore unseen scenarios [18].

The Q-learning algorithm at its simplest stores the Q-value in tables, see Fig. 1. Its combination with deep neural networks has given rise to the Deep Q-Learning, where neural networks are used to approximate Q-value functions [1]. More specifically, Deep Q-Learning algorithms aim at computing the agent policy $\pi(\cdot|s)$ represented via a deep neural network, see Fig. 1.

As for the Snake game, a Deep Q-Learning based approach is necessary to handle its complexity and the large number of states. Moreover, there is no a priori knowledge about the optimal policy, thus a compact representation of the policy via neural networks can facilitate the learning process.

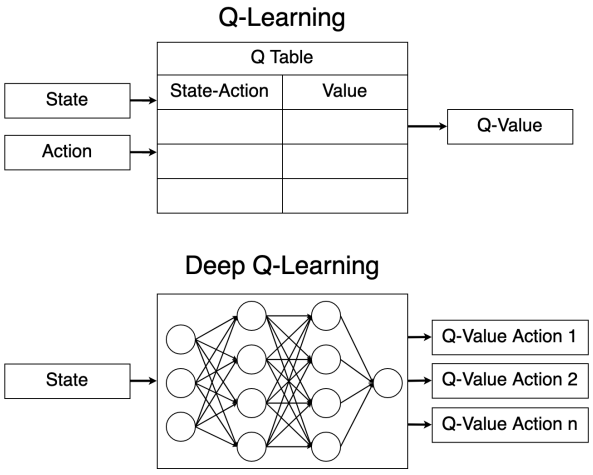

Fig. 1. Q-Learning vs. Deep Q-Learning.

III. The proposed RL system for the Snake game

This section describes all the elements of the RL system for the Snake game. To start with, its environment is composed of the game board, the fruit, and the snake itself. The game board is a $20x20$ grid. Each cell can contain either a fruit or the snake's head or an element of the snake's tail. The game board walls can not be crossed by the snake.

During the game a fruit is spawned randomly on the board. Whenever it is eaten by the snake, the agent receives a reward equal to 10. There can only be one fruit at a time on the board. If the snake eats the fruit, a new fruit is spawned in a random position.

Each game starts with a one-piece long snake. Whenever the snake eats a fruit, the game points counter is incremented by 1 and the tail increases by 1 piece. If the snake bumbs into a wall or its tail, the game ends and the game points counter is reset to 0. Moreover, the total reward is decreased by 10. Note that the reward is used to train the DQN based agent, while the game points counter to measure the performance of a given agent.

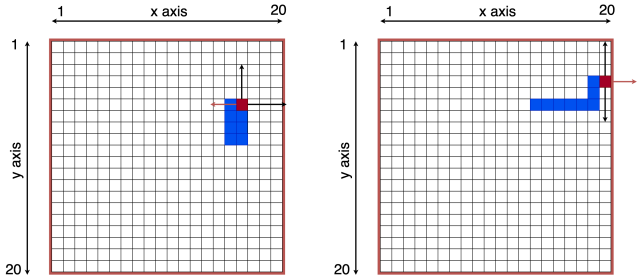

Fig. 2. Example of dangerous positions for the snake.

The state is a vector of 11 binary values defined as follows

$$\begin{aligned} x(k) = [&danger\ straight,\\ &danger\ right,\\ &danger\ left,\\ &moving\ left,\\ &moving\ right,\\ &moving\ up,\\ &moving\ down,\\ &food\ left,\\ &food\ right,\\ &food\ up,\\ &food\ down]. \end{aligned} \tag{5}$$

The first three entries of the state vector indicate if snake is in a dangerous position, which happens when the snake is close to a wall or to its tail (see Fig. 2). Thanks to its on-board proximity sensors (e.g. ultrasonic sensor), the snake can perceive the presence of the wall (or of its tail) located in a cell adjacent to its head. The next four entries show how the snake moves:

- If the x-velocity is equal to 1, then the snake moves to its right and the moving right entry is set to 1.
- If the x-velocity is equal to -1, then the snake moves to its left and the moving left entry is set to 1.
- If the y-velocity is equal to 1, then the snake moves downwards and the moving down entry is set to 1.
- If the y-velocity is equal to -1, then the snake moves upwards and the moving up entry is set to 1.

The snake position update rule is given by the following expressions

$$x_{pos}(t+1) = x_{pos}(t) + x_{velocity} \tag{6a}$$
$$y_{pos}(t+1) = y_{pos}(t) + y_{velocity}, \tag{6b}$$

where $x_{velocity}$ and $y_{velocity}$ are set to 1.

The next four entries are related to the position of the snake's head with respect to the fruit (see Fig. 3). Such information can be determined via fruit localization sensors (e.g. sonar, radar, lidar etc.). For instance, if the snake head x-position is greater than the fruit x-position, the "food left" entry is set to 1. For each state, the agent can perform the following three actions (see Fig. 4): Straight, Left, and Right.

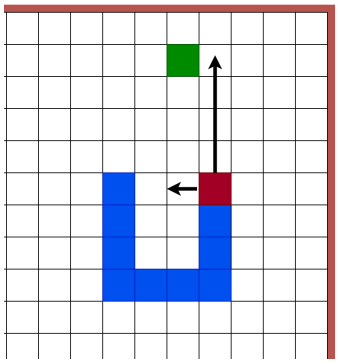

Fig. 3. Example of the snake approaching the fruit.

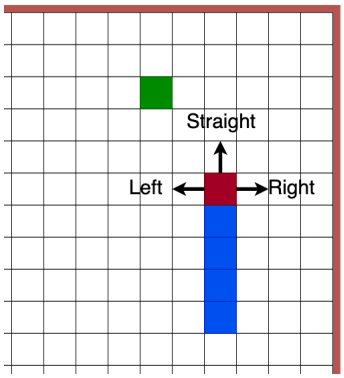

Fig. 4. Example of the action space.

The action selected at a given time t affects the snake state entries related to its movement.

IV. The Deep Q-Learning Based Agent for the Snake Game

This section describes the proposed DQN based agent architecture and the algorithm used to learn how to play the Snake game. The agent is a deep neural network used to represent the agent policy $\pi(\cdot|s)$ and to determine the best action to be executed at each state. This policy is learnt by performing a significant number of training episodes (i.e., the actual games). Every time the agent executes an action, the environment provides a reward. Such reward can be positive or negative, based on how adequate the action was from that particular state. The goal of the training process is to learn which action maximises the expected total reward from each state.

The DQN agent architecture is shown in Fig. 5. It is composed of 5 fully connected layers. The first layer is an input layer, composed of 11 nodes used for feeding the network with the state vector entries. The next three layers are used to extract the features and to predict the best action for each visited state. These layers have 100 fully connected nodes. Each layer is followed by a ReLU (Rectified Linear Unit) and a Dropout layer. The last layer is the output layer, and predicts the best action at the given state by using the softmax function [18]. The algorithm used to train the DQN

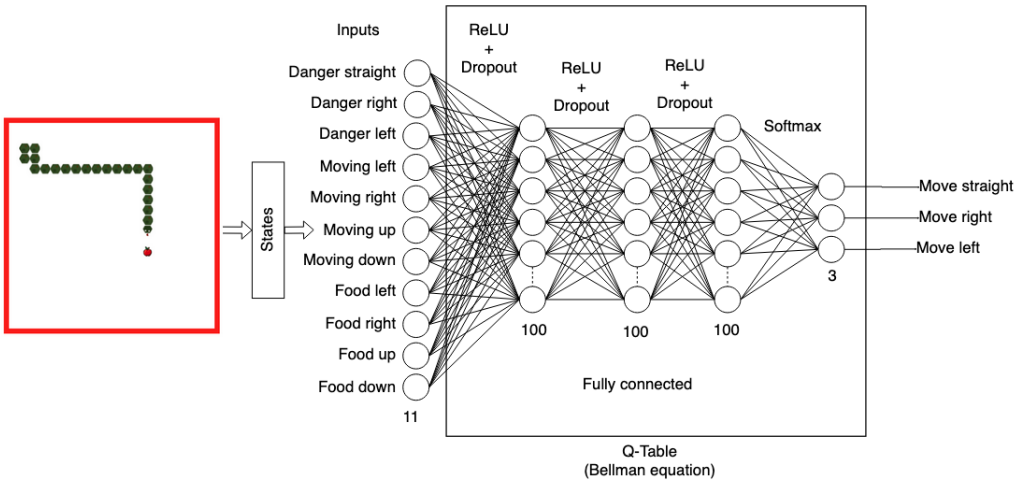

Fig. 5. Deep Neural Network architecture.

based agent can be summarised as follows:

1) At the start of the DQN based agent training process, the Q-values $Q(s, a)$ are randomly initialised.
2) For each epoch (which corresponds to a game), and for each visited state:
   a) The agent selects and executes an action by using the exploration-exploitation trade-off [18].
   b) Based on the selected action, a reward signal is generated and the system enters into a new state.
   c) The agent stores the original visited state, the selected action, the reward signal, and the new state.
   d) The steps a), b), and c) are repeated until the game ends.
3) After each epoch, the deep neural network of Fig. 5 is trained by using a subset of data collected during step (2).

Such deep neural network needs both the input and the expected output values to be trained. In this regard, during the training phase, the expected output values are computed by using the expression (4). The network is trained via the Back Propagation approach, with the Adam optimiser [19] and the Mean Square Error (MSE) as loss function.

After the training phase, the resulting deep neural network can be used to play new games. This way, we can validate the agent architecture configuration and test its performance. Like the training phase, the agent processes the state entries and predicts the outcome of each action. The one maximising the neural network output is chosen and executed.

V. Numerical Simulations and Results

This section presents the numerical simulation results and discusses the performance achieved by the proposed DQN based agent for the Snake game. The performance values computed for specific configurations of the deep neural network are also shown.

The numerical simulator has been developed in Python by means of the Pygame library. This library has been used to implement the GUI (Graphical Users Interface) and the game rules. Fig. 6 shows the GUI of the Snake game. The agent of Fig. 5 has been implemented by using the Tensorflow-Keras library. Thanks to the simulator, it is possible to monitor

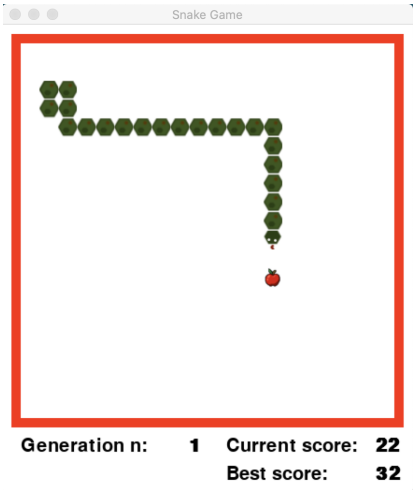

Fig. 6. The Snake game GUI.

the learning process by visual interpretation and numerical analysis. The GUI shows the score of the on-going game and the best score within a given game session (note that a session can coincide with the overall training). Fig. 7 highlights the performance improvement of the DQN based agent during the training process, e.g., after 79 epochs, the best achieved score is 52.

A. The hyper-parameters tuning process

In order to select the best neural network hyperparameter values, different model configurations were trained and tested. Table I shows 13 different model configurations. The

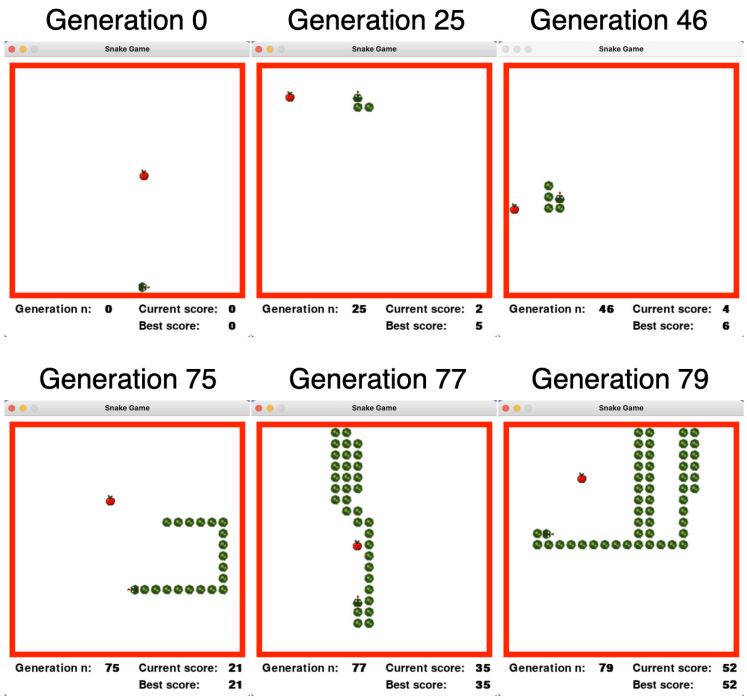

Fig. 7. The Snake Game - Training of Deep Q-Learning Network.

first column of the table defines the model configuration identifier, the second one defines, the ϵ parameter used for the exploration-exploitation trade-off, the next three ones define the number of nodes for each layer, the seventh one indicates the dropout value ("-" means that the dropout layer is bypassed), the eighth one indicates the number of epochs within the training session. The ninth column defines the number of state-action-reward-next state samples stored during the epoch. The next column represents the size of each batch used for the actual training. The last column indicates the average score for 5 different tests performed for each trained configuration. The best configuration 11 achieves the best mean score of 56, see Table I. To prove the effectiveness of this configuration, further 30 tests were carried out. These tests, grouped in five plots, are showed in Fig. 8; the best reached score was 62. Fig. 8 also shows an interesting behaviour, indeed each test shows that, for about the first 70 epochs of training, the score is approximately equal to 2; then, for the following epochs the score starts to increase very rapidly. This aspect can be explained by different factors, such as the choice of the loss function and the optimiser. Finally, we can note that configuration 12 achieves poor performance values: this can be explained by the fact that its exploration ability has been reduced (see parameter ϵ in Table I).

B. The Dropout layer role

Regularisation is a machine learning approach and prevents model over-fitting by adding a penalty to the loss function. Thanks to this, the model can be trained in a way that it does not learn an interdependent set of features weights [20].

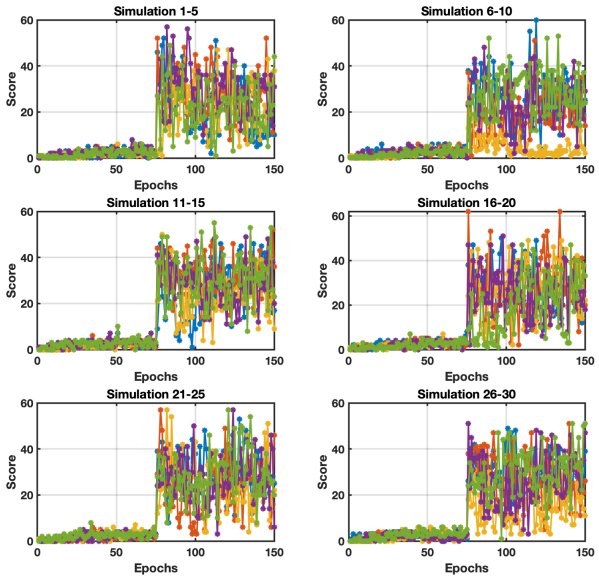

Fig. 8. Training trends of 30 simulations, each composed of 150 epochs.

Despite this, the best performance was achieved by the configuration 11, which does not use any dropout layer. In fact, during the various tests, it emerged that the absence of the dropout layers caused a different behaviour of the snake. Indeed, in case of no dropout, the snake tended to move by keeping the portions of his tail close together. On the other hand, when the dropout layers were used, the snake moved by keeping the different parts of the tail quite far from each other. This phenomenon means that, as the tail

Config.	ε	l.r.	Nodes 1st	Nodes 2nd	Nodes 3rd	Dropout	Epochs	Memory Size	Batch Size	Best Mean Score
0	1/75	0.0005	150	150	150	0.3	150	2500	500	44.0
1	1/75	0.0005	100	100	100	0.3	150	2500	500	41.6
2	1/75	0.0005	200	200	200	0.3	150	2500	500	36.0
3	1/75	0.0005	50	50	50	0.3	150	2500	500	33.4
4	1/75	0.0005	100	100	100	0.3	400	2500	500	47.8
5	1/75	0.0005	100	100	100	0.3	150	3500	500	38.2
6	1/75	0.0005	100	100	100	0.3	150	1500	500	45.6
7	1/75	0.0005	100	100	100	0.3	150	2500	200	37.8
8	1/75	0.0005	100	100	100	0.3	150	2500	800	43.8
9	1/75	0.0005	100	100	100	-	150	1500	800	45.6
10	1/75	0.0001	100	100	100	-	150	1500	800	50.4
11	**1/75**	**0.00005**	**100**	**100**	**100**	**-**	**150**	**1500**	**800**	**56.0**
12	1/200	0.0001	100	100	100	-	150	1500	800	7.0
13	1/30	0.0001	100	100	100	-	150	1500	800	44.6

TABLE I

PERFORMANCE VALUES FOR DIFFERENT AGENT ARCHITECTURE CONFIGURATIONS.

grows, a head-tail collision becomes much more frequent.

VI. CONCLUSION

This paper has presented a Deep Q-Learning based approach for learning how to play the Snake game. The related Reinforcement Learning framework has been presented and discussed. Unlike the solutions available in the literature, the proposed agent uses sensor data measurements (and not image pixels), which allows the adoption of a more straightforward (and easier to be trained) deep neural network with respect to other methods, e.g., convolution neural networks. A particular attention has been given to the neural network hyperparameters tuning process, which influences the level of the achieved performance. Several numerical simulations have proved the effectiveness of the proposed approach.

In the near future, we plan to analyse in more detail the role of the Dropout layers, since this layer should support the training process and allow the achievement of better performance values. Another important aspect is to investigate the adoption of different loss functions and optimisation algorithms [21], [22].

REFERENCES

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[2] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A survey of deep learning applications to autonomous vehicle control," *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[3] Y. Li, K. Fu, H. Sun, and X. Sun, "An aircraft detection framework based on reinforcement learning and convolutional neural networks in remote sensing images," *Remote Sensing*, vol. 10, no. 2, p. 243, 2018.

[4] D. P. Bertsekas, "Feature-based aggregation and deep reinforcement learning: A survey and some new implementations," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 1, pp. 1–31, 2018.

[5] ——, *Dynamic programming and optimal control, Vol. II*. Athena scientific Belmont, MA, 2012.

[6] A. Forootani, D. Liuzza, M. Tipaldi, and L. Glielmo, "Allocating resources via price management systems: a dynamic programming-based approach," *International Journal of Control*, pp. 1–21, 2019.

[7] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[8] M. P. Del Rosso, A. Sebastianelli, and L. U. Silvia, *Artificial Intelligence Applied to Satellite-based Remote Sensing Data for Earth Observation*. Book under contract with a scheduled publication of spring/summer2021. IET (Institute of Engineering and Technology) Digital Library.

[9] G. D'Angelo, M. Tipaldi, F. Palmieri, and L. Glielmo, "A data-driven approximate dynamic programming approach based on association rule learning: Spacecraft autonomy as a case study," *Information Sciences*, vol. 504, pp. 501–519, 2019.

[10] M. Tipaldi, L. Feruglio, P. Denis, and G. D'Angelo, "On applying AI-driven flight data analysis for operational spacecraft model-based diagnostics," *Annual Reviews in Control*, vol. 49, pp. 197–211, 2020.

[11] A. Almalki and P. Wocjan, "Exploration of reinforcement learning to play snake game," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 377–381.

[12] M. Comi, "How to teach ai to play games: Deep reinforcement learning," https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a.

[13] E. A. O. Diallo, A. Sugiyama, and T. Sugawara, "Learning to coordinate with deep reinforcement learning in doubles pong game," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017, pp. 14–19.

[14] S. Yoon and K. J. Kim, "Deep q networks for visual fighting game AI," in *2017 IEEE conference on computational intelligence and games (CIG)*, 2017, pp. 306–308.

[15] 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.

[16] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou, "Autonomous agents in snake game via deep reinforcement learning," in *2018 IEEE International Conference on Agents (ICA)*, 2018, pp. 20–25.

[17] A. Finnson and V. Molnö, "Deep reinforcement learning for snake," https://pdfs.semanticscholar.org/6fcc/13f9bfaa8de167f1f3a96d5c0ce7e7ce4542.pdf.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] P. Baldi and P. J. Sadowski, "Understanding dropout," *Advances in neural information processing systems*, vol. 26, pp. 2814–2822, 2013.

[21] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," *arXiv preprint arXiv:1910.05446*, 2019.

[22] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas, "Learning to learn by gradient descent by gradient descent," in *Advances in neural information processing systems*, 2016, pp. 3981–3989.