Using Deep Reinforcement Learning for Autonomous Navigation of Agent

Suat BAYIR - 2023913109

1. Introduction

Autonomous navigation has emerged as a transformative technology driving innovations in robotics, drones and autonomous vehicles [1]. In real-world situations such as urban transportation, warehouse automation and search and rescue missions, the ability of machines to navigate safely and efficiently without human intervention is critical. For example, driverless cars need to navigate congested city streets avoiding pedestrians and other vehicles, while drones tasked with delivering goods need to maneuver around obstacles in complex airspaces. These challenges require robust, intelligent systems that can adapt to dynamic and unpredictable environments.

Achieving effective autonomous navigation involves solving complex problems such as environmental sensing, path planning, decision making and obstacle avoidance. Traditional methods rely heavily on rule-based algorithms and predefined parameters, which often fail to adapt to new or unpredictable conditions. In contrast, modern approaches such as deep reinforcement learning (DRL) show significant promise [2]. By enabling agents to learn directly from their interactions with the environment, DRL provides a framework for developing adaptive and flexible navigation systems. With algorithms such as Deep Q-Networks (DQN) [3] and Proximal Policy Optimization (PPO) [4], machines can learn optimal navigation strategies, maximizing rewards while minimizing risks, even in high-dimensional and uncertain environments.

This work aims to explore the application of deep reinforcement learning to autonomous navigation, investigating its potential to address real-world challenges and providing efficient, adaptive solutions. By implementing and analyzing the DQN and PPO algorithms in the OpenAI Gym simulation environment, it aims to compare the success of deep reinforcement learning algorithms in the path-finding autonomous navigation problem.

2. Algorithm and Problem Review

2.1. Autonomous Navigation

Autonomous navigation refers to the ability of an agent, such as a robot or vehicle, to plan and execute paths from a starting point to a destination using its own decision-making capabilities without any outside intervention. This capability is important for various fields such as unmanned aerial vehicles, robotics and autonomous vehicles. An autonomous navigation system basically involves sensing the environment, understanding its structure and making decisions to reach destinations while avoiding obstacles. The system must also handle uncertainties and changes in the environment in real time.

The problem of autonomous navigation is challenging due to dynamic obstacles, incomplete information, and the complexity and variability of real-world environments. Solutions to this problem require sophisticated algorithms that can sense the environment, generate feasible paths, and efficiently adapt to unpredictable situations. Deep reinforcement learning (DRL) has emerged as a promising approach for autonomous navigation by exploiting the ability to learn complex decision-making problems.

2.2. Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a machine learning approach that combines Reinforcement Learning (RL) and Deep Learning (DL) to solve complex decision-making problems [5]. DRL enables an agent to learn optimal policies by interacting with its environment, receiving feedback in the form of rewards, and using deep neural networks to handle high-dimensional input spaces.

In project applications, DRL is especially effective for tasks requiring adaptive behavior in dynamic and uncertain environments, such as robotics, gaming, and autonomous systems. Popular DRL algorithms include Deep Q-Networks (DQN), Proximal Policy Optimization (PPO) and Actor-Critic methods. While DRL offers scalability and adaptability, challenges like computational demands and training stability must be addressed to achieve practical deployment.

2.3. Deep Q-Networks (DQN)

Deep Q-Network (DQN) is a reinforcement learning algorithm that uses deep neural networks to approximate the Q-value function and enables decision making in complex environments with high-dimensional state spaces. It estimates the expected cumulative reward for taking an action in each state and then following the optimal policy. DQN uses experience repetition to store past interactions and random sampling to train the model, which reduces correlations in the training data and increases learning stability. In addition, a target network is used to stabilize updates by providing a constant reference for Q-value computation across multiple training steps. This approach allows DQN to learn efficient policies for tasks such as gaming, robotics and autonomous systems.

The Bellman Equation (1) enables DQN to iteratively improve Q-value estimates, converging towards the optimal policy. It provides the theoretical backbone for evaluating actions and guiding the learning process in reinforcement learning environments.

$$Q(s,a) = r + \gamma\, maxa' Q(s',a') \tag{1}$$

- $\boldsymbol{Q(s,a)}$**:** The Q-value of state s and action a.
- $\boldsymbol{r}$**:** The immediate reward after acting a in state s.
- $\boldsymbol{\gamma}$**:** The discount factor, which balances the importance of future rewards ($0 \leq \gamma \leq 10$).
- $\boldsymbol{s'}$**:** The next state resulting from action a.
- $\boldsymbol{maxa' Q(s',a')}$**:** The maximum Q-value over all possible actions a' in the next state s'.

2.4. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that optimizes policies by balancing exploration and exploitation while maintaining stability and efficiency. It uses a neural network to parameterize the policy and uses a surrogate objective function to iteratively improve the policy. PPO imposes a constraint on the size of policy updates by trimming the likelihood ratio between new and old policies, ensuring that updates stay within a trust region. The objective function uses the Equation (2).

$$L(\theta) = E[min(rt(\theta)A\hat{}t, clip(rt(\theta),1-\epsilon,1+\epsilon)A\hat{}t)] \tag{2}$$

- $\boldsymbol{rt(\theta)}$**:** Measures the change in action probabilities between the new policy and the old policy.
- $\boldsymbol{A\hat{}t}$**:** Indicates how much better or worse an action performed compared to the expected baseline.
- $\boldsymbol{1-\epsilon, 1+\epsilon}$ **(Clipping Range):** Restricts the policy update magnitude to prevent overly large changes and ensure stable learning.
- $\mathbf{E}$ **(Expectation):** Averages the objective over sampled trajectories to compute a stable update for the policy.

3. Literature Review

Recent advances in autonomous navigation are increasingly using Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) algorithms to improve the decision-making and path planning capabilities of systems in unknown environments.

In recent years, reinforcement learning, which has become one of the key methods to achieve autonomous navigation of robots, continuously optimizes decision making based on real-time feedback reward signals through continuous interaction with the environment. Wang et al. [6] presented a method combining DQN and PPO to optimize path planning and decision making for autonomous robots. Through continuous interaction with the environment and real-time feedback, their approach improved navigation capabilities in unknown environments.

Collision-free motion is essential for mobile robots. Most approaches to collision-free and efficient navigation with wheeled robots require parameter tuning by experts to achieve good navigation behavior. Taheri and Hosseini [7] proposed an improved neural network structure within the PPO algorithm to improve safe mobile robot navigation. Using LiDAR sensor data and a deep neural network, the method generated control signals that guide robots to designated destinations while avoiding obstacles. Experimental results obtained in both obstacle-filled and obstacle-free environments underlined the effectiveness of their approach.

An exciting and promising frontier for Deep Reinforcement Learning (DRL) is its application to real-world robotic systems. While modern DRL approaches have achieved remarkable success in many robotic scenarios (including mobile robotics, surgical assistance and autonomous driving), unpredictable and non-stationary environments can pose critical challenges for such methods. Corsi et al. [8] introduced a novel benchmark environment for aquatic navigation to evaluate the performance of DRL algorithms, including PPO. Their study highlighted the challenges posed by unpredictable and non-stationary environments, showing that even the most advanced DRL approaches can struggle to produce reliable policies in such environments.

De Moraes et al. [9] compared Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) algorithms for maples navigation of ground-based mobile robots. The research methodology primarily involves a comparative analysis between a Deep-RL strategy based on the basic DQN algorithm and an alternative approach based on the DDQN algorithm. Their work has shown that Double Deep structures significantly improve the navigation capabilities of mobile robots compared to simple Q structures, even with low-dimensional sensory inputs.

In a recent study [10], an optimized autonomous drone navigation system using Dual Deep Q-Learning (DDQN) is proposed. The proposed system helps to automatically generate three-dimensional (3D)

networks for any type of object, building or scenario using drones with monocular RGB cameras. All these targets are large and located outdoors, which makes it possible to use drones for their capture. The approach balances sample efficiency and convergence speed, enabling drones to effectively navigate complex environments. Alternative reinforcement learning algorithms such as PPO, DQN and DDPG also show promise in UAV navigation, while the choice of DDQN balances sample efficiency and convergence speed.

4. Methodology

4.1 Software Specifications

In this study, the codes required to perform the experiments were written in Python. Keras and Tensorflow libraries were used for model training. OpenAI Gym environment was used for the simulation environment. The software requirements used in this implementation are listed below.

- Python 3.11.4
- Tensorflow 2.12.0
- Keras 2.12.0
- Numpy 1.23.5
- Gym 0.9.7
- Matplotlib 3.8.0
- Pyglet 1.5.27

4.2. Simulation Environment

The OpenAI Gym simulation environment was chosen to develop the robot's autonomous navigation ability in an unknown environment. Figure 1 shows an example of an obstacle grid world-like environment (25x25) with the target as a red state and the agent state as a green square. The chosen environment is a square grid with 25 rows and 25 columns, the states are simply the row and column numbers of each square in the grid (r, c), so there are 625 different states in the environment and 4 available actions (up, down, left and right) for each state. The environment is fully determined, meaning that each action maps one state to the next state separately.

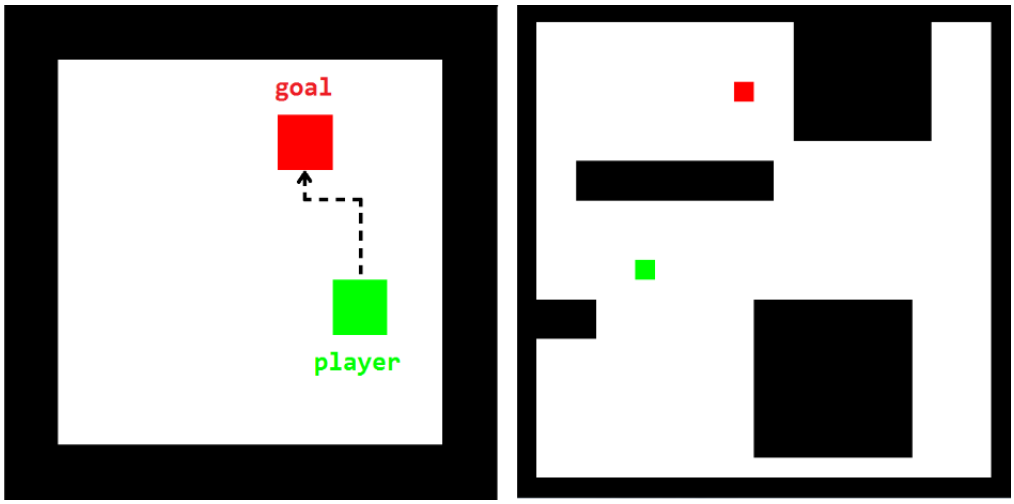

Figure 1. OpenAI Gym path-finding simulation environment

Action space: In this grid, there are 4 possible actions up, down, left, right and they can be represented by an array of 4 items [0, 1, 2, 3].

Observation space: Set of values reflective of the environment state that the agent has access to (25x25). The free cells are represented by a 0, obstacles represented by 1, the agent is represented by a 2 and finally the target is represented by 3.

```
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 2 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 3 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]]
```

Figure 2. Environment state

4.3 DQN Implementation

In DQN implementation, the agent uses a neural network to approximate the Q-value function, which estimates the expected cumulative reward for taking a specific action in each state. The network consists of an input layer matching the size of the flattened environment (625 neurons for a 25x25 grid), one hidden layer with 32 neurons, and an output layer corresponding to the number of actions in the environment. The network uses *tanh* activation for hidden layers and a *linear* activation for the output layer. The agent trains the model with the *Adam* optimizer, minimizing the mean squared error (MSE) loss.

The agent employs an epsilon-greedy exploration strategy to balance exploration and exploitation. Initially, the agent selects actions randomly with high probability (high epsilon) and gradually reduces this randomness as training progresses (epsilon decay). The training process uses experience replay, where the agent stores past experiences in a memory buffer and samples batches of them to update the Q-network. This approach reduces the correlation between consecutive experiences and stabilizes learning. A target network is periodically updated to ensure stable Q-value updates by providing fixed Q-value targets during training.

The training loop involves the agent interacting with the environment over multiple episodes. At each time step, the agent selects an action, observes the resulting reward, and transitions to the next state. It then stores this experience in the replay buffer. After accumulating sufficient experience, the agent samples a

minibatch to train its Q-network. During training, the Q-value for the selected action is updated using the Bellman equation, incorporating the immediate reward and the discounted future reward estimated by the target network. The implementation visualizes the agent's progress by plotting the number of steps per episode and cumulative rewards, highlighting the agent's improvement in performance over time.

Hyper-parameters	Value
Episode count	500
Step count	2000
Learning rate	0.001
Discount factor	0.99
Memory size	20000
Epsilon decay	0.995
Exploration rate	1 - 0.01b

Table 1. DQN Hyper-parameters

4.4 PPO Implementation

PPO provides stability in training by introducing a trimming mechanism that restricts updates to the policy network to a predefined interval. In this application there is an agent with two separate neural networks: a policy network that outputs a probability distribution over actions (the actor) and a value network that estimates the expected payoff for a given state (the critic). Both networks have a hidden layer of 128 neurons using ReLU activation functions, while the output layers of the actor and critic use softmax activations. The models are optimized using the Adam optimizer with a learning rate of 0.001.

During training, the agent interacts with a path-finding environment and collects state-action-reward trajectories. After each episode, the rewards-to-go are computed to represent the discounted sum of future rewards. The advantages are calculated by subtracting the predicted value estimates from these rewards-to-go. The policy is updated by maximizing a surrogate objective function, which uses the clipped ratio of new and old action probabilities to ensure that policy updates remain within a stable range. Simultaneously, the value network is trained to minimize the mean squared error (MSE) between predicted values and the rewards-to-go.

The main training loop iterates through episodes, where the agent performs actions based on the current policy and receives rewards from the environment. Post-episode, the policy and value networks are updated using the collected experience. The implementation includes plots for visualizing the agent's performance, such as the number of steps taken per episode and cumulative rewards, highlighting the agent's learning progress over time.

Hyper-parameters	Value
Episode count	500
Step count	2000
Learning rate	0.001
Discount factor (gamma)	0.99
Clip ratio	0.2

Table 2. PPO Hyper-parameters

5. Results

This section evaluates the results of the DQN and PPO implementations mentioned in the previous sections on the OpenAI Gym path-finding simulation environment. To make an equal comparison, the common parameters such as episodes, steps and learning rate of both algorithms are kept equal.

5.1 DQN Algorithm

The DQN Algorithm is simulated in Figure 3 for 500 episodes, each episode is limited to a maximum of 2000 steps to avoid infinite loops. The network was trained by episodically sampling a batch from the replay buffer. After 350 episodes, the agent started to learn the optimal path and was able to reach the goal within 2000 steps. DQN training is computationally long and consumes many resources. For 500 episodes, it took about 8 hours for the agent to run.

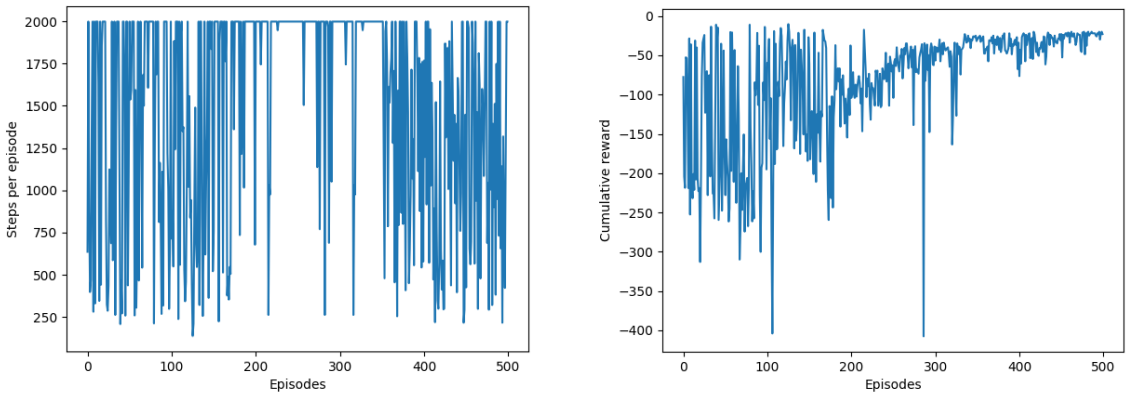

Figure 3. DQN episode/reward graph

As shown in Table 3, the DQN agent took 350 episodes to converge to its optimal policy, which allows the agent to reach the goal in 138 steps with a cumulative reward of 10.27 and a success rate of 51%. The DQN agent converges relatively slowly with a small success rate, taking approximately 8 hours to train.

Measure	Value
Maximum cumulative reward	10.27
Convergence speed	350 episodes
Success rate	%51
Number of actions of the optimal policy	138
Training time	493 minutes

Table 3. DQN experiment results

5.2 PPO Algorithm

The PPO Algorithm is simulated in Figure 4 for 500 episodes, each episode is limited to a maximum of 2000 steps to avoid infinite loops. The agent started to learn the optimal path after 250 episodes. It took about 41 minutes for the agent to run for 500 episodes in PPO training.

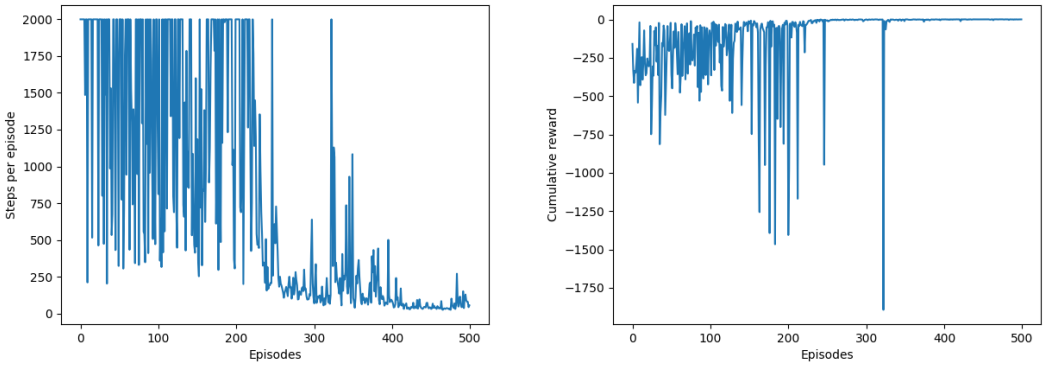

Figure 4. PPO episode/reward graph

As shown in Table 4, it took the PPO agent 250 episodes to converge to its optimal policy, which allows the agent to reach the goal in only 26 steps with a cumulative reward of 0.74 and a success rate of 76%. The PPO agent took about 41 minutes to train and successfully found the optimal path after 250 episodes

Measure	Value
Maximum cumulative reward	0.74
Convergence speed	250 episodes
Success rate	%76
Number of actions of the optimal policy	26 actions
Training time	41 minutes

Table 4. PPO experiment results

5. Conclusion

This work aims to find a solution to the autonomous navigation of agents using reinforcement learning. Within the scope of the study, DQN and PPO algorithms are implemented, and experimental results are shared. To validate the algorithms used, simulations were designed and implemented in the Python based OpenAI Gym environment. The designed algorithms are compared according to 3 main criteria: convergence speed, success rate and cumulative reward. According to the experimental results, the graphs show that the PPO algorithm gives more successful results than the DQN algorithm.

References

[1] Bagnell, J. A., Bradley, D., Silver, D., Sofman, B., & Stentz, A. (2010). Learning for autonomous navigation. *IEEE Robotics & Automation Magazine*, *17*(2), 74-84.

[2] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26-38.

[3] Fan, J., Wang, Z., Xie, Y., & Yang, Z. (2020, July). A theoretical analysis of deep Q-learning. In *Learning for dynamics and control* (pp. 486-489). PMLR.

[4] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[5] Li, Y. (2017). Deep Reinforcement Learning: An Overview. *arXiv preprint arXiv:1701.07274*.

[6] Wang, Z., Yan, H., Wang, Z., Xu, Z., Wu, Z., & Wang, Y. (2024, July). Research on autonomous robots navigation based on reinforcement learning. In *2024 3rd International Conference on Robotics, Artificial Intelligence and Intelligent Control (RAIIC)* (pp. 78-81). IEEE.

[7] Taheri, H., Hosseini, S. R., & Nekoui, M. A. (2024). Deep reinforcement learning with enhanced ppo for safe mobile robot navigation. *arXiv preprint arXiv:2405.16266*.

[8] Sajjad, S., Akhter, N., & Sajjad, L. (2024). Formal Modelling and Model Checking of a Flood Monitoring and Rescue System: A Case Study of Safety-Critical System. *VFAST Transactions on Software Engineering*, *12*(3), 114-137.

[9] De Moraes, L. D., Kich, V. A., Kolling, A. H., Bottega, J. A., Grando, R. B., Cukla, A. R., & Gamarra, D. F. T. (2023, October). Enhanced Low-Dimensional Sensing Mapless Navigation of Terrestrial Mobile Robots Using Double Deep Reinforcement Learning Techniques. In *2023 Latin American Robotics Symposium (LARS), 2023 Brazilian Symposium on Robotics (SBR), and 2023 Workshop on Robotics in Education (WRE)* (pp. 337-342). IEEE.

[10] Sánchez-Soriano, J., Rojo-Gala, M. Á., Pérez-Pérez, G., Bemposta Rosende, S., & Gordo-Herrera, N. (2024). Optimized Autonomous Drone Navigation Using Double Deep Q-Learning for Enhanced Real-Time 3D Image Capture. *Drones*, *8*(12), 725.