Use of Neural Networks for rocket piloting in 3D space, with simulation created in Godot, training with ES, PSO and DQN, without the use of external libraries.

Gabriel Bomfim[1], Victor Caus [2], Gabriel Tedesco [3],

***Abstract*— This study presents the results and discussions of three algorithms tested for piloting a spaceship in a 3D environment using neural networks. The algorithms tested are Evolution Strategies (ES), Particle Swarm Optimization (PSO), and Deep Q-Network (DQN). For ES, the convergence curve showed a slow but stable convergence to find optimal solutions. PSO exhibits a more fluctuating pattern, indicating potential instability in its convergence under unconstrained conditions. DQN, however, faces significant challenges in the Godot environment due to computational limitations and difficulties implementing matrix mathematics. As a result, DQN fails to achieve true convergence to the optimal solution. In general, among the tested algorithms, ES and PSO show promising performance in piloting the spaceship in the 3D environment. DQN faces intrinsic difficulties in the Godot engine. While the implemented approach serves as a basis for future work, further improvements and optimizations are necessary for real-life applications of neural network-based spaceship piloting. The present study contributes to the development of neural networks in game environments and provides insights for future research in this area.**

***IndexTerms*—Godot, Neural Network, 3D piloting, Aerospace AI, GDscript, ES, DQN, PSO.**

I. INTRODUCTION

The problem of AI piloting learning is certainly one of the most important for the development of the aerospace sector. Due to the fact that space is a biologically adverse environment for human conditions, considering the lack of oxygen and extreme temperatures, and furthermore, piloting being a high-risk profession, especially when it involves fighter jets and military vehicles, it is of utmost importance to seek alternatives to human piloting to save lives and perform better work. Such a system is already used, for example, in drones, for reconnaissance, surveillance, and even remote attacks in conflict situations. The Gulf War (1990-1991) serves as an example, where the US Air Force used Predator drones for surveillance and reconnaissance functions (although they were not controlled by AI). These drones were capable of carrying Hellfire missiles and conducting targeted attacks from a remote control, showing a new potential for military application. Besides military applications, spacecraft piloting can lead to a better exploration of the universe without the need to risk human lives, as mentioned before.

In this context, the core of the current work is to simulate a region of space with practically zero gravitational field and train a Neural Network so that, given a path defined by rings, it becomes capable of traversing that path in the shortest time possible.

For this problem, the Godot Engine was used, in which the execution environment and the neural network used were built from scratch, considering three mechanisms: an evolutionary algorithm, another one based on Particle Swarm Optimization (PSO), and a third one that uses Deep Q-Network for reinforcement learning. In this way, it is evident that this work also contributes to game development with the implementation of neural networks and their functions within an engine that lacks such documentation.

A. Fully Connected Neural Networks

Fully connected neural networks, also known as multilayer perceptrons, are a type of neural network in which each neuron in a layer is connected to all neurons in the previous layer. These networks consist of multiple layers of neurons, including an input layer, one or more hidden layers, and an output layer.

The input layer receives the input data and forwards it to the first hidden layer. Each neuron in the hidden layer applies an activation function to the weighted sum of the inputs and sends its output to the next layer. This process is repeated until the output of the network is calculated at the output layer.

$$y = f^{(n)}(...f^{(2)}(f^{(1)}(xW^{(1)} + b^{(1)})W^{(2)} + b^{(2)})...W^{(n)} + b^{(n)})$$

In the equation above, x represents the input vector, $W^{(i)}$ represents the weight matrix between layer i and layer $i+1$, $b^{(i)}$ represents the bias vector of layer $i+1$, and $f^{(i)}$ represents the activation function of layer $i+1$. The output of the network, y, is computed by successively applying the activation functions to the inputs in each layer.

Fully connected neural networks are capable of learning complex representations of input data and can be used to solve a wide range of problems. They are trained using supervised learning algorithms, such as backpropagation, which adjust the network's weights and biases to minimize a

*This is the final work of the course CT-213 taught by Marcos Máximo, first semester, 2023. Special thanks to his very good classes

[1]Gabriel Bomfim is an engineering student at Instituto Tecnológico de Aeronáutica, Rua H8C, CTA, São José dos Campos, Brazil gabriel.bomfim.8860@ga.ita.br

[2]Victor Caus is an engineering student at Instituto Tecnológico de Aeronáutica, Rua H8C, CTA, São José dos Campos, Brazil b.d.researcher@ieee.org

[2]Gabriel Tedesco is an engineering student at Instituto Tecnológico de Aeronáutica, Rua H8C, CTA, São José dos Campos, Brazil gabriel.tedesco.8723@ga.ita.br

cost function that measures the error between the network's outputs and the target values.

B. Evolution Strategies

Evolution Strategies (ES) [1] are a class of optimization algorithms inspired by the principles of biological evolution. They were originally developed to solve global optimization problems and have applications in various fields, including Artificial Intelligence (AI). In this work, they were inspired by Charles Darwin's evolutionary mechanism and applied to the evolution of neural networks with specific objectives in mind.

In the context of AI, Evolution Strategies are mainly used to solve search and optimization problems, especially when the search landscape is complex, unknown, or non-differentiable. These problems are common in areas like machine learning, deep learning, and parameter optimization in complex systems. In this case, Evolution Strategies are applied to mutate the neural network within a certain population and then perform selection based on a reward function. (if desired, you can write more)

C. Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization algorithm inspired by social and collective behaviors of animals, such as the migration of bird flocks or the movement of fish schools. It was proposed by Eberhart and Kennedy in 1995 [2] and has since been applied in various fields, including engineering, computing, economics, and artificial intelligence.

PSO is a global optimization method that seeks to find the best solution to a problem by using a population of particles (or agents) that move through the search space in search of promising solutions. Each particle represents a possible solution to the problem and is associated with a position and a velocity in the space.

The position of each particle is updated in each iteration of the algorithm based on two pieces of information: its personal experience (the best position it has been in so far) and the group's experience (the best position achieved by any particle in the swarm). These pieces of information are combined to determine the particle's new direction of movement, and the velocity and position are updated accordingly.

The updates of the positions and velocities of the particles in PSO are governed by the following equations:

$$v_i^{t+1} = w \cdot v_i^t + c_1 \cdot r_1 \cdot (pbest_i - x_i^t) + c_2 \cdot r_2 \cdot (gbest - x_i^t) \quad (1)$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \quad (2)$$

where:

- v_i^t is the velocity of particle i at iteration t,
- x_i^t is the position of particle i at iteration t,
- w is the inertia coefficient, controlling the inertia of movement,
- c_1 and c_2 are cognitive and social acceleration constants, respectively,
- r_1 and r_2 are random numbers between 0 and 1,
- $pbest_i$ is the best position ever reached by particle i,
- $gbest$ is the best position achieved by any particle in the swarm.

The PSO algorithm continues iterating until a termination criterion is met, such as a maximum number of iterations or convergence to a satisfactory result.

PSO has been widely used in optimization problems, especially in multimodal or high-dimensional problems, due to its ability to efficiently explore the search space and find approximate solutions to a wide variety of problems.

D. Deep Q-Learning

The Deep Q-Network (DQN) [3] algorithm is a reinforcement learning method that combines Q-learning with deep neural networks. It was developed to solve sequential decision-making problems, such as games and control tasks, where the agent must learn to take actions from high-dimensional observations, such as images.

The DQN algorithm uses a neural network to approximate the Q-function, which estimates the expected future reward for each state-action pair. The network is trained to minimize the difference between its predictions and the Q-values calculated using the Bellman equation.

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \quad (3)$$

In the equation above, s represents the current state, [4] a represents the action taken, r represents the received reward, γ represents the discount factor, s' represents the next state, and a' represents the next action. The Bellman equation expresses the Q-value of a state-action pair as the sum of the immediate reward and the maximum Q-value of the next state discounted by the factor γ.

The DQN algorithm has been successfully applied to various challenging problems, including Atari 2600 games and continuous control tasks. It is considered a significant breakthrough in the field of reinforcement learning and has been widely used in both research and practical applications.

1) GLIE: To ensure that the DQN algorithm converges to an optimal policy, it is essential that the epsilon schedule is "GLIE", i.e., that it is Greedy in the Limit with Infinite Exploration. This property ensures that, as the number of interactions with the environment (episodes) tends to infinity, the policy followed by the agent becomes increasingly greedy. Formally, an epsilon schedule is GLIE if:

- The probability of exploiting (ε) decreases over time, becoming closer and closer to zero.
- The probability of exploring (ε) is not completely eliminated, ensuring that the agent continues to explore the environment to a lesser extent, even after long training.

This combination of continuous and decreasing exploration is crucial to allow the agent to effectively learn the optimal policy by sufficiently exploring the environment at the beginning of training and subsequently focusing on actions with higher expected reward as it accumulates knowledge.

2) Fixed Q-Target: A technique used by the DQN algorithm to stabilize training is the use of a target network. The target network is a copy of the main network that is used to calculate the Q-values in the Bellman equation. The target network is periodically updated by copying the weights from the main network.

3) Experience Replay: The DQN algorithm uses a technique called experience replay to stabilize the training of the neural network. Experience replay stores the agent's transitions in the form of (state, action, reward, next state) tuples in a replay buffer. During training, random batches of these transitions are sampled from the buffer and used to train the neural network. The goal of replay is to break the temporal correlation between training experiences, thus reducing bias and variance in the gradients computed during the learning process. This way, the replay memory enhances learning efficiency, promoting faster and more stable convergence of the agent's policy.

4) Loss Function: The process of adjusting a neural network using gradient descent is a fundamental step in training the network to approximate complex functions and make accurate predictions. Gradient descent is an optimization algorithm used to minimize the loss function of the neural network by iteratively updating the network's parameters (weights and biases) in the direction that reduces the loss.

The loss function, also known as the cost function or objective function, quantifies the difference between the predicted output of the neural network and the actual target values. In regression problems, a common choice for the loss function is the Mean Squared Error (MSE) function. For a given neural network with parameters θ, the MSE loss function is defined as:

$$\text{MSE}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{4}$$

where:

- N is the number of training samples;
- y_i is the actual target value for the i-th sample;
- $\hat{y}_i$ is the predicted output of the neural network for the i-th sample, given the input features and the current network parameters.

The goal of training the neural network is to find the optimal set of parameters θ that minimizes the MSE loss function.

5) Gradient Descent: To update the neural network parameters and minimize the loss function, gradient descent computes the gradient of the loss function with respect to the parameters θ. The gradient is essentially the vector of partial derivatives of the loss function with respect to each parameter.

The parameter update in gradient descent is given by:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha\nabla\text{MSE}(\theta_{\text{old}}), \tag{5}$$

where:

- θ_{old} and θ_{new} are the old and updated parameter values, respectively;
- α is the learning rate, which controls the step size in the parameter space;
- $\nabla\text{MSE}(\theta_{\text{old}})$ is the gradient of the MSE loss function evaluated at the current parameter values θ_{old}.

By iteratively updating the parameters using gradient descent, the neural network learns to adjust its weights and biases to better approximate the target function, reducing the prediction errors over the training data.

E. Quadratic Loss Function and Derivative

For illustration purposes, let's assume the loss function is a quadratic function of the form:

$$\text{MSE}(\theta) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{6}$$

Then, the derivative of the quadratic loss function with respect to the j-th parameter of the neural network, denoted as $\frac{\partial\text{MSE}}{\partial\theta_j}$, is given by:

$$\frac{\partial\text{MSE}}{\partial\theta_j} = -\sum_{i=1}^{N}(y_i - \hat{y}_i)\frac{\partial\hat{y}_i}{\partial\theta_j}, \tag{7}$$

where:

- y_i is the actual target value for the i-th sample;
- $\hat{y}_i$ is the predicted output of the neural network for the i-th sample, given the input features and the current network parameters;
- $\frac{\partial\hat{y}_i}{\partial\theta_j}$ is the partial derivative of the predicted output $\hat{y}_i$ with respect to the j-th parameter θ_j.

The gradient descent algorithm uses this derivative to update the parameters and minimize the quadratic loss function.

F. Godot

Godot [5] is a free and open-source game engine designed for 2D and 3D game development. It was created by Juan Linietsky and Ariel Manzur and initially released in 2014. Since then, it has become a popular tool for game developers due to its flexibility, user-friendliness, and powerful capabilities for game creation.

Unlike other game engines, Godot uses an object-oriented architecture to organize and manage game elements. It provides an intuitive node-based interface that allows developers to build game logic by creating and interconnecting different nodes, representing elements such as scenes, objects, characters, and visual effects.

Godot also offers its own scripting language called GDScript, which is easy to learn and allows developers to efficiently program game logic. Additionally, it supports C# and provides a wide range of features, such as physics, animation, audio, particles, and advanced lighting. However, integrating Godot with Python and external libraries is not as straightforward, which motivated this group to program everything from scratch, including the neural network.

II. IMPLEMENTATION

The implementation of the codes in Godot uses as a whole GD script. However, it does not have its own library for neural networks and their strategies, so the biggest challenges of this work were to implement from scratch some of the known learning algorithms so that there was a test and the rocket learned to follow its goals.

At first, the environment in which the evolution of the rocket's trajectory would occur was built. It was created as a Node3D within Godot, for which a camera was defined in view of the spherical coordinates, that can be controlled by the A,W,S,D buttons (camera movement) and the mouse scroll (camera zoom). Thus, you have a greater notion of space, in which a skybox was plotted for graphic purposes. With the scenario set up, the next step was to create a neural network and associate it with a spaceship. Table I shows the configuration of the network layers used for the first two algorithms to be described. The input for this configuration is 18 numbers, i.e., 18 pieces of information, representing a total of 6 trines of three numbers, each indicating the following properties, present in Figure 2. Its output is four continuous numbers for propulsion (forward), and rocket torque in the four main axes, x, y and z. [6]

TABLE I
MULTI LAYER PERCEPTRON CONFIGURATION

Layer	Output shape	Activation Function
Dense	18	Tanh
Dense	32	Tanh
Dense	32	Tanh
Dense	4	Tanh

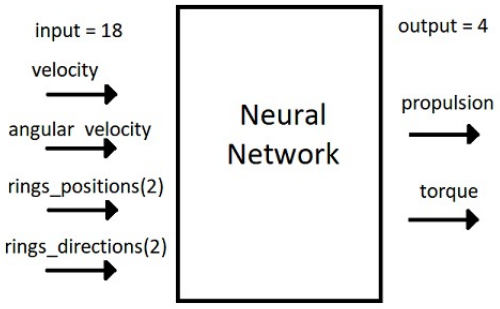

Fig. 1. Diagram of the inputs and outputs of the neural network used in the first and second problems

In Table II we have the first network adapted to the reinforcement learning problem. Note that the activation function was changed to sigmoid due to the ease of derivation, necessary to perform the back propagation, fundamental to the DQN method. In addition, another adaptation from the previous network to the new DQN network was a reduction in the size of the network due to the reduced processing of the computer, which, because the GODOT platform does not have good matrix arithmetic, all methods and multiplications needed to be done in summations, so that it was necessary to slightly reduce the size of the network. A big advantage of ready-made libraries is that algorithms such as backpropagation are vectorised already, calculating much more efficiently than the one in our project.

The number of outputs was adapted and discretised due to the need to implement reinforcement learning following the policy of taking actions, and such actions were discretised. That is, for each of the 4 changes of the standard neural network, three actions were assigned at its core: -1, 0 or 1. In this way, for example, for propulsion, the rocket can accelerate, not accelerate or brake, in a discrete way. The output, in this sense, is the reward for taking a certain action. The policy will thus choose the best reward.

TABLE II
DQN - NEURAL NETWORK

Layer	Output shape	Activation Function
Dense	18	Sigmoid
Dense	20	Sigmoid
Dense	20	Sigmoid
Dense	12	Sigmoid

However, after a negative result, as will be explained, after countless frustrated hours of debugging, in an attempt to resemble the Deep Q-Networks made by DeepMind to play Atari, the style of the neural network output was changed: Now, in addition to being discrete, each action (forward thrust and torque in each direction) is mutually exclusive, so that the greedy action to be taken is the one with the highest action-value stipulated by the neural network. That is, it would now be necessary to do one thing at a time. It was not concerned with a neutral action (staying still) because the torque on the rocket's shaft does not contribute much to the motion (it can spin like a top if it wants to stay still). We named this new endeavour DQN2, whose neural network is as follows:

TABLE III
DQN2 - NEURAL NETWORK

Layer	Output shape	Activation Function
Dense	18	Sigmoid
Dense	20	Sigmoid
Dense	20	Sigmoid
Dense	4	Sigmoid

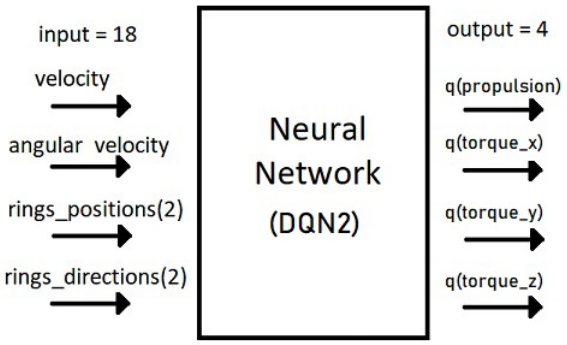

Fig. 2. Diagram of the inputs and outputs of the neural network used in the last attempt of DQN.

A. Evolution Strategy

To implement the evolutionary strategy, 50 rockets are generated in the space created in Godot, each rocket representing its own neural network. The reward function, in this case, is given by adding 1 for each ring passed and subtracting a certain value from its distance to the ring. At each generation, the best neural network is taken, i.e. the one that obtained the highest value, and it is replicated, following a mutation, indicated by Deviation (A mutation changes the bias and weights of the neural network stochastically, but proportional to the Deviation). In the scene, for each ring crossed, another is generated in a queue, and the rings which has been passed becomes a little more transparent for better visualization. For visual purposes, within the interface, the worse the rocket performs, the more transparent it is. You can save the best neural network and load it. If necessary, you can change the hyperparameters of the system, in order to check convergence and different learning.

If desired, in this system, one can also follow the view of the best rocket from the ”Spaceship View” button. In order to avoid overfitting, it was determined that the rings would be instantiated randomly with a distance of 6 meters from each other. In addition, in order to speed up the training of the network, a limitation of the number of rings crossed to 24 was made, in order to encourage the increase of the speed of the rockets and a better training of the neural network. However, by easily removing such limitation in the lines of the code, we can see the number of rings crossed increase indefinitely, and so the reward function. But, as the mark of 24 rings is achieved, the objective is to maximize the velocity, so that is done.

B. Particle Swarm Optimization

To implement PSO, again 50 rockets are initialized in the scene, each rocket representing its own neural network, such that at each generation the method governing the replication of the rockets is PSO, that will be responsible for the obtaining of the global optimum. The PSO algorithm is used to optimize the weights and biases of the neural network by updating their values based on the best values found so far by the spaceship (unique particle) and the best values found so far by the entire swarm (represented by the best spaceship). The inertia_weight, cognitive_parameter, and social_parameter arguments control the influence of the particle’s previous velocity, its own best position, and the global best position, respectively, on the new velocity of the particle.

The code iterates over each neuron in the layer and updates its weights and biases using the PSO algorithm. The new velocity of each weight and bias is calculated as a weighted sum of its previous velocity, its distance from its own best value, and its distance from the global best value, where the latter two are multiplied by a random factor between 0 and 1, as shown in equation 1. The new value of each weight and bias is then calculated by adding its new velocity to its current value, as in equation 2.

In this case, the value of inertia_weight used was 0.9, the value of cognitive_parameter was 0.6 and the value of social_parameter was 0.8. In this sense, we do not have a complete valorisation of the previous speed (as it would be with an inertia_weight of 1.0), but we also do not have a devaluation of the previous speed. In addition, there is a trade off between exploration (diversity) and exploitation (convergence) in the search space. The social parameter indicates how much it will explore, while the cognitive factor indicates how much it will exploit, so a balance has been made between these parameters so that the algorithm works well.

Previously, a schedule of the inertia_weight was made, that is, at each iteration its value was decreased so that the particles tend more to follow the optimal values. However, it was removed in order to increase exploration and avoid local maxima.

C. Deep Q-Learning

Unlike the other algorithms, which were population-based, DQN creates only one spaceship. So, for each physical step, unlike most implementations in which the current state, the action, the reward obtained and the prediction of the next state are taken as experience, in the simulation in Godot this is not possible. To circumvent this, the experience is counted in the frame of the ”new state”. Thus, the previous state, the action taken in the previous state, the instant reward in the new state and the reward in the new state are encapsulated in an ”experience” object, as well as a boolean indicating whether it is the last state of the episode or not, and added to the replay buffer.

The instant reward is given by a small negative constant times the distance to the target ring, plus a large reward when passing a ring. The ”states” are basically the inputs of the neural network that have already been explained. What we call action is a vector of actions taken: for each part of the movement (thrust and torque in x, y and z), one chooses between three possibilities (-1, 0 and 1). The action for each of these is evaluated with an epsilon-greedy policy: There is an epsilon chance of taking a random action and the rest choose the one with the highest value. The epsilon goes through a schedule to ensure that the GLIE conditions

(explained in the introduction) are satisfied: at each epoch, it is reduced at a rate, until it reaches a small minimum epsilon.

The check whether that is the last state is done in a greedy way: if the current time plus Godot's time step exceeds the end-of-episode time, that is the final state. This assumes that the rocket will not pass the ring at the exact last frame of the episode (because then the time would be increased and the episode would actually continue), but this is statistically unlikely to happen, so it should not influence it. Always put in the buffer, check if the current frame is not the first of the episode (because if it were we would not have information of the new state and the new reward the way it was implemented). The buffer has a maximum size and automatically removes older elements when it reaches this limit.

At each physical step, if there is already enough data (at least twice the batch_size), a random mini_batch is taken from the replay buffer, and then, for each sample, the TD target is given by the equation below:

$$\text{TD Target} = \begin{cases} R_t, & \text{if "done",} \\ R_t + \gamma \cdot \max_{a'} Q(S_{t+1}, a'; \theta^-), & \text{if "ongoing",} \end{cases} \tag{8}$$

where:

- R_t is the reward obtained after transitioning from state S_t to state S_{t+1} via action a_t;
- γ is the discount factor controlling the importance of future rewards compared to immediate rewards;
- $\max_{a'} Q(S_{t+1}, a'; \theta^-)$ is the maximum Q-value estimated for the next state S_{t+1} considering all possible actions a', where θ^- represents the target network's parameters.

This defines the targets to be used in the loss function, that is MSE (see equation 4). Then, the neural network is updated using gradient descent (explained in the introduction) with this Loss function. Everything, including the gradient descent was implemented from scratch, without any library.

Since, unfortunately, the algorithm did not converge, as verified in the results, we tried to simplify the problem to mutually exclusive actions (DQN2), as already seen. Due to the generality of the implementation of DQN, when trying to make the specific case, it was enough to change the interaction of the outputs with the environment, that is, the application of the actions in the forces, which will imply the physics of Godot.

III. RESULTS AND DISCUSSIONS

A. Interface of the Neural Network training

First, the interface is highly adaptable and can change the hyperparameters that govern the scene and the rocket learning. In this scene, it is possible to increase or decrease the size of the ring, the distance between them, the time that the algorithm gains for crossing a ring and also, when it comes to evolutionary algorithms, you can also change the mutation factor if you need to perform larger tests. The global view is given by Figure 3, which is centered on the coordinate center of space, where the 50 rockets are generated at each iteration.

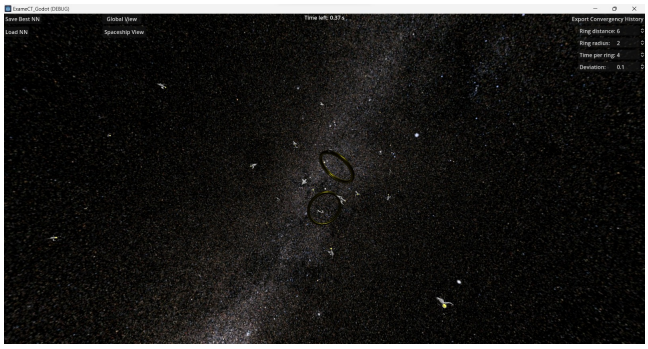

Fig. 3. Training 3D Simulation (Global View)

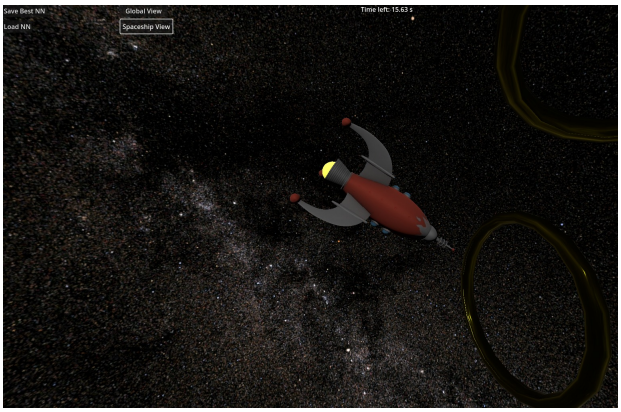

Fig. 4. Training 3D Simulation (Best Spaceship View)

Note that it is also possible to follow the best rocket by fixing the center of the radial camera on the rocket that has the best reward, as shown in Figure 4, by just clicking the button Spaceship View. In the case of DQN, which has only one rocket, it fixes the camera on him.

B. Evolution Curves

For the evolutionary strategies algorithm, the neural network is started with random factors and, from stochastic changes of the deviation in view of a normal distribution around the weight value of the previous iteration, the network performs its mutation. Note that the process is random, but guarantees good solution since it performs an exploration around the best previous generation and exploits the fact of the best reward. As the goal is to cross more rings in the shortest possible time, we plotted the graph of the crossed rings per iteration and also the graph of the reward function of the best rocket per iteration, which are plotted on the curves present in Image 5.

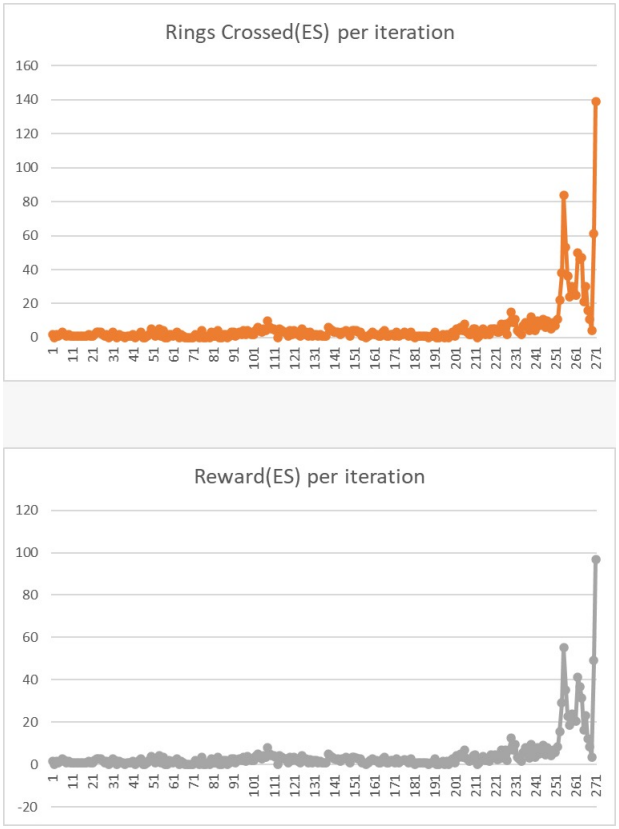

Fig. 5. Evolution algorithm curves

The graph has a very large variation, but it would be possible to let the algorithm run more iterations, so that the performance would improve. In this version, the following variable hyperparameters were used: 2 for the ring size, 6 for the distance between the center of two rings and 4 for the time added when the algorithm traverses a ring. In the final code version, we limited the number of rings arbitrarily to 24 so that the computer running Godot does not need to generate too many rings and thereby consume too much computing power.

As the algorithm is random, its improving can happen faster or slower. Running the code another time, this time with the number of rings limited to 24, we have that the evolution this time occurred within a scope of 125 iterations, with after that, the algorithm being forced to try to find a faster way. The number of iterations was lower than the previous one due to purely stochastic factors.

In this case, as the rings appear in random positions, it may be that the network that performed well in the past will not perform very well in the future. This was done to avoid overfitting. Now, it is interesting to check the final strategy adopted by the rocket: it goes towards the ring and starts describing circumferences until it reaches the target, which is the ring. Thus, it manages to reach the target, describing the ring path.

C. PSO Curves

In this section, we present the curves analysis of the Particle Swarm Optimization (PSO) algorithm for the optimization problem at hand. Unlike previous experiments where a time limit of 24 iterations was imposed, the curves presented here is obtained without any specific time constraint. The purpose of this analysis is to evaluate the stability and performance of PSO under unconstrained conditions.

The curves, depicted in Figure 6, reveals interesting insights into the behavior of PSO during the optimization process. It is observed that the curves exhibits a more fluctuating pattern compared to previous experiments, suggesting a higher degree of instability in the algorithm's convergence. These fluctuations may arise due to the unrestricted optimization duration, allowing the algorithm to explore the search space more extensively, which could lead to oscillations in the particle positions.

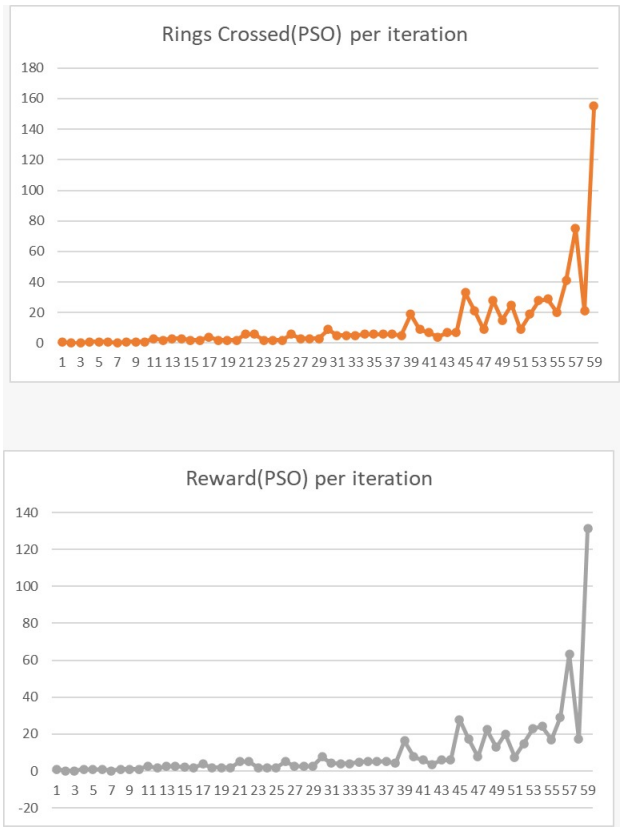

Fig. 6. PSO algorithm curves

Upon closer examination, it becomes evident that PSO find a good solution more rapidly during the initial training iterations, especially when optimizing the objective function, y. This characteristic indicates that PSO is particularly effective for simpler optimization problems or scenarios where quick convergence is crucial.

Furthermore, it is worth noting that PSO's convergence heavily relies on the initial condition. The algorithm's final solution is profoundly influenced by the random initialization of particles, which can either lead to successful convergence to the global optimum or trap the algorithm in local optima. As a result, careful consideration and tuning of the initial parameters are vital to achieve desirable results with PSO.

D. DQN Curves

One of the biggest challenges to perform DQN or DQN2 is computational, since, at each frame, the computer needs to compute gradients and back-propagation of a relatively complex naural network in view of a relatively large batch. In this sense, a lot of computer power is required and therefore the batch was reduced, which theoretically would increase the convergence time. Another major challenge is to implement all the functions without the matrix mathematics that Python, for example, provides, and Godot does not have libraries like Keras, already implemented in Python. In this sense, the challenge of implementing DQN in Godot proved to be too great for the time frame of the project work. Even so, the code was executed, which may not be 100% correct given the difficulties imposed by the Godot environment. The result is shown in Image 7 below.

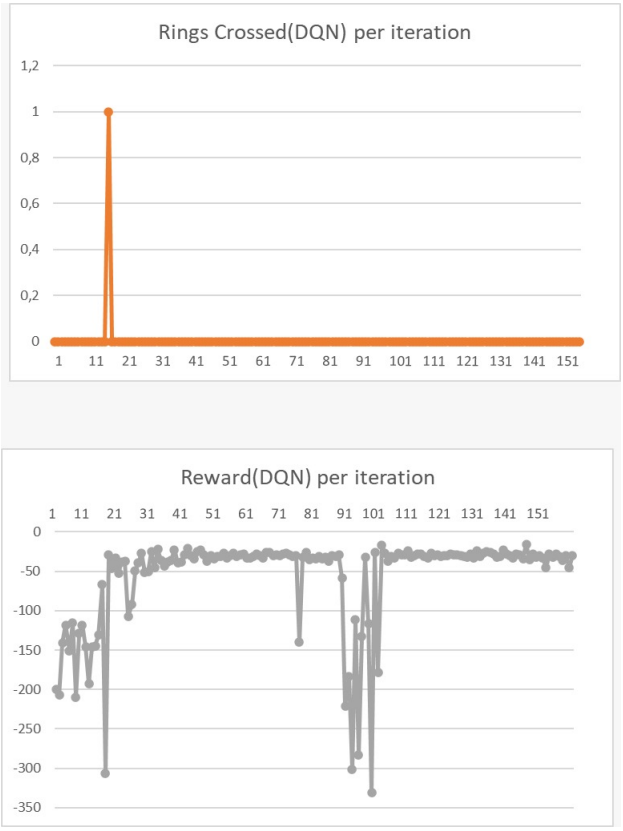

Fig. 7. DQN2 algorithm curves

This result is after the change to DQN2. It seemed to want to stay close to the ring when we heavily punished the distance to the ring, which was an improvement on the previous one which seemed to run away from the target. However, none of the algorithms were able to come in and update the logic of the targets. The only convergence seen was the rocket wanting to stay close to the ring. In this sense, a difficulty in making Deep Q-learning work in Godot is identified. This is explained by the difficulty of implementing matrix mathematics in this code, especially back-propagation, a function that requires a lot of Engine. In addition, the computational cost of calculating the optimal policies at each frame is absurd, in the sense that one does not have much freedom in increasing the size of the network or increasing the size of the batch. Therefore, there wasn't a true convergence to the best solution.

IV. CONCLUSIONS

In summary, of the three algorithms tested in Godot, the ones that had the best performance in the engine in the rocket learning problem were the ES and the PSO. The DQN presented intrinsic difficulties of Godot, such as the lack of libraries and high computational cost, which implicated in greater problems in its implementation and performance. ES and PSO were very good among themselves, ES converged more slowly but found more stable solutions. The PSO, on the other hand, is chaotic but also converges to the optimal solution faster.

In this context, solving a piloting problem using neural networks in 3D space is a complex task that involves multiple ranges of variables. The current work has achieved an approximation of a Spaceship, but the real-life application conditions are considerably broader. Thus, this work represents only a step towards the proper development of neural networks capable of learning and piloting a Spaceship.

In addition, what has been implemented here can serve as a basis for future work on games using the Godot engine, since most of the codes were implemented from scratch, especially the DQN, since there is no library dedicated to neural networks in GDscript as there is in Python. It is considered that the objective was achieved, but that the method used can be improved in several aspects.

ACKNOWLEDGMENT

Special thanks to Professor Marcos Maximo for having introduced us to this subject. Artificial intelligence is certainly the future. Many thanks also to the Godot team, for having made a simple software open to the public.

REFERENCES

[1] NORVIG, Peter; RUSSELL, Stuart. Artificial Intelligence: A Modern Approach. Pearson, 2009.
[2] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," Proceedings of IEEE International Conference on Neural Networks, 1995.
[3] GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. Deep Learning. The MIT Press, 2016.
[4] HUGGING FACE. Deep Q-Network. Available at: https://huggingface.co/learn/deep-rl-course/unit3/deep-q-network?fw=pt. Accessed on: 19/07/2023.
[5] Godot Engine. (2023, July 21). Official Godot Engine documentation. Retrieved from https://docs.godotengine.org/
[6] SORRENTINO, John. Coding A Neural Network FROM SCRATCH!. Published by John Sorrentino channel, 2017. 1 video. Available at: https://www.youtube.com/watch?v=yyS5hjyOFDo. Accessed on: 13/07/2023
[7] SUTTON, R. S.; BARTO, A. G. Reinforcement Learning: An Introduction. 2. ed. Cambridge: MIT Press, 2018.
[8] Professor Marcos Máximo's Slides from course CT-213