Learning to play a simple game with Genetic Neuroevolution

Roger Creus Castanyer

December 2020

1 Introduction

In this work one can find the documentation of the design and implementation of a Genetic Neuroevolution Algorithm (GNA) that enables an agent to succeed in a simple environment. The algorithm emulates the survival of the fittest methodology in a very similar way. Generations of individuals (agents) explore the environment, and according to a defined fitness function the best performing agents are chosen to reproduce and suffer mutations to span the following generation. With this methodology, the goal is to find the convergence to an optimal policy given the environment and the fitness function [10].

For this project, the environment is defined as a 2-dimensional grid where a simple game is played and defined as follows. It consists of a pursuit inside a grid where an agent needs to avoid collisions with the walls and with an enemy that chases it. At each time step, the agent chooses a direction to move in Right, Left, Up and Down, and does move *player_mov* units towards the chosen direction. Then, in the same time step, the enemy moves in a direct way in the agent's direction for *enemy_mov* units.

With this definition, the fitness function to evaluate the agent's performance is defined to be the number of time steps survived, without colliding with the enemy or the limits of the grid. For this set-up the optimal policy will be one that can survive for a given value of *max_time_steps* without colliding with the walls or with the enemy.

Both the environment and the agent are implemented using Matlab version 9.9.0.1524771 (R2020b) Update 2 on Windows.

2 Background

In Machine Learning (ML) the Multi Layer Perceptron (MLP) is a type of Neural Network (NN) used in a wide range of areas in the domain of prediction problems. A common usage of this architecture is in the Regression and Classification frameworks. The reason of this is for the ability of the MLP to become an appropriate universal function estimator when adequately configured. Tipically, the MLP is used in Supervised Learning (SL) and Unsupervised Learning (UL), and more rarely in Reinforcement Learning (RL), where no tagged examples are fed to the model (either ground truth as in SL or crafted as in UL).

However, for the prediction problems in RL the policy is supplied, and the goal is to measure how well it performs. That is, to predict the expected total reward from any given state assuming the function finding the optimal policy is fixed. For these problems, the distribution of the state-values needs to be approximated with estimators of the reward distribution for each state in which the agent can be. In contrast, there also exist the control problems, where the policy is not fixed, and the goal is to find the optimal policy, which maximizes the expected obtained reward in the environment. Moreover, in deep RL, more complex techniques as the NN are used in order to avoid the assumption of availability of tabular state-actions sets, meaning that these techniques adapt to environments with unreachable large number of possible states [1]. In this field, the GNA would represent the laziest version of deep RL for solving control problems. This can be said because many of the typical problems in deep RL are reduced to the

survival of the fittest methodology. These include the control of the level of exploration, the usage of the reward distribution, which is generalized to the fitness function, and the optimization of the policy. The major drawback of this technique is that it requires a considerable number of hyperparameter tuning, and if added to the MLP, even more. So an implementation that allows fast and efficient executions of the program for evaluating different configurations is highly recommended.

In this work the agent takes the form of an MLP. This architecture adapts well to the deep RL domain for its capabilities of receiving numerical inputs and giving probability outputs for each action in a defined set. The usage of the MLP is very similar to the traditional ones in SL and UL but differs in the training method. It makes use of the connections weights, neurons biases, hidden layers and activation function for reproducing the feed-forward pass, but it replaces the back-propagation of the loss function for the neuroevolutive convergence to the sparse solution. The solution is sparse because among all possible configuration of layers, weights and biases the optimal policy, which is not unique, makes use of a little and very specific subset of neurons and connections of arbitrary values. With all this settings, the hyperparameter tuning difficulties of the MLP add to the same ones of the GNA.

3 The program

In this section the implementation of the proposed work is documented. The general idea is that the weights and biases of the MLP agents are stored in 3-dimensional matrices where the size of dimension 3 is equal to *generation_size*. This way, the feed-forward pass can be computed for each agent and time step at the same time. In other words, all the agents in the same generation play at the same time. In each generation, while all the agents play, two vectors named **L** and **R** of sizes *(1 x generation_size)* are kept in memory and updated so that $L(i)_t = 1$ if the agent i is alive at time step t and $L(i)_t = 0$ otherwise. Also, $R(i)_t$ is equal to the fitness achieved by the agent i at time step t. Furthermore, two vectors named **mean_performance** and **max_performance** of sizes *(1 x num_generations)* are created so that they store the the mean and max fitness obtained in each generation for visualization purposes after training.

The visualization of the gameplay will be reserved for the optimal agent for efficiency purposes.

An important point is that Matlab does not actually support an explicit optimized way for computing 3-dimensional matrix multiplications. This is solved by means of the usage of the **mtimesx** package, which makes use of a C compiler for providing that service [11].

3.1 Environment

The environment consists of a simple game that is played in a 2-dimensional grid, where an agent takes the form of a blue square and the enemy of a red one. Since the enemy moves in a deterministic way towards the agent with freedom of movement within the grid, its movement units of distance are set to be smaller than the player ones: *player_mov* > *enemy_mov*. The definition of the game is straightforward and in the following list a valid example of initialization is provided:

```
map_x = 10;
map_y = 10;
init_player_x = -5;
init_player_y = 5;
init_enemy_x = 5;
init_enemy_y = -5;
player_mov = 2;
enemy_mov = 1;
enemy_distance_collision = 0.1;
```

The map is of total size of *(map_x*2)x(map_y*2)* and it is centered in (0 , 0). Additionally, the *enemy_distance_collision* variable encodes the maximum distance allowed between the agent and the enemy so that the agent does not die by colliding with it.

3.1.1 Visualization

For the purpose of enabling the visualization of the environment and the gameplay two auxiliary functions are created to draw a background grid and to place the blue and red squares in their positions at each time step. The background is drawn once in the beginning. Then, at each frame a white square is placed in the previous positions of both the player and the enemy and the new positions are colored in blue and red respectively in order to create a dynamic visualization. All the graphics displayed for the visualization of the game are created with *fill* [3]. The sequence of generated plots is stored into a *movie* object by means of the *getframe* method and the resulting movie is played at the end [5] [2].

This functionality is only executed over the best ever seen agent in a run of the training phase. The best agent configuration is both saved and loaded from text files (.txt) using the *writematrix* and *readmatrix* methods [9] [7].

3.2 MLP

The first concern in the definition of the MLP is setting the input and output layers appropriately. The role of the input layer is to become the eyes of the agent, and the output layer must allow the agent to select all the possible actions. In an MLP, defining a layer means setting a number of neurons on it. So for defining *input_size* and *output_size* it has been considered that the player must know at each time step its self position and the enemy's position in the grid with X and Y coordinates. With this information the agent must be capable of moving in the 4 directions so both the *input_size* and the *output_size* definitions are set straightforward.

```
1 input_size = 4;
2 output_size = 4;
```

It is reasonable to think that if the agent is wanted to avoid collisions with the wall the distances to the four walls should also be given to the agent in the input layer. However, it is better to not give redundant information to the agent and it will be assumed in the program that the agent can have this information by simply learning what is the maximum and minimum X and Y values in its position that can be reached without dying.

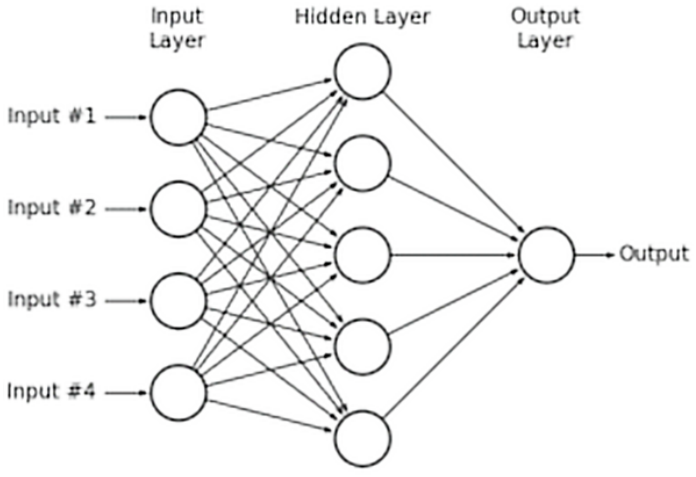


Figure 1: MLP architecture. (Not adapted to the proposed definition of the hidden & output sizes)

After the configuration of the input and output layers the focus is put on deciding a number of hidden layers and hidden neurons. The appropriate hidden configuration might be hard to find because the behaviours of the agent can vary outside of the human comprehension with small changes in it. In addition to this, the range of possible values that the weights and biases can take are two more hyperparameters. These ranges can have an undesired influence on the training phase depending on the magnitude of the input

data values. A reasonable choice is setting the ranges to [-range, range] so that the agent can learn to decide both to or not to perform an action given the input conditions. The best way to tune the hidden configuration is by trial and error, since no formula of optimization exists. For the program a number of hidden layers is set to 2 and the number of neurons in each of this hidden layers can be configured in *hidden_size*. See an example of the simple definition in the following list:

```
input_size = 4;
output_size = 4;
hidden_size = 8;
weight_range = 1;
bias_range = 0.5;
```

The range of the weight and bias values for the MLP agents is set to be limited to 1 in the example because the input data will be normalized to the [-1, 1] range as described in detail in the following subsections. With all this, it is expected to enable the agents to obtain knowledge from the differences between input value changes at an equal scale as the differences between input values magnitudes.

However, what the previous code box definition implies is not that simple. An MLP with this configuration consists of the weight matrices **W1** of size *(input_size) x (hidden_size)*, **W2** of size *(hidden_size) x (hidden_size)* and **W3** of size *(hidden_size) x (output_size)*. In this matrices, each position $W_{ij} = w_{ij}$ stores the weight of the connection of the neuron **i** in previous layer with neuron **j** in the current one. Note that consecutive layers are fully connected meaning that each neuron in a layer has a connection with all neurons in the following layer, as seen in Figure 1. The MLP definition also implies the definition of the bias vectors **B1** and **B2** of sizes *(1 x hidden_size)* and **B3** of size *(1 x output_size)*. These bias values belong to each neuron in the hidden and output layers, and are added to the activation formula in the feed-forward step. Note that the input layer neurons do not have a bias because the information that the agent reads from the game is wanted to be pure and clear, hence, unbiased.

For the activation function, the *ReLU* function is implemented. See its definition:

$$ReLU(x) = max(0, x)$$

With all the definitions, the feed-forward pass can be applied. This procedure is the one that allows the numerical information in the input layer get to the output layer so a decision can be taken. It consists of a recursive computation of the neurons values in each layer depending on the values, weights and biases of the previous one in the following way. E.g. Value of neuron j in the first hidden layer given the mentioned configuration will be:

$$n_j = act(W_{1j} * I_1 + W_{2j} * I_2 + W_{3j} * I_3 + W_{4j} * I_4 + b_j)$$

Where I_i are the values of the input neurons, b_j is the bias of current neuron n_j and act(x) is the activation function.

In general:

$$n_j^{(l)} = act((\sum_{i=0}^{size(l-1)} W_{ij}^{(l)} * n_i^{(l-1)}) + b_j^{(l)})$$

Where l specifies the layer and $n_i^{(l)}$ contains the actual value of neuron i in layer l.

There is a difference in the activation function call in the output layer, where the activation function is replaced by the *Softmax* in order to obtain probabilities of taking each action so that they sum to 1. See the definition of the function:

$$Softmax(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{j=0}^{output\_size} e^{\mathbf{x}_j}}$$

Where $\mathbf{x}$ is the vector containing the outputs, and the function is applied to each neuron i in the output layer .

Finally, the agent just takes the action with maximum probability among all neurons in the output layer. As a consideration, in the program the following associations are set for the values in the input and output layers:

- $I_1 = Player\_X$
- $I_2 = Player\_Y$
- $I_3 = Enemy\_X$
- $I_4 = Enemy\_Y$
- $O_1 = Right$
- $O_2 = Left$
- $O_3 = Up$
- $O_4 = Down$

3.3 GNA

The algorithm is called so that it generates *num_generations* generations of size *generation_size*. The first generation is special because the values in the weights and biases are random in the defined ranges [-*weight_range, weight_range*] and [-*bias_range, bias_range*]. It is important that the first generation contains the maximum amount of variability because the new behaviours added to the selected agents will come by means of the mutations, and with low probability. So the potential optimal agent must appear in the first generation, even though it might fail hard in playing the game in the beginning.

Once all the agents have played the game, as long as no agent has survived for *max_time_steps*, the best k agents are selected to reproduce. In this case k is saved in *crossover_size*, and this subset of agents is called the parents. For the purpose of achieving an efficient implementation of this GNA, the new agents are generated with information coming from only two of the parents.In this case, reproducing agents means generating new ones that contain values in its weights and biases that come from the parents. Each generated agent selects a random parent (in the chosen pair) for setting each of the values of its weights and biases, so variability is ensured.

A part from this, each of this agents suffers a mutation in each of its weights with probability *weight_mutation_prob* and in each of its biases with probability *bias_mutation_prob*. In this case, suffering a mutation means replacing the old value of these selected connection/bias to a new random one in the same defined ranges. This ensures a new level of exploration in the agents behaviours.

With this procedure, if the optimal policy is obtained in convergence, it turns out to be a black box. There is no way of interpreting the reason of why the values in the connections weights and biases work well in the environment. It is just set that if they have proven to work they must be inherited in the new generation.

In the following list there is an example of definition of the GNA hyperparameters:

```
num_generations = 50;
generation_size = 8000;
crossover_size = 4;
```

```
weight_mutation_prob = 0.08;
bias_mutation_prob = 0.05;
max_time_steps = 500;
current_time_step = 1;
```

3.4 Training the agent

In the following section the structure of the program pipeline is described and complemented with pieces of code shown in the appendices section. Following the aforementioned declaration of variables to set up the Environment, GNA and MLP, the main program links the following sections. Initialization, Gameplay and Reproduction. These are repeated cyclically as seen in Figure 2.

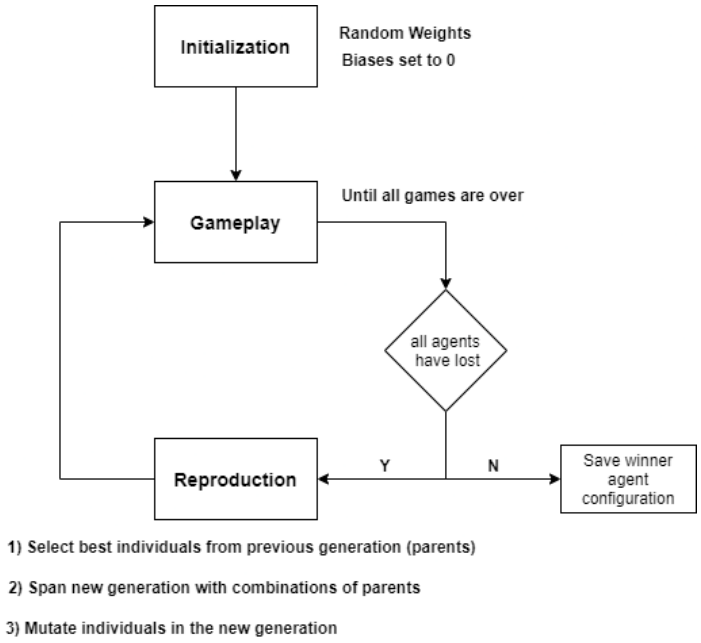


Figure 2: Cyclic schema of the program

3.4.1 Initialization

The weights and biases of the first generation are randomly initialized. These consist of 3-dimensional matrices of a size that both matches the MLP structure in the first two dimensions and is equal to the *generation_size* in the third dimension. With this configuration, and if the input vector (which stores the position of the player and the enemy at a current time step) is extended to be 3-dimensional as well, the feed-forward pass can be computed for all agents in a generation at the same time, a fact that produces a very fast and efficient training phase.

The auxiliary functions for the initialization of the 3-dimensional agents and inputs are defined as follows. Note that the weight and bias matrices are initialized in 3 dimensions and that the initial input vectors are spanned to match this third dimension with the *repmat* function [8]. Also note that the biases are initialized with zeros, which is a common methodology that leaves the bias changes only to the probability of suffering mutations on them for each agent at the beginning of each generation.

```
function [W1, W2, W3] = initialize_weights(input_size, hidden_size, output_size, range, ...
    generation_size)
    a = range;
    b = -range;

    W1 = (b-a).*rand(input_size, hidden_size, generation_size) + a;
```

```
6       W2 = (b-a).*rand(hidden_size, hidden_size, generation_size) + a;
7       W3 = (b-a).*rand(hidden_size, output_size, generation_size) + a;
8   end
9
10  function [B1, B2, B3] = initialize_bias(input_size, hidden_size, output_size, ...
        generation_size)
11      B1 = zeros(1, hidden_size, generation_size);
12      B2 = zeros(1, hidden_size, generation_size);
13      B3 = zeros(1, output_size, generation_size);
14  end
15
16  function I = generate_initial_inputs(map_x, map_y, init_player_x, ...
        init_player_y,init_enemy_x, init_enemy_y, generation_size)
17      I = [init_player_x/(map_x) init_player_y/(map_y) init_enemy_x/(map_x) ...
            init_enemy_y/(map_y)];
18      I = repmat(I, 1, 1, generation_size);
19  end
```

As seen in the code box, the values of the positions of both the agent and the enemy in the input vectors are normalized with the map length to map them to the [-1, 1] range. This is done for the purpose of allowing efficient learning to the MLP. It is a common methodology that enables the MLP to focus on the more relevant descriptors in the inputs rather than in the strong differences in the magnitudes of the input values. Also, when the positions of these in the grid change they are updated according to this normalized scale.

3.4.2 Gameplay

In this phase all the agents of the generation play the game with continuous record at each time step of which of them are alive and their value of the fitness function achieved.

As mentioned in the above section, the auxiliary function that generates the initial input vectors does so in a way that each input vector matches its according agent in the third dimension, so these input vectors can be updated for the next time step in an appropriate way for each different action that each agent might have taken.

At a given time step, after the outputs of all the agents are computed, different actions are selected according to the configurations of the agents, and hence, different actions are taken by the enemy, generating the need of updating the input vectors for each agent. In the Figure 3 it is shown how the enemy (red square) computes its updates in its position depending on the player's current position. The **d** vector is the distance vector, and the **sin(A)** and **cos(A)** values are the necessary updates in the X and Y coordinates so that the number of distance units moved by the enemy is equal to 1 ($enemy\_mov = 1$) .

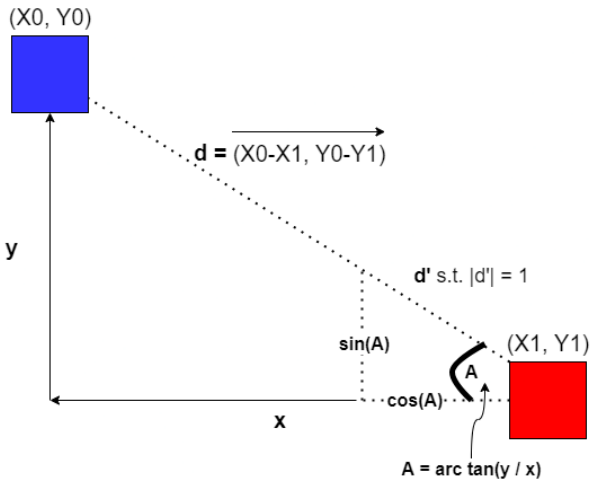


Figure 3: Schema of the enemy position update

These updates are computed in the *take_action()* auxiliary function where the updated inputs, R, and L vectors are returned. See the definition of the gameplay pipeline plus the function that updates the input vectors in the appendices section.

3.5 Reproduction

This phase consists of the last stage of the training cycle and starts when all the agents in the previous generation have ended its gameplay phase. This event is identified at time step t when the condition of $L_t(i) = 0$ *for all* i is met. Also, reproduction is only needed as long as no individual has survived for a number of steps bigger than *max_time_steps*, which would mean that we have empirically obtained an optimal policy. The output of this last stage must be a new generation of agents containing variants of the information of the best-performing agents of the previous generation. The hyperparameter tuning has a strong influence in this phase on balancing the trade-off between convergence and exploration.

As a first step, the k best-performing agents of the recently terminated generation are identified by looking to the indices of the k maximum values of the **R** vector, which stores the fitness values achieved for each of the agents in the previous generation. These subset of agents contains the parents of the new generation and is usually of a very small size compared to the generation size. This fact is due to the will of propagating the knowledge of solutions in a sparse way, giving priority to convergence rather than to exploration.

The second step consists of creating the new generation from the subset of parents and is the crossover step. Several approaches exist on the creation of new generations by crossover. They vary in the way the new agents are influenced by the parents. In this work, the agents randomly select 2 parents and set the values of each of their weights and biases to one in the chosen parents. In other words, for each parameter of the new agents a coin is flipped to select from which parent will the genetic information come. Another possible approach that is not implemented in this work could be to mix information of the entire subset of parents to each of the agents. Note that with the implemented approach the random choices of parents can produce exact copies of a parent in a child. However, that happening might not be a drawback due to the other upcoming level of agent modification, which is the mutation-based. In the following code box the implementation that makes use of the 3-dimensional size of the matrices that define the agents for the reproduction phase is provided. The example shows how in 4 lines of code the weight matrices **W1** (which contain the weights of the connections between the input and first hidden layer) of **the new entire generation** is spanned.

```
%Set W1 for each agent of the new generation to be equal to one of the parents ...
    (different for each agent)
W1 = best_W1(:,:,1+floor((crossover_size)*(rand(1,generation_size))));
%Choose random positions where 2cnd parent will be set
P1 = rand(input_size, hidden_size, generation_size);
%Choose new parent for each individual in the new generation
new_P1 = best_W1(:,:,1+floor((crossover_size)*(rand(1,generation_size))));
%Set 2nd parent weights
W1(P1≤0.5) = new_P1(P1≤0.5);
```

As mentioned above, the last step of the reproduction phase consists of introducing to the agents the mutation-based modifications. These enable the smooth introduction of variability in the agents behaviours for allowing more exploration in the solution space. With this, it is expected to obtain a more balanced trade-off between exploitation and exploration that allows better convergence to the optimal policy. However, since the crossover step implies great variability in the new generation it is not expected that the mutations imply huge changes to all the agents generated and hence, the values of the probabilities of suffering mutations, which are hyperparameters, are usually of small scale. As it happened before when implementing the crossover operation, the mutation-based modifications can also be implemented following different principles. In this work suffering a mutation implies replacing the inherited value of a weight or bias to a new one that is randomly generated (defined in the same range of values). In the following code box an example of mutation on the same **W1** matrix that is provided in the previous example is shown.

```
a_w = weight_range;
b_w = -weight_range;
```

```
3
4  P1 = rand(input_size, hidden_size, generation_size);
5  W1 = current_W1;
6  W1(P1 ≤ weight_prob) = (b_w-a_w).*rand(1, size(W1(P1 ≤ weight_prob),1)) + a_w;
```

Note that again, with very few lines of code, the mutation operation is being performed across the third dimension for all the agents in the recently spanned generation.

The output of the reproduction phase, which consist of the three aforementioned steps is a new generation that is ready to be tested again in the environment.

4 Results

In the following section it is described in detail the results obtained when executing the proposed pipeline for achieving learning of an environment by an MLP that is trained with the GNA. Furthermore, for empirically estimating the impact of the hyperparameter tuning on the agent performance a comparative analysis is developed.

The described program is finally integrated in a Matlab App tool. In the app the user may define the properties of the environment regarding the size and the initial positions of both the player and the enemy, and the MLP and GNA hyperparameters can also be modified and given as an input to the program.

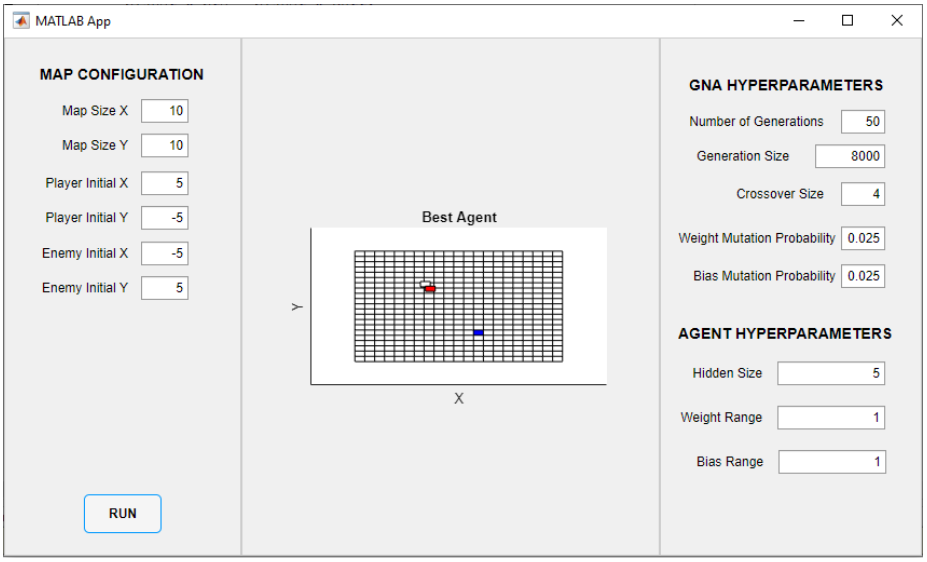
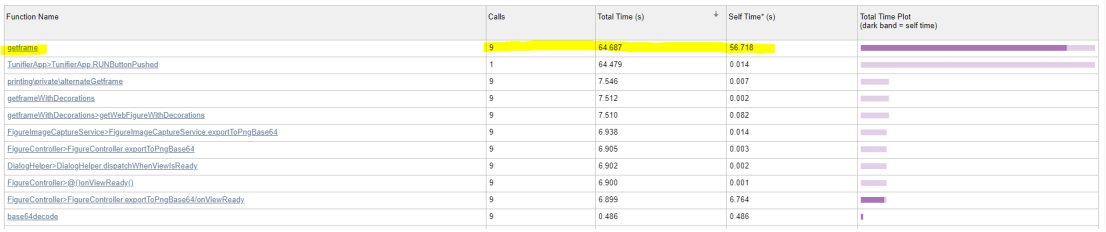


Figure 4: Matlab GUI App

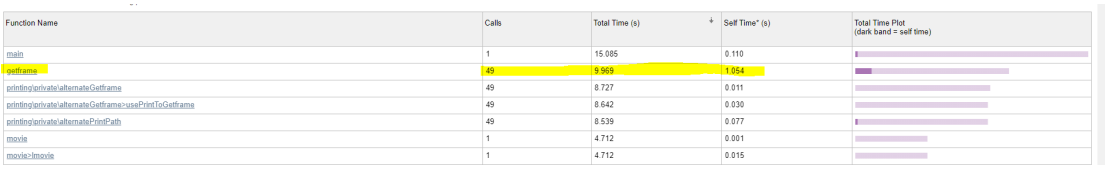
Finally, from this platform the user may run the program and visualize the game of the best-performing observed agent during the configured algorithm run. However, the performance (fluency) of the animations is deprecated when the program is executed in the Matlab App since the *movie* and *getframe* methods used for creating the animation are not supported in the Matlab App Designer tool [4]. In Figures 5 and 6 automatically generated profiles that report the time cost of each executed operation are displayed [6].

In Figure 5 it is shown how the *getframe* operation is not supported when launching it from the Matlab App Designer tool. However, the animation of the gameplay of the best-performing agent is correctly displayed (and fluent) when executing it from the code.



Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
getframe	9	64.687	56.718	
TunifierApp>TunifierApp.RUNButtonPushed	1	64.479	0.014	
printing\private\alternateGetframe	9	7.546	0.007	
getframeWithDecorations	9	7.512	0.002	
getframeWithDecorations>getWebFigureWithDecorations	9	7.510	0.082	
FigureImageCaptureService>FigureImageCaptureService.exportToPngBase64	9	6.938	0.014	
FigureController>FigureController.exportToPngBase64	9	6.905	0.003	
DialogHelper>DialogHelper.dispatchWhenViewIsReady	9	6.902	0.002	
FigureController>@()onViewReady()	9	6.900	0.001	
FigureController>FigureController.exportToPngBase64/onViewReady	9	6.899	6.764	
base64decode	9	0.486	0.486	

Figure 5: Generated profile from the execution in the Matlab App



Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
main	1	15.085	0.110	
getframe	49	9.969	1.054	
printing\private\alternateGetframe	49	8.727	0.011	
printing\private\alternateGetframe>usePrintToGetframe	49	8.642	0.030	
printing\private\alternatePrintPath	49	8.539	0.077	
movie	1	4.712	0.001	
movie>lmovie	1	4.712	0.015	

Figure 6: Generated profile from the execution of the raw code

4.1 Achievement of knowledge

The 3-dimensional computations of the feed-forward pass together with the crossover and mutation operations that also operate in the third dimension allow to obtain a very efficient and parallel simulation of the agent-environment interactions. This capability is key for performing a large number of experiments and eventually configuring the hyperparameters with the best settings. **The program is capable of executing approximately 13.000 games per second and with the appropriate configuration of hyperparameters it finds convergence to the optimal policy 92% of the times**. The speed of execution and the number of games that are run simultaneously at the same moment depends on the length of these games. Actually, the number of games played per second varies from 10.000 to 16.000.

In Figures 7 and 8 it is shown the evolution of the survived time steps in each generation from 1 to 50 in a single run of the algorithm. Although these plots are extracted from only one run with a specific configuration, the pattern observed is clear and generalizes well for any configuration and number of runs. The two figures clearly show the sparsity of the problem/solution relationship. It is seen in Figure 8 that in generation 39 an agent succeeds to survive for 500 time steps, when the maximum ever seen value of steps survived before that event was less than 100. Furthermore, the mean performance of generation 39 is considerably worse than previous generations, meaning that from the same parents there came out a lot of worse new agents but also the best ever seen with big difference. Moreover, no increasing trend is ever shown in the mean performance of the agents in each generation. This happens because the fact of containing the near-optimal genetic information in the configuration of an agent does not imply that this is manifested given the input conditions. Hence, one small change coming from the crossover or mutation operations can produce a massive impact in the fitness function achieved. In other words, an agent can go from performing very bad to succeeding in the environment with a change in a very little part of its genetic information. The job that the GNA does in this work is empirically looking for the maximum number of these little information changes to explore and find the way to the optimal policy. Since it does so in a manner that is highly influenced by random numbers that are generated in the initialization, reproduction and mutation phases, the optimal policy can be found either in the first or in the last generations. Moreover, there exists a considerably large number of optimal policies (very small compared to the number of possible policies) so the outputs of each run of the algorithm when it converges are optimal policies found by an agent that succeeds in the environment, and all these do it in a relatively different way. Furthermore, the same configuration of the program can produce divergent runs where no optimal policy is found. For this reason, in the following section, the fast implementation of the program is exploited for producing a statistical significant analysis of the performance depending on the hyperparameters.

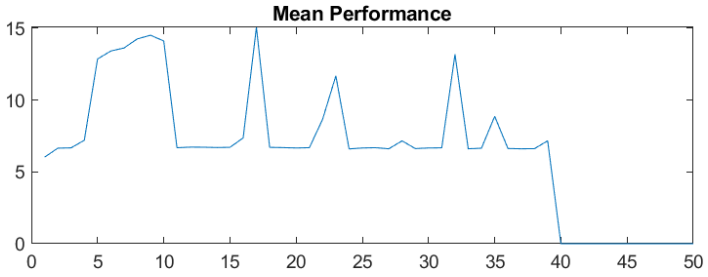


Figure 7: Mean number of steps survived per agent in each generation

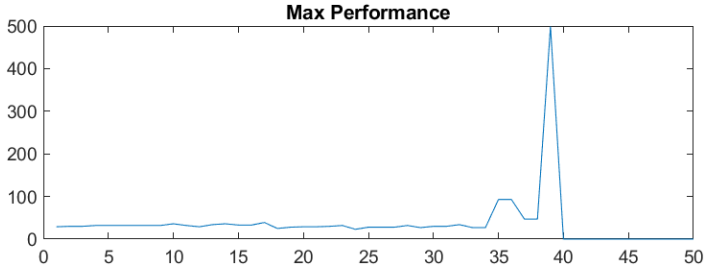


Figure 8: Maximum number of steps survived by an agent of each generation

4.2 Comparative Analaysis

In this section it is described in detail the executions of multiple runs of the algorithm with different configurations for assessing the performance of the implementation. With this it is intended to provide both evidence of the functional implementation, showing that the algorithm converges to the optimal policy with an appropriate configuration, and also insides on the distribution of the hyperparameters, relating these to the changes in the algorithm performance. For all the experiments performed, there are common and fixed values of a subset of hyperparameters, which are listed as follows.

- **Maximum time steps** = 500. If an agent survives for 500 time steps its policy is considered optimal.
- **Number of generations** = 50. If no agent survives for 500 time steps in any of the 50 generations the run is considered to be divergent.
- **Number of runs** = 50. Each configuration of the algorithm is run 50 times for 50 generations for obtaining statistically significant results.
- **Number of Hidden Layers** = 2. The tunaable parameter of the MLP hidden layers is the *hidden_size* which encodes the number of neurons in these 2 layers.

In a first experiment it is wanted to provide insides on the speed of convergence to the optimal solution depending on the generation size, crossover size (length of the subset of chosen parents) and hidden size (number of neurons in the two hidden layers). The results on the average needed generations for achieving the optimal policy, the fastest observed convergence in the 50 runs, and the number of runs that have not converged to the optimal policy are shown in Table 1.

In Table 1 several insides of the distribution of the appropriate values of the hyperparameters are identified. Regarding the generation size, if it considerably decreases from 8000 the output runs are not robust, meaning that the algorithm diverges too many times (e.g with generation size equal to 8000 the algorithm finds convergence to an optimal policy 46/50 runs; with generation size equal to 3000 it converges 26/50 times, and if equal to 1000 it converges 4/50 times). This happens mainly because of the need of a great amount of variability presence in the first generation, so all future policies can be reached. Regarding the crossover size, it is seen that the difference between choosing 4 and 12 parents of each generation for reproduction is key for the algorithm performance. Taking into account too many agents (although

Generation Size	Crossover Size	Hidden Size	Avg. Generations Needed	Fastest Learning Observed	Diverged Runs
8000	4	5	14.5	2	4
3000	4	5	14.42	3	24
1000	4	5	1.16	2	46
8000	12	5	∞	∞	50
3000	12	5	∞	∞	50
1000	12	5	∞	∞	50
8000	4	10	18.54	2	10
3000	4	10	7.26	2	34
1000	4	10	25.73	19	48
8000	4	2	17.22	2	14
3000	4	2	4.28	4	40
1000	4	2	∞	∞	50

Table 1: Speed of convergence (max. and avg.) plus divergence depending on the sizes of the generations, the subsets of parents and the hidden layers.

12 out of 8000 might sound appropriate) causes the new generations to consider too much undesired genetic information. See that with the crossover size equal to 12, no matter what values are given to the other hyperparameters, the algorithm diverges 100% of the times. As it has been described previously, the problem/solution relationship is very sparse in this case and the algorithm must follow this principle of sparsity, selecting very few agents for reproduction. Regarding the hidden size, it is shown how the usage of deeper networks does not involve better performance. The best results for all configurations are obtained when the hidden size is tuned to 4, better than 2 and 10. There is no formula for optimizing this hyperparameter so being able to execute the algorithm so many times allows good statistical-driven decisions. Finally, notice that in 50 runs, if enough variability is ensured in the first generation it almost always happens that there is one run where the optimal solution is found in generation 2, right after the first randomized generation.

In the next experiment, it is wanted to obtain insights in the influence of the probabilities of mutation of the weights and biases and also of the range of values that these take. The provided metrics are the same as the ones in Table 1: Avg. Generations Needed, Fastest Learning Observed and Diverged Runs. The configuration of the generation size, crossover size and hidden size is now fixed to the one in the first row of Table 1, which has shown to be the most robust one. Results are provided in Table 2.

Weight Mutation Pr.	Bias Mutation Pr.	Weight Range	Bias Range	Avg. Generations Needed	Fastest Learning Observed	Diverged Runs
0.08	0.05	-1/1	-0.5/0.5	15.45	2	6
0.025	0.25	-1/1	-0.5/0.5	13.16	2	0
0.15	0.15	-1/1	-0.5/0.5	13.38	2	37
0.08	0.05	-3/3	-1.5/1.5	16.82	2	3
0.08	0.05	-1/1	-1/1	19.78	2	3
0.08	0.05	-3/3	-3/3	20.06	5	7
0	0	-3/3	-3/3	5	3	37
0	0	-1/1	-1/1	3.14	2	36

Table 2: Speed of convergence (max. and avg.) plus divergence depending on the mutation probabilities and parameter ranges.

With this second experiment the key role of the mutations is clear. With too large values for the mutation probabilities the algorithm struggles in finding convergence to an optimal policy since too much exploration is introduced. The same happens if no mutations are introduced, but because of the opposite reason, no exploration through mutations causes the algorithm to stuck in a non-optimal part of the learning phase, causing divergence in the majority of cases. However, with the appropriate values of the mutation probabilities together with the most robust configuration from Table 1 it can be reached the 100% convergence rate.

Regarding the magnitude of the weight and bias parameters, no increase in performance is shown when increasing these to the -3/3 range. However, with the experimented configurations, setting the same range for both weights and biases and equal to -1/1 shows the best rate of convergence. This might be because of the link in the scale of the generated input data, which also lies in the -1/1 range.

5 Conclusions

A functional and efficient implementation of a GNA that trains MLP agents has been deployed in the Matlab framework. The training technique has been verified to be feasible for succeeding in a simple gamified environment. For problems with sparse solutions the GNA has shown that is able to balance the convergence/exploration trade-off when appropriately tuned.

The results on the impact of a set of hyperparameters has been provided with statistically significant conclusions, which allow understanding of the solution capabilities. So, from this work one can identify the role of each of the hyperparameters that are involved in the GNA and the MLP configurations, and can extend the technique capabilities including variants in the key steps of the GNA.

Moreover, the program pipeline has been deployed in a Matlab Application with the drawback of bad performance of the visual animations. However, the built application still lets the user to tune the map and algorithm configurations and train and save an optimal agent correctly.

Finally, it has been observed that the usage of the third dimension in the data structures has been key for enabling parallel computation in the program. This capability has proved to be very useful for adjusting the distribution of the values of the hyperparameters. This has been possible because of the exploitation of the computation capabilities of the Matlab framework. However, the three-dimensional matrix multiplication operation should be optimized in the native Matlab framework because of its usefulness in optimization-like problem solving.

6 Discussion

In this work an specific version of a GNA is implemented to train MLP agents in a given environment. However, there exist several variants that can enhance the functionality of this combined technique. Some of them have been tested in the GNA literature and some of them are proposed in this work (marked in bold). They all are listed as follows.

- The crossover subset of agents (parents) can include random individuals of the previous generation to introduce a new level of exploration.
- The new agents in each generation can obtain genetic information from each of the parents, not only from a pair of them.
- The mutation strength magnitude could be tuned for achieving more control over the exploration levels, so that a mutation would involve more little changes in the behaviour of an agent.
- **After the successful training of x agents, these could be reproduced in order to span new generations that consist of combinations of all-optimal agents in order to allow more generalization in the environment**.
- **A not-deterministic environment could be used to make the agents capable of adapting to changes in this**. e.g. Introducing pseudo-randomized enemy movement, introduction of a new enemy in a chosen generation x, introduction of obstacles (walls, cliffs) in different positions in each run.

References

[1] University of Edinburgh. *"Reinforcement Learning with Neuroevolution: Final Report"*. 2019. URL: http://www.inf.ed.ac.uk/teaching/courses/mlp/2019-20/example_cw4/finalReport-G109.pdf.

[2] MathWorks. *"Capture axes or figure as movie frame"*. URL: https://es.mathworks.com/help/matlab/ref/getframe.html.

[3] MathWorks. *"Fill 2-D polygons"*. URL: https://es.mathworks.com/help/matlab/ref/fill.html.

[4] MathWorks. *"Graphics support in App Designer"*. URL: https://es.mathworks.com/help/matlab/creating_guis/graphics-support-in-app-designer.html.

[5] MathWorks. *"Play recorded movie frames"*. URL: https://es.mathworks.com/help/matlab/ref/movie.html.

[6] MathWorks. *"Profile execution time for functions"*. URL: https://es.mathworks.com/help/matlab/ref/profile.html.

[7] MathWorks. *"Read matrix from a file"*. URL: https://es.mathworks.com/help/matlab/ref/readmatrix.html.

[8] MathWorks. *"Repeat copies of an array"*. URL: https://es.mathworks.com/help/matlab/ref/repmat.html.

[9] MathWorks. *"Write matrix to a file"*. URL: https://es.mathworks.com/help/matlab/ref/writematrix.html.

[10] Felipe Petroski Such et al. ""Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning"". In: (2017). URL: https://arxiv.org/abs/1712.06567.

[11] James Tursa. *"MTIMESX - Fast Matrix Multiply with Multi-Dimensional Support"*. URL: es.mathworks.com/matlabcentral/fileexchange/25977-mtimesx-fast-matrix-multiply-with-multi-dimensional-support.

Appendices

In this Section one can find the code of the main script for executing the training pipeline. Only one of the auxiliary functions is provided in the Appendices Section.

See more details about the code in the README.txt file attached.

```
%%%%%% TRAINING PHASE %%%%%%

%%%%% INITIALIZATION %%%%%
[W1, W2, W3] = initialize_weights(input_size, hidden_size, output_size, weight_range, ...
    generation_size);
[B1, B2, B3] = initialize_bias(input_size, hidden_size, output_size, generation_size);

mean_performance = zeros(1,num_generations);
max_performance = zeros(1,num_generations);
best_performance = 0;

%%%%% GAMEPLAY %%%%%
for i=1:num_generations
    L = ones(1,generation_size);
    R = zeros(1,generation_size);
    I = generate_initial_inputs(map_x, map_y, init_player_x, init_player_y,init_enemy_x, ...
        init_enemy_y, generation_size);
    current_time_step = 1;

    %%% GAMEPLAY %%%
    while current_time_step < max_time_steps
        dead = find(L==0);
```

```
        %Check if all agents have died
        if size(dead,2) == generation_size
            break
        end

        I(:,:,dead) = 0;
        W1(:,:,dead) = 0;
        W2(:,:,dead) = 0;
        W3(:,:,dead) = 0;
        B1(:,:,dead) = 0;
        B2(:,:,dead) = 0;
        B3(:,:,dead) = 0;

        %3D Feed Forward pass (dead agents set to 0)
        OUTPUTS = softmax(mtimesx(ReLU(mtimesx(ReLU(mtimesx(I,W1) + B1), W2) + B2),W3) + ...
            B3);

        %Agent takes action depending on OUTPUTS and enemy does too accordingly
        [I,L,R] = take_action(I, OUTPUTS, L, R, map_x, map_y, player_adv, ...
            enemy_distance_collision);

        current_time_step = current_time_step+1;
    end

    %%% REPRODUCTION + MUTATION %%%
    best_ind = select_best_individuals(crossover_size, R);

    [W1,W2,W3,B1,B2,B3] = create_new_generation(W1(:,:,best_ind), W2(:,:,best_ind), ...
        W3(:,:,best_ind), B1(:,:,best_ind), B2(:,:,best_ind), B3(:,:,best_ind), ...
        generation_size);

    [W1,W2,W3,B1,B2,B3] = ...
        mutate_generation(W1,W2,W3,B1,B2,B3,weight_mutation_prob,bias_mutation_prob,
    weight_range,bias_range,generation_size);

    mean_performance(i) = mean(R);
    max_performance(i) = max(R);
    games_played = games_played + generation_size;

    %Stopping if game is solved at any generation
    if max(R) == max_time_steps-1
        break
    end
end
```

The function that updates the player's and enemy's position in the input vectors at each time step is defined as follows. Notice that it only operates for the input vectors for which its matching agent is alive.

```
function [update, L, R] = take_action(inputs, outputs, L_, R_, map_x, map_y, ...
    player_adv,enemy_dist_coll)
    update = inputs;
    R = R_;
    L = L_;

    %Iterate on agents that are alive
    for i=find(L)
       %AGENT ACTION
       [M,idx] = max(outputs(:,:,i));

       if idx == 1
           update(1,1,i) = inputs(1,1,i) + (player_adv / (map_x));
       elseif idx == 2
           update(1,1,i) = inputs(1,1,i) - (player_adv / (map_x));
       elseif idx == 3
           update(1,2,i) = inputs(1,2,i) + (player_adv / (map_y));
       elseif idx == 4
           update(1,2,i) = inputs(1,2,i) - (player_adv / (map_y));
       end

       %ENEMY ACTION
       enemy_direction_Y_pre = update(1,2,i) - update(1,4,i);
```

```
        enemy_direction_X_pre = update(1,1,i) - update(1,3,i);

        tan = atand(abs(enemy_direction_Y_pre)/abs(enemy_direction_X_pre));

        enemy_direction_Y = sind(tan) / (map_x);
        enemy_direction_X = cosd(tan) / (map_y);

        if enemy_direction_X_pre < 0
            enemy_direction_X = -enemy_direction_X;
        end

        if enemy_direction_Y_pre < 0
            enemy_direction_Y = -enemy_direction_Y;
        end

        update(1,3,i) = update(1,3,i) + enemy_direction_X;
        update(1,4,i) = update(1,4,i) + enemy_direction_Y;

        %Crash into wall check
        if abs(update(1,1,i)) ≥ 1 ||  abs(update(1,2,i)) ≥ 1
           L(i) = 0;
        %Catched by enemy check
        elseif  abs(update(1,1,i)-update(1,3,i)) ≤ enemy_dist_coll && ...
            abs(update(1,2,i)-update(1,4,i)) ≤ enemy_dist_coll
           L(i) = 0;
        else
           R(i) = R(i) + 1;
        end
    end
end
```