Playing Mega Man II with Neuroevolution

Fernando Ishikawa, Leandro Z. Trovões, Leonardo Carmo, Fabrício Olivetti de França, Denis G. Fantinato
Center for Mathematics, Computing and Cognition (CMCC)
Federal University of ABC (UFABC)
Santo André, Brasil
{fernando.ishikawa,leandro.trovoes,leonardo.carmo}@aluno.ufabc.edu.br,
{folivetti,denis.fantinato}@ufabc.edu.br

***Abstract*—The problem of developing Game-Playing Agents provides a controlled environment with varying levels of difficulty in order to test different Artificial Intelligence algorithms. A recently proposed framework for testing such algorithms is called EvoMan and was created based on a classic and challenging game called MegaMan II. In this framework, the agent must defeat a number of different enemies equipped with a diverse set of weapons with different behaviors. This paper follows up the Evoman: Game-playing Competition hosted at the World Conference on Computational Intelligence in 2020 with the objective of finding a general strategy capable of defeating all of the bosses training only on a subset of those. Our approach is composed of manually crafted inputs based on the available sensors fed into a Neuroevolution algorithm composed of a Genetic Algorithm evolving the weights of a Multilayer Perceptron. Our results obtained the first place on the competition and was capable of defeating the entire set of enemies.**

***Index Terms*—Neuroevolution, Genetic Algorithm, Artificial Neural Network**

I. Introduction

The use of Computational Intelligence techniques in games has received great attention of the research community and is experiencing a rapid development [1]. As an established research field, this topic focuses on creating game playing agents, adapting the difficulty of the game to the human player, creating randomized environments, etc. [2]–[4]. Since games are able to pose different levels of challenges and require the adequate use of a wide sort of skills, they emerge as a relevant problem for Artificial Intelligence (AI) algorithms. Different contexts of a game demands a set of skills to be learned in different proficiency levels [1].

Within this research field, a prominent learning technique widely used is the neuroevolution (NE) [5], [6], which can either make use of optimization metaheuristics to adapt the weights of an Artificial Neural Network (ANN) [7] with fixed topology, or adapt the weights and the topology itself. The optimization metaheuristics are usually from the family of evolutionary algorithms [8]. Not only in games, this technique has found application in a diverse set of tasks, such as robotics [9], biophysics [10] and others [11]. The main advantage of this technique in comparison with classical ANNs – trained by the backpropagation algorithm [7] – is that it is more robust against local minima convergence and can be more efficient in finding the suitable topology [5].

Notably, neuroevolution techniques were capable of achieving remarkable performance in a large variety of games, which includes Pac-Man [12], Quake II [13], Unreal Tournament [14] racing and fighting games [15], [16]. However, since each game has its own particular set of actions/actuators, there is not an unique solution that fits all games. Each game poses a different challenge for the learning algorithms [17]. In that sense, in this work, we aim at applying a neuroevolution technique in Mega Man II, a very challenging platforming game [18] from the 80s. More specifically, our method will be analyzed within the Evoman framework [19], a public domain version clone of the original game.

The rest of this paper is organized as follows. In Section II the Evoman competition challenge is described. Section III presents the background on neuroevolution, detailing the artificial neural networks and the genetic algorithms. The attributes selection and the classification methods are presented in Sections IV and V. The performance of our proposal is shown in Section VI and, finally, the conclusions are presented in Section VII.

II. Evoman Competition

Evoman is an open source video game playing framework inspired by the game Mega Man II for developing artificial intelligence algorithms [18], [20]. In Evoman, a playable character has the objective of defeating a total of 8 different Master Robots, without the stages that precede them in the original game, each one having different patterns and behavior and supported by different scenarios. The player has at their disposal the ability to move forward and backward, jumping and shooting from an arm cannon. These moves can be done simultaneously, so one can jump forward while shooting.

The framework uses the Python library Pygame, a cross-platform set of Python modules, with the purpose of supporting the creation of games. It consists of computer graphics and sound libraries designed to be used with the Python programming language.

The objective of this work is to create a general agent capable of defeating all eight opponents while learning only from a subset of four of those enemies.

Since each enemy behavior is notably different from each other, the player should learn general strategies and reactions

978-1-7281-2547-3/20/$31.00 ©2020 IEEE

from common patterns of the enemies behavior such as avoiding being shot and shoot in the direction of the enemy. This has already been proven to be a challenge to different algorithms [18].

To properly measure the knowledge acquired at the training stage, the performance of the intelligent agent is indirectly evaluated as an harmonic mean of the following function applied to each boss fight, as the framework limits its direct measure: Based on the knowledge acquired at the training stage, the performance of the intelligent agent is evaluated as an harmonic mean of the following function applied to each boss fight:

$$J = 100.01 + ep - ee, \tag{1}$$

where ee and ep are the final amount of energy of the enemy and the player, respectively. The value of 100.01 is added so that the harmonic mean always yields valid results ($J > 0$) . The goal is the maximization of the harmonic mean, i.e., the defeat of every boss without getting hit. Both the agent and the enemies start the game with 100 energy points. Every time one player gets hit, it loses some points. Whoever reaches 0 points loses the match.

There are several ways to categorize the environment where the agent should learn, providing relevant features that can assist the analysis of alternative ways for solving the problem. For this particular case, Evoman can be described as:

- **Partially observable**, due to the knowledge of relative positions of the enemy and projectiles, but no information about the field or your own shoots.
- **Multi-agent competitive**, the enemy tries to maximize the damage given by pursuing and shooting at the player while the agent needs to keep alive to defeat the opponent.
- **Nondeterministic**, there is a level of randomness on the bosses actions, so a deterministic sequence of actions cannot return the optimal strategy for different games.
- **Sequential**, the agent's actions affect the future movements of the enemy which modifies the fight procedure.
- **Static**, the game procedure is dependent on the agent's actions, the game only continues once its response is returned.
- **Discrete**, there is a finite number of possible states, although this number is considerably large.

To solve the problem, there are several approaches that are either difficult or impossible considering some features, mainly because it is nondeterministic and partially observable. For instance, one of the most efficient algorithm to solve game related problems are reinforcement learning [21], however the inability to identify successful shots would affect its implementation, since there is no immediate feedback.

III. Background

The solution developed for training the agent was composed of a neuroevolution algorithm [5], which, by using a search metaheuristics such as the Genetic Algorithm [8], is responsible for searching for the best weights of an Artificial Neural Network (ANN) [7] in order to achieve an optimal solution for the majority of bosses, according to Eq. (1). The two main elements of the adopted neuroevolution approach are described bellow.

A. Artificial Neural Networks

An artificial neural network is an information processing system inspired by the human brain cells: it is composed of several units called neurons and, by connecting them in net, they are capable of doing complex analysis on stimulus. The basic element, an artificial neuron, is also called perceptron and has three basic elements: a set of synapses weights, a linear combiner and a nonlinear activation function [22]. The synapses weights are related with the relevance of the input for the neuron, which could be positive, negative or null. The neuron can also include a bias term, a value that adds a new fixed input signal, with the effect of applying an affine transformation which translates the output of the linear combiner [22]. Next, the linear combiner performs the summation of all signals received and weighted by the synapses. At last, the activation function is responsible for limiting the amplitude of the neuron output to some finite value and also to apply a nonlinear transformation – usual activation functions are the hyperbolic tangent and the ReLU [7]. Mathematically, the perceptron's output can be defined as:

$$y = f\left(\mathbf{w}^T\mathbf{x}\right), \tag{2}$$

where $\mathbf{x} \in \mathcal{R}^{M\times 1}$ is the input vector with M attributes, $\mathbf{w} \in \mathcal{R}^{M\times 1}$ is the vector with the synaptic weights and $f(\cdot)$ is the nonlinear activation function.

Generally speaking, we can classify ANNs in three categories regarding their architectures (which depends on how the artificial neurons are connected): Single-Layer Feedforward Networks, Multilayer Feedforward Networks and Recurrent Networks [22]. In this work, we will focus on the Multilayer Feedforward Networks or Multilayer Perceptrons (MLPs), which are characterized by the presence of one or more intermediary layers, also known as hidden layers, between the input and the output ones. In this case, the neurons are structured along the exclusive connection of the outputs of one layer to the inputs of the next, defining the single direction of *data flow*, characterizing the feedforward approach. The hidden layers make it possible to handle nonlinear problems by extracting more complex patterns of the input. Finally, the last layer (output) is responsible for combining the information processed by the previous layers and to yield an output that can be used for decision taking [22]. Fig. 1 depicts a visual representation of a MLP with one hidden layer and two outputs. The arrows indicate the flow of information between neurons.

B. Genetic Algorithms

Genetic algorithm (GA) is a metaheuristic that can perform the search for a solution based on the natural evolution proposed by John Holland in 1975 [23]. The algorithm starts with a set of randomly initialized agents, with each agent called an *individual* and the set being called *population*. The

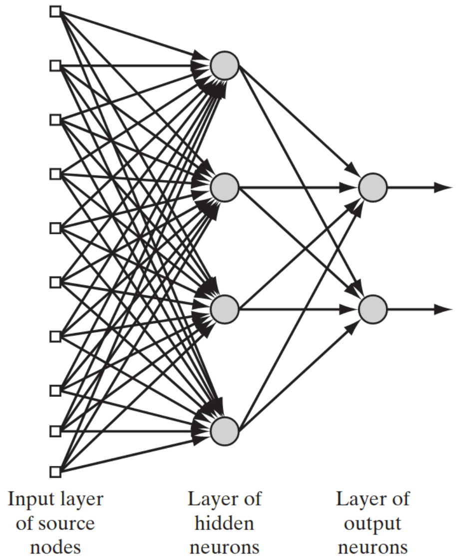

Fig. 1. Graphic representation of a MLP (extracted from [22]).

quality of each individual is then numerically assessed by means of a fitness function with a highest value representing better solutions [21].

Each state, named as individual, is rated by the fitness function, a function that calculates a value that represents how close it is to a satisfying solution [21].

At every iteration, called generation, the population goes through the evolution steps by combining their solutions with the crossover operator and slightly changing them with the mutation operator. The crossover operator combines a pair of selected solutions – called parents – by copying parts of their chromosome into a new individual called offspring. These offsprings can also go through a slightly change in their solution, called mutation [21]. Finally, a sample of individuals from the combined population of parents and offsprings is selected for replacing the current population.

There are many variations for the *selection, crossover, mutation* operators. For the selection operator we can choose from Boltzmann selection, tournament selection, rank selection, steady-state selection, elitism selection, among others [24]. For the crossover, there are three main types of crossover operators, namely as single-point, two-point and uniform crossover [24]. One should select the appropriate ones based on the specific problem to solve. Alg. 1 describes the main steps of the GA algorithm, being P the population and F the fitness value associated with each individual.

C. Neuroevolution

To solve the problem of finding the parameters of artificial neural networks, the usual solution consists on the backpropagation algorithm, which uses the current error calculated on an iteration to adjust the preceding layers weights. However, on this specific problem there are no means of determining the quality of the outcome of a single action of the agent, we

Algorithm 1: Genetic Algorithm

P = initialPopulation();
while *not shouldTerminate()* **do**
  F = calculateFitness(P);
  P = selection(P, F);
  P = crossover(P);
  P = mutation(P);
end

can only measure the overall quality of the agent after the end of a match.

Also, this process could lead to a local optimal solution [25] that might not be satisfactory for the problem. Less susceptible to this issue, metaheuristics can be used instead [5]. Although not able to guarantee optimal convergence, they generally find very promising solutions. Some of the other advantages are that they provide a more flexible way of training the ANN when there is no clear definition of error function; they could be used to determine a good starting point for back-propagation [5] and they allow other hyperparameters, such as number of layers or layer size, to be optimized [26].

Hence, in this work, we apply the GA for adapting the weights of the ANN.

IV. Introducing new features

The Evoman game provides, in real time, 20 sensors for the agent:

- **Distance to enemy**: the horizontal and vertical distances, in pixels, between the player and the enemy (total of 2 attributes).
- **Distance to projectiles**: the horizontal and vertical distances, in pixels, between the player and each of the eight projectiles (total of 16 attributes).
- **Directions**: the direction both player and enemy are facing (total of 2 attributes).

These attributes could be used as the input for the ANN. However, using 20 attributes may be an excessive amount, since it would require a larger ANN dimension (number of neurons and layers) and, as a consequence, a larger number of parameters for the GA to search. Besides that, some of these attributes may not be relevant, contributing for a performance loss. In that sense, we used this information to create other attributes for the ANN:

- From the horizontal distance between the characters and the direction the player is facing, we calculate whether or not the player is facing the enemy; This would be a better indication to the player that his shots can be effective.
- The horizontal and vertical closest distances between the projectiles and the player; we mapped the euclidean distance for each one of them and removed the farthest three of the projectiles, even when there were less than 4 of them;

- Finally, we preserved the sensors of the horizontal and vertical distances between the player and the boss and the direction the enemy is facing.

By limiting the number of projectiles the agent has to deal with, we also limit the number of parameters the genetic algorithm has to find, simplifying the search and prioritizing the most relevant information to take a decision. A projectile that is far away is unlikely to influence the outcome of the match, also, as not every boss uses them all, this difference could bring unexpected behaviours to the ANN when values that did not have a meaning in the training phase are used during the test.

For every distance x, we also applied the following transformation:

$$\text{dist}(x) = \begin{cases} 0 & \text{if x = 0} \\ g(x) & \text{if x >0} \\ -g(x) & \text{if x <0} \end{cases} \tag{3}$$

being

$$g(x) = 2^{-(x/150)^2}. \tag{4}$$

Hence, Eq. (3) is used to normalize the values between -1 and 1 and assign greater weights to the changes in projectiles closer to the player. Fig. 2 depicts the behavior of function $dist(x)$, by varying x from -600 to 600.

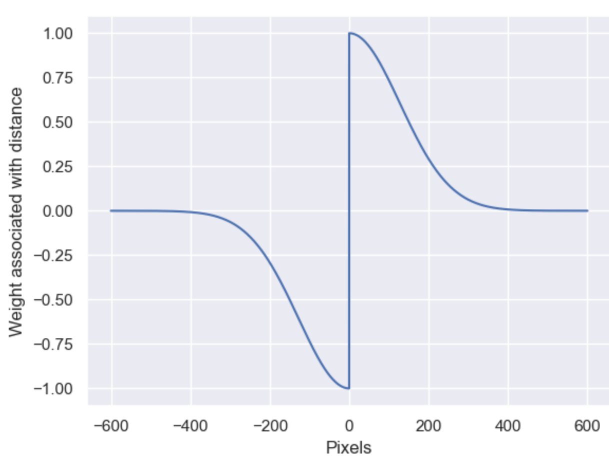

Fig. 2. Graphic representation of the function $dist(x)$.

V. Classifier

We used an ANN to decide the actions the agent should perform on each iteration of the game. It consisted of 14 input neurons, two intermediate layers, containing 32 and 12 neurons respectively, and five non-exclusive outputs. Each output corresponds to the actions the player can perform: right, left, shoot, jump and release jump. All layers used a sigmoid activation function, with the output having a threshold of 0.5.

The weights of this ANN were optimized by a search performed by the GA adapted to our problem, in which the individuals were composed of a set of weights between each pair of neurons of the network. Starting with 10 individuals, each one was initialized with values from a uniform distribution inside the -1 and $+1$ interval.

One by one, all individuals were then put to fight against each of the four training bosses. At the end of the four fights, they were evaluated using the match duration time and the remaining life of the fighters.

The fitness function used to evaluate the individual against one boss was

$$\begin{aligned} fightResult =& 0.7 * (100 - enemyLife) \\ &+ 0.3 * playerLife - ln(time), \end{aligned} \tag{5}$$

and then the individual fitness was calculated using

$$fitness = mean(fightResult)/2 + min(fightResult). \tag{6}$$

For each generation, at the crossover step, we used a convex combination between two parents, as used in Evolutive Strategies algorithms [8] – aimed to deal with real valued problems.

We selected each pair of the parents individuals by sampling without replacement the entire population, then their values were multiplied by complementary values chosen randomly inside the 0 to 1 interval and added together.

After that, the individuals from both the previous generation and the resulting ones from the recombination were submitted to a mutation process, in which one generates five new individuals. The newborns were then evaluated the same way as before; if one had performed better than their predecessor, it would substitute the original individual.

The mutation was performed by adding values from a gaussian distribution with zero mean and standard deviation σ on each weight of the ANN. In the case the individual was created at the current generation, the value of σ were fixed at 0.5. However, if the individual comes from the previous generations, the value would be selected as $(150 - current)/150$, where 150 is the number of generations run by the algorithm and $current$ the number of generations so far. This was done to benefit the exploration of the search space on the beginning iterations and gradually increasing exploitation to try to achieve an optimal solution.

Finally, the best individuals are selected to survive the next generation and the remainder are discarded in order to keep the size of the population constant.

At every 20 iterations, the worst performing half of the population is discarded and new individuals are sampled in order to increase exploration of new regions of the search space and stall convergence. This sampling is performed the same way as in the population initialization.

The algorithm performs 150 iterations and store the best individual of each population. At the end, to prevent the agent specialization on the train set, these individuals are tested against all enemies and the one with the highest weighted score among all bosses is chosen as the final weights of the ANN. To achieve this, a more common approach like early stopping could not be used. The small amount of enemies available

TABLE I
HYPERPARAMETERS

Parameter	Value
Hidden layers	2
Neurons	$(32, 12)$
Population Size	10
Generations	150
Mutation	Gaussian
Crossover	Convex combination
Selection	Elitism
Resampling rate	20 gens
Training bosses	$1, 4, 6, 7$

TABLE II
RESULTS OF BEST AGENT

Boss	***Energy player***	***Energy enemy***	***Time***
1	64.00	0.00	184
2	68.00	0.00	288
3	8.00	0.00	394
4	34.60	0.00	860
5	87.40	0.00	254
6	17.80	0.00	600
7	81.40	0.00	150
8	58.60	0.00	422
Harmonic Mean	28.44	0.00	290.96

preclude the selection of a part exclusive for this verification. Also, the high volatility of results between individuals against the testing set, at each round, could be misinterpreted as a sign of overfitting.

VI. RESULTS

For the sake of clarity, the hyperparameters and configurations used in our approach are depicted in Table I. As we can see from this table, the enemies selected for the training stage were 1, 4, 6 and 7. The enemies choices showed to be a relevant step of the algorithm, as particularities of certain combat fields — in specific, the water field of boss 7 — are not present in others i.e. cannot be learned if the boss is not present on training.

A summary of the results is reported in Table II. From this table, we can see that the best agent succeeded in defeating every enemy of the enemies set (see **enemy energy** column). As expected, the worst performance was obtained against a boss not pertaining to the training set. Particularly, one of the possible shots of this enemy is a circle of leaves that comes from the top, instead of the usual horizontal bullets. For this reason, our agent might have found more difficulty on beating this particular boss.

Curiously, the best performance was obtained against another boss outside the training set. Unlike most of the bosses, this boss usually stays at one side of the screen, allowing the player to just jump and shoot while staying at the other side.

The longest fight was obtained against the fourth boss. One of the attacks of this boss is a fire dash in which it makes the enemy intangible during the attack. For this reason, the player has less opportunities to shoot at the enemy.

TABLE III
COMPARISON BETWEEN THE UPPER BOUNDS AND OUR BEST AGENT.

Algorithm	**Gain**
Our approach	147.12
GAP	139.64
GA10	143.74
GA50	149.43
LOP	0.04
LO10	104.01
LO50	79.32
NEAT	185.67

Since the energy of each enemy is zero for every stage, the gain is simplified to $J = 100.01 + ep$. The harmonic mean of the gain is then 147.12.

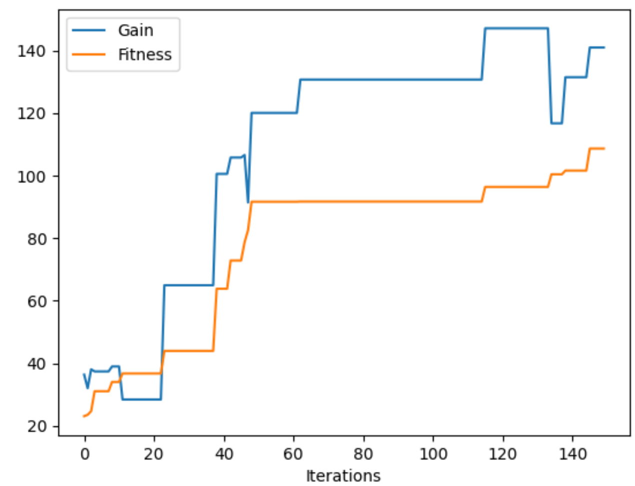

Fig. 3. Fitness function against train bosses and general gain of best agent of each iteration.

As means of comparison, in Table III we replicate the upper bounds reported in [19] that serves as a baseline for this competition. It includes the results of two ANNs: (*i*) a 1-layer perceptron and (*ii*) a 2-layers perceptron with 10 and 50 neurons for the hidden layer, whose weights were adjusted by GA (denoted by 'GAP' for the single layer perceptron, and 'GA10' and 'GA50', for the 2-layers perceptron with 10 and 50 neurons, respectively) and by LinkedOpt algorihtm [27] (denoted, analogously to GA, as 'LOP', 'LO10' and 'LO50'). The latter strategy comprises the evolution of a Neural Network topology along with their weights by means of the NEAT algorithm [28].

In Table IV, we present the main results obtained in the World Conference on Computational Intelligence (WCCI) competition, where the other strategies encompassed the adjustment of an ANN – two hidden layers, with 32 neurons each – through the Proximal Policy Optimization (PPO) algorithm, and the MultiNEAT, an extension of the NEAT algorithm to a multi-populational coevolutionary strategy [28]. The tests were

TABLE IV
RESULTS OF "EVOMAN: GAME-PLAYING COMPETITION FOR WCCI 2020"

Algorithm		**Boss**								**Gain**
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
Our	Player	64	68	8	35	84	18	81	59	147
	Enemy	0	0	0	0	0	0	0	0	
PPO	Player	100	90	7	3	57	94	90	45	138
	Enemy	0	0	20	22	0	0	0	0	
Multi Neat	Player	0	16	0	37	41	23	36	80	56
	Enemy	80	0	80	0	0	0	0	0	

Displayed values are rounded.

executed considering a time limit of 3000 game time steps per fight, the default value of the framework.

Notice that the results from Table III were obtained by training different agents specialized at a single specific boss, an easier challenge than the one presented in this paper. As one can see, our agent succeeded on finding a general strategy, presenting a performance comparable with that obtained by the set of specialist agents.

Also, from Table IV, it is possible to note that our agent performed better than the other competitors, being the only one able to defeat all bosses.

VII. CONCLUSION

In this paper, we created an intelligent agent for the game playing framework Evoman with the objective to learn actions that are general enough to fight against several enemies with different behaviours, while training with only four of the eight enemies.

For doing so, we selected a subset of the 20 sensors provided by the game, as part of them were not much useful for decision taking, and introduced a new feature to help on shooting. These information were presented to an ANN at each tick of the game to select the subsequent action. This ANN were trained using a GA as an alternative to handle the absence of a clear outcome for each action.

The results were successful as the agent managed to defeat all the eight bosses in the game. Despite the small quantity of enemies, this is a evidence of how generalist the agent is.

However, as there are a small numbers of enemies, one possible future test to a better outcome could include more enemies to be more confident that the agent generalizes well.

Other than that, more sensors could be provided to possibly give a better certainty on the agent decision making.

REFERENCES

[1] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 1, pp. 25–41, 2015.

[2] K. Chellapilla and D. B. Fogel, "Evolution, neural networks, games, and intelligence," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1471–1496, 1999.

[3] R. Miikkulainen, "Creating intelligent agents in games," in *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2006 Symposium*. National Academies Press, 2007, pp. 15–15.

[4] R. Miikkulainen, B. D. Bryant, R. Cornelius, I. V. Karpov, K. O. Stanley, and C. H. Yong, "Computational intelligence in games," *Computational Intelligence: Principles and Practice*, pp. 155–191, 2006.

[5] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[6] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[7] C. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[8] I. BoussaïD, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information sciences*, vol. 237, pp. 82–117, 2013.

[9] S. Nolfi, D. Floreano, and D. D. Floreano, *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press, 2000.

[10] J. Clune, J.-B. Mouret, and H. Lipson, "The evolutionary origins of modularity," *Proceedings of the Royal Society b: Biological sciences*, vol. 280, no. 1755, p. 20122863, 2013.

[11] A. K. Hoover, P. A. Szerlip, M. E. Norton, T. A. Brindle, Z. Merritt, and K. O. Stanley, "Generating a complete multipart musical composition from a single monophonic melody with functional scaffolding." in *ICCC*. Citeseer, 2012, pp. 111–118.

[12] S. M. Lucas, "Evolving a neural network location evaluator to play ms. pac-man." in *CIG*. Citeseer, 2005.

[13] M. Parker and B. D. Bryant, "Visual control in quake ii with a cyclic controller," in *2008 IEEE Symposium On Computational Intelligence and Games*. IEEE, 2008, pp. 151–158.

[14] R. Kadlec, "Evolution of intelligent agent behaviour in computer games," *Master's thesis, Charles University in Prague*, p. 75, 2008.

[15] A. Agapitos, J. Togelius, and S. M. Lucas, "Evolving controllers for simulated car racing using object oriented genetic programming," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1543–1550.

[16] T. Graepel, R. Herbrich, and J. Gold, "Learning to fight," in *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*. Citeseer, 2004, pp. 193–200.

[17] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super Mario Evolution," in *IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 156–161.

[18] K. d. S. M. de Araújo and F. O. de França, "An electronic-game framework for evaluating coevolutionary algorithms," *arXiv preprint arXiv:1604.00644*, 2016.

[19] F. O. de Franca, D. Fantinato, K. Miras, A. Eiben, and P. Vargas, "Evoman: Game-playing competition," *arXiv preprint arXiv:1912.10445*, 2019.

[20] K. Miras and F. O. de Franca, "Evolving a generalized strategy for an action-platformer video game framework," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 1303–1310.

[21] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 2002.

[22] S. S. Haykin *et al.*, "Neural networks and learning machines." 2009.

[23] J. H. Holland *et al.*, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[24] L. Haldurai, T. Madhubala, and R. Rajalakshmi, "A study on genetic algorithm and its applications," *International Journal of Computer Sciences and Engineering*, vol. 4, no. 10, p. 139, 2016.

[25] M. Gori and A. Tesi, "On the problem of local minima in backpropagation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 1, pp. 76–86, 1992.

[26] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton, "Optimizing deep learning hyper-parameters through an evolutionary algorithm," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015, pp. 1–5.

[27] F. O. de Franca, "Maximization of a dissimilarity measure for multimodal optimization," in *2015 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2015, pp. 2002–2009.

[28] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.