Evolving a Generalized Strategy for an Action-Platformer Video Game Framework

Karine da Silva Miras de Araujo and Fabrício Olivetti de Franca, *Member, IEEE*
Universidade Federal do ABC (UFABC)
Center of Mathematics, Computing and Cognition (CMCC)
R. Santa Adélia 166, CEP 09210-170, Santo André, Brazil
Email: karine.smiras@gmail.com, folivetti@ufabc.edu.br

***Abstract*—Computational Intelligence in Games comprises many challenges such as the procedural level generation, evolving adversary difficulty and the learning of autonomous playing agents. This last challenge has the objective of creating an autonomous playing agent capable of winning against an opponent on an specific game. Whereas a human being can learn a general winning strategy (i.e., avoid the obstacles and defeat the enemies), learning algorithms have a tendency to overspecialize for a given training scenario (i.e., perform an exact sequence of actions to win), not being able to face variations of the original scenario. To further study this problem, we have applied three variations of Neuroevolution algorithms to the EvoMan game playing learning framework with the main objective of developing an autonomous agent capable of playing in different scenarios than those observed during the training stages. This framework is based on the bosses fights of the well known game called Mega Man. The experiments show that the evolved agents are not capable of winning every challenge imposed to them but they are still capable of learning a generalized behavior.**

***Keywords*—video game playing, autonomous agent, neuroevolution.**

I. Introduction

Computational Intelligence in Games [1] is the field that studies the application of machine learning techniques with the goal of creating autonomous agents capable of learning to play games with human-like abilities or super-human abilities. This study can lead to the developing of autonomous agents capable of making decisions in order to solve problems involving uncertainties.

The games may comprise the classic board games [2], like chess, checkers, GO, or the most modern eletronic Video Games [3]. In many board games, there are one or more optimal strategies (for winning or forcing a draw) but that are usually unfeasible to compute. Regarding electronic Video Games, they usually have some uncertainties associated to the consequences of the agent action (i.e., there is not enough information in order to build a game tree).

As such, in Video Game playing, the learning agent relies on learning from the measurable results after a sequence of actions. These measurable results can be: points earned, enemies defeated, agent dying, etc. One problem that the learning algorithms may face is the over-specialization, in which they learn how to beat an specific stage really well but become incapable of generalizing their knowledge to further stages.

This happens because the different stages of a Video Game may have variations of the patterns for the obstacles and enemies. For example, on a *platformer* game, an ideal general strategy would be to move the agent in such a way to avoid the obstacles. However, during the learning stage the agent may face obstacles that move only on an straight horizontal pattern, and so it would learn to move up or down accordingly. Yet, on the next stage, the obstacles could start moving on a zigzag pattern, requiring a different sequence of actions from the agent.

The challenge of reaching a general game playing agent is mainly due to overfitting [4]. Most learning algorithms tend to overspecialize their solutions over the training data set. In the previous example, the agent could overspecialize on avoiding horizontal moving obstacles and fail to avoid a zigzag pattern.

A common technique used to create a game playing agent is the Neuroevolution [5], [6]. In this technique the weights and/or topology of a Neural Network are evolved in order to maximize a fitness function regarding successive applications of the Network into the problem. The evolutionary part is necessary since there exists only an indicative of the overall quality of the Neural Network, instead of a sequence of pairs of input and desired output to be learned with gradient descent methods.

The evolution of a Neural Network may also be characterized by a multimodal search space, and as such, there can be different equally good solutions that translate to different playing strategies. On one hand, we may assume that there exists one single global optimum strategy that can generalize to different patterns of the game but, it is also possible that there are multiple complementary optimal strategies that should be learned to beat the game.

As such, a Multimodal Optimization [7], [8], [9], [10], [11] algorithm could also be employed in order to evolve a set of strategies.

Multimodal optimization algorithms try to return multiple solutions for a given optimization problem that are located at different local optima. The main motivation for this comes from real world constraints that cannot be modelled, and also from the risk of rendering a single impractical local optima solution. Besides, a single solution may not be useful for every desired situation.

This paper has two goals: i) to introduce to the international community a framework based on the commercial electronic game Mega Man [12] depicting eight *fights* with different enemies, first introduced in [13], [14] and; ii) to create a baseline for the generalization of this game strategy in order to assess its difficulty and possibilities.

The paper is organized as follows: Section II and III will introduce the proposed framework. Section IV explains the basics of the Neuroevolution algorithms used during the experiments. Section V will present two experiments using the proposed framework, one of individual strategy learning and one of general strategy learning. Finally, Section VI concludes this papers with some insights for future work.

II. An Action-platformer Video Game Playing Benchmark: EvoMan

The EvoMan framework was created as a multipurpose Video Game Computational Intelligence framework [13], [14]. This framework is inspired by the boss fight stages of the game called Mega Man [12], a platformer game created in 1987 by Capcom. The game main character is a robot equipped with an arm cannon with the goal of defeating eight different robots equipped with a diverse range of weapons. Each one of the weapons assigned to each enemy have an unique pattern that should be learned by the player in order to succeed in battle. EvoMan tries to replicate these battles in order to test the creation of autonomous agents.

This framework is composed of the player agent and a set of enemy agents, with both having the possibility of being controlled by manual input or an autonomous agent. Additionaly, each enemy agent has also a standard rule-based heuristic mimicking the original game. Both the player and the enemies start with 100 points of energy that decrease whenever they are in contact with each other or are hit by the enemy projectile.

Each one of the current available stages are inspired by the boss fights from Mega Man 2, but the modularization of the framework allows to add more enemies as desired. In the beggining of each stage, both agents are placed at the opposing extremes of the game screen. The game ends when the energy of one of the players reaches a value of zero. The game is composed of discrete time steps and at every time step the player can read from 20 different available sensors:

- **Vertical and Horizontal distance:** the absolute distance between both agents in x and y-axis (total of two values).
- **Direction:** the direction each character is facing (total of two values).
- **Projectile distances:** the absolute distance on x, y-axis for each of the adversary projectiles towards the player. The game allows a maximum of eight projectiles at once (total of sixteen values).

An illustration to the available sensors is given in Fig. 1. Besides these values, the API also provides the current energy of both players, to be used by the objective-function of any optimization algorithm. After reading the values from every sensor, the agent can apply one or more of the following actions:

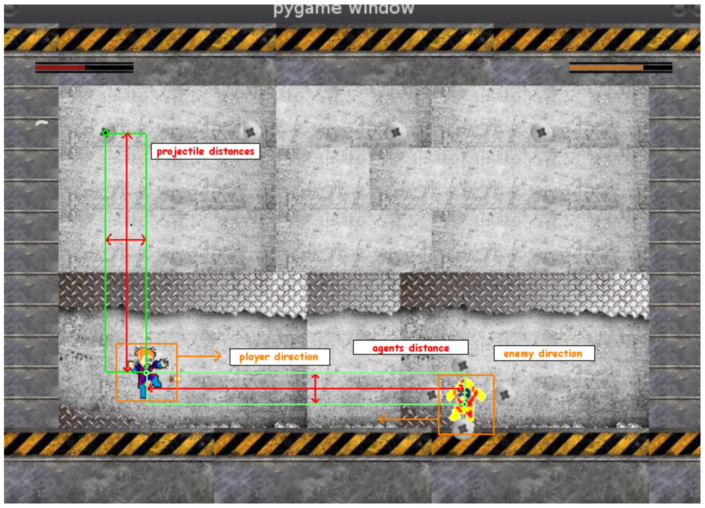

Fig. 1. The main game screen with the available sensors. The character to the left is the main player, the character to the right is the current enemy.

- **move:** the agent will move one distance unit to the right or to the left, if both actions are performed at the same time, the agent will stand still.
- **shoot:** the agent will shoot a bullet towards the direction it is currently facing.
- **jump:** the agent will jump to its maximum height except if the *interrupt* action is performed.
- **interrupt:** the jump action will be interrupted and the agent will start falling to the ground. If the jump action is not currently active, this action will have no effect.

The eight enemies exhibit different behaviors and abilities that demand different strategies from the player. They mimic the behavior of the original Mega Man 2 game as described below.

Flashman behavior follows the player throughout the stage and, from time to time, it freezes the player movements while shooting eight projectiles.

Airman stands still on one side of the screen while shooting 6 projectiles at the player from time to time. The projectiles are randomly spread accross the screen and move towards the player. After some attacks, *Airman* crosses the screen to the opposite side.

Woodman moves towards the player by small steps while attacking. His attack consists of 4 projectiles shot vertically from the top of the screen.

Heatman fires a sequence of 3 projectiles at the player. Whenever the player attacks him, it turns into a projectile and moves fast torward the player.

The *Metalman* stage has the particularity that the floor moves the player towards to or away from the enemy. The enemy shoots randomly at the player and also shoots back whenever it is attacked.

Crashman keeps jumping over the player, throwing a bomb at him from the top. If the player shoots him, it drops an extra bomb.

The *Bubbleman* stage has some spikes on the ceil and a different physics that makes the player jump higher. If the player touches the spikes it loses all of his energy at once. Meanwhile, *Bubbleman* shoots a sequence of projectiles at the player.

Finally, *Quickman* just jumps at the player and shoots projectiles at random.

Currently, this framework is written in Python 2.7 with the use of the library Pygame. The source code along with the documentation is available at https://github.com/karinemiras/evoman_framework.

III. EvoMan as a Video Game Playing Testbed

The EvoMan framework can be used as a testbed for different problems of Computational Intelligence in Games community. One example is to evolve the behavior of the opponent in order to gradually increase the difficulty [15], [16], [17] for the human player while he becomes better at the game at hand. Traditionally, the game developers artificially increase the difficult of a game by increasing the damage an enemy attack deals to the player energy. By means of a competitive coevolution, the behavior of the enemy could be changed in order to reflect a more interesting challenge to the player, with an evolving AI behavior. This procedure was initially explored in [17] for EvoMan.

Another possibility is to simply use the testbed for testing the difficulty and feasibility of a newly created enemy. By evolving the player agent with the objective to beat a single enemy [2], [18], [19], the developer can measure the overall difficulty of winning and also if the enemy is even beatable.

Finally, another possibility is to treat each enemy as a different stage of the game in order to test algorithms for the generalization of Video Game Playing strategies. In this scenario, the agent must learn a general strategy from a sample of enemies. After the learning phase, the autonomous agent should be successful against the remaining enemies.

In this work we will show and assess the difficulty of the last two mentioned testbeds applied to the EvoMan framework.

IV. Neuroevolution

Whenever the supervised learning problem at hand lacks a sequence of tuples (x, y) such as x is any given input and y is the expected correct output, the use of any gradient-based learning algorithm becomes infeasible. In such cases, the weights assigned to the connections of a Neural Network can be optimized by gradientless optimization algorithms, such as an evolutionary approach [6], by measuring the quality of the outcome after a sequence of input-output.

In Game Playing learning, for example, there is no clear indicative of the desired output y corresponding to the current input x chiefly because the output is influenced by past decisions made by the Neural Network [5], [20].

The evolution of a Neural Network may regard just the adjustment of the weights. In this case any evolutionary algorithm could be applied, or the concurrent evolution of the weights and topology (i.e., connections and activation function). In this last case the algorithm NeuroEvolution of Augmenting Topologies (NEAT) [21] is often applied.

A. *Evolving weights with Genetic Algorithm*

A simple way to perform Neuroevolution is to define a fixed topology and represent the evolved solutions as a vector of weights. By doing so, the evolution process may be performed by a simple Genetic Algorithm [22], [23].

In this work we will use a standard Genetic Algorithm with the chromossome encoded as a real-valued vector. This algorithm comprehends three main steps:

1) Given a population of solutions, generating a new offspring by applying a crossover operator on pairs of solutions.
2) Randomly mutating the offspring in order to introduce variations to the current weight values, thus promoting diversity.
3) Selecting the next generation among the current population and the offspring.

The crossover operator randomly chooses a pair of solutions from the current population and combines both solutions through an weighted average:

$$x' = \alpha \cdot x_1 + (1 - \alpha) \cdot x_2 \quad (1)$$

where x' is the offspring, x_1, x_2 are the chosen solutions and α is a random value uniformly sampled from $[0, 1]$. This operator is applied repeatedly until an offspring population of the same size of the current population is created.

The mutation operator just randomly disturbes the value of some chromossomes with a random value uniformly sampled from the problem domain.

Finally, the selection stage chooses the individuals that will be part of the next generation by using Tournament Selection of size 2. In Tournament Selection n individuals are randomly sampled and the fittest is chosen to be part of the next generation. This process is repeated until the new generation has the desired number of individuals.

B. *NeuroEvolution of Augmenting Topologies*

NeuroEvolution of Augmenting Topologies [21], or NEAT, is a metaheuristic based on Genetic Algorithms that tries to evolve the best topology and corresponding weights of a Neural Network for a given task. The evolution is performed incrementally, so each individual of the population starts off as a simple perceptron topology. At every evolution step, new neurons and connections are created by means of a mutation operator.

The building blocks of good topologies are inhereted and combined by the future generations of solutions in order to find the simplest topology capable of maximizing the fitness for the problem. Each solution may encode single or multi-layers topologies, each node of the network may have a different activation function and recurrent connections.

This algorithm encodes each solution as a list of connections, and each connection is represented by a tuple of nodes (A, B) where the information flows from node A to node B and a set of flags indicating whether this connection is active or not. Also, each connection have an unique id, called innovation number, that helps identifying corresponding connections from different chromossomes.

As many evolutionary algorithms, these evolutions rely on three main heuristics: crossover, mutation and selection. The mutation operator is responsible for the exploration process of the optimal topology, this heuristic simply adds a node and/or a new connection between two nodes. The node addition procedure actually creates two new edges that connect existing nodes through a newly created intermediate node. Additionally, the mutation procedure can disable a given gene, eliminating a node or connection from the inferred topology while not discarding the created gene.

The crossover procedure simply creates a network with the same topology of the fittest parent and the weights inherited by both parents, chosen at random. After creating the offspring, every disabled gene have a 25% chance of being reenabled. Finally, the selection heuristic is a simple tournament, but the diversity is stimulated through a process of fitness sharing.

In [24], the NEAT algorithm was used to evolve a controller for the game Frogger. In this game, it was given twelve sensors measuring the proximity from the obstacles to the player. The fitness measure was inversely proportional to the distance of the player to the goal. Also, in [25], the NEAT algorithm was successfully applied to the racing game TORCS. In this work, the Neural Network was evolved by using just the central line position, angle and speed of the car as the input. Finally, in [26], NEAT was applied to the game Ms. Pacman, with the goal of avoiding the enemies (ghosts) while *eating* the pellets for maximizing the points earned.

C. Multimodal Optimization for Neuroevolution

As mentioned in the previous section, the NEAT algorithm implements a niching technique in order to maintain the diversity of solutions and stimulate the exploratory process. However, this diversity of solutions may have another positive implication. Considering that the problem at hand is multimodal, i.e., have multiple equally good solutions, a set of distinct Neural Networks capable of achieving the stipulated goal may lead to different levels of generalization.

For example, supposing that a multimodal algorithm finds three different sets of weights for the Neural Network that are capable of defeating the current enemy. These may translate into three different strategies, each strategy, though currently reaching the main goal, may be used afterwards as alternative strategies that may generalize different sets of enemies.

One algorithm capable of finding multiple optima, named *LinkedOpt* [11] was introduced in 2015 with the idea of optimizing a diversity measure instead of seeking the global optimum directly. The reported results are competitive with the current state-of-the-art and the source code is available at https://github.com/folivetti/LINKEDOPT.

The main idea of the *LinkedOpt* algorithm is described as a pseudo-algorithm in Alg. 1 and further detailed afterwards.

The function *SelectNodes()* will sample $nnodes$ nodes from the network at random but with probability inversely proportional to their degree. This probability ensures that the nodes located at unexplored areas (the leaves of a tree-like structure) will have preference to be expanded, while those at already explored areas are less likely to continue its expansion.

```
input : max iterations $maxIT$, number of expanded
        nodes per iteration $nnodes$, radius of similarity
        $thrE$, $thrL$, lower and upper bound of the
        objective-function $lb$, $ub$
output: network of solutions $G$
$x_0 \leftarrow lb + 0.5 \cdot (ub - lb)$;
$InsertNode(G, x_0)$;
for $it \leftarrow 1$ to $maxIT$ do
    $nodes \leftarrow SelectNodes(G, nnodes)$;
    $G \leftarrow ExpandNodes(G, nodes, thrE)$;
    $G \leftarrow Suppress(G, thrE, thrL)$;
end
```
Algorithm 1: LinkedOpt meta-heuristic

After the node selection, each selected node is expanded by sampling an unity vector d representing a random direction. Then, the function $F(\alpha) = LD(x, x + \alpha \cdot d)$ is maximized.

After the optimization of each segment, some decisions are performed to whether to replace the current node or to connect it to the new solution. Given an optimal α of one segment, we define $x^* = x + \alpha \cdot d$ and $x_m = 0.5 \cdot (x - x^*)$, also, y, y^*, y_m will refer to the objective-function values of the corresponding points.

If $y_m > y, y^*$ or $y^* > y, y_m$, then x cannot possibly be a global maximum and then its corresponding node is replaced by x_m or x^* accordingly. If $y_m < y, y^*$, then it is assumed that x_m is a local minima, and x and x^* are located at the basis of attraction of different optima. In this situation, x and x^* are both kept in the population and x^* is linked to the closest node inside the network.

V. Experimental Results

As mentioned in previous sections, in this paper we will test the EvoMan framework in two different scenarios. The first one will assess the capability of the neuroevolution algorithms to find an specific winning strategy for every single enemy of this framework. The second scenario will verify whether the algorithms can evolve a generalized strategy for different enemies.

Since the Genetic Algorithm (GA) and LinkedOpt (LO) algorithms will evolve just the weights of the Neural Network, we have tested the neuroevolution with these two algorithms with the Perceptron and one-hidden layer with 10 and 50 neurons topologies. The NEAT algorithm evolves the topology together with the weights, and as such just a single version was used for the experiments.

The inputs for every Neural Network were normalized within the range $[-1, 1]$ and the fitness function was defined as:

$$f = 0.9 \cdot (100 - e) + 0.1 \cdot p - \log t \qquad (2)$$

where p, e are the current energy level of the player and the enemy, respectively, and t is the total number of timesteps used to end the fight.

TABLE I. PARAMETERS USED FOR GENETIC ALGORITHM DURING THE EXPERIMENTS.

Parameter	Value
Population Size	100
Crossover Prob.	100%
Mutation Prob.	20%

TABLE II. PARAMETERS USED FOR LINKEDOPT ALGORITHM DURING THE EXPERIMENTS.

Parameter	Value
step size	1
thr	0.5
thrL	0.5

This fitness function aims at maximizing the energy of the player by the end of the level, while minimizing the energy of the enemy. The last term penalizes fights that take too long to be finished.

The parameters used for each algorithm were fine-tuned by evaluating the average fitness during the first 10 iterations for 30 repetitions. The parameters chosen for each algorithm are described in Table I for the Genetic Algorithm, Table II for the LinkedOpt and Table III for the NEAT algorithm.

For Genetic Algorithm and LinkedOpt, the activation functions used for the hidden and output layers was the Sigmoid function. The output of each Neural Network was composed of 5 values, each one corresponding to a possible action. The value ranged in the interval $[0, 1]$ and a value equal or higher than 0.5 means that the corresponding action will be performed by the agent.

Each algorithm evolved their solutions throughout 100 generations and these experiments were repeated 30 times. The results will be reported by means of the average of the obtained values.

A. Individual Evolutions

In Table IV the final energy of each agent after the end of the fight is reported. A value of 0 means that the agent has lost the fight. In this Table we can see that every enemy in this framework is beatable by any algorithm given the correct network topology. NEAT algorithm had a much better performance when compared to the other two algorithms, being surpassed only in two occasions by the LinkedOpt algorithm.

The strategy adopted by the agents for the first three enemies (*FlashMan*, *AirMan* and *Woodman*) consists of just jumping at the same place while shooting, and this simple strategy suffices to win these matches. The *AirMan* is particularly easy to beat with such strategy, since this enemy stays static during most of the battle. Ideally, for *Woodman*, the agent would move at small steps increment, forward or backward, for a perfect win.

TABLE III. PARAMETERS USED FOR NEAT ALGORITHM DURING THE EXPERIMENTS.

Parameter	Value
Population Size	100
Compatibility threshold	3
Excess coefficient	1
Disjoints coefficient	1
Weight Coefficient	0.4
Link insertion Prob.	5%
Node insertion Prob.	3%
Weight Mutation Prob.	90%

TABLE IV. FINAL SCORES FOR THE SINGLE ENEMY CHALLENGE IN EVOMAN. THE BEST RESULTS ARE MARKED IN BOLD.

Enemy	NEAT	GA	GA10	GA50	LO	LO10	LO50
FlashMan	90	90	90	90	0	**96**	0
AirMan	**94**	90	82	78	90	82	88
WoodMan	**80**	58	58	36	24	0	16
HeatMan	**94**	4	30	69	11	0	19
MetalMan	**94**	80	88	79	78	81	88
CrashMan	**73**	0	0	10	39	34	0
BubbleMan	78	68	56	18	86	69	**90**
QuickMan	**86**	77	83	78	0	82	83

Regarding *Heatman*, the agent should jump over the shots and the enemy itself, so a general strategy found by the learning algorithms was to jump forward by small steps while shooting against the enemy. This one can be particularly challenging since it requires precise jumps.

Metalman, despite having a moving scenario, can be beaten by the simple strategy of jumping and shooting, like the first three enemies.

The sixth enemy, *Crashman*, is particularly challenging because of his behavior of attacking from above the agent. The agent must evolve an strategy of carefuly avoiding the enemy and the attacks while shooting at the very few opportunities that arise.

BubbleMan requires a special care while jumping because of the spikes on the top of the stage and the changed physics that makes the jumps higher than normal. For this enemy the learning algorithms must evolve the behavior of releasing the jump button at the right moment. As it can be seen from the results, this is not an easy behavior to achieve.

Finally, *QuickMan* requires a similar strategy to *HeatMan*, so the agent must learn to jump towards the enemy while shooting. Even if the agent does not try to avoid the projectiles, it can still win the battle.

B. General Strategies

For the next set of experiments we will perform the training stage of the Neural Network through Neuroevolution by using sets of two and three enemies (denoted *training enemies*). In order to do so, we will calculate a new fitness function f' that is equal to the average of the obtained fitness for each one of the training enemies.

After the evolution process, the evolved agents were tested on the remaining enemies (hereby *validation enemies*) with the objective of maximizing victories.

Due to the computationaly expensive nature of the training process, we first performed a simple experiment with every possible pair of enemies using NEAT. From these experiments, the best pairs were chosen to be tested with the GA and LO algorithms.

In Table V we can see the obtained results by means of number of victories against the training enemies, the number of victories against the validation enemies and the total gain, calculated as:

$$g = \sum_{i=1}^{n} (p_i - e_i), \tag{3}$$

where n is the number of enemies (of the training and validation sets), p_i is the remaining energy for the player on scenario i and e_i is the remaining energy for the enemy on scenario i. The higher the gain, the more general the agent is.

From this Table we can see that the best pairs to use as a training set were $(2, 4), (2, 6), (7, 8)$ while the pair $(1, 5)$ obtained the highest fitness. For the triples of enemies, we have chosen to test against $(1, 2, 5)$, because the enemy 2 was a recurrent good enemy from the pairs and $(1, 5)$ was the pair with the highest fitness. The triple $(2, 5, 6)$ that corresponds to the best pair and one of the enemies from the best fitness, and $(1, 5, 6)$ that combines one enemy from the best pair and both enemies from the pair with the best fitness.

TABLE V. TEST OF DIFFERENT COMBINATIONS OF TRAINING SET WITH NEAT ALGORITHM. THE COLUMNS LABELED *1st* AND *2nd* CORRESPOND TO THE TWO ENEMIES USED DURING THE TRAINING STAGE. THE COLUMN LABELED *fitness* CORRESPONDS TO THE FINAL FITNESS AS THE AVERAGE FITNESS OF THE AGENT WITH RESPECT TO THE TRAINING ENEMIES. THE *training* AND *validation* COLUMNS ARE THE NUMBER OF VICTORIES OBTAINED ON THE TRAINING AND VALIDATIONS SETS, RESPECTIVELY. FINALLY, THE *Gain* COLUMN REFERS TO THE TOTAL GAIN AS DESCRIBED IN EQ. 3.

1st	2nd	Fitness	Training	Validation	Gain
1	2	66	1	0	-469
1	3	66	0	2	-250
1	4	67	1	1	-269
1	**5**	**94**	2	0	-275
1	6	66	1	1	-86
1	7	87	2	1	-347
1	8	58	0	1	-285
2	3	91	2	1	-88
2	**4**	90	**2**	**2**	**-76**
2	5	91	2	0	-110
2	**6**	92	**2**	**2**	**-65**
2	7	92	2	0	-71
2	8	92	2	0	-194
3	4	75	1	0	-362
3	5	89	2	0	-410
3	6	87	2	1	-313
3	7	85	2	1	-290
3	8	90	0	3	-178
4	5	90	2	1	-164
4	6	84	2	0	-347
4	7	89	2	0	-325
4	8	87	2	0	-197
5	6	90	2	0	-243
5	7	89	2	0	-240
5	8	87	2	0	-279
6	7	89	2	0	-181
6	8	87	2	1	-167
7	**8**	88	**2**	**2**	**-143**

The experiments were performed for 30 repetitions and the average gain, alongside with the standard deviation are reported for each algorithm. Also, the configuration of victories against the training and validation sets are graphically reported. The strategy adopted for each agent was the final best solution of each run with the exception of LinkedOpt algorithm, in which we report the combination of the results from the two most different strategies with a difference in fitness of at most 5 points.

First, in Table VI, the average gain achieved by each strategy is reported. From this Table we can see that NEAT performed better on three of the seven tests, GAP obtained the best result in one of the tests, GA10 was best in two different sets and LOP achieved the best result in one of the tests. The best gain was obtained by GA10 on the training set $(7, 8)$. Curiously, GA underperformed on these two enemies on the individual tests when compared to NEAT and LO, this may indicate that a harder training scenario may lead to a better generalization for the validation set. The next best result, and the only other positive gain, was for the triple $(1, 5, 6)$ with NEAT algorithm.

In Table VII we can see the victories obtained by each evolved agent for the pairs of training enemies. With this table it is easy to see that the enemies 2, 5 and 8 are frequently beaten even when they are not part of the training set. On the other hand, the enemies 4, 6 and 7 were only frequently beaten whenever they are part of the training set. This means that the set $(2, 5, 8)$ is robust to different strategies while the set $(4, 6, 7)$ requires specific variations of a general strategy, thus implying they are more challenging enemies. Finally, the enemies 1 and 3 were never beaten by any of the evolved strategies. The best achieved result for this set was obtained by GAP with the training set $(7, 8)$ with a total of five victories.

For the training set composed of triples of enemies, as depicted in Table VIII, we can see that they generated a tendency of evolving an specific strategy to the training set, the exception being the training set $(1, 5, 6)$ which also generalized for the enemy 2. The best obtained result for this set was with LO10 with the training set $(1, 2, 5)$ which achieved a generalization strategy for five enemies.

Comparing both results, we can see that GAP with training set $(7, 8)$ and LO10 with training set $(1, 2, 5)$ were the ones that maximized the generalization, obtaining the total of five victories. We should notice that GAP obtained three victories on the validation set, while LO10 could achieve just two victories. Also, LO10 result is a combination of two solutions obtained by the algorithm, while GAP used only its best strategy. On the other hand, the scores obtained by LO10 victories were higher than those obtained by GAP. Notice also that GAP cannot always beat the sixth enemy leading to a large variation and lower gain.

In order to visualize the quality of the generated solutions, the execution of the best result obtained by NEAT, Genetic Algorithm and LinkedOpt against each enemy was recorded and hosted in https://youtu.be/w97qitczuL8, https://youtu.be/vQ_xCShU1gQ, and https://youtu.be/ic68VCjMIMI, respectively.

From these videos we can perceive some of the behaviors evolved by each algorithm. The NEAT algorithm, for example, seems to have evolved a more neutral behavior in which the agent tries to avoid getting hit by the enemy and its projectiles while shooting at the enemy whenever it had an opportunity. This tactic worked well on average but it was weak against opponents that pursued the player and, also, against the changed physics of *BubbleMan*.

The general strategy of the GAP algorithm, on the other hand, focused solely on the attack. The agent just jumped forward towards the enemy while continuously shooting. Though this strategy could win against more enemies than the strategy

TABLE VI. Overall gains obtained by NEAT, GA and LO for each of the training sets tested. The reported results are the average and standard deviation after 30 repetitions of each experiment.

Training	NEAT	GAP	GA10	GA50	LOP	LO10	LO50
(2, 4)	$\mathbf{-73 \pm 9}$	-259 ± 32	-76 ± 8	-77 ± 12	-85 ± 8	-268 ± 18	-124 ± 33
(2, 6)	$\mathbf{-52 \pm 14}$	-382 ± 31	-283 ± 9	-256 ± 8	-175 ± 4	-96 ± 11	-239 ± 17
(7, 8)	-166 ± 18	-58 ± 37	$\mathbf{40 \pm 11}$	-225 ± 4	-126 ± 4	-4 ± 4	-253 ± 9
(2, 5, 1)	-143 ± 5	$\mathbf{-66 \pm 10}$	-233 ± 5	-144 ± 7	-144 ± 21	-71 ± 10	-285 ± 20
(2, 5, 6)	-83 ± 16	-272 ± 9	$\mathbf{-43 \pm 7}$	-157 ± 20	-189 ± 8	-141 ± 14	-120 ± 28
(1, 5, 6)	$\mathbf{33 \pm 9}$	-129 ± 15	-201 ± 20	-131 ± 12	-130 ± 12	-185 ± 15	-336 ± 22
(4, 6, 7)	-169 ± 29	-269 ± 7	-142 ± 22	-232 ± 24	$\mathbf{-38 \pm 34}$	-63 ± 15	-257 ± 5

TABLE VII. Victories scores for the generalization tests with pairs of enemies serving as the training set. Each square corresponds to one of the enemies, in order, and the values represent the final energy of the player. The green mark highlights whenever the remaining energy was above the 50 mark, the yellow mark shows whenever the remaining energy was below that mark and the absence of values means that the player lost that level.

Training	(2,4)							
NEAT		70		66	30			18
GAP		70		29	34			
GA10		75			45			19
GA50		76			54			19
LOP		60		35	41			5
LO10		24			35			21
LO50		72		23	65			26

Training	(2,6)							
NEAT		74			43	88		4
GAP		26						
GA10		13			42	13		
GA50					34			21
LOP		45			51	10	5	
LO10		56			50	2		29
LO50		47			18			

Training	(7,8)							
NEAT		67			6		42	60
GAP		15			45	2	53	56
GA10		74		43	60			30
GA50		8					19	45
LOP		24			52		70	18
LO10		77			41		68	50
LO50				25	61			26

TABLE VIII. Victories scores for the generalization tests with triples of enemies serving as the training set. Each square correspond to one of the enemies, in order, and the values represent the final energy of the player. The green mark highlights whenever the remaining energy was above the 50 mark, the yellow mark shows whenever the remaining energy was below that mark and the absence of values means that the player lost that level.

Training	(1,2,5)							
NEAT	96	76			55			
GAP	96	74			74			
GA10		74			80			
GA50		74			54			4
LOP		48			57			11
LO10	96	74	11		71			44
LO50		31			14			44

Training	(2,5,6)							
NEAT		52			58	58		30
GAP		78			77			
GA10		89			50			35
GA50		76			29			21
LOP		48			10			27
LO10		76			32			21
LO50		10			49			54

Training	(1,5,6)							
NEAT	96				71	63		
GAP		74			51	2		
GA10		8				1	1	9
GA50		74			48			
LOP		7			47	1		52
LO10		8			1			44
LO50					72			46

adopted by NEAT, it can be seen that the agent barely won most of the battles.

Finally, the LO10 algorithm adopted a more defensive strategy in which the agent just stands still at the same position using only vertical jumps to avoid the projectiles while continuously shooting towards the enemy. This strategy also coped well on average but, similarly to the NEAT algorithm, it presented a weaker performance against enemies that pursued the player.

It is curious to notice that the three algorithms adopted different strategies to win the game. This is another evidence to the hypothesis that there are different possible winning strategies characterizing a multimodal search space.

VI. Conclusion

This paper experiments with an Action-Platformer Video Game Playing Framework, called EvoMan, by first trying to evolve individual agents each one capable of defeating one of the enemies in the framework. Afterwards, we have tried to evolve a general agent capable of beating the complete set of enemies while learning by playing against a subset of the enemies set.

The framework is initially composed of a set of eight enemies with different pre-programmed behaviors. Additionally, the main player and the enemies can be controlled by manual input (i.e., human player) or by an autonomous agent. The autonomous agents can read a set of 20 sensors in order to infer a decision for each action it will perform. The actions can be one or more of the set: jump, go right, go left, shoot, release jump.

In order to evaluate the framework, two different experiments were tested: individual strategy learning, in which the agent learns how to beat each enemy individually; and general strategy learning, in which the agent tries to find a general strategy to beat every enemy, by learning from a set of two or three enemies.

The experiments showed that every enemy in the framework is beatable by simple neuroevolution agents. On the other hand, a general strategy proves to be a challenge and not yet achieved by the tested algorithms. The best general strategies were obtained by Genetic Algorithm and LinkedOpt algorithms, with both beating a total of five enemies.

For the next steps, we will test different topologies of Neural Network, like the Recurrent Neural Networks [27] and different learning strategies like Incremental Evolution [28], [29]. We will also verify the possibility of connecting the different strategies through a Neural Network that will choose which strategy to apply for each enemy, thus exploring effectively the multimodality of the search space.

Regarding the framework we will explore the Co-Evolutionary module further in order to devise a learning algorithm capable of gradually raising the difficulty of the game, while maximizing the fun for the player.

References

[1] S. M. Lucas, "Computational intelligence and games: Challenges and opportunities," *International Journal of Automation and Computing*, vol. 5, no. 1, pp. 45–57, 2008.

[2] S. M. Lucas and G. Kendall, "Evolutionary computation and games," *Computational Intelligence Magazine, IEEE*, vol. 1, no. 1, pp. 10–18, 2006.

[3] G. N. Yannakakis, "Game ai revisited," in *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, pp. 285–292.

[4] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[5] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the nero video game," *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 653–668, 2005.

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

[7] K.-C. Wong, "Evolutionary multimodal optimization: A short survey," *arXiv preprint arXiv:1508.00457*, 2015.

[8] M. Preuss, "Multimodal optimization," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15. New York, NY, USA: ACM, 2015, pp. 293–312. [Online]. Available: http://doi.acm.org/10.1145/2739482.2756572

[9] F. O. de Franca, F. J. Von Zuben, and L. N. de Castro, "An artificial immune network for multimodal function optimization on dynamic environments," in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 2005, pp. 289–296.

[10] F. O. de França, G. P. Coelho, P. A. Castro, and F. J. Von Zuben, "Conceptual and practical aspects of the ainet family of algorithms," 2012.

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

[12] M. MEGA, "Produced by capcom, distributed by capcom, 1987," *System: NES*.

[13] K. S. M. ARAUJO and F. O. DE FRANCA, "Um ambiente de jogo eletrnico para avaliar algoritmos co-evolutivos," in *Proceedings do Congresso Brasileiro de Inteligłncia Computacional*. IEEE, to appear.

[14] K. da Silva Miras de Araújo and F. Olivetti de França, "An electronic-game framework for evaluating coevolutionary algorithms," *ArXiv e-prints*, Apr. 2016.

[15] P. Demasi and J. d. O. Adriano, "On-line coevolution for action games," *International Journal of Intelligent Games & Simulation*, vol. 2, no. 2, 2003.

[16] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma, "Difficulty scaling of game ai," in *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004)*, 2004, pp. 33–37.

[17] K. S. M. de Araujo and F. O. de Franca, "Um ambiente de jogo eletrnico para avaliar algoritmos coevolutivos," in *Proceedings of the 12th Congresso Brasileiro de Inteligłncia Computacional*, 2015.

[18] D. Perez, M. Nicolau, M. ONeill, and A. Brabazon, "Evolving behaviour trees for the mario ai competition using grammatical evolution," in *Applications of Evolutionary Computation*. Springer, 2011, pp. 123–132.

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

[20] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, no. 4, pp. 355–366, 2014.

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

[22] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[23] D. E. Goldberg, *Genetic algorithms*. Pearson Education India, 2006.

[24] D. Ancona and J. Weiner, "Developing frogger player intelligence using neat and a score driven fitness function."

[25] L. Cardamone, D. Loiacono, and P. L. Lanzi, "On-line neuroevolution applied to the open racing car simulator," in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE, 2009, pp. 2622–2629.

[26] M. Wittkamp, L. Barone, and P. Hingston, "Using neat for continuous adaptation and teamwork formation in pacman," in *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*. IEEE, 2008, pp. 234–242.

[27] H. Jaeger, *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. GMD-Forschungszentrum Informationstechnik, 2002.

[28] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super mario evolution," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009, pp. 156–161.

[29] J. Togelius, S. M. Lucas, and R. De Nardi, "Computational intelligence in racing games," in *Advanced Intelligent Paradigms in Computer Games*. Springer, 2007, pp. 39–69.