Multi Populational NeuroEvolution of Augmenting Topologies for the EvoMan Framework

Augusto Dantas *, Aurora Pozo †

Department of Computer Science
Federal University of Paraná
Curitiba, Brasil
Email: *aldantas@inf.ufpr.br, †aurora@inf.ufpr.br

***Abstract*—Artificial Intelligence and Games is a growing research area that had some remarkable breakthroughs in the past few years. Electronic video games, in particular, offer a great testbed for AI researchers and practitioners due to their challenging nature. Recently, some works have highlighted the possibility to use evolutionary algorithms for learning game playing agents, instead of the more widely used reinforcement learning methods. This work investigates a cooperative coevolutionary strategy for learning a generalized agent to play a platform level game called EvoMan. In this game, a player is faced against one enemy at each level, totalling at eight levels and enemies. The goal is to learn a strategy able to beat all enemies, but using only four of them during the evolutionary process. We used the NeuroEvolution of Augmenting Topologies algorithm to evolve player agents against a set of enemies in a novel multi-populational scheme. Our agent was able to beat 6 enemies out of 8, including the four enemies used for training and two other unseen enemies.**

***Index Terms*—artificial intelligence and games, neuroevolution, cooperative coevolution**

I. Introduction

This is a submission to the EvoMan Competition [1]. The goal is to achieve a generalized playing agent to the EvoMan framework [1], a platform level game based on the famous MegaMan game. In EvoMan, there are eight possible enemies, each one with complete different behaviours, that the player must defeat. Therefore, a single agent should ideally generalize and be able to beat all the eights enemies, which is not a trivial task [2]. The rewards in EvoMan are episodic, which means that the feedback is only given at the end of a complete run of a level. Therefore, it is mainly intended as a testbed for evolutionary strategies.

The generalization to different enemies can be seen as a multi-objeticve task. One way of dealing with multi-objective problems is to decompose it into different mono objective problems and solve them under a coevolutionary strategy, with multiple populations sharing their information [3].

This work investigates the use of the NeuroEvolution of Augmenting Topology (NEAT) algorithm [4] in a multi-populational scheme (MultiNEAT) for learning a general agent for the EvoMan game. The wide success of NEAT in the literature is due to some simple, yet important, designs aspects that it introduces. First, the algorithm starts with a population of minimal networks that increase in complexity over time trough mutation. Also, it protects innovative nodes and connections by speciation and fitness sharing. Finally, it handles the problem of matching different topologies by marking the genes with unique keys that are used to easily match genomes during reproduction. NEAT, and its variations, has been already successfully applied to games such as Ms. Pacman [5] and some classic Atari games [6].

In MultiNEAT, each subpopulation have a NEAT instance that evolves individuals for performing one task (enemy). Then, the best genomes of each population are passed to a master population that evaluates the individuals for all the training tasks (enemies). The idea is to allow the networks to evolve in a simpler environment and make them compete in a harsher condition afterwards. The best genomes from the master population are then introduced back into the single enemy populations for the next iteration. These genomes can be used during reproduction in the subpopulations.

The paper is organized in the following way: Sections II and III respectively detail the original NEAT algorithm and our MultiNEAT approach. The experimental setup is described in Section IV, followed by the results in Section V. At last, conclusions and future works are drawn in Section VI.

II. NEAT

A key aspect of the NeuroEvolution of Augmenting Topologies algorithm is that the initial population is composed of minimal structured networks. Then, these networks becomes more complex over the generations through mutation and crossover. The advantage is that the solutions can become more tailored to the problem while reducing the search space [4]. There are several implementations of NEAT available. In this work, we used a python implementation called `NEAT-Python` [2].

In NEAT, the genomes (individuals) are encoded by a set of node genes and a set of connection genes. Each gene have a key identifier that allows the algorithm to easily crossover or compute the distance between two genomes. In this implementation, the node genes have four attributes: the values for bias

[1] http://pesquisa.ufabc.edu.br/hal/Evoman.html

[2] https://neat-python.readthedocs.io

and response, and the activation and aggregation functions. These attributes determine the output of a node according to:

$$\textit{output} = \texttt{activation}(\textit{bias} + (\textit{response} * \texttt{aggregation}(\textit{inputs}))) \quad (1)$$

Meanwhile, a connection gene contains the weight value and a flag indicating if it is enabled or disabled. The key of the connection gene is a tuple with the keys of both nodes it is connecting.

At each generation, new genomes are created through reproduction and mutation. The $s\%$ best genomes of each species (with s being a configurable parameter) are considered for reproduction pool. For each new genome, two parents are chosen at random from this survivors pool for mating. Then, the genes from both parents that have identical keys are passed to the child (the gene values are inherited from either parent at random). The remaining genes are called the disjoint genes, and only the fittest parent is allowed to pass them to the offspring.

Next, the new genome goes through mutation, which can be random perturbations on gene values (weight in connection genes, bias and response in node genes), or structural mutation, that adds new nodes and/or connections. An add connection mutation simply connects two nodes with a random weight. An add node mutation inserts a new node in the place of an existing connection, then create two new connections to link the previous and next nodes that were connected before. After this, all offspring replace the genomes from the population for the next generation (except the best genomes if elitism is enabled).

A common problem when evolving topologies is that the addition of new structures (a node or connection) usually leads to an immediate decrease in performance of the network [4]. Hence, in order to protect innovative mutations, NEAT uses a speciation scheme, in which the genomes are divided into species based on topological similarity. This is done by calculating a distance measure between a genome and a randomly chosen member of each species. If this distance is less than a pre-defined threshold, the genome is placed into the respective species. If no compatible species is found, then a new species is created for placing that individual. The distance between two genomes is given by the amount of disjoint genes (genes with unmatched keys), multiplied by a coefficient parameter, plus the sum of the absolute differences between the values of the paired genes, also multiplied by its coefficient parameter.

III. MULTINEAT

MultiNEAT uses several NEAT instances in order to achieve a generalized solution. Therefore, instead of only evolving solution on a multi-objective task, it also creates subpopulations to handle the tasks individually. The principle of this is to alternate the evolution between simpler and more complex environments, so the individuals can improve for specific objectives individually before competing in the whole set of objectives.

Algorithm 1 shows the workflow of our MultiNEAT approach. For a problem with n objective task, $n + 1$ NEAT populations are created: one master population that is evolved against all objectives and n subpopulations with one objective assigned to each of them (lines 1-5). Then, for max_iter iterations, each subpopulation is evolved for max_gen generations , and the n_best solutions at the last generation are selected to compose the migrants set (lines 11-12). This migrants set is used to replace the worst solutions in the current master population (line 14). Additionally, the `replaceWorsts` function also evaluates the migrants in the new environment (in this case, a multitask one) and calls the NEAT speciation function to update the species scheme. Finally, the master population is then evolved by NEAT for max_gen_master generations, and the n_best_master from the last generation are chosen as the elite set (lines 15-16).

Algorithm 1: MultiNEAT

Input: A set $\mathsf{T} = \{t_1, t_2, \ldots, t_n\}$ of n objective tasks
Output: The winner solution of each population
1 master_population ← `initializePopulation`(T)
2 populations ← []
3 **foreach** *task* $t_i \in \mathsf{T}$ **do**
4 | populations[i] ← `initializePopulation`(t_i)
5 **end**
6 master_elite ← []
7 **for** $i \leftarrow 1$ **to** *max_iter* **do**
8 | migrants ← []
9 | **foreach** *population* $p_i \in$ populations **do**
10 | | `setArchive`(p_i, master_elite)
11 | | `NEAT`(p_i, *max_gen*)
12 | | migrants ← migrants ∪ `selectBests`(p_i, *n_best*)
13 | **end**
14 | `replaceWorsts`(master_population, migrants)
15 | `NEAT`(master_population, *max_gen_master*)
16 | master_elite ← `selectBests`(master_population, *n_best_master*)
17 **end**

In the next iteration, the best solutions from the master population are introduced to the subpopulations in the form of an archive. In MultiNEAT, the archive is a fixed set of solutions across generations that can be used for reproduction. The `setArchive` method (line 10) defines the external solutions as the archive and reevaluates them according to the respective population objective function.

During the reproduction in the subpopulations, the first parent is randomly selected from the species pool (as in standard NEAT). However, the second parent is chosen with a two tournament selection between a random solution from the species pool and a random solution from the archive. In this way, the best generalized genomes may guide the search on the single task environments. Therefore, MultiNEAT performs cycles where the subpopulations feed their information to the master population and vice-versa.

In the context of the EvoMan framework, the fitness of each subpopulation is the outcome of the play against one enemy. Meanwhile, the fitness in the master population uses multiple mode for playing against the whole set of training enemies. The fitness in the multiple mode is the average fitness of each enemy minus the standard deviation.

IV. Experimental Setup

For each experiment, we executed 10 independent runs with the same set of 10 distinct seeds. We used the GNU Parallel [7] tool to manage the execution of the trials on a Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz with 6 cores (2 threads each). The experiment lasted 20 hours on average.

Table I displays the main NEAT parameter values that we used for all populations. The values for the compatibility threshold, the distance coefficients, and the weight mutation probability are the same used by the framework authors in their previous study [2]. In that work, they evaluated NEAT (and a few other evolutionary algorithms) on three triples of enemies as the training sets, which was done based on the observed performance of NEAT on different pairs of enemies.

TABLE I: NEAT main parameters

Parameter	Value
Population sizes	50
Species compatibility threshold	3
Weight distance coefficient	0.4
Disjoint distance coefficient	1
Add node probability	20%
Add connection probability	50%
Weight mutation probability	90%
Survival threshold	20%
Elitism inside species	1

However, for the add node mutation and add connection mutation probabilities, we choose higher values from the original authors, which were 3% and 5% respectively. We made this decision because our approach is based on alternating runs of NEAT with few generations, as described in Section III. Therefore, we wanted to assure that, during these runs, the populations could be able to have more structural changes.

Additionally, there are some other NEAT parameters that were not reported in [2], such as the elitism inside a species and the survival threshold (the proportion of the best species individuals able to reproduce). The initial solutions are feedforward networks in which all inputs are connected to all outputs with random weights and no hidden layer. The `NEAT-Python` implementation also allows to evolve the activation and aggregation functions from 1, however, we fixed them to the sigmoid and sum functions, respectively. The whole parameter configuration file is available together with the source code.

Moreover, the MultiNEAT have some parameters of its own. Besides the number of iterations, we must define the amount of generations for both the subpopulations and the master populations per iteration. Other parameters are the number of migrants the subpopulations will send to the master and how many solutions from the master population will be introduced in the subpopulations. Table II shows the values for these MultiNEAT parameters. Notice that we decided to set the same generation number for all populations, but this relationship between sub and master populations evolutions should be further investigated.

The choice of 10 for the migrants from each subpopulation parameter is to allow the master population the keep its best solutions from the previous iteration, and make them compete with the best solutions from each mono task NEAT.

TABLE II: MultiNEAT parameters

Parameter	Value
Iterations	20
Subpopulation generations	10
Master generations	10
Solutions from subpopulations	10
Solutions from master	10

TABLE III: Times an agent beats the enemy

	Test enemy							
Training Set	1	2	3	4	5	6	7	8
1,3,4,7	3	0	5	4	10	0	6	3
1,4,6,7	2	5	0	7	5	6	2	5
2,4,6,7	0	10	0	7	5	8	3	6

During preliminar experiments, we noticed that enemies 5 and 8 were beaten more often, indicating that they are easier than the others, so we did non include them for training. Then, we experimented with a few variations of four enemies with the remaining six. The best results were achieved with the set of enemies $\{2,4,6,7\}$, hence, our final agent was evolved using this set.

A. Player controller

The player controller is a feedforward neural network generated by NEAT, with 20 inputs and 5 outputs. The inputs are normalized in the range $[-1, 1]$, except the two input sensors that indicate the direction the characters are facing, because they already are either 1 or -1. The output of the network are five values in the range $[0, 1]$, where each value corresponds to one of the five possible actions. The respective action is performed if the value is higher than 0.5.

V. Results

Table III shows hoy many agents trained with a given enemy set was able to beat the enemies. In total, there are 10 agents for each training set. The set $\{2,4,6,7\}$ had the most wins, although it was never able to beat enemies 1 and 3. The next results we show are related to this training set.

In Table IV we can observe the performance of the agents trained with the set $\{2,4,6,7\}$, each with a different initial random seed. The reported values are the player energy minus the enemy energy, hence, a value higher than zero means that the agent have won the match. The enemy 2 was always beaten, meanwhile, the enemy 7, that is present in the training, was beaten only three times and with low scores. Interestingly, the best score was achieved against enemy 8, which was not present in the training set.

Nevertheless, the most successful agent (ID 8 in Table IV) was able to beat 6 out of 8 enemies. This is the agent that we submitted to the competition. Table V reports the required performance metrics: the player energy, the enemy energy and the time of the match.

TABLE IV: Performance of the 10 agents trained with the set {2,4,6,7}

	Test enemy							
Agent ID	1	2	3	4	5	6	7	8
1	-70.00	56.00	-60.00	48.40	-50.00	-10.00	-10.00	7.00
2	-80.00	24.00	-60.00	-10.00	22.60	-10.00	27.40	44.20
3	-70.00	74.00	-20.00	42.40	55.60	15.40	-10.00	62.80
4	-70.00	38.00	-60.00	0.40	-70.00	29.20	-30.00	-20.00
5	-70.00	4.00	-80.00	-10.00	-80.00	19.60	-10.00	-50.00
6	-80.00	28.00	-60.00	17.80	61.00	1.60	-10.00	-70.00
7	-70.00	52.00	-70.00	36.40	51.40	26.20	-10.00	61.60
8	-80.00	16.00	-80.00	37.00	38.80	22.60	35.80	79.60
9	-100.00	56.00	-80.00	34.00	35.20	5.80	-20.00	-60.00
10	-80.00	48.00	-60.00	-10.00	20.20	14.80	19.00	19.00

TABLE V: Final agent performance metrics

Enemy	Player energy	Enemy energy	Match time
1	0	80	294
2	16	0	242
3	0	80	206
4	37	0	755
5	38.8	0	481
6	22.6	0	327
7	35.8	0	460
8	79.6	0	262

VI. Conclusion

This work investigated a multi populational NEAT strategy for cooperative coevolution of game playing agents. The goal was to achieve a generalized agent able to beat the set of eight enemies, but without seeing four of them during the evolutionary process. In the present state, our approach was able the learn a strategy that beats 6 out of 8 enemies. Besides the low achieved scores, the agent performed well against one enemy that it did not see during training. This indicates the generalization potential of the approach and that further investigation must be done to better identify its properties.

Future works include to perform a thorough analysis on the influence of the archive solutions into the subpopulations and to monitor the fitness evolution across populations. Additionally, a multi-objective evolutionary algorithm could be applied on the master population, then guiding the search also on the diversity of strategies.

References

[1] K. d. S. M. de Araújo and F. O. de França, "An electronic-game framework for evaluating coevolutionary algorithms," *arXiv:1604.00644 [cs]*, Apr. 2016, arXiv: 1604.00644. [Online]. Available: http://arxiv.org/abs/1604.00644

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

[3] L. Miguel Antonio and C. A. Coello Coello, "Coevolutionary Multiobjective Evolutionary Algorithms: Survey of the State-of-the-Art," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 6, pp. 851–865, Dec. 2018.

[4] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, Jun. 2002. [Online]. Available: https://doi.org/10.1162/106365602320169811

[5] J. Schrum and R. Miikkulainen, "Evolving multimodal behavior with modular neural networks in Ms. Pac-Man," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14. Vancouver, BC, Canada: Association for Computing Machinery, Jul. 2014, pp. 325–332. [Online]. Available: https://doi.org/10.1145/2576768.2598234

[6] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A Neuroevolution Approach to General Atari Game Playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 355–366, Dec. 2014.

[7] O. Tange *et al.*, "Gnu parallel-the command-line power tool," *The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.