Evolutionary Computing - Standard Task 1: Specialist Agent Group 22

Classical crossover functions for fixed topology neural networks

Alessio Ferrarini
2821024
alessio.ferrarini@studenti.unipd.it

Daniele Quartinieri
2767570
d.quartinieri@student.vu.nl

Lars de Wolf
2820753
l.de.wolf@student.vu.nl

Leonardo Dominici
2829197
leonardo.dominici@student.uva.nl

Mahdi Kazemi
2807658
m.kazemi@student.vu.nl

30 September, 2023

1 INTRODUCTION

Classical crossover operators are not optimal choices in those situations in which evolutionary algorithms modify directly the weights of a neural network. For this reason, algorithms like NEAT [8] rely on custom crossover operators, which also take the topology of the neural networks into account; the main issue that arises with such approaches is they require an evolving topology, which is not feasible in every domain.

There is already a theory that classical crossover operators create functionally unstable networks [1], also known as the permutation problem. But in practice, especially in the field of engineering, classical evolutionary algorithms are still used to evolve fixed topology neural networks. These algorithms often use classical crossover operators, although in some cases [6], they can operate while having discarded the crossover operation completely [7], while still obtaining satisfying results.

Since crossover operators vary in the way they act on the weights of the neural network, our goal is to compare operators that rearrange the weight of the neural network like the *2 point crossover* with operators which computes a function of the weights like *whole arithmetic crossover*. The toy problem that we will be using to train the neural networks is called Evoman [4] since we have the results obtained by applying NEAT to compare to [3].

1.1 Hypothesis

Even though [1] shows that all classical recombination operators are destructive, we expect to see *2 point crossover* perform better than *whole arithmetic crossover*. This expectation arises from the observation that the former keeps most of the weights in the offspring similar to the parents' weights in both position and value keeping most of the neural network structure intact and functional minimizing the damages described in [1], which cannot be said for the latter which compute a weighted average of the parents' weights completely changing the synergy between weights.

2 METHODS

In order to extensively compare the two EC algorithms, we ran each algorithm against 3 enemies. Every algorithm has been run 10 times against each enemy, this is necessary given the stochastic nature of EC. To gather additional information on the best-performing individuals, the best individual in each test is selected and run against its enemy.

2.1 Software implementations

For our software implementations, two python3 codebases have been developed, one "vanilla", inspired from "*optimization_dummy.py*" and one using LEAP [2], a general-purpose Evolutionary Computation package. Since both implementations gave similar results, the choice has fallen to the LEAP package, due to code cleanness.

2.2 Metrics

The population is evaluated using the default fitness function. The remaining enemy health E_{health} has the most impact on the fitness value, however, the remaining player health P_{health}, as well as the logarithm of the elapsed time also contributes to the fitness value, as shown in equation 2.2.1. This implementation of the fitness function favors a more aggressive strategy, where reducing E_{health} results in higher fitness scores.

$$f_i = 0.9(100 - E_{health}) + 0.1P_{health} - \ln(t) \tag{2.1}$$

The best individual is defined as the individual with the highest gain. The gain I_{gain} is calculated by subtracting the enemy's health from the player's health, formally in equation 2.2.2. Although this metric takes fewer factors into account compared to the fitness metric, the individual gain gives a better insight into the performance of the individual.

$$I_{gain} = P_{health} - E_{health} \tag{2.2}$$

The genetic diversity D of a population is calculated using a pairwise squared distance metric. This takes the sum of the squared Euclidean distances of genomes between individuals.

$$D(population) = \sum_{i=1}^{n} \sum_{j=1}^{n} ||x_i - x_j||^2 \tag{2.3}$$

2.3 Enemy selection

We decided to pick enemies 3 (Woodman), 6 (Crashman), and 7 (Bubbleman). This is due to the fact that from the empirical results in [3], these three should be the hardest enemies. These enemies were chosen because specialist agents get very good quite quickly, and with easier enemies, the random initialization was already giving us very capable agents. This resulted in unfair environments to compare the algorithms.

3 EA INSTANCES

To compare the differences between crossover functions, we implemented two evolutionary algorithms with overlapping components.

Both algorithms are generational, meaning that none of the individuals of the current generation can survive to the next generation. The individual components of the algorithms are described below.

3.1 Selection

In every generation, parents for generating suitable offspring are selected using a tournament selection. An individual is selected from the population, and its fitness is compared to k other randomly selected individuals. This is repeated until enough parents have been selected to produce population_size children. In both algorithms, we used a k of 10 and a population_size of 100.

3.2 Crossover

Two types of crossover have been tested, two-point crossover and whole arithmetic recombination. Two-point crossover operates by slicing the parent's genes into two randomly selected places, obtaining 3 sections for each, and swapping them. Whole arithmetic recombination uses the weighted average of the two parents to generate two children. Children can be more similar to their parents by varying α, which was set equal to 0.5. Formulas for the children of the whole arithmetic recombination are listed in equations 3.1 and 3.2.

In both algorithms, the probability of a crossover happening was set to 100%.

$$Child1 = \alpha x + (1 - \alpha)y \tag{3.1}$$
$$Child2 = \alpha y + (1 - \alpha)x \tag{3.2}$$

3.3 Mutation

Children were mutated by adding values drawn from a Gaussian distribution. Each allele in the genotype of the child went through the transformation described in 3.3.

$$x' = x + \mathcal{N}(0, \sigma) \tag{3.3}$$

The mutation_rate, σ, was kept constant in both algorithms. For both experiments, we used a mutation_rate of 0.6.

3.4 Parameter Tuning

To ensure reliable performance, we conducted a targeted parameter tuning process for the mutation rate, population size, selection mechanism, and size of the tournament in tournament selection.

We opted for a manual exploration of the parameters, refraining from relying on automatic tuning algorithms. This choice stemmed from the consideration that utilizing such algorithms would necessitate selecting a specific EA for their application, potentially introducing bias and unfairness to the evaluation of other EA in our study. Most of the parameters correspond with the suggested value [5] and were adjusted if needed to ensure a favorable selection pressure.

4 RESULTS

4.1 Numerical Data

We ran each algorithm 10 times against 3 enemies and plotted: the average maximum fitness with standard deviation, the average average fitness with standard deviation, the diversity of the population with standard deviation, and the individual gain of the best agent produced in each run.

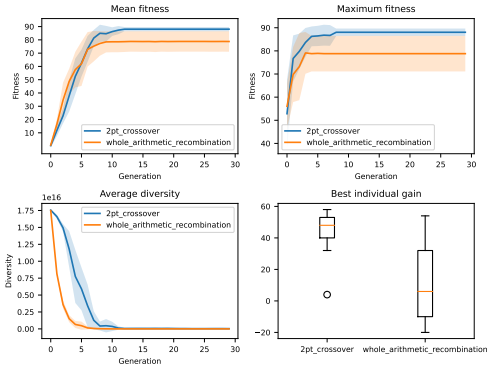

Figure 1: Numerical results of 10 runs of our EAs against enemy 3

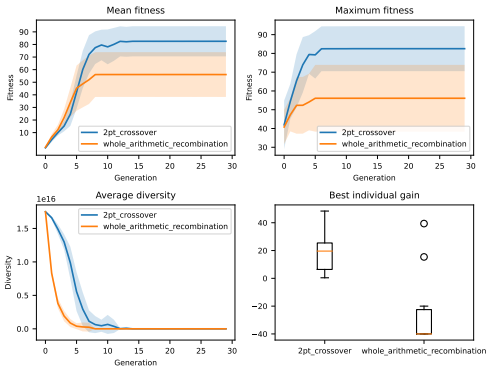

Figure 2: Numerical results of 10 runs of our EAs against enemy 6

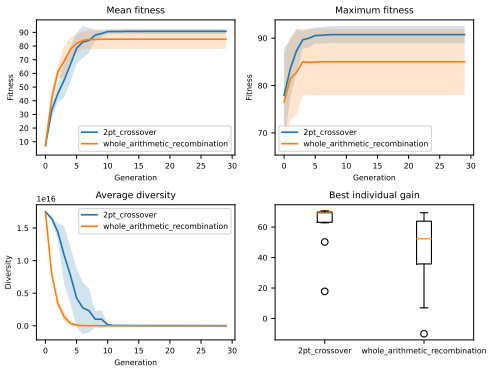

Figure 3: Numerical results of 10 runs of our EAs against enemy 7

4.2 Comparison between our EAs

As we can see from plots 1, 2, and 3; *2 point crossover* results better performing against *whole arithmetic crossover*, against the easier enemies (3 and 7) the differences between from a *fitness* standpoint is very small but still noticeable, against the harder enemy 6 the difference are more explicit favoring *2 point crossover* by far.

From the very high standard deviation in the results obtained by *whole arithmetic crossover* we can infer that this method is strongly dependent on a good random initialization, the same thing doesn't happen with *2 point crossover*.

Another important point to notice is the trend in diversity *whole arithmetic crossover* is worst also at keeping diversity in the population usually converging to a local optimum really fast, meanwhile *2 point crossover* takes more generation to converge to a fixed population and usually having spikes in diversity corresponding to spikes in both the best and mean fitness, this could be explained by the fact that theoretically *whole arithmetic crossover* produces worst offspring and also would explain his strong dependence on random initialization.

Enemy	p-value
3	0.003862
6	0.000507
7	0.096712

Table 1: P-Values calculated based on the individual gain of the best agents produced each run

From table 1 we are obtaining sufficient p-values to claim the difference between the 2 GAs is not caused by simply chance, the p-value computed against enemy 7 could be lower but is still more than an acceptable result, the issue probably arises from the very big standard deviation between the runs of the GA using *whole arithmetic crossover* this could be fixed by running the experiments more times.

Against enemy 6 the p-value is extremely low confirming the statistical significance of our experiment against the hardest enemy.

4.3 Comparison with NEAT

Enemy	NEAT [3]	GA [3]	2 pt. crossover	WA crossover
3	80	58	58	54
6	73	0	48	39
7	78	68	71	69

Table 2: Comparison of our EA instances to [3] reporting the energy of each best agent at the end of the battle.

As we can see from table 2 against each enemy, except 7, we underperformed against agents evolved with NEAT but we obtained big improvements against the classical evolutionary algorithm, especially taking into account that we have the same initial population but performed only 30 generations instead of 100, the crossover function that is used for the EA in [3] is a modified version of *whole arithmetic recombination* where the parameter α is chosen randomly.

Enemy	2 pt. crossover	WA crossover
3	10	5
6	10	2
7	10	9

Table 3: Number of runs that were able to produce a best agent able to beat the enemy (out of 10)

Even if from 2 we can suspect that both *2 point crossover* and *whole arithmetic crossover* are performing very similarly it's important to note that the first is much more stable as we can see from table 3 this fact could also be inferred from the lower standard deviations in plot 2.

4.4 Convergence

From the low standard deviation noticeable in plots 1, 2 and 3 and the fact that the agents were always able to beat the agents produced by the genetic algorithm in [3] we are confident that the EA using *2 point crossover* is converging to a global optimum, meanwhile, the EA using *whole arithmetic crossover* is clearly getting stuck in local optimums on runs with a "bad" random initialization, the same reasoning is also supported by the number of runs that were able to produce a winning agent as seen in table 3.

5 CONCLUSIONS

As we can see from the numerical results the EA using *2 point crossover* always performed better than the one using *whole arithmetic crossover* with an important statistical, even if the two enemies 3 and 7 weren't hard enough to contribute in an important way to our research.

As expected we weren't able to produce agents as good as the agents produced by a more sophisticated algorithm like NEAT but with minor tweaking we were able to produce agents that are substantially better than the one produced by other classical evolutionary algorithms in [3].

The results seem to support our thesis but as we will explain in the sections below it not enough experimental evidence to confirm it.

5.1 Scope of the research

It's important to note that all the 3 tasks on which the 2 genetic algorithms were tested are pretty similar to each other, so we cannot know if there is some inherent bias in the problem that favours *2 point crossover* over *whole arithmetic crossover* that is only related to the fact that we are choosing the weight of a neural network.

5.2 Future research

Clearly more testing for our hypothesis is needed, it would be interesting to try different problems that are substantially different from game playing and harder problems to prove that the difference between the methods is not implicit in the nature of the task.

To strengthen our hypothesis a possible step could be analysing other crossover function that have the same behaviour as the one tested for example *n-point crossover* or *uniform crossover* in place of *2 point crossover* and *blend crossover* in place of *whole arithmetic crossover*, an experiment that gives a winning result for the first would strengthen our hypothesis on why *2 point crossover* performs better than *whole arithmetic crossover* when evolving neural networks. We also believe that interesting results can be obtained by using self-adapting parameters, due to the fact that it is not always preferable to have constant parameter values throughout the whole evolution.

Also would be interesting to try if the same results are replicated in more complex evolutionary algorithms for example using self adaptive mutation or the island model even if they could hide the difference by compensating for the shortcomings of the crossover operators.

ACKNOWLEDGMENTS

We would like to express our gratitude for the invaluable contributions provided by our Teaching Assistants. Their insights served as a source of inspiration for aspects of our research and greatly assisted us in the process of structuring and presenting our findings in a rigorous scientific manner.

REFERENCES

[1] Peter Angeline, Gregory Saunders, and Jordan Pollack. 1994. An evolutionary approach that construct recurrent neural networks. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 5 (02 1994), 54–65. https://doi.org/10.1109/72.265960

[2] Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. 2020. Library for Evolutionary Algorithms in Python (LEAP). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1571–1579. https://doi.org/10.1145/3377929.3398147

[3] Karine da Silva Miras de Araujo and Fabrício Olivetti de Franca. 2016. Evolving a generalized strategy for an action-platformer video game framework. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. 1303–1310. https://doi.org/10.1109/CEC.2016.7743938

[4] Fabricio Olivetti de Franca, Denis Fantinato, Karine Miras, A. E. Eiben, and Patricia A. Vargas. 2020. EvoMan: Game-playing Competition. (2020). arXiv:cs.AI/1912.10445

[5] A. E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing* (2nd ed.). Springer Publishing Company, Incorporated.

[6] Jose Antonio Martin H and Javier Asiaín. 2009. Learning Autonomous Helicopter Flight with Evolutionary Reinforcement Learning, Vol. 5717. 75–82. https://doi.org/10.1007/978-3-642-04772-5_11

[7] Torsten Reil and Phil Husbands. 2002. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Trans. Evol. Comput.* 6 (2002), 159–168. https://api.semanticscholar.org/CorpusID:7092624

[8] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127. https://doi.org/10.1162/106365602320169811