VU Amsterdam Evolutionary Computing 2022

The Standard Assignment - Task 1 - Group 47

Dionysios Kyriazopoulos
2727624

Antonios Georgakopoulos
2775847

Sarper Okuyan
2740753

Marco Guglielmi
2746817

Priyakshi Goswami
2762110

ACM Reference Format:
Dionysios Kyriazopoulos, Antonios Georgakopoulos, Sarper Okuyan, Marco Guglielmi, and Priyakshi Goswami. 2022. VU Amsterdam Evolutionary Computing 2022: The Standard Assignment - Task 1 - Group 47. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Evolutionary Algorithms

The increasing complexity of the problems (in fields such as machine learning, data mining, bioinformatics) caused the task of optimization to be more difficult. [6] The diversity in the types of search methods can be useful because this creates the opportunity to solve a problem with different techniques when a particular one falls short. For example, a well-known, widely used optimization algorithm called gradient descent takes the slope of the loss surface into account [4], but this approach may not be optimal when the loss function is not differentiable. On the other hand, the demand for problem-solver automation is also increasing because of the limited capacity in resources to be used on deep analysis of problems and algorithm design. These two reasons creates a demand for a problem-solver which can be applied on wide range and complex problems. Fortunately, a class of optimization algorithms is proposed as capable to satisfy these requirements, called evolutionary algorithms (EAs). [3]

The term *evolutionary algorithms* describes a set of algorithms that are inspired from Darwinian evolution.[1] They are population-based non-deterministic search algorithms which follow the concept of *natural selection*, one the main pillars of evolution theory. To do so, candidate solutions are considered as individuals which form a population. Each individual is evaluated with a fitness function. In order for the population to evolve, new individuals are required which are new solutions produced by 'mating' between chosen current solutions. The variation is provided by *mutation* and *recombination* operators. The population size should be fixed to apply selection pressure on the individuals. Therefore, individuals are ranked by fitness values and the worst ones are discarded which leads to a new population. The process is repeated until a certain level is reached. [3]

1.2 EvoMan Framework

EvoMan is a game-playing framework for testing and comparing optimization algorithms. It is based on the game called 'MegaMan II' and it contains 8 different predator-prey games. A single game ends when one of them is dead which means to be out of energy. Both the enemy (predator) and the player (prey) can be controlled by either an AI algorithm (specifically, a neural network) or default behaviour which follows some fixed rules. There are 20 sensor values available which mostly distance measures. These variables will be fed into the AI algorithm. The output will point out the predicted optimal move of the player. Lastly, an algorithm can be trained as a specialist agent or as a generalist agent. The former means to play against a single enemy while the latter means to play against multiple enemies in a single training. [2]

1.3 Research Question

In this assignment, our task is to implement two EAs into 'static enemy' version of the framework in order to train a specialist algorithm. A custom-made algorithm for a specific problem may provide best performances, yet the resources for a thorough analysis and an algorithm design is not unlimited, as mentioned earlier. Fortunately, a prefabricated algorithm may perform well enough. Therefore, we chose to design a (Simple) Genetic Algorithm from scratch and to implement a Genetic Algorithm library called NEAT. Our goal is to compare solution performances of their specialist agents in three different scenarios.

2 METHODS

2.1 The Genetic Algorithm

2.1.1 Motivation. Coding a GA almost from scratch is the best way to delve into evolutionary algorithms. The decisions you have to take are crucial for every step. Experimenting with the various techniques and observing the results is paramount in obtaining a satisfying result. Our goal with this approach is to understand the components of a GA and select the best techniques for our solution.

2.1.2 Representation and Fitness Function. We decided to use the neural network architecture that was available to us through the `demo_controller_py` to represent the individuals. Each individual is represented by a neural network. Each neural network maps the inputs, which are the inputs of the 20 sensors of the game, to the outputs, which are the five actions a player can do in the game. Therefore, each individual is represented by real values. Also, each neural network contains biases for the hidden neurons, weights for the connections, and a sigmoid activation function. We are also using the default fitness function of the framework to evaluate the individuals. After each evaluation, we receive the fitness, the player energy, and the enemy energy for each individual. The number of individuals in a population is 75, which was providing equal or better results than 100.

2.1.3 Parent Selection, Crossover and Mutation. Parent Selection, Crossover, and Mutation are essential in exploring and exploiting the search space. Parent selection is the first step in producing great offspring. We are mating 2 parents to produce randomly 4 to 8 children in total. Thus the λ/μ (λ children and μ parents) factor is approximately 3, which is the most popular setting today. For each parent, we implement a tournament selection. One tournament hosts 6 competitors and the fittest one, according to the fitness function, is deemed to be the parent. After selecting the parents, we apply the blend crossover, because we want to explore the space beyond the rectangle spanned by the parents. All other crossover options provide significantly less space to explore. In blend crossover, each genome z_i of the offspring is obtained from moving each genome of one parent by $\gamma = (1 - 2\alpha)u - \alpha$ and by $1 - \gamma$ for the other parent. The value u is a random value between 0 and 1 and α is the value that determines the search space for the crossover. The crossover is applied with probability 1 because we always want to exploit and explore more space. Finally, the mutation causes small and random variations to the genomes of the offspring and aims to provide connections in the search space and diversity. We initially experimented with uniform mutation which is a very simple technique and did not give us the wanted results. Ultimately, we chose to use a self-adaptive mutation model and, specifically, the uncorrelated mutation with one step size. In this technique, each chromosome has its step size σ, which influences the mutation of all the genomes. Simultaneously σ also mutates itself before every mutation. Each genome is mutated according to

the function $x_i' = x_i + \sigma' * N_i(0,1)$. The mutation probability and the evolution of the step size are discussed in section 2.1.5 .

2.1.4 Survivor Selection and Population Management. Survivor selection is extremely important in improving the population and maintaining diversity. We had to choose between "comma strategy" (μ, λ) and "plus strategy" $(\mu + \lambda)$. Both are fitness-based selections and both gave us great results. Plus strategy is based on both parents and offsprings, meaning that the next population is comprised of the fittest individuals. Comma strategy depends only on offsprings, as it completely deletes the parents and keeps the fittest offsprings. Comma strategy provided the overall best results, while also helping in avoiding local optima and preserving diversity. The comma strategy also fits the uncorrelated mutation's σ step size. The initial number of individuals is maintained for every generation. What happens when evolution does not improve for many generations? We created a mechanism that "reshuffles" the population if there is no improvement after 10 generations. By improvement, we mean that the best solution is never topped for the next 10 generations. This mechanism keeps 2/3 of the population and replaces the remaining 1/3 with either the best solution or a random one. This choice is made also randomly. This technique helps us explore new areas in the search space and avoid local optima. Finally, a copy of the best solution so far is always kept.

2.1.5 Parameter Settings. Setting the parameters of an algorithm is paramount to performance. Although no parameter tuning took part in the genetic algorithm, the widely regarded optimal values were chosen. These parameters are:

- $N = 75$: The number of the individuals in the initial population, as well as in every generation. We selected 75 individuals.
- $Size_t = 6$: Tournament size for parent selection. Six competitors are fighting for a spot.
- $P_c = 1$: The possibility of a crossover happening between two parents. This value was set to 1.
- $\alpha_c = 0.5$: The value that determines the search space for the crossover. The value is 0,5 because the chosen values are equally likely to lie inside the two parent values as outside, so balancing exploration and exploitation.
- $P_m = 0.2$: The possibility that an offspring mutates.

Additionally, we used parameter control to tune a parameter online. Step size σ of the self-adaptive mutation was tuned online. Initially, we selected the value of 1 for σ. Finally, we decide to mutate the σ with the following function: $\sigma' = \sigma * e^{N(0,\sigma)}$. This function makes sure that σ is unpredictable and almost out of the user's control. Furthermore, the evolution of σ is a result of natural selection.

2.2 The NEAT Algorithm

2.2.1 Motivation. By using the NEAT algorithm[5] we can take advantage of the different approach of the evolution process and the benefits that it provides. The NEAT algorithm is able to find not only the optimal weights for a neural network, but also the topology(architecture) for the neural network that works best for the problem we are trying to solve, as opposed to a fixed-topology neural evolution approach. Thus, it reduces the overall structure complexity of a network and it prevents the algorithm from creating an overly large and complex network that is not optimal for the given problem. Moreover, it protects the topological innovation by separating networks into different species, creating the potential of overall better solutions in the long run. For this approach we used the NEAT-Python library to run the experiments.

2.2.2 Representation and Fitness Function. A genome in the NEAT algorithm is represented by two lists that contain information about the corresponding neural network. The 'nodes genes' list provides information about the type of each node in the network, while the 'connection genes' list specifies the details about the connection of two nodes in the network for example the in-node, the out-node, the weight of the connection etc. An important value in the representation of the genome is the innovation number that is attached to a specific gene. So when a new gene is created, a global innovation number is incremented and assigned to that gene. These innovation numbers do not change and are being used as historical markings because it is easier not only to track structural changes, but also to find the origin of a gene. For the fitness function we are using the default fitness function of the framework to evaluate the individuals.

2.2.3 Mutation and Crossover. The mutation in the NEAT algorithm changes both the network structure and the weights of the network. Connection weights are modified in the standard way as in any neuroevolution algorithm. There are two ways that the mutation of the network structure can be achieved. The first way is by adding a new connection with a random weight that connects two previously unconnected nodes. The other way is by adding a new node inside the network. For this to happen a connection must be split and the node is placed in the same place as the old connection used to be. As a result this connection is disabled and two new connections are added to the network. For the crossover operation the NEAT algorithm uses the innovation numbers to allow the algorithm to track the origins of different genes and define which genes of different genomes match up. So when the crossover operation is performed, the genes are tracked by chronological order and the similar ones are aligned. The genes that do not match are inherited from the fittest parent.

2.2.4 Speciation and Explicit Fitness Sharing. In NeuroEvolution processes usually the newly created individual will do poorly in terms of performance because of their low fitness level and thus they are having less chances of surviving because the selection process is based on the fitness value. In order to protect this new topological innovation, NEAT separates the individuals into species and as a result gives these new innovations time to evolve and optimise in order to compete with the rest of the species population. This separation is based on topological similarity between the individuals, and the correlation between genomes can be calculated using the historical markings. For the comparison between different genomes a value δ (distance measure) is defined in order to find if this genome belongs to an existing species or not. The value of δ depends on the disjoint genes, the excess genes and the weight

difference of the matching genes between the current genome and a sample genome from each species. If δ is less than a predefined threshold δ_t (compatibility threshold) then the current genome is placed in that species. Otherwise the current genome's δ is compared with other species' δ and if the genome is not compatible with any of the previous species then a new species is created. The NEAT algorithm is also using a technique called explicit fitness sharing that protects new innovation within the species. Before the selection process the fitness of each individual is divided by the number of individuals in that specific species. As a result the older species are penalised and newer species can be active for longer and thus preventing a single species from taking over the population.

2.2.5 NEAT-Python Framework. We decided to implement the NEAT-Python framework for this approach due to its clear documentation and the plethora of configuration choices that we could make. Moreover the instantiation of the NEAT environment was convenient and the integration with the EvoMan framework's methods was troubleless. We created a class named `NeatPopulation` that inherits from the `neat.Population` class in order to make the proper modifications for the best individual to be calculated based on their individual gain first and then their fitness value. As with approach 1, we did not perform parameter tuning but after some trials we concluded that satisfying results could be reached with 100 individuals in the population, 30 generations in total and using the sigmoid as the activation function.

3 RESULTS AND DISCUSSION

3.1 Genetic Algorithm Results

The results of the GA were impressive early on. Initially, we would always get a few winning individuals for the first population, but the mean fitness was below 10. Entering the second generation, the mean fitness is already scaling upwards of 90 for enemies 2,8 and upwards of 86 for enemy 5. The algorithm provided better results for enemies 2,8, where the average max fitness almost reaches 94 in most generations. Two of the best solutions for enemy 2, achieve an individual gain of 98. Furthermore, the high standard deviation of a few generations is due to the population management mechanism, where the population gets reshuffled to explore new areas. Overall, the performance was pleasing and the results were satisfactory, although we narrowly missed the 100 mark for the individual gain.

3.2 NEAT Algorithm Results

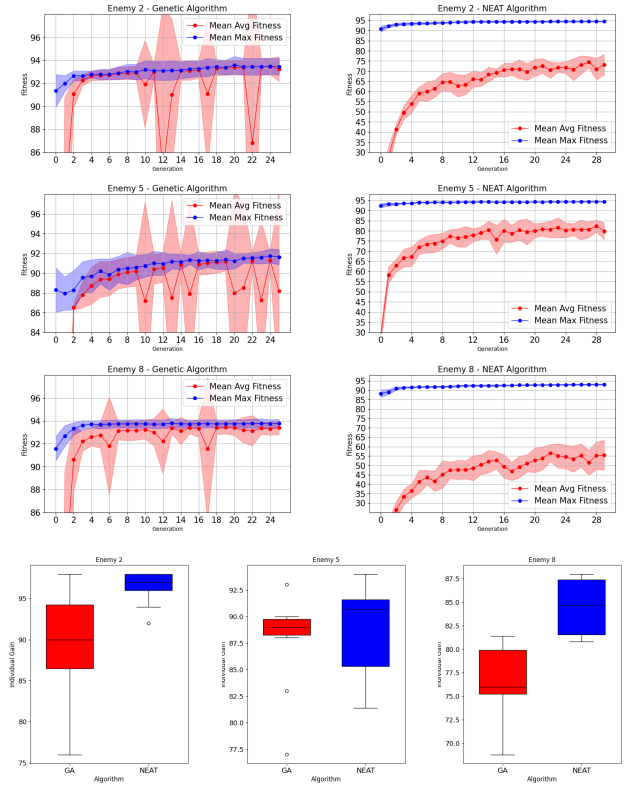
The NEAT algorithm managed to achieve great fitness values, close to 90, from the very first generations. There was a further upward trend with the maximum fitness levels reaching close to 93 for each enemy. An interesting observation is that the average fitness for all enemies showed a rising trend for the first 12 generations and then the numbers dipped significantly. After that the average fitness values continued rising until the last generation.

3.3 Discussion

When comparing the results of both algorithms the findings are interesting. Both algorithms achieve exceptionally high mean maximum fitness, as we observe in Figure 1. However, the NEAT algorithm reaches higher individual gain than the genetic algorithm in every enemy. The big difference lies in the mean average fitness. In the genetic algorithm, the mean average fitness tends to equalize or be very close to the mean maximum fitness. On the other hand, the mean average fitness of the NEAT algorithm is always 20 fitness points below the mean maximum fitness. Finally, in terms of max average fitness, the NEAT algorithm outperforms the genetic algorithm only for enemy 5.

Our genetic algorithm outperforms the genetic algorithm of the original paper[2] for enemies 2,5 and 8. The GA has the best individual gain values of 94,90,80 for these enemies respectively, while the original paper's values are equal to or well below 90. Also, our NEAT algorithm equals the paper's NEAT algorithm for enemies 2 and 8 but gets outperformed for enemy 5.

Figure 1: Visualized performances (fitness graphs and boxplots) of algorithms in Enemy 2, 5 and 8 cases.



4 CONCLUSION

The NEAT performed significantly better than GA for Enemy 2 and 8. For Enemy 5, we can't say that there is a significant difference between them because the median values of the boxplots are on the same level with the other box. We can state that the NEAT algorithm did not significantly perform worst in all cases. In the end, the prefabricated algorithm seemed *good enough* compared to designed algorithm. The usage of the former could save time and resources while getting a sufficient performance. In the future, it would be interesting to test them while training a generalist agent.

REFERENCES

[1] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. 2014. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4, 3 (2014), 178–195.

[2] 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)*. IEEE, 1303–1310.

[3] Agoston E Eiben, James E Smith, et al. 2003. *Introduction to evolutionary computing*. Vol. 53. Springer.

[4] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[5] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[6] Pradnya A Vikhar. 2016. Evolutionary algorithms: A critical review and its future prospects. (2016), 261–265.