Evolutionary Computing - Assignment 1: Specialist agent

Team 88 - October 2, 2021

Nedim Azar (2728313)
Vrije Universiteit
Amsterdam, Netherlands
n.azar@student.vu.nl

Simone Montali (2741135)
Università di Bologna
Bologna, Italy
s.montali@student.vu.nl

Giuseppe Murro (2740494)
Università di Bologna
Bologna, Italy
g.murro@student.vu.nl

Martin Pucheu Avilés (2687903)
Vrije Universiteit
Amsterdam, Netherlands
m.i.pucheuaviles@student.vu.nl

ACM Reference Format:
Nedim Azar (2728313), Simone Montali (2741135), Giuseppe Murro (2740494), and Martin Pucheu Avilés (2687903). 2021. Evolutionary Computing - Assignment 1: Specialist agent: Team 88 - October 2, 2021. In *Proceedings of X400111 (Evolutionary Computing).* , **??** pages. https://doi.org/

1 Introduction

1.1 Evolutionary computing

The term **evolutionary computing** refers to a field that is concerned with designing, applying, and studying evolutionary concepts in computing. We could consider it as the link between **problem-solving** and **biological evolution**. By applying these evolutionary concepts to candidate solutions of a problem (which, in EC, are equivalent to **Individuals**), we can find the optimum in a fast, efficient and non-deterministic way.

1.2 The EvoMan framework

As with every other optimization field, we're often interested in testing the performance of **Evolutionary Algorithms** in a standard way. The **EvoMan** framework was created with this purpose in mind: it consists of a game in which a player has to fight against an enemy that shoots, jumps and moves. The framework provides **8 different enemies**, each having different behavior and ways of winning. 20 *sensors* are available, providing information about the environment. Having these sensors as input, we would like to find a *function* that maps these values to an **optimal action** for the player. We would like to be able to represent a neural network's structure in our evolutionary algorithm, to be able to find the optimal one. The **fitness** of these individuals will just represent the score our player obtains in the game.

1.3 Research question

Open source heavily supported the progress research: a multitude of genetic algorithms can be found on the web. But is it always better to use open source? We know from the literature that fine-tuning an algorithm to perfectly suit a problem is **often the best choice**, but sometimes our main concern may not be solution quality: for example, in a *production perspective*, we can say that a correctly working solution is already good enough. On the other hand, open-source libraries like *NEAT* are usually well-supported by the community and were created with a way **higher effort** in terms of time, the number of researchers, and community. This might even result in better performances compared to a custom-tailored algorithm missing the **highly specialized optimizations** that only time and experiments can lead to. Furthermore, from the academic point of view, this research is thought to be interesting from a future work perspective: while implementing an algorithm from scratch allows us to **practice all the theoretical concepts** we learned during lectures, using an open-source library allows us to discover the benefits of a **faster**, yet **more expressive** implementation, and possibly **higher performances**.

2 Methods

2.1 Two different approaches

For the first approach, we **developed an evolutionary algorithm from scratch**, while for our second approach we **implemented an existing neuroevolution algorithm** called NEAT.

2.2 Approach 1: coding the GA from scratch

2.2.1 Overview. In this approach, **three main components** interact:

- the `GameRunner` object, that is in charge of initializing the EvoMan framework for a new game;
- the `PlayerController` object, in charge of the translation from the genotype (a Python dictionary containing a Numpy array for the neural network, and a mutation step-size as seen in section 2.2.3) to the phenotype, a neural network;
- the `GeneticOptimizer` object, in charge of the search for an optimal individual through evolution.

2.2.2 Representation, genotypes and phenotypes. Choosing the **correct representation** for a problem is the most important task in the design of an **evolutionary algorithm**. As mentioned in section 1.2, we would like to obtain an **individual** representing an **optimal neural network** that is able to **map sensors'** inputs to a player **action**. The most straightforward approach to represent a NN would be using a real-valued **array** containing the **weights and biases** for the neural network. We could keep this array **flat**, then **reshape it** when needed, as we know the network structure. In fact, to convert the **genotype** to a phenotype (the neural network), it's sufficient to remove the last $\sum_{i=w_n}^{i\leq n} 1$ values representing the **biases**, then reshaping the remaining array with `numpy` to represent the layers. The network structure is a conventional one: the layers are fully connected and a *ReLU activation function* is used for the hidden layers, while the output layer is activated by a *sigmoid function* that gives a probability for each of the possible actions. Since "left"/"right" and "jumping"/"releasing" actions are mutually exclusive, only the ones with the highest probability are considered activated.

2.2.3 Reproduction and mutation. **Reproduction** and **mutation** are crucial steps for a good exploration of the search space. The first thing that is needed to obtain a good offspring is applying what is known as *mating selection* (seen in section 2.2.4). Then, two of the chosen individuals are cloned into the offspring, and **blend crossover** is applied with a probability $p_{crossover}$. This type of crossover was introduced in [4] to generate offspring in a region that is **bigger than the n-dimensional rectangle** spanned by the parents. This allows for a **more thorough exploration** of the search space, moving each attribute of the genome of parent 1 by a quantity $\gamma = (1 - 2\alpha)u - \alpha$ where u is a random number between 0 and 1, and $1-\lambda$ for parent 2. **Mutation** is the other important operation that allows us to explore the search space: each individual has a probability $p_{mutation}$ of mutating, and each attribute in this individual mutates with probability $p_{ind\_mutation}$. To achieve a non-uniform mutation, we implemented what is called **self-adaptive mutation**. This type of mutation sees a varying σ value, which is added to the genome of an individual and evolves together with it. Every attribute that has to mutate will then mutate of a quantity $x_i' = x_i + \sigma' \cdot N_i(0, 1)$. To further investigate how σ is updated, see section 2.2.5.

2.2.4 Selection. While variation operators strive for exploration, **selection** is concerned with exploitation. It allows us to select good individuals in the population, allowing them to survive and mate. We can talk about two types of selection: *mating selection* and *survival selection*. The first selects which **parents will be able to reproduce**, while the latter is used to decide **which individuals to replace** (since we're maintaining a fixed population size, we can talk about replacement). To perform *mating selection*, we used a **tournament selection**. This type of selection allows us to only consider the fitnesses of a small subset of individuals, then select the best among these. What happens is that we extract $size_{tournament}$ random elements from the population, then pick the best one. We repeat this process for k times, obtaining, consequently, k individuals. To perform survival selection, two main choices could be taken: the first is $(\mu+\lambda)$-selection, which considers both the present population and the offspring in the selection of the survivors. The second approach is (μ, λ)-selection, where all the parents are discarded in spite of a sometimes bigger offspring. As seen in [3] at section 5.3.2, the latter is usually preferred, as it's better at leaving local optima, forgetting outdated solutions, and the σ we're using for *self-adaptive mutation*. Because of this, we're performing (μ, λ)-selection to choose our offspring, using **best selection**: only the μ best individuals are kept, whereas the other $\lambda \cdot \mu$ are forgotten. Literature research suggested a λ between 5 and 10, and several tests led us to choose $\lambda = 7$.

2.2.5 Parameter tuning and control. Parameter setting is a crucial part of every Artificial Intelligence algorithm, and Genetic Algorithms make no exception. We have to distinguish between *parameter control* and *parameter tuning*: the latter happens *offline*, before a run, while the first modifies the parameters continuously during a run. To perform **parameter tuning**, we exploited a Python library that allows users to explore the hyperparameter space, searching for the highest performances: `hyperopt` [1]. To achieve faster testing, we integrated our hyperparameter testing into a distributed computing platform, **Apache Spark**, then set up a cluster of cloud VPS to run experiments parallelly. The set that achieved the best results was the following:

Parameter	**Value**
$alpha_{crossover}$	0.45
$p_{crossover}$	0.75
$nodes_1$	20
$nodes_2$	24
$p_{mutation}$	0.62
$p_{ind\_mutation}$	0.70
$size_{niche}$	8
$size_{population}$	75
$size_{tournament}$	7

Parameter control was a concern too: we implemented a **self-adaptive mutation step size**. This consists in adding the mutation step size to the genome of an individual, making it part of evolution. Ideally, we'll want to have a high step size in the beginning (fixed to a default value of 1.0), to better **explore** the search space, then start **exploitation** later in the evolution. We mutate the step-size as seen in section 4.4.1 of [3]:

$$\sigma' = \sigma \cdot e^{\cdot N(0,\tau)}$$

2.3 Approach 2: Implementation of NEAT

2.3.1 Overview. For the second approach, we decided to study and implement the Neuroevolution of Augmenting Topologies (NEAT) algorithm created by Kenneth O. Stanley and Risto Miikkulainen[5], which has been proved to be capable of getting excellent results for our given task[2]. For this purpose, we made use of the NEAT-Python library which is a pure Python implementation of NEAT.

2.3.2 Representation. Every genome of the population is composed of two lists that represent the complete structure of a network: the **connection genes and node genes lists**. Thus, every *connection gene* specifies both the in and out *node genes*, the weight of the connections, a bit that indicates if the connection is enabled, and an **innovation number**.

2.3.3 Reproduction and Mutation. Mutation in NEAT can modify both the structure of the network and the weights of the connections. The weights are modified as in any neuroevolution algorithm, while the **structure mutation expands the genome space** in two different ways. On one hand, there is an **add connection mutation** that adds a single connection between two node genes previously unconnected. On the other hand, we have an **add node mutation** that replaces an existing connection, adding a new node where the old connection used to be. In this way, the genome space is expanded through the creation of new networks with size-varying connections. The crossover operator out stands for its simplicity at the moment of **matching networks with different topologies**. Every time a new gene is created through mutation, a **global innovation number** is increased and assigned to the new gene. The innovation number never changes for a specific gene so in this way **the historical origin of a gene is well known through all the evolution process**. Then, when crossover occurs, the genes from the networks involved are lined up by their innovation numbers, **recasting the problem from topological analysis to historical matching**, which is significantly simpler.

2.3.4 Speciation, Fitness Sharing, and Minimized Dimensionality. Speciation allows new structures, which usually have reduced fitness, to have time to evolve into better solutions. In this way, the **diversity of the population is preserved and the innovation is protected**, having individuals competing most with only other individuals from the same species. Innovation number is also used for this purpose. The more disjoint two genomes are (genes with different innovation numbers), the less they have in common, which permits the classification of those genomes in different species. NEAT also implements **explicit fitness sharing** as a method to prevent an individual to take over the population. Individuals from the same species adjust their fitness depending on the species size and the species can grow or shrink depending on the average fitness from all species. Finally, having speciation and fitness sharing allows NEAT to start the population with minimal networks that grow through structural mutation.

2.3.5 NEAT-Python Framework and Implementation. For the implementation of the algorithm, we used the NEAT-Python framework. This gave us the chance of using pre-existent classes, built with almost everything we needed for the evolutionary process. Classes such as `neat.Population`, which is initialized randomly and is the base for the searching of new solutions. For our work purpose, we made a subclass named `EvomanPopulation` with the customizations that were needed. We also did the same for the class `neat.BaseReporter`, writing our own class `EvomanReporter`. For the rest of the implementation, we made use of the standard classes that NEAT-python provides, such as the `neat.Configuration` class, where all the parameters that are going to be involved in the evolution are specified, such as the *population size*, the *node add mutation probability* or the *compatibility weight coefficient*, just to mention some. We also needed a *controller class* that does the mapping through the NEAT neural network created by any solution, and the actions taken by the EvoMan player at any given time.

2.3.6 Parameter tuning and control. As we did in our first approach, we chose to use `hyperopt`, an automated strategy to find optimized values for the `neat.Configuration` file described before. Some of those values are expressed in the next table:

Parameter	**Value**
pop_size	97
activation_mutate_rate	0.38
compatibility_disjoint_coefficient	0.94
conn_add_prob	0.92
...	...
node_add_prob	0.36
num_hidden	13
weight_mutate_rate	0.67

3 Results

3.1 Results obtained with approach 1

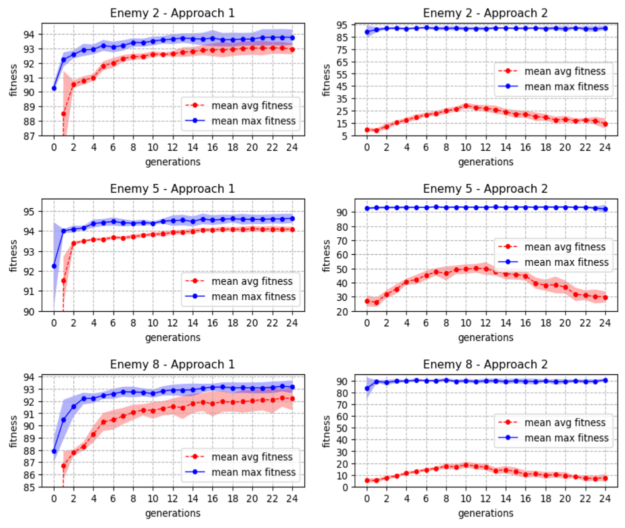
Starting from the first generation, results look interesting: the random initialization of the individuals achieves a **low fitness** (as expected), but as soon as an offspring is generated (note that $\lambda = 7$, meaning that the size of the offspring is quite large) and selection happens, **good individuals start to emerge** fast. The three enemies achieved **similar results**, with enemy 5 achieving the highest score, followed by 2 and 5. A fitness higher than 90 (which was reached with all the enemies) means that the player is **winning**, and it's doing so **fast**. It rarely happens that the player loses a game by finding itself in uncommon situations.

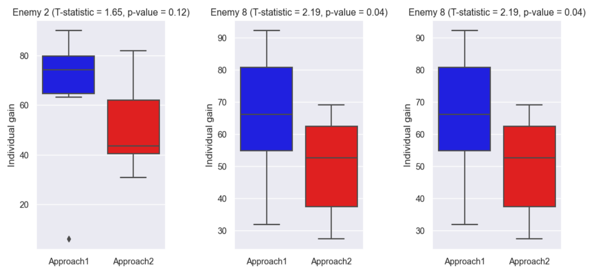
3.2 Results obtained with approach 2

Results obtained with this approach show that even in the **early generations** we can obtain very good individuals, reaching **fitness** values above 90 for all three enemies. Nevertheless, it is interesting to observe that, also for all enemies, the average fitness of the population reaches its highest point around the **twelfth generation**, and then it decreases. It is also interesting to mention that this decrease occurs without losing the fittest individuals. Finally, the algorithm has proved to be capable of finding solutions which were good enough to defeat all the three enemies.

3.3 Comparison

As seen in the previous sections, the first approach (coding from scratch) had **slightly better performances**. This proves that even though an open-source library might be **well-optimized**, designing algorithms from scratch delivers **better results** in these kinds of situations. Probably, these depend on the **application** too: in some cases, different decisions may be more profitable. We can see that both approaches have a steep learning curve, resulting in high fitnesses from the first generations already. Approach 2 learns **almost as fast** as approach 1, but we can notice that it **explores the search space** more: the maximum fitness is oscillating across all generations, and the average is more distant from it than in 1. Therefore, approach 1 has a smoother growth: the average fitness looks like an *increasing function*. The results are **very similar** to the ones seen in [2], which is not surprising: the paper uses similar approaches and algorithms to solve the same problem. We can note that the *p-values* are low, meaning that this comparison is **statistically significant**. To have a deeper insight on these graphs, check out the `plots` folder in the project.





4 Conclusions

This research was a good way of **understanding the basics** of Evolutionary Algorithms and how they can change the world we live in. We feel like, in the last years, most of the scientific research around computing has seen Machine Learning and Deep Learning as the sole protagonists of development. Evolutionary Algorithms could prove themselves as **highly powerful alternatives** that can explore the search space in a less *deterministic* way, being able to find uncommon solutions to common problems.

4.0.1 Work division. Montali and Murro: approach 1. Pucheu Avilés and Azar: approach 2. All members: implementation, experimentation, comparison of results, and report writing.

References

[1] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. JMLR.org, I–115–I–123.

[2] Karine da Silva Miras de Araujo and Fabrício Olivetti de Franca. 2016. Evolving a generalized strategy for an action-platformer video game framework. (2016), 1303-1310 pages. https://doi.org/10.1109/CEC.2016.7743938

[3] A.E. Eiben and J.E. Smith. 2015. *Introduction to Evolutionary Computing*. Springer. https://doi.org/10.1007/978-3-662-44874-8 Gebeurtenis: 2nd edition.

[4] Larry J. Eshelman and J. David Schaffer. 1993. Real-Coded Genetic Algorithms and Interval-Schemata. In *Foundations of Genetic Algorithms*, L. DARRELL WHITLEY (Ed.). Foundations of Genetic Algorithms, Vol. 2. Elsevier, 187–202. https://doi.org/10.1016/B978-0-08-094832-4.50018-0

[5] K.O. Stanley and R. Miikkulainen. 2002. Efficient evolution of neural network topologies. (2002), 1757-1762 vol.2 pages. https://doi.org/10.1109/CEC.2002.1004508