Genetic Fuzzy and Neuro Evolution: Genetic Optimization Algorithm Analysis

Tom Pawelek, *tp1333@msstate.edu*

***Abstract*—In this document, I present my applied study of evolutionary algorithms used in fintech. This is a continuation of my two previous papers: ”Genetic Evolution of Expert AI Systems in Financial Underwriting” [1] and ”Fintech Use of Genetic Algorithms” [2]. Here, I present a more structured and mathematically formal approach to evolutionary training complexity and explain why it is a good fit for my particular applications. I also emphasize the duality of genetic training, with chromosomes responsible for parameters encoding and fitness functions acting as black boxes representing the underlying business logic.**

***Index Terms*—genetic algorithm, artificial intelligence, time complexity, space complexity, genetic fuzzy, genetic expert systems, neuro evolution**

I. Introduction

Financial industry was one of the first adopters of Artificial Intelligence. With ample structured data (ledgers, records, audit logs) and clearly defined goals (underwriting, performance, margins), it is an excellent candidate for supervised machine learning.

Fig. 1. John Holland - University of Michigan

In the 80s and 90s, the dominant approach was to use expert systems based on fuzzy logic, and then run business cases through its decision trees and probability distributions. Recently, most of the focus has been on neural networks in their various shapes and forms.

Both classes of algorithms have one thing in common: non-trivial training. Despite vastly different approaches (expert systems relying on extraction of human knowledge and neural networks utilizing gradient descent feedback loops), both remain costly and resource-intense.

II. Genetic approach

Similar to neural networks taking inspiration from a biological construct, researchers in the 1960s mimicked another natural phenomenon: evolution. This resulted in 1975 publication [3] by John Holland (Fig. 1) formalizing the idea. What makes them unique is that they de-couple an operational process of an existing algorithm from its training procedure. By saving the underlying parameters into a continuous data stream (referred to as a chromosome, because it matches the role of RNA/DNA), we can focus on the evolutionary improvement of their performance without any dependency on the underlying business logic (represented as a fitness function).

Originally applied to engineering, scheduling and design, this genetic approach proved to be equally efficient at training more computationally-intense protocols [4], including neural networks (NEAT [5]). In that case, weights and biases are encoded on the chromosome which then undergoes regular evolutionary training. Once optimized, they are applied back to the neural polynomials and resulting network can be used as usual.

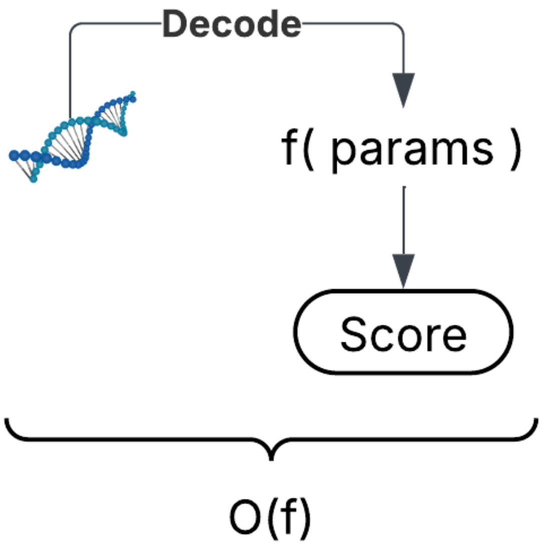

Fig. 2. Evaluation of Fitness Function

Similarly, in my previous MSU research, I was able to apply this approach to expert systems, by representing the values of their decision trees on a 42-bit chromosome [1]. When applied to real-life loan portfolio, a genetically trained system outperformed a reference platform configured using traditional knowledge extraction.

This flexible training process is the main motivation for my continuous research of the evolutionary approach. Not only does it provide a training layer for any abstract problem, where parameters can be binary encoded (fuzzy), but it also adds robustness and resource saving to other algorithms which are traditionally very expensive to train and prone to local optima (neural).

III. Complexity Analysis

In order to establish time and space complexities for our case studies, it is important to note that we consider our fitness function (f) as a black box of O(f).

Figure 2 shows a single execution of our function. There are a couple of reasons that justify this approach:

- The focus of this paper is on the genetic algorithms themselves, not the underlying logical payload.
- Fitness function evaluation cost does not depend on the genetic composition of our training setup (as defined below).
- In real-life environment, we tend to consume (not train) computationally heavy algorithms on ASIC hardware.

Our "genetic setup" therefore consists of the following variables:

- chromosome length (c),
- population size (n),
- number of generations (g).

Those parameters are fine-tuned to optimize the training process, but they do not affect the underlying network being trained (expert system or neural).

In the following section, I will describe the training process and derive base formulas for time and space complexities. Later on, I will show their practical counterparts based on real-life applications.

For my analysis, I break the process down into simple elements, resulting in direct O(x) complexities: lower and upper bounds are the same.

IV. Time Complexity

To optimize any algorithm using evolutionary approach, we take the parameters of an underlying system (weights, biases, probabilities, tree conditions, etc.) and represent them in a continuous binary variable, called chromosome. A fitness function f represents a single run of this underlying business logic given a particular chromosome, as shown in Figure 2.

The training process consists of multiple steps:

A. Cross-Over

Two chromosomes are picked and they exchange parts of their bits at a given cross point. This location can be fully random (which might result in encoded information being split) or it could be random with consideration for information binary length (equivalent to DNA gene). The process is shown in Figure 3.

In either case, this process does not depend on chromosome length, therefore it's **O(1)**.

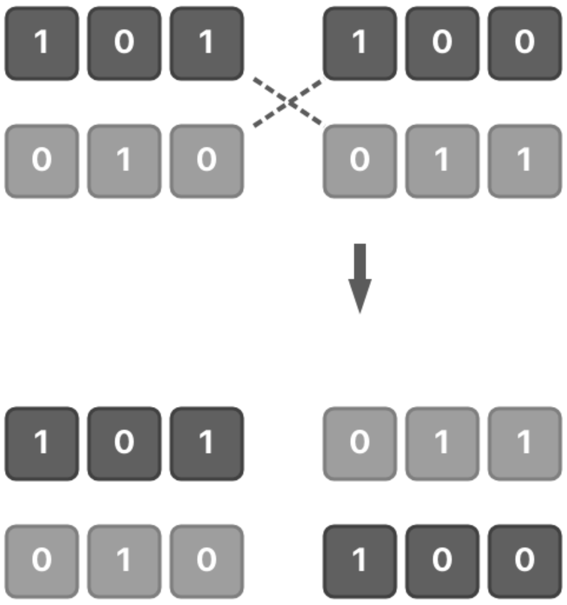

Fig. 3. Genetic Cross-Over

B. Mutation

To increase variability and ensure all possible solution space is evaluated, we can introduce mutation. When undergoing mutation, rantom bits of a chromosome have their values flipped between 0 and 1.

As with the cross-over, this process has as fixed complexity of **O(1)**.

C. Survival of the fittest

In order to determine which chromosomes produce off-spring, we need to evaluate the fitness of each one of them. This is done by executing of the fitness function f with parameters provided by a given chromosome.

As established, each call of function f costs us **O(f)**.

D. Entire population

We need to iterate this evalution across the whole population of chromosomes. Given its count of n, this yields a cost of **O(n)**.

E. Multiple generations

Finally, the whole process is repeated for each generation. We have g generations to analyze, therefore a complexity of **O(g)**.

Time complexities of each step are shown in Table I.

Combined, this results in a Time Completixy of:

$$O(f * n * g) \tag{1}$$

Step	Time Complexity
Cross-Over	O(1)
Mutation	O(1)
Fitness evaluation	O(f)
Population iteration	O(n)
Multiple generations	O(g)

TABLE I
GENETIC TIME COMPLEXITY

Step	Time Complexity
Cross-Over	O(1)
Mutation	O(1)
Fitness evaluation	O(c)
Population iteration	O(n)
Multiple generations	O(1)

TABLE II
GENETIC SPACE COMPLEXITY

V. SPACE COMPLEXITY

Same as in previous section, I am going to only focus on the evolutionary part of the training: in this case, we do not allocate any resources to execution of the fitness function. The reasoning remains the same - with the added notion of most commercially viable algorithms requiring vastly larger memory resources than the genetic optimization itself.

With that in mind, memory requirements for our process are as follows:

A. Cross-Over

We can edit chromosomes in-place, resulting in **O(1)**.

B. Mutation

Same as with Cross-Over, we edit in-place, therefore yielding **O(1)**.

C. Survival of the fittest

To run a fitness function we need to store its chromosome values, which gives us **O(c)**.

D. Entire population

Each individual has a separate chromosome, so our whole population requires **O(n)**.

E. Multiple generations

Finally, even though we run the process g times, we can discard the results of any previous generation.

Effectively, we only keep track of one (current) generation, which gives us **O(1)**.

Space complexities of each step are shown in Table II.

Combined, this results in a Space Completixy of:

$$O(c * n) \tag{2}$$

VI. ANALYSIS OF FORMAL COMPLEXITIES

Our overall formal analysis yields time complexity of O(f * n * g) and space complexity of O(c * n). Before we look at practical application and real-time equivalences of both, we can formulate some preliminary observations based on those values alone.

As my experiments have shown, in financial realms, optimization usually plateaus fairly quickly, with g ranging from 20 to 200. Similarly, chromosome size cannot be too large, as it would require exponentially bigger population size for a meaningful fitness comparison.

Therefore, we can see that the single most impactful variable at this stage is the generation size (g), appearing as a multiplication factor in both time and space complexities. The following sections will further reaffirm this conclusion.

VII. PARALLELIZATION

One of the reasons why I find genetic optimization a good fit for modern algorithm optimization is how easy it is to parallelize.

The main reasons are:

- Every fitness evaluation is independent from another.
- Chromosomes are of minimal size, making them trivial to distribute in a message queue.
- Similarly, once a fitness score is calculated, we only need to return that value back to the pool.
- For heavier algorithms (such as neural network), there are affordable ASICs found in consumer grade hardware (GPUs, ARM CPUs, etc.)

A. Parallel Time Complexity

Assuming we have n ASIC cores optimized for execution of function f (which can be achieved with consumer grade hardware [6]), we can perform the following simplification of our initial time complexity of O(f * n * g).

First, the business logic execution can be considered fixed and independent from our genetic environment (f = 1), simplifying it to O(n * g).

Then, since we can run all fitness evaluations at the same time, we can also substitute n = 1.

Overall, our real-life parallel time complexity becomes **O(g)**. This matches our finding from the formal analysis.

B. Parallel Space Complexity

In a similar way, we begin with base space complexity of O(n * c). But since we run on n cores at the same time, we can substitute n = 1, giving us O(c).

However, we mentioned before that realistic chromosomes are fairly limited in size, due to exponential growth of population size they would require. In my analysis, I found 50 bits to be a realistic threshold. As such, we can also treat this variable and fixed and substitute c = 1.

This gives us final space complexity of **O(1)**. Indeed, with separate fitness function execution, the memory requirements for genetic algorithms are almost negligible.

Complexity type	Theoretical	Practical
Time	O(f * n * g)	O(g)
Space	O(c * n)	O(1)

TABLE III
COMPARISON OF THEORETICAL AND PRACTICAL COMPLEXITIES

Table III shows a comparison of both theoretical and practical complexities. Those findings match my real-life implementations - where number of generations is the main determinant of how long an evolutionary optimization is going to take.

VIII. Practical Considerations and Conclusions

The complexities that were derived in previous sections truthfully represent applied approach to evolutionary optimization.

A. Application with Expert Systems and Neural Networks

This approach is an excellent training supplement to well established algorithms:

- **Expert Systems** - genetic optimization provides a way to easily train them in an unattended setup. Not only does it reduce a cost vs knowledge extraction from Subject Matter Experts (SMEs), but it also provide a more reliable timeline of the process.
- **Neural Networks** - a properly chosen chromosome size can help avoid local optima and analyze a wider range of solution domain. It also requires less resources that traditional backpropagation due to abundance of dedicated neural execution hardware.

B. Chromosome size

As mentioned before, I found the chromosome size of under 50 bits to be the optimal for most of my real life applications. While larger sizes are possible, they usually result in exploding population size, which dramatically increases the cost of optimization.

One way of addressing this issue, is to copy mother nature again - rather than slicing chromosomes at random binary locations, we can introduce a concept of genes and cross-over those entire data chunks.

In this case, we usually require a larger mutation rate to allow for a complete discovery of result domain.

C. When to use it, when not to use it

In my experience, the fields which are best suited to benefit from genetic optimization are:

- Fintech
- Information Security
- Engineering

Basically, we want our data to be structured, somewhat discrete (due to binary encoding of chromosomes) and of limited range. Those requirements allow for great levels of autonomy - in fact, genetic training has been shown to deliver great results in training artificial players of computer games [7].

For tasks with larger granularity of parameters, or even continuous values, such as generative imagining or weather forecasting, backpropagation might be a better choice. Many researchers set the threshold at under 1,000,000 - 10,000,000 weights [8] of a neural network to be a good candidate for genetic optimization.

D. Future Research

Based on previous successful deployments of evolutionary algorithms, I am planning to continue my research of their fintech applications.

In particular, I am currently evaluating the following cases:

- Evaluation system for receivables ERV (Expected Recovery Value)
- Underwriting automation for returning credit customers
- Early warning system for credit default in retail

Considering the ease of use, with fitness function being fully ring-fenced from optimization environment, I am certain there will be other applications where genetic algorithm will provide a better return for investment vs their traditional counterparts.

References

[1] T. Pawelek, “Genetic evolution of expert AI systems in financial underwriting,” https://tompaw.net/genetic-evolution-of-expert-ai-systems/, January 2025, mSU research paper on evolutionary training of CLIPS fuzzy expert systems for consumer loan underwriting. [Online]. Available: https://tompaw.net/wp-content/uploads/2025/01/genetic_experts.pdf

[2] ——, “Fintech use of genetic algorithms,” https://tompaw.net/fintech-use-of-genetic-algorithms/, May 2025, mSU seminar paper on genetic-assisted AI in fintech applications including ledger forecasting, anti-fraud, and forensic audits. [Online]. Available: https://tompaw.net/wp-content/uploads/2025/06/fintech_gen.pdf

[3] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press, 1975.

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[5] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[6] J.-R. Cheng and M. Gen, “Accelerating genetic algorithms with gpu computing: A selective overview,” *Computers & Industrial Engineering*, vol. 128, pp. 514–525, 2019.

[7] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, “Designing neural networks through neuroevolution,” *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019.

[8] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, “Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning,” *arXiv preprint arXiv:1712.06567*, 2017.