EvoTank: Optimization of Video Game Artificial Intelligence Via Evolution

Justin Bishay

May 16, 2020

Abstract

Artificial intelligence (AI) has been a significant part of video games ever since their inception and continues to have profound importance in games today. These AI systems often require a large amount of work to properly tune for the gameplay experience. Many hours and people are needed to not only develop but also test AI in games. EvoTank is a simple 2D tank game developed to test whether the optimization of a game's AI can be automated by utilizing a genetic algorithm to evolve the AI.

1 Introduction

1.1 Artificial Intelligence in Video Games

Artificial intelligence is a crucial component of video games, especially in the modern day. The creation and fine tuning of AI systems is extremely important in most video games in order to achieve high levels of immersion and generate quality gameplay experiences. The process of optimizing such AI systems can be quite arduous and require much guesswork since the best parameters for an AI created from the ground up for a game that is also created from the ground up is not immediately known.

Most video games do not make use of advanced machine learning models that can be trained on player data. While these machine learning models are excellent for teaching AI how to play games at superhuman levels, they often become nearly unbeatable and lead to a less enjoyable gameplay experience for players [1]. Thus, it is actually desirable for AI in video games to be imperfect in terms of its performance.

1.2 Game AI Optimization

Much play testing is needed when designing an AI system in a video game. Developers already spend a hefty amount of time playing through their games during development to test various aspects of the game. Some studios even reach out to the public to gather more people to act as playtesters. A tremendous amount of quantitative and qualitative data is gathered from the play tests and used by the developers to make adjustments to the components of their game including its AI. If some parts of this process were to be automated in some way, it could lead to a great reduction in the time and resources needed to gather the essential data needed to improve the game. The goal of this paper is to see if an evolutionary approach could be used to automate the optimization of a game's AI.

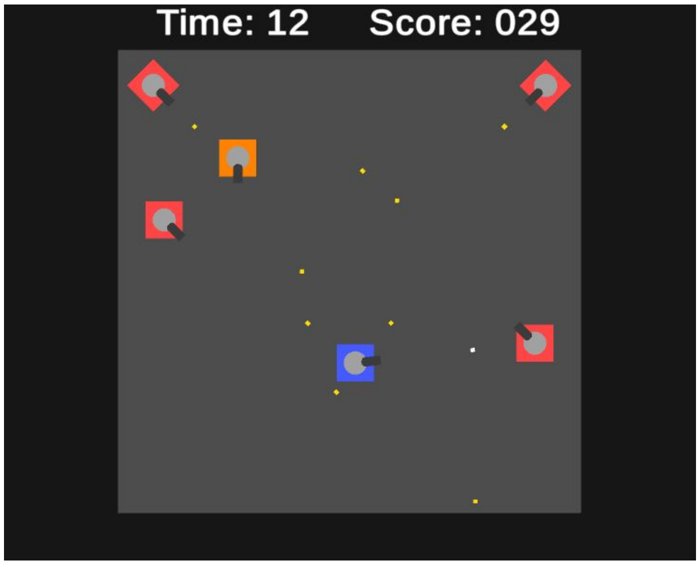
1.3 Related Work

Other studies have been conducted with AI in games. IBM constructed an AI system in 1997 that managed to win against a grandmaster player in chess. However, research like this is often done more so with the intention of using a game to perform a benchmark test on the AI [2]. Similar research has been done with the use of machine learning models such as neural networks to learn how to play games. One such neural network, MarI/O, uses evolution to construct a neural network that is capable of learning and playing a 2D mario game [3]. However, MarI/O focuses on learning the game, whereas this paper is examining the optimization of a more rigidly constructed AI component in the game.

2 Design

2.1 The Game

The evolutionary optimization will be performed on an AI player in a 2D tank game called EvoTank. The tank will be placed in an enclosed arena where it will battle against a constant barrage of enemy tanks for 60 seconds. The AI tank's score will be recorded either when the tank loses or when it reaches 60 seconds. The AI tank player will be restricted to moving up, down, left, or right and will receive 7 points for every 5 seconds it survives and 3 points for each enemy it defeats.



The enemy tanks will either be stationary tanks that sit in one spot until destroyed, or mobile tanks that constantly move in a straight line and can exit the arena without being destroyed. The game will run a live simulation of the battle on 20 AI tanks for each generation. To ensure consistent results between each of the tanks, the sequence of enemies will be the same across all tanks and generations.

The parameters of the AI tank are divided into two categories: static parameters and adjustable parameters. Static parameters are traits that can be adjusted but will remain constant throughout the evolution. The static parameters are the tank's reload speed and movement speed. The reason these two are static parameters is because the optimal setting for these parameters is logically intuitive; a faster movement speed allows the tank to evade more successfully, and a quicker reload speed allows the tank to eliminate more enemies in a shorter amount of time. Thus, it is presumably not worthwhile to adjust them through evolution. Instead, the static parameters will be considered the natural abilities of the tank that it will have to evolve around.

The adjustable parameters are the traits that will be adjusted via evolution. This includes its movement behavior, radius to detect danger, target prioritization, and shooting angle. Should the tank move randomly when not in danger, or should it only move when necessary? How soon should the tank detect approaching danger; is it possible for the radius to be too far? Should the tank shoot the closest or furthest enemies first, or is random selection of targets actually better? When there are moving targets, how far ahead should the tank aim based on the current movement vector of the target? These are all the questions the evolutionary algorithm aims to answer for the AI tank.

The adjustable parameters will have the following ranges:

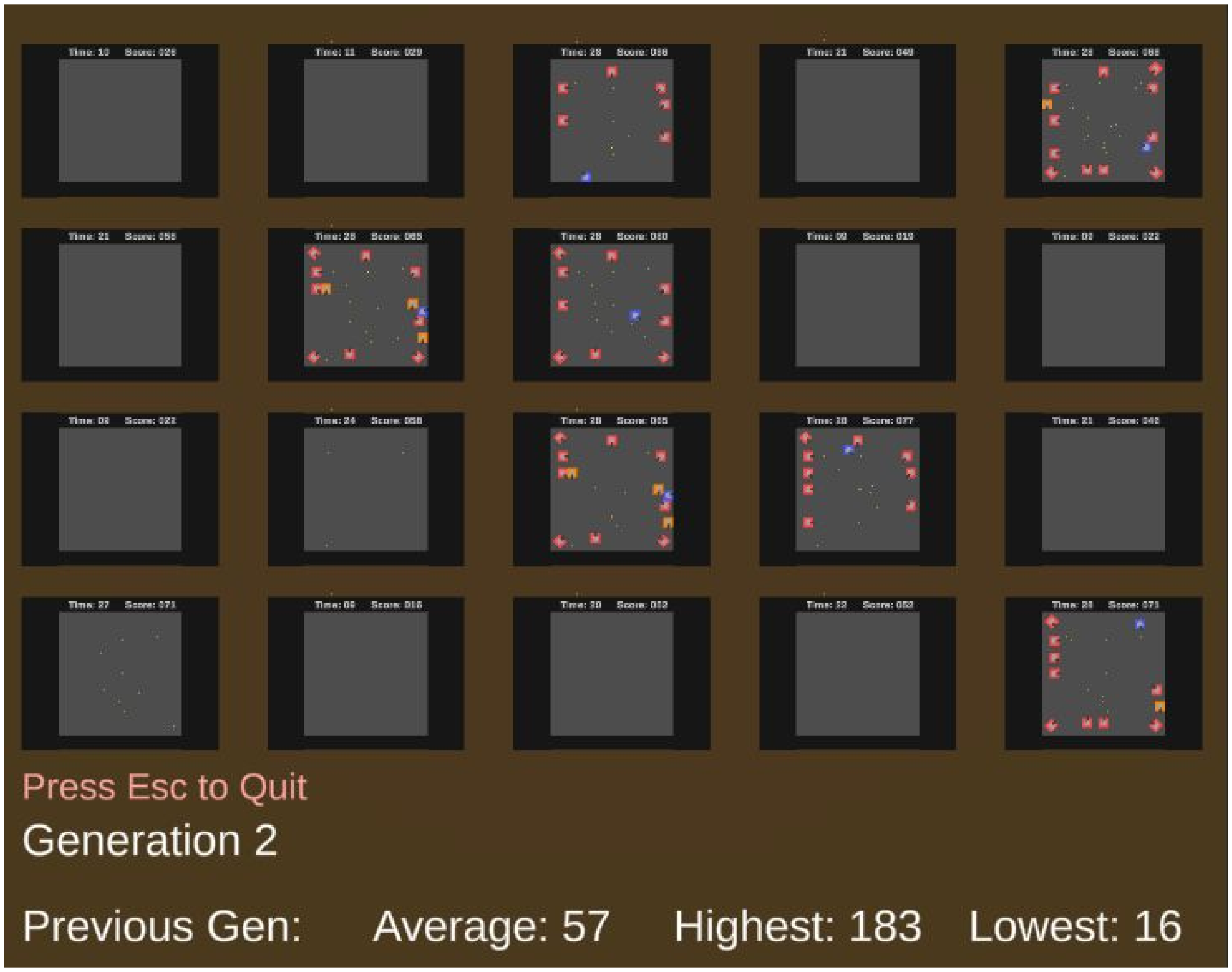
- Movement Behavior
    - 0 (Random) or 1 (Necessary)
- Detection Radius
    - Range of decimals from [1, 4]
- Target Prioritization
    - 0 (Random), 1 (Closest), or 2 (Furthest)
- Shot Angle
    - Range of decimals from [0.25, 3]

2.2 Search Space

The search space is defined as the set of all possible combinations of the adjustable parameters for the AI tank. The goal is to optimize the performance of the AI to find what the best combination of parameters is for it. The fitness of each AI tank will be the score it achieves.

2.3 Population Size
Every generation will have 20 individuals in its population. The population will have its parameters randomized at the start. The population will continue undergoing evolution until it goes through 10 iterations where no new optimum has been found.

2.3 Selection
Ten individuals will be selected to be parents for reproduction at the end of the generation. This selection will be done via tournament selection which will compare 2 individuals at a time and select the one that is more fit. This guarantees that the best member of the population is selected for reproduction. The 10 parents will form random pairs and will crossbreed 20 children for the new population. The crossbreeding is simply done by randomly selecting one parent's parameter value for the child's respective parameter. The child's fitness will be initialized as the average fitness of its parents.



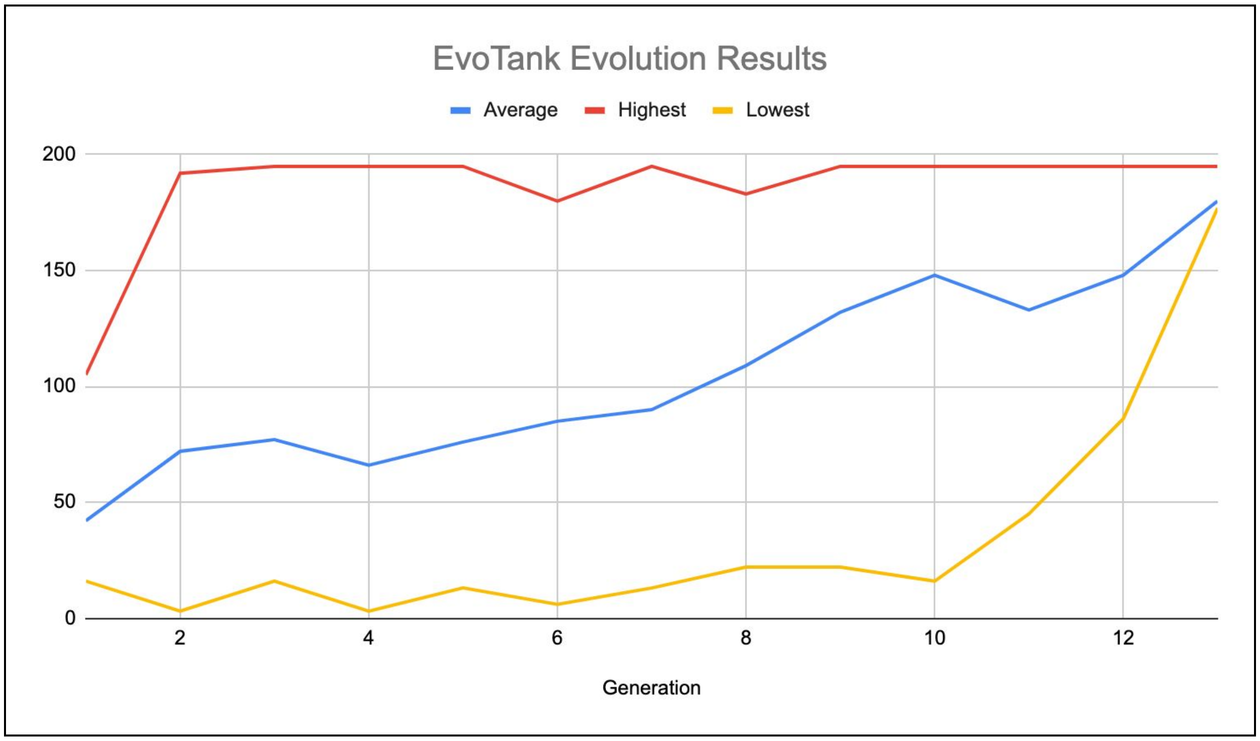
2.4 Mutation
Following reproduction, each individual in the new population will have a 10% chance of mutation applied to each of its parameters. If a parameter mutates, it will do so by means of k-nearest neighbors mutation. This mutation method is done by finding the k members in the previous generation that have the smallest euclidean distance when compared to the child's fitness and parameter undergoing mutation. For example: if a child undergoes mutation for its detection radius, the euclidean distances for each member of the previous generation are computed as $\sqrt{(child.fitness - elder.fitness)^2 + (child.radius - elder.radius)^2}$. The new value is the mean of the parameter values of the k-nearest neighbors.

The k-nearest neighbors mutation approach has been shown to accelerate the convergence of genetic algorithms [4]. To ensure that the algorithm does not converge on suboptimal parameters, only neighbors that have fitness values greater than the child's will be considered. If there are fewer than k such neighbors, then the algorithm will take the mean of however many neighbors with a higher fitness exist. In the case that there are no neighbors with a higher fitness, the algorithm will select a random value outside of the range of values that were present in the previous population. If a population's detection radius values ranged from [1.12, 2.78], for instance, the new detection radius value would be randomly selected from the range [1, 1.12) and (2.78, 4]. This is done with the intention of allowing the algorithm to explore parameters that have not yet been tested and avoid converging on an optimum before thoroughly examining all possibilities.

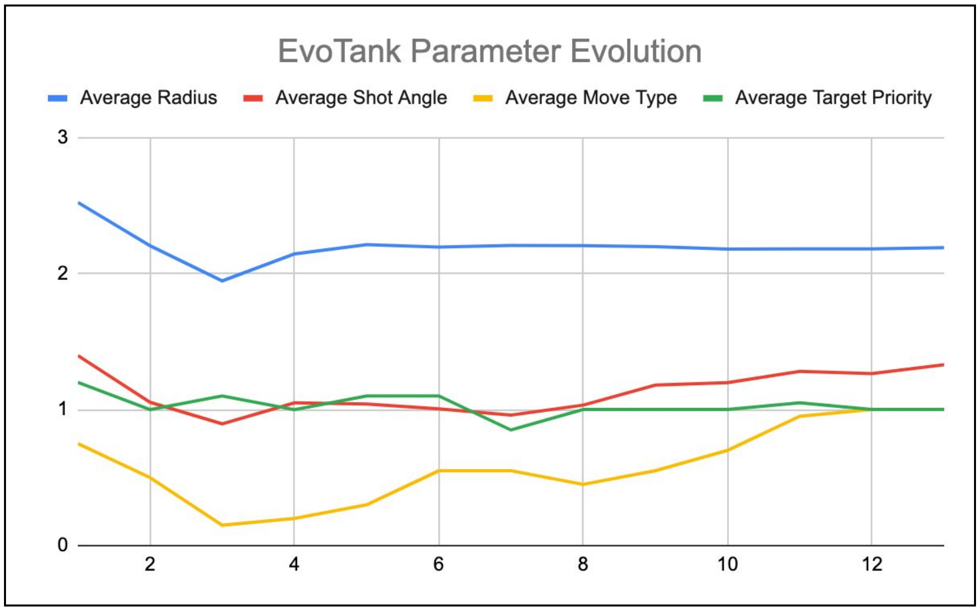
The k value that will be used for mutation will be 3. This value was determined after running multiple tests on the algorithm. A k value of 3 enables the population to consider a wider range of values when mutating.

3 Results

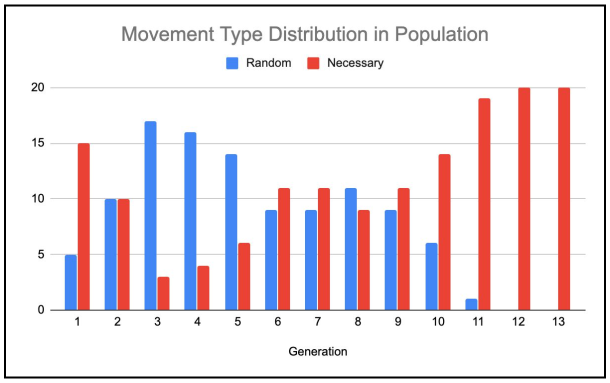
The game ran for 13 generations and reached a high score of 195 points. The AI's parameters converged on a value of 1 (necessary) for movement behavior, a value of about 2.18 for the detection radius, a value of 1 (closest) for the target prioritization, and a value of about 1.11 for the shot angle. The AI actually managed to achieve high scores in earlier generations when either the movement behavior or target prioritization were set to 0 (random). These early high scores often emerged as outliers as their respective generation's average was much lower. Though this does provide interesting insight to how random behavior can lead to desirable results but is generally less reliable at doing so.

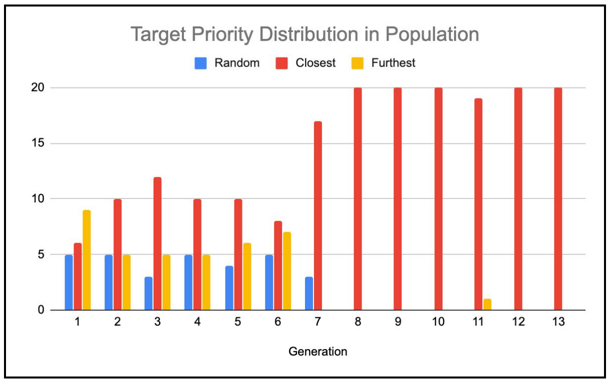


Although the algorithm managed to reach the optimum by the 3rd generation, the average fitness of the population increased at a much slower rate. The above graph shows how the population gradually converges on the discovered optimum over time. The worst member of each population maintained very low fitness throughout most of the evolution. It was not until the 11th generation that its fitness began to sharply improve.



The first two parameters that converged on their optimal value was the detection radius and the target priority. Both of these parameters started to settle on their eventual optimum around the 4th generation which was about the same time the optimal score was first reached. The average shot angle and movement behavior continued to fluctuate until the 11th generation. The average and lowest fitness of each population saw the most amount of improvement between generations 8 and 13 which is when the average shot angle and movement behavior values began to converge toward their respective optimum. This may suggest that the shot angle and movement behavior are the more influential parameters on the AI tank's performance.





The distribution of movement behavior and target priority values in the populations may also support this claim. There was a fairly even split of random and necessary movement behavior tanks until the 10th generation where the majority of the population shifted toward necessary movement. This was coincidentally around the time when the average fitness was experiencing its greatest increase, and the lowest fitness began its sharp rise in the 11th generation when almost all of the population had shifted to necessary movement. Conversely, the target priority of the tanks dramatically shifted toward closest target priority at the 7th generation. At this point of the evolution, the average fitness was increasing at a slower rate than it did in later generations and the lowest fitness had seen minimal improvement.

4 Limitations

It is important to note that this test was performed on the same sequence of enemy encounters every time. This means that the resulting AI is optimized specifically for this setting. It is unclear how the AI would perform in a foreign or randomized environment with the parameters that were selected through its evolution. The MarI/O neural network designed to learn and play 2D mario games encountered a similar problem. The neural network managed to learn how to play the first level of various 2D Mario games, but it often struggled and failed to complete the second level of these games [5]. It may be worthwhile to run the evolutionary algorithm for the

AI tank on enemy sequences that are completely randomized every time. Attempting to optimize the AI tank on random enemy sequences would expose it to more unique scenarios which could lead to a more well rounded selection of parameters. However, this would need to run for a longer amount of time to avoid the AI settling on suboptimal parameters.

5 Conclusion

The AI tank player managed to modify itself through evolution to achieve a higher score on a specific level. Though the true performance of the optimized AI tank would need to be evaluated across numerous levels, it is very likely that the set of parameters found through evolution will perform better than randomly selected parameters majority of the time.

The most interesting take away from the study was that the AI tank's optimized parameters led it to consistently execute a strategy where it would defeat enemies in a specific order and ultimately found a spot in the arena where it could sit still without being threatened. This essentially exposed an exploit in the level where the tank did not need to ever dodge after a certain point, and any player could use this to their advantage if discovered. Such a tactic could take countless trials by numerous players to finally be discovered, yet the AI managed to discover this during evolution and actually optimized itself for it. This reveals another useful application of optimizing AI systems in games–uncovering design flaws.

Such design flaws are not always easy to find through play testing. The exploit in EvoTank is the result of eliminating enemies in a specific order that leaves a spot on the right side of the arena becoming safe from enemy fire around the 20 second mark. Even after that, this spot only remains safe by continuing to target enemies a certain way. The problem could be resolved by tweaking the enemy sequence to ensure an enemy is present to fire at the space where the AI tank player sits still to force it to keep dodging. However, it is likely this will just lead the AI tank to find a different set of parameters that leads it to another exploitable strategy. Ultimately, developers will need to examine such outcomes to determine whether or not these occurrences are acceptable.

References

[1] L. Maass, A. Luc, “Artificial Intelligence in Video Games,” *Towards Data Science*, July, 2019. https://towardsdatascience.com/artificial-intelligence-in-video-games-3e2566d59c22

[2] N. Statt, “How Artificial Intelligence Will Revolutionize the Way Video Games are Developed and Played,” *The Verge*, March, 2019. https://www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning

[3] S. Hendrickson, “MarI/O - Machine Learning for Video Games,” June, 2015. https://www.youtube.com/watch?v=qv6UVOQ0F44&t=48s

[4] G. Liu, C. Wu, “Differential Evolution with k-Nearest-Neighbour-Based Mutation Operator,” *International Journal of Computational Science and Engineering*, Vol. 19 No. 4, pp. 538 - 545, 2019. doi: 10.1504/IJCSE.2019.101884

[5] S. Hendrickson, “MarI/O Followup: Super Mario Bros, Donut Plains 4, and Yoshi's Island 1,” June, 2015. https://www.youtube.com/watch?v=iakFfOmanJU