Self-Driving Cars Using Genetic Algorithms and Neural Networks

James Peralta
Department of Computer Science
University of Calgary
Calgary, Canada
james.peralta@ucalgary.ca

Albert Choi
Department of Computer Science
University of Calgary
Calgary, Canada
albert.choi1@ucalgary.ca

Nathaniel Habtegergesa
Department of Computer Science
University of Calgary
Calgary, Canada
nathaniel.habtegerge@ucalgary.ca

***Abstract*—Throughout the project, we developed a self-driving car that is controlled by a neural network. A genetic algorithm (GA) was used to train the weights of the neural network and every part of the neural network was implemented from scratch using Unity. We created a sandbox in Unity to allow users to test these neural networks. This sandbox includes a car attempting to navigate a race track. If it hits a wall it will disappear and spawn a new one. Over time you will see that the car will get better and better at navigating the race track. We developed a 2D simulation where we tested the genetic algorithm and neural network before creating our 3D simulation. As a result, our system can have a car learn to successfully navigate through a track and overcome the difficulties of curves, hills, and slopes.**

***Index Terms*—autonomous vehicles, genetic algorithms, neural networks, unity**

I. Introduction

When it comes to self-driving cars, the future was supposed to be now, but self-driving cars turned out to be harder than people expected. Self-driving cars, also known as autonomous or "driverless" vehicles have been prevalent for decades and are poised to revolutionize the transportation industry. An autonomous car is a vehicle capable of sensing its environment and operating without human involvement. Having self-driving cars could give people more free time, increase mobility for disabled individuals and potentially reduce the number of road accidents. Various self-driving technologies such as lane correction, potential collision detection, and automated parking have already been developed by Tesla, Google, Uber, and many more major automakers. Autonomous vehicles have come a long way. Uber's self-driving prototypes use sixty-four laser beams, along with other sensors, to construct their internal map; Google's prototypes have, at various stages, used lasers, radar, high-powered cameras, and sonar. Despite extraordinary efforts from many of the leading names in tech and in the automaking industry, fully autonomous cars are still out of reach except in special trial programs.

In our project, we are using Unity to create a genetic algorithm (GA) to train the weights of a neural network that will drive a car. Inspired by evolutionary principles this will allow us to make training more efficient and effective. Our system converts sensor data into actions. This way the car can react to its changing environment and make the right real-time decision. There are different types of optimization functions used for Artificial Neural Networks (ANNs), including derivative optimizations and derivative-free optimizations. Derivative-based optimizations (DBOs) iteratively improve on the parameters by using the derivative to find the best search direction and it essentially climbs the hill towards the global maxima. DBOs have been shown to have problems when there are disconnected areas in the search space and when the search space is multimodal causing the algorithm to settle in local maxima, instead of the global one. DBO techniques have been very popular for training the latest and greatest neural networks and the most popular DBO technique is the backpropagation algorithm. Derivative free optimization (DFO) is also a mathematical optimization technique but doesn't require derivative information to optimize a function. DFOs have been used to address some of the challenges that come with DBO techniques but have mostly been stuck as an active research topic. Motivated by the potential of this emerging topic we have decided to use a popular DFO algorithm (a genetic algorithm) to find the optimal weights of a neural network that will be used to drive a car in Unity.

The general purpose of our project was to have a car learn to complete the track without crashing. In this state, it was not enough to just make the car drive itself. To make it drive we needed to train a neural network to optimize the weights and that's where genetic algorithms came in. Genetic algorithms are a promising answer because it can create multiple solutions to a given problem and evolves them through several generations. Each solution holds all parameters that might help improve the results, in our case these were the weights of the neural network. We will be discussing our success and walking you through the process of how we achieved these results.

Throughout this paper, we demonstrate how genetic algorithms can be used in conjunction with neural networks to drive a car. We will also be describing a series of experiments that we performed in Unity. Our paper consists of some examples of similar work done in the field, the design of our simulation, an explanation of our algorithm, results we obtained through running our simulation and finally, the conclusion.

II. Related Work

Some of the earliest literature on genetic algorithms used to optimize the weights of an Artificial Neural Network date back to the 1990s and is still very alive in academia and work that's done today [1]. A great example is David J. Montana [2] who was able to demonstrate that a GA was capable of outperforming the backpropagation algorithm on a sonar image classification problem. Their GA contained mutations and crossovers when searching for the correct weights, but they also introduced a gradient operator. Kinjal Jadav [3] also attempted to use a GA to optimize weights and obtained some helpful results. He was able to show that a very promising way to breed generations is to use a Tournament selection method with Elitism to select the best individuals from a given population. He also showed different ways of encoding chromosomes to improve the efficiency of the genetic algorithm.

Another interesting variation to the same problem was presented by Dario Floreano [4] in 2004. Using computer visions, he created a model that visually recognizes edges, corners, and height, rather than proximity sensors. Dario implemented a GA for tuning the neural network. The system consisted of a feedforward neural network of artificial neurons with evolvable thresholds and discrete-time, fully recurrent connections at the output layer. Their outputs would encode acceleration, braking, and steering parameters of the car. The neural network was composed of 25 visual neurons (arranged on a 5 x 5 grid) and two proprioceptive neurons. Dario's work began resembling strategies observed in simple insects which obtained great results. The performances of best-evolved individuals were equal to or better than those of well-trained human drivers tested on the same circuits.

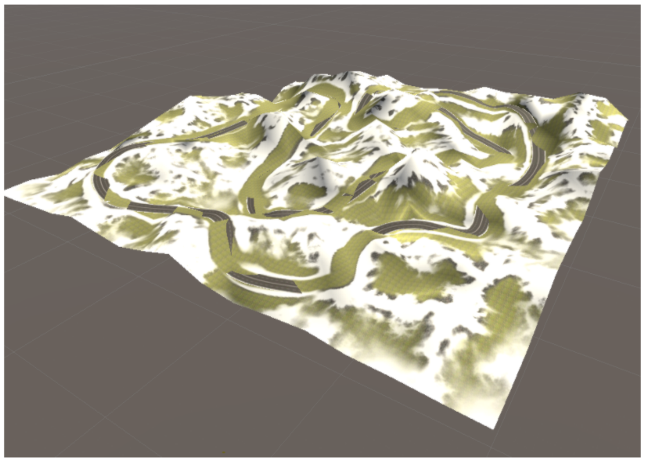

Fig. 1. 3D track that a candidate solution is required to complete.

III. Methods

The main goal of our research was to train a neural network to autonomously navigate a track without crashing. The most popular way to train neural networks is through back propagation but this requires a labeled dataset which is very difficult and time-consuming to create for this problem. A genetic algorithm was the most attractive approach because it doesn't require a labeled dataset, it only requires an environment to test candidate solutions. Our experiment consisted of two components: developing a genetic algorithm to evolve candidate solutions and creating a simulation to test these solutions.

A. Simulation

We built the simulation using Unity to leverage its pre-built physics engine. This allowed us to create a realistic driving environment with less overhead. Our simulation contains a 3D track imported from the Unity store [5] that consists of a road and barriers on each side of the road (see Fig. 1). The road is the surface that each car can drive on and the barriers are used to keep cars within the road. When a car crashes into a barrier, it is stopped. In addition to these components, we created checkpoints which are invisible markers spread out evenly across the course. The track contains sharp turns and steep hills (see Fig. 2) which introduced an interesting set of challenges that candidate solutions had to adapt to. If a car is accelerating to fast around the turn they will not be able to complete it. The steep hills require a car to gather enough speed to successfully climb it and apply sufficient breaking when rolling down to avoid gaining too much speed.

We imported a vehicle from the Unity store [6] that contained the body and wheels of the car (see Fig. 3) but did not contain any movement scripts. Using the techniques shown in [7], we were able to simulate the acceleration, braking, and turning of the car. Afterwards, we added proximity sensors to the car which gave it the ability to probe its environment. Three proximity sensors traveled 30 meters north, northeast, and northwest of the car. These proximity sensors were implemented using Unity raycasting [8] and return a value between 0 and 30 depending on the closest barrier that the raycast collides with. The higher the value that is returned from the sensor, the closer that sensor is to a wall. For example, if a car is in open space and there are no walls around, each sensor will return a value of 0. On the other hand, if a car is speeding up towards a wall, the sensor's value will increase rapidly towards 30. The final step for the simulation was to implement the ability to generate populations of these vehicles on the same track without them crashing into one another. Unity can ignore collisions between specific assets [9] which allows us to ignore car-to-car collisions and still have each car correctly interact with the road and barriers.

B. Algorithm

Genetic algorithms require an environment, a genotype, a phenotype, operators, and a fitness function. The environment that each candidate solution will interact with is the 3D track

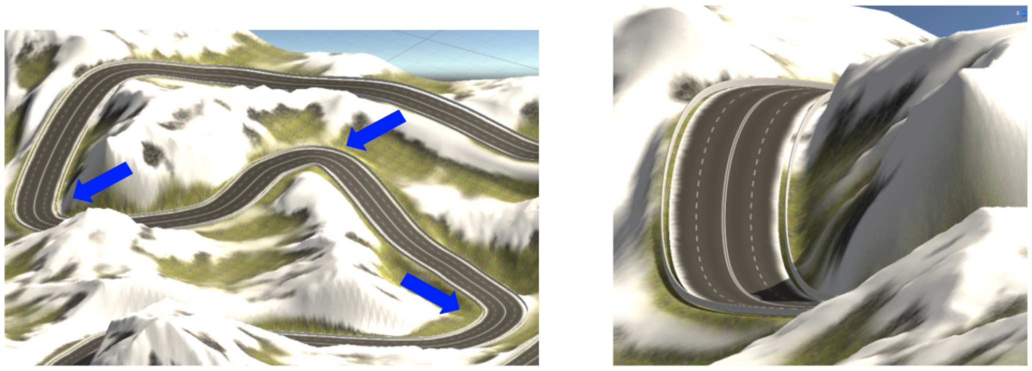

Fig. 2. Depicts the difficulty of the track which contains sharp turns and steep hills.

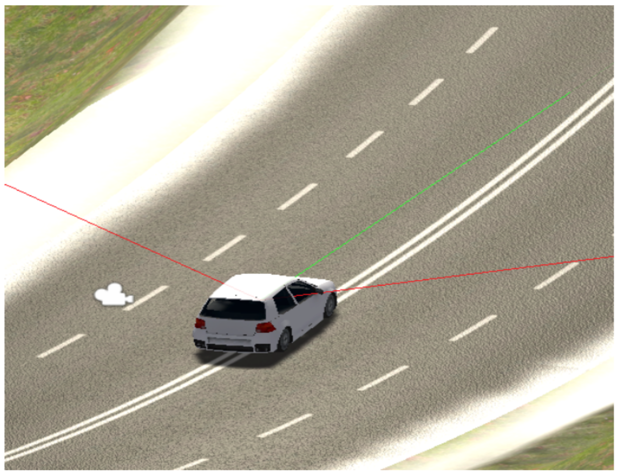

Fig. 3. A 3D car asset equipped with proximity sensors.

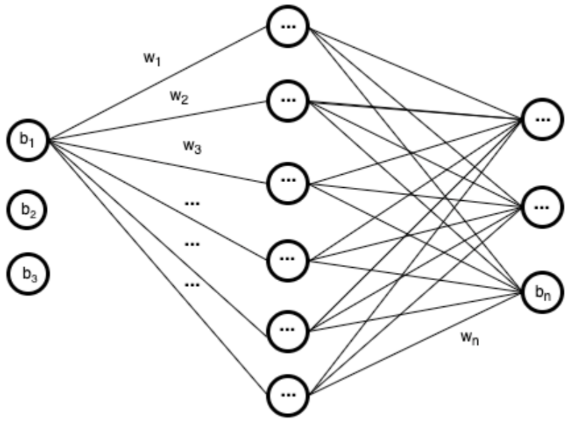

Fig. 4. Visualization of how the genome defined in (1) is converted into a configuration for the neural network instantiated in each car.

we developed. The genotype is a list of floats (1) where each element has a direct mapping to weight or bias in the neural network (see Fig. 4).

$$\left[w_1,\ w_2,\ w_3,\ \cdots,\ w_{\mathrm{n}},\ b_1,\ b_2,\ b_3,\ \cdots,\ b_{\mathrm{n}} \right] w, b \in \mathbb{R} \quad (1)$$

The phenotype is the combination of an instantiated vehicle and a neural network configured based on a given genome. The neural network architecture we settled with was a fully connected feed-forward dense neural network (see Fig. 5). Our network has an input layer of four nodes, one hidden layer of seven, and an output layer of four. The input consists of the values for the three proximity sensors as well as the speed in meters per second. The inputs will travel through the neural network in a feed forward fashion using tanh as the activation function in each neuron. There are four output neurons which represent urges to accelerate, break, turn the wheel left, or turn the wheel right. We use argmax to determine which output neuron contains the most stimulus. The urge associated with the most stimulated neuron will be applied to the car.

The checkpoints mentioned previously are used to calculate the fitness of a car. When a car crosses a checkpoint they will increase their fitness score by one. Therefore, the car which passes the most checkpoints will also be the car that has traveled the furthest and will have the highest fitness score. In terms of operations, our implementation only consists of a mutation function. We have implemented a Gaussian mutation function where each gene has a chance to be mutated based on the user-defined mutation rate. For each element in the genome, we calculate a random number $r_{\mathrm{i}} \in$ [-mutation radius, mutation radius], where the mutation radius is a float that can be set by the user. We then generate the mutated genome g' from g using (2). Our genetic algorithm utilizes a plus scheme in conjunction with elitism. The complete algorithm is as follows:

1) Generate a population of n number of cars with randomly initialized neural networks. The weights and biases are set to a random distribution of values between [-0.5, 0.5].
2) Spawn all cars into the environment.

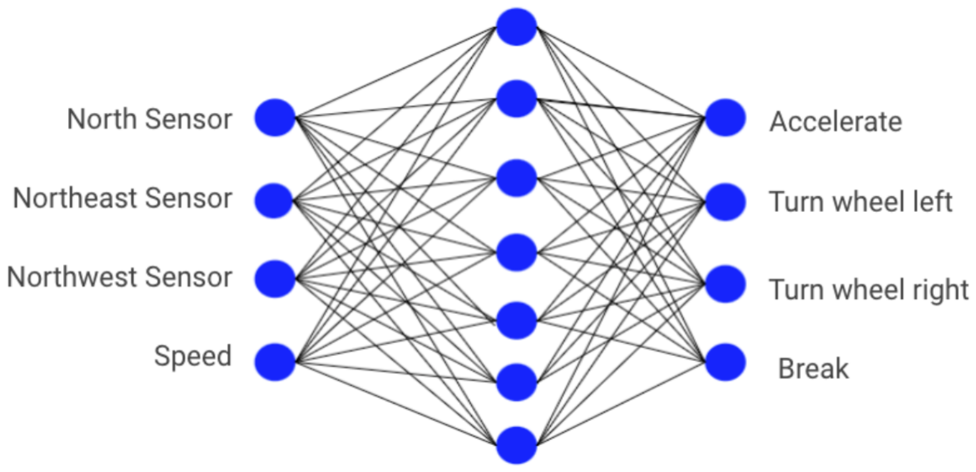

Fig. 5. The architecture of the neural network used in this project.

3) Wait until all cars are crashed or not progressing. Each car has 10 seconds to reach the next checkpoint before they are stopped.
4) Sort cars based on their fitness and select the top 50 percentile to move onto the next generation without mutation.
5) For each car selected, clone this car, mutate it, and then insert in into the next generation.
6) Repeat step 2 until a car can successfully complete the track.

$$g_{\mathrm{i}}' = \begin{cases} g_{\mathrm{i}} & \text{if } r_{\mathrm{i}} \leq \text{mutation rate} \\ g+m & \text{otherwise} \end{cases} \quad i = 1, ..., |g| \tag{2}$$

IV. Results

The main question that we set out to answer when starting this project was, "Can a genetic algorithm be used to optimize the weights of a neural network?" Our observations showed the genetic algorithm optimizing the weights so that the car was eventually able to traverse around the entire track. We observed significant improvements from the genetic algorithm tuning the neural network that was visible in our simulation. The convergence that we observed was better than our initial expectations. Initially, we were not expecting to observe such significant progress between each generation but by watching the simulation, the improvements between each generation are noticeable.

To ensure the simulation would be successful in the given time frame we had to complete the project, we developed a 2D version of the simulation in Unity before scaling up to 3D. After transitioning our project over to the 3D version, we had concerns that the car would not perform as well given that the track was a lot harder. Despite these concerns, the car was still able to complete the 3D track, due to the increased difficulty it resulted in taking more generations to converge. We also had to make a few adjustments in our neural network to take into account the 3D physics. In our 2D environment, we only used the three proximity sensors as input into the neural network. However, our 3D environment had an area on the map where there was a steep hill with a sharp turn at the bottom. When the cars had no sense of speed they would gain too much speed on this hill and were never able to complete this hill. After noticing this, we decided to add speed as an input into our neural network. This was very beneficial for the algorithm and after this addition, the cars were able to converge.

The most interesting behavior that we noticed is the "hotspots" (see Fig. 6) around the track where groups of cars would get stuck at the same difficult points on the track such as hills or sharp corners. We did not expect to see such discrete spots on the map that showed the performance of the different neural networks. The cars that also performed the best were the ones that went at a moderate speed, not too fast and not too slow. We attributed this to the fact that the fast cars would be too fast when turning corners and would hit the wall and crash and the slow cars were often too slow and would not make it past the next checkpoint in the time limit of 10 seconds between checkpoints that we set out. The cars also exhibited behaviors similar to what a human driver by slowing down around corners and accelerating when going around relatively straight parts of a track.

As we can see in the graph (see Fig. 7), for a mutation rate of 50% and below it seems as though the genetic algorithm optimized better on the population size of 50 and 100 compared to 10. But when we get to a mutation rate of 80% it is interesting to see that the genetic algorithm performed similarly across the different population sizes with a convergence at around 36 generations (see Fig. 8). A mutation radius of 0.5 was selected as it was one the only mutation radius that we were able to observe convergence for the variety of population and mutation rates that we selected.

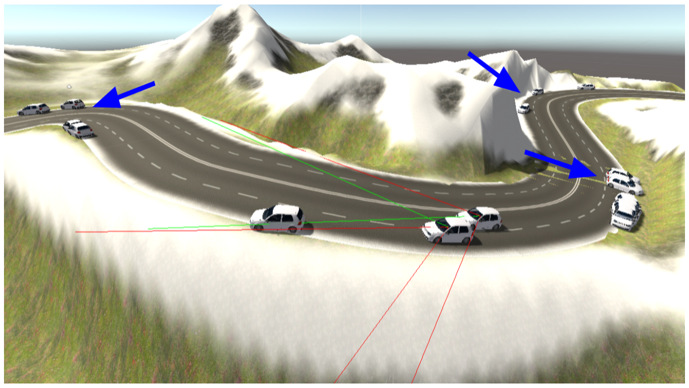

Fig. 6. Hotspots for crashing on the track.

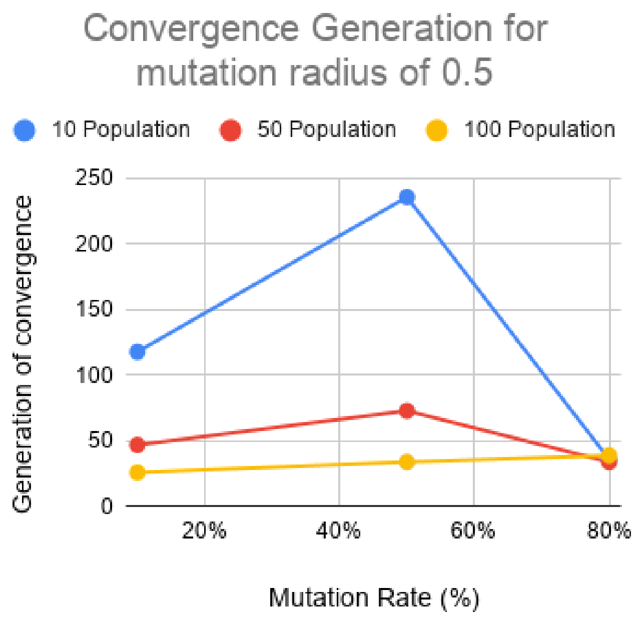

Fig. 7. Convergence generation for mutation radius of 0.5.

Population Size	Mutation Rate	Mutation Radius	Convergence Generation
10	10%	0.5	118
10	50%	0.5	236
10	80%	0.5	36
50	10%	0.5	47
50	50%	0.5	73
50	80%	0.5	34
100	10%	0.5	26
100	50%	0.5	34
100	80%	0.5	39
50	10%	1	28

Fig. 8. Convergence generation for various configurations.

V. Future Work

The behaviors that we observed in this simulation were very interesting, and it gave us insight into how we would be able to perform further exploration on this topic. We thought that making the car physics and road physics more realistic would be a good addition in any future iterations of our project. We could also generate new tracks for each generation of the run of the simulation to see how the improvement of the neural network differs.

The implementation of traffic using additional cars could also add a collision-avoidance aspect in our simulation. Observation of the cars traversing the track showed the cars staying in a single lane for portions of the track. After this observation, we thought it would be an interesting addition to add lane detection to our simulation in the future. For even further exploration we could add communication between multiple cars using a networking system to see how communication between cars changes their behaviours.

VI. Conclusion

In this paper, we demonstrate how genetic algorithms can be used to optimize the weights of neural networks. At the start of this project, we were unsure that our neural network would be to perform optimally and complete the track but after implementing the neural network along with the genetic algorithm tuning the weights, our observation exceeded our initial expectation. In experiments shown throughout this paper we were able to come up with great results just by applying four simple rules to the car (accelerate, turn the wheel left, turn wheel right, break). These four simple rules gave the car the ability to maneuver through tight turns, overcome big hills and stay in control over steep slopes.

References

[1] K. De Jong, D. Fogel, and H.-P. Schwefel, “A history of evolutionary computation”, in. Jan. 1997, A2.3:1–12.
[2] D. J. Montana and L. Davis, “Training feedforward neural networks using genetic algorithms.”, in IJCAI, vol. 89, 1989, pp. 762–767.
[3] K. Jadav and M. Panchal, “Optimizing weights of artificial neural networks using genetic algorithms”, Int J Adv Res Comput Sci Electron Eng, vol. 1, no. 10, pp. 47–51, 2012.
[4] D. Floreano, T. Kato, and E. Sauser, “Coevolution of active vision and feature selection”, in. March. 2004, Biol. Cybern. 90, 218–228.
[5] S. Awan, “Unity Asset Store”, 2019. [Online]. Available: https://assetstore.Unity.com/packages/3d/environments/landscapes/mountain-race-tracks-110408. [Accessed: 18-April-2020]
[6] 1Poly, “Unity Asset Store”, 2016. [Online]. Available: https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-destructible-2-cars-no-8-45368. [Accessed: 18-April-2020]
[7] Renaissance Coders, “Unity3D How To: Driving With Wheel Colliders”, 2018. [Online]. Available: https://www.youtube.com/watch?v=j6_SMdWeGFI. [Accessed: 18-April-2020]
[8] Unity, “Unity | Documentation”, 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html. [Accessed: 18-April-2020]
[9] Unity, “Unity | Documentation”, 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/Physics.IgnoreCollision.html. [Accessed: 18-April-2020]
[10] Code Monkey, “Code Monkey - Create a Graph”, 2018. [Online]. Available: https://unitycodemonkey.com/video.php?v=CmU5-v-v1Qo. [Accessed: 18-April-2020]