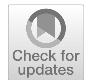
SN Computer Science (2022) 3:147
https://doi.org/10.1007/s42979-022-01030-2



ORIGINAL RESEARCH

Implementation of Genetic Algorithm for Path Estimation in Self Driving Car

Jatin Luthra[1] · Abhishek Sharma[1] · Shubham Kaushik[1]

Received: 25 August 2021 / Accepted: 7 January 2022 / Published online: 31 January 2022
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2022

Abstract
With the recent advancement in artificial intelligence, autonomous vehicles have been a significant area of reach. Companies like Tesla and Waymo by Google are leading examples in this area. This paper focuses on path allocation and trajectory mapping research by creating the 3D environment and implementing a genetic algorithm. The work presented in this paper has three significant contributions: the first step is to develop a simulation of a real-world environment for self-driving cars using the Unity3D Engine, a real-time creation tool. The second step is to implement genetic algorithms to perform training related to path allocation and obstacle avoidance. In the last step, performance analysis for the algorithm in the simulation environment is described, and the benefits are explored later in this work. The novelty in the approach lies in checkpoints and crash penalties as fitness functions. Moreover, it includes two separate training, one for consistency and one for efficiency. The model was trained for 330 generations with 75 agents of genetic algorithm in both modes of training. The study presented in this work helps to decide the optimal path in the most diminutive time frame for self-driving cars.

Keywords Autonomous vehicles · Neural networks · Genetic algorithms · Simulations

Introduction

Autonomous vehicles are capable of interpreting data from the environment through the sensor interface in the vehicle. It takes decisions without requiring human intervention as ECU (Electronic Control Unit) with algorithms works in real-time for path allocation and trajectory estimation. Since the boost in AI (after the rise in data and computational efficiency) and the development of few major algorithms with concrete research [1–3], this topic has become one of the most significant applications of AI to be followed by many researchers [4–7]. Companies like Tesla and Waymo have already done quite a good amount of research in this area, but still, it is far from perfect. The issues related to incompetency, infrastructure, and driving scenarios are the primary area of contributions done by many researchers [8–11]. Apart from that, human safety is a significant concern that is related to driver-less cars application development. Perfection should be accurate because human life should not be dealt with chances. Even after so much progress, it also gives the reason that consumer-ready driverless cars are far from reach in daily lives. Also, since there are many unpredictable situations, human intelligence and awareness can easily manage compared to machine-based intelligence. Accidents happen daily due to careless and distracted drivers, which calls for such technology to prevent it. Many researchers and organizations are trying to collect daily data to add more and more parameters and complex scenarios for models to train on to reach the global optimal solution. Once they are ready, these cars can be made available to the consumers and general public.

✉ Abhishek Sharma
abhisheksharma@lnmiit.ac.in

Jatin Luthra
jatinluthra14@gmail.com

Shubham Kaushik
18ucc088@lnmiit.ac.in

[1] The LNM Institute of Information Technology, Rupa ki Nangal, Post-Sumel, Via-Jamdoli, Jaipur, Rajasthan 302031, India

Related Work

In the work presented by A simplified dynamic model with driver's NMS characteristic for human-vehicle shared control of autonomous vehicle [12], Adaptive Genetic Algorithm (AGA) has been used to identify the parameters for

the Human-Vehicle Shared Control (HVSC) dynamic model for a 2-DOF vehicle model. Deep Learning-Based Resource Allocation Scheme For Vehicle to Everything Communication [13] proposes a deep learning genetic algorithm model with the added benefit of reducing dimensions providing 20% better throughput than a vanilla genetic algorithm. Research on AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems [14] shows a great way to convert a locally optimal solution to a globally optimal one through fuzzing. This helps in finding the best or safe solution for the evolving traffic. In a communication scheme for delay sensitive perception tasks of autonomous vehicles [15], a genetic algorithm is implemented to minimize the processing delay of sensory data converging to less than 60 ms for a multi-autonomous vehicle. In work presented by Diagnosis of Sensor Faults in Hypersonic Vehicles Using Wavelet Packet Translation Based Support Vector Regressive Classifier [16], Fault pattern recognition is done using Support Vector Regression (SVR). Again to optimise it, genetic algorithms are used. Research on Robustness and performance of Deep Reinforcement Learning [17] shows the use of Deep Reinforcement Learning Network for the self-driving car application. However, to increase the accuracy, they use a genetic algorithm for optimisation. Analysis of MPC path-planner for autonomous driving solved by genetic algorithm technique [18] shows that they propose a trajectory planner for which genetic algorithms achieve the solution. In Driving Cars by Means of Genetic Algorithms [19], Genetic algorithms are used to train the cars, and the performance is also compared to other simulator bots. However, it took almost a thousand generations to get an optimal result on a circular track. In contrast, it took only three hundred generations to do so for a typical road system in this work. In a work presented by Comparative Study of NeuroEvolution Algorithms in Reinforcement Learning for Self-Driving Cars [20], the aim is to use neuroevolution with reinforcement learning to keep the cars in the middle lane for as long as possible. The cars presented in this work automatically change the routes for the quickest turns possible, either left or right, due to the one-way roads. Research presented in Self-Driving Cars Using Genetic Algorithm [21], A work very similar to the work presented in this manuscript is done but in a 2D simulation environment rather than the 3D environment in this work. A comparison of genetic algorithm and reinforcement learning for autonomous driving [22] compares two different algorithms for such simulation environments. It is concluded that genetic algorithms did outperform reinforcement learning, but the latter was more stable and safe in terms of real-world driving. Similar work is done in Application of Neuroevolution in Autonomous Cars [23]. A comparison is also made across the various combinations of genetic algorithm parameters such as crossover rate, mutation rate, and population. This, in turn, led to a different number of generations to achieve an optimal result which in one case was as low as 12 generations. A summary is also provided in Table 1

The novelty in this work as compared to all the works presented above is not just in using genetic algorithms but also the correct fitness function of checkpoints and penalty on collision to provide an incentive to take optimal turns as discussed in further sections. Also, to ensure both consistency and efficiency, another fitness function was introduced to improve the time taken by the agents to reach a certain checkpoint.

Method Overview

A vehicle that hasn't analysed all the scenarios is imperfect and, hence, unfit to drive on roads. Simulations can do this analysis. Simulation is a virtual world where virtual models can traverse infinitely. Due to this, it is the perfect way to train driverless cars. According to Waymo, they run 20 million miles each day in simulation. The simulation not only involves straight roads but turns, pedestrian crossings, traffic lights, etc.

Through the simulation, the intention is to build a simpler model of straightforward interpretation and computation. This can be done by generating own virtual terrain of straight and turn roads. The Unity3D [24] game engine is incorporated to visualise and check how the application performs on various test cases. Unity allows to assimilate real-world physics and use an almost accurate representation of real cars in a designed model. With the implementation of openly available assets for roads, side rails, grassy terrain, and most importantly, the model of car from the Unity Asset Store [25] (UAS), this visualisation will be more accessible, and the prime focus will be on application development.

The development process includes multiple steps such as terrain formation, the evolution of agents, gene crossover, etc., as depicted in Fig. 1 which explains the overall methodology adopted in this work. The motive here is to get a general overview of the procedure to get through the working functionality of the model. In the beginning, a virtual terrain is generated with roads and grass. In the background, five thousand terrains are generated, and the best is selected according to the maximum distance covered. Since all these calculations are done mathematically (coordinates to place roads and tiles), the terrain is visually rendered after the choice is made. In the next step, seventy-five cars are generated with a set maximum lifetime of 15 s which will be changed later according to progress. Then the agents start evaluating the sensor values and calculate the action (throttle and steer's magnitude and direction) with a randomly generated neural network. After all the cars are destroyed, either due to the collision with side rails or their maximum lifetime, the genetic algorithm prepares new set of seventy

Table 1 Related work comparison

Study	References	Main goal	Major findings	Comparison with our work
Driving Cars by Means of Genetic Algorithms	[19]	Learn to drive and optimize lap times	Optimal solution in 1000 generations but path becomes invalid based on initial state and sensor noise	Optimal solution in 330 generations on a real-world road system
Comparative Study of NeuroEvolution Algorithms in Reinforcement Learning for Self-Driving Cars	[20]	Compare neuroevolutionary algorithms to DDQN algorithm	Evolutionary Strategies (ES) outperforms with middle lane accuracy of 97.13%	Automatic change in lanes to achieve greater efficiency while turning
Self-Driving Cars Using Genetic Algorithm	[21]	Learn to drive in a simple environment with basic obstacles	Genetic Algorithms can be used to update weights using real-time sensor values instead of backpropagation in 573 generations	3D simulated environment is used instead of 2D trained for 330 generations to ensure consistency
A comparison of genetic algorithm and reinforcement learning for autonomous driving	[22]	Compare the learning efficiency of genetic algorithms and reinforcement learning	Genetic Algorithms outperform but isn't stable and safe for real-world	A real-world road system is used instead of circular track with better fitness function
Application of Neuroevolution in Autonomous Cars	[23]	Train self-driving cars without the need of large datasets	Correct settings can lead to optimal solution in just 12 generations	With such low population and lesser mutation rate, our work was able to perform consistently for 330 generations

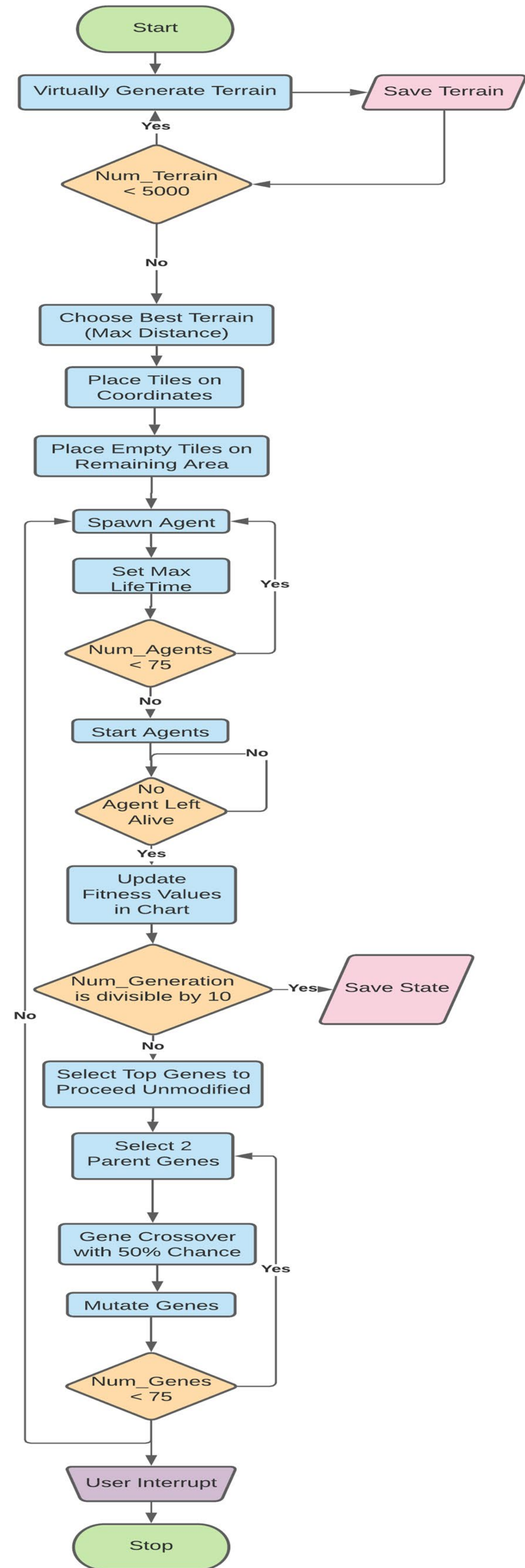


Fig. 1 Overall FlowChart

five cars for the next generation and this is repeated until stopped. Each of the steps are discussed in more detail in the subsequent sections. Finally, there are two separate training modes with separate fitness functions applied to them. The first training mode proves that the solution is consistent and the second training mode proves that the solution is efficient.

Environment Preparation

This section explains the procedure of terrain generation, including the coordinates calculation, tile-based approach, car models, and visually rendering the complete simulation environment.

Procedural Terrain Generation

Random terrain is needed to evaluate agents' performance better; otherwise, it might just be overfitting and remembering when to turn right exclusive to the map. Since roads are generally straight and wavy patterns are rare, it is not preferred to use things like Perlin Noise for terrain generation. So instead of that, the tile-based approach is preferred.

In the tile-based approach, tiles are placed on the terrain according to few conditions, further explained. The first logical thing to do is design an empty tile. This tile will be a cuboid of 60×60×0.5 size since tiles need not be thick as the under terrain portion won't be visible. The area was decided based on the turn road length, which will be discussed later. Then a grass material was imposed on this blank Tile to form the "Empty Tile," as shown in Fig. 2a.

For roads, two 3d shapes were taken, and road-like material was imposed on them; the tile-based approach helps take advantage of such models. These models are a straight road and a road curved towards the right. These were imposed on separate" Empty Tiles" and scaled to a perfect fit forming the" Straight Tile" (ST), and the "Turn Tile" (TT) depicted by Fig. 2b, c respectively. Only three of these tiles are required because all the directions can be formed by rotating the TT around the y-axis. We'll label the directions as two parts which would be from where to where we're going. So a regular TT, as mentioned above, will be NE because, before the turn, The travelling direction would be north and then east. If seen from the other side, The same Tile could also be labelled WS as coming from west to south. Hence each rotation of TT encompasses two diagonal directions making 8 of them possible. The ST will only have a single direction N, S, E or W. Then, some rules can be set up regarding the options of tiles available which can be connected to a tile. For example, an NE tile can be connected to tiles such as W, WN, WS since it's impossible to connect a tile to an EN tile that will not be connected. Instead of hard-coding such rules in the code, they are generalised.

At this point, the issue of overlaps and possible tile placements in the same position needs to be addressed. The latter can be solved by keeping a record of all the positions previous tiles are placed. However, the former is tricky and can be avoided by tightening the rules, but overlaps are inevitable. So this issue is solved by creating some terrains without actually placing the tiles and choosing the best terrain, followed by actual tile placement.

The best terrain is decided by the number of overlaps and the total straight line distance from start to end. Since all the terrains are just numbers at first, it doesn't affect the performance, and the whole process, including tile placement, doesn't even take a second. The number of terrains to be generated is controlled by a parameter set by default to 5000. Moreover, the benchmark was done in less than a second. After these, the empty positions on the terrain are filled with empty tiles completing the terrain generation process. It was decided to keep it a significant area for future work to be extensible as much as possible.

In total, twenty road tiles are used. One last parameter for generation is "Straight Road Percentage" (SRP), the overall percentage of roads that need to be straight. This parameter can help control turns in the road system, and by default, 40% straight roads are set. A top-view or birds-eye view was captured for demonstration, as shown in Fig. 3. The sub Fig. 3a, b show some of the sample-generated terrains at the default settings. Sub Fig. 3c, d show the generated terrains with 10% straight roads and 35 total roads, respectively. It can be observed that by decreasing the SRP, almost every other road results in a turn. Also, by increasing the total roads, more area of the empty terrain is covered. These are a few of the infinite possible terrains on tuning the settings, and each of them shown above is the maximum distance possible with the defined settings.

Car

The car model consists of the body mesh, wheels, and physics-related settings like suspension, torque, throttle, steering angle, etc. The car model is an asset named Arcade Car Physics [26] imported from the UAS, depicted in Fig. 4. The asset was chosen to focus more on the path estimation rather than fine-tuning the physics parameters of cars which would be just reinventing the wheel. The model is a "Low Poly" design which means it has fewer details and looks very rough. This model, however, is best for our use case since such methods give a better computational performance. Optimised performance for each car means we can have more cars in our genetic algorithm, bringing faster and accurate improvements. The vehicles can also be called "agents" since multiple cars will be spawned independently of each other. The complete set of agents in each generation is known as population. Independence means that the

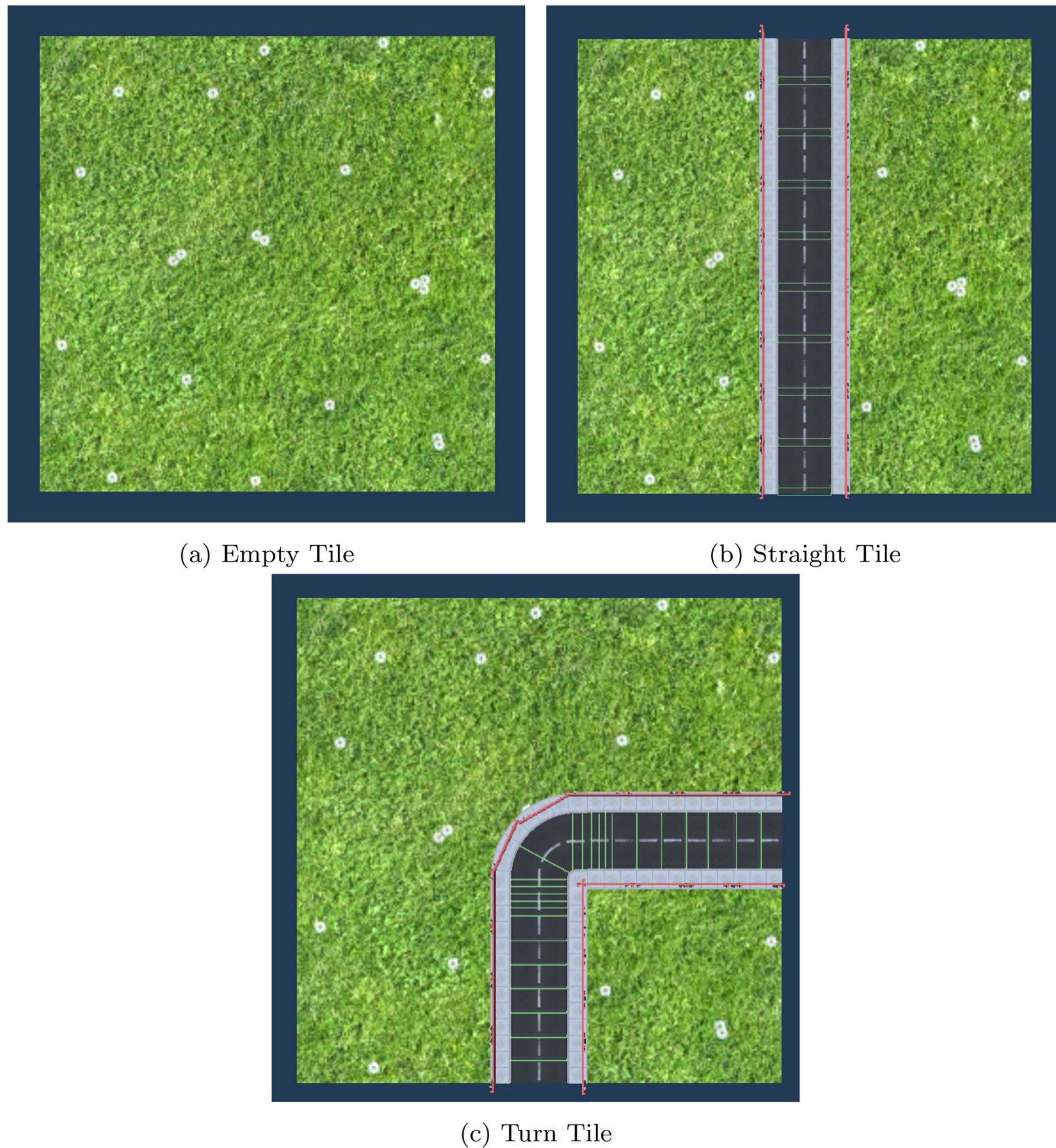


Fig. 2 Road tile set with checkpoints

vehicles won't interact like inter-car collisions, but each of them will interact with the environment perfectly.

Neural Networks

The car model requires only two floating-point inputs, throttle and steering. The range of these inputs is $[-1.0, 1.0]$ where the $+/-ve$ sign is used for direction. A positive throttle means forward movement, and a positive steer means towards the right. The negative sign is opposite, i.e., and negative throttle means reverse direction and negative steer means towards left. The magnitude of these values decides the intensity of throttle and steering. Since these inputs have to be calculated in each pass/frame of physics rendering, neural networks will be used. The neural network will act as the agent's brain since, at each frame, the values of throttle and steer will be calculated based on inputs such as distance sensor values and current speed, which will also be discussed in detail in the following subsection.

(a) Sample 1

(b) Sample 2

(c) Sample with 10% Straight Roads

(d) Sample with 35 Roads

Fig. 3 Sample generated terrains

Fig. 4 Car Model

Architecture

There is no image or sequence processing involved, so straightforward and shallow neural networks are possible. Again, a significant advantage would be the performance, which is inversely proportional to the density of such networks. Since the processing scripts are written in the C# programming language, a compatible neural network library is required. There are many such libraries available, but there are few problems in using these libraries. Training data is needed to train the neural network, which means the correct output for a given set of inputs must be present so that the weights of individual neurons can be adjusted. However, a

genetic algorithm is being used to make adjustments that rely on separate metrics known as fitness scores. Also, it is essential to save and reload the best neural network among all the cars for each agent, which must be done efficiently. These problems can be addressed by creating a custom implementation of a neural network. The only library used here would be the matrices library to perform matrix operations efficiently as performance matters a lot. This becomes necessary when there would be many agents trying to run their neural network at each frame which would be a lot of calculations per second.

The most important part here would be the structure of the neural network. It comprises six input neurons, five of which are sensor values. Each agent is equipped with five distance sensors starting from right to left with a 45-degree angle between them. The purpose of these sensors is to give the agent a sense of the environment. Red rays were drawn for all the sensors to visualise properly for debugging purposes, which can be observed in Fig. 6. The maximum range of these sensors was clamped to induce the sense of perspective and horizon. The sixth input is the agent's speed so that it can adjust its acceleration and steering accordingly. These values are collected for each agent at each frame and fed into the six neurons of the input layer.

Only one hidden layer was selected, with only five neurons because, as discussed earlier, It is necessary to keep it as simple as possible for better performance. If there is any problem, the architecture can always be modified. And finally, the output layer has only two neurons, the first one for the throttle and the second one for the steering, each with the sigmoid activation function since the desired values are from – 1 to 1 centred around 0 depicting the acceleration and reverse in the first case and the steering direction in the second. For each layer, the weights are multiplied with the node values and biases are added at each layer. Since each layer has multiple nodes, matrix multiplication is applied instead of nominal multiplication. In the final layer, the output is a real number according to the activation function applied. Since the sigmoid activation is applied here, the values can only be from – 1 to 1. In the case of throttle, maximum throttle is applied in the forward direction when the output is 1 and maximum throttle is applied in the reverse direction when the output is – 1. The intermediate values are mapped to varying throttle power since the value of 0 means no or zero power. Thus, values from 0 to 1 imply varying throttle power from minimum to maximum in the forward direction and values from 0 to – 1 imply varying throttle power from minimum to maximum in the reverse direction. Similar is the case with steering. The values are mapped in a similar fashion where -1 means left direction and 1 means right direction. The architecture is also visualised in Fig. 5

The weights and biases of all the neurons are initialised randomly so that each agent has a different perception of the inputs. Hence, each agent has a unique set of actions. However, there is no backward propagation involved to tune these weights and biases. Instead, we'll benefit from the solid concept of evolution through genetic algorithms or neuro-evolution, which will be explained more in the following significant section (see Fig. 6).

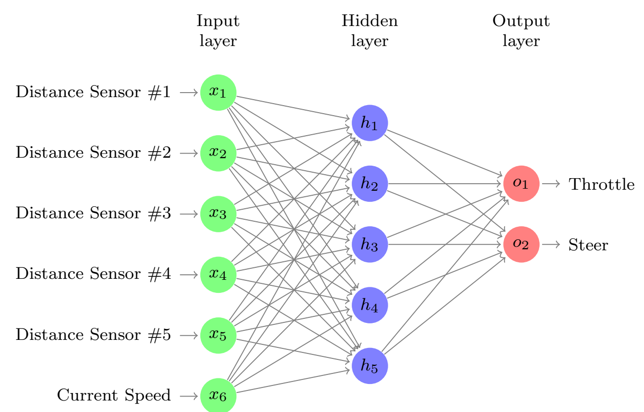


Fig. 5 Neural Network Architecture

Fig. 6 Car with active sensors

Implementation

Many neural network implementations are available in the form of C# libraries, but they were neglected. That is because complete control over neural networks, which means how the weights are updated and saved, can only be achieved by creating a custom implementation. Since neural networks use matrix multiplications, it was futile to reiterate that in the form of arrays. But instead of its matrix implementation, the Math.Net Numerics [27] library was lucrative. The matrix implementation should be efficient as multiple neural

networks run and process inputs at each frame. Since it is a game in the broader picture, any decrease in FPS or "Frames Per Second" is unfavourable since it would introduce stutters and potential inconsistencies. Due to this, each agent might not be able to collect the inputs at each frame which is a problem for both generational improvement and reproducibility. Creating own Neural Network implementation has many benefits. It helped integrate the genetic algorithm and the reproducibility factor much more quickly, which is discussed further.

Genetic Algorithm

Generations

Each agent is spawned with a maximum lifetime it can be alive. This is because if an agent decides to do nothing at all and just stay in a fixed position, we'll see a stalemate. So with max lifetime, either the agent dies after crashing or after a specified time.

This fixed time is set to a short time, like 15 s at the start, so each generation is guaranteed to take a maximum possible time of 15 s. But there is one problem. The procedural track generated is too long and has many turns. So 15 s is not feasible. Instead, another parameter is introduced known as "Generation Improvement" (GI). Another thing to keep in mind is that if the agents have learned best to drive in that 15 s, the max fitness of generations will remain the same, i.e., a straight line in the graph. With this parameter, it is possible to set the number of generations after which if the max fitness is constant, the max lifetime is increased by 5 s. If the GI is set to fifteen, then if the max fitness remains constant for fifteen generations, the max lifetime will increase to 20 s. This ensures that the agents have mastered the timing for each max lifetime and have given their best. It also adds the element of progressive or iterative nature to the algorithm.

Methodology

Gene Pool Formation

A gene pool is a set of all the agents' neural networks. Each agent is added to the set fitness number of times. What fitness is and how it is decided will be discussed after the explanation of steps. This means if there are two agents with the fitness score of two and eight each, agent one will be added to the pool two times and agent two will be added to the pool eight times. This is because, during the selection process, the agent with more fitness should have more chance of being selected as a parent to pass the genes to the next generation.

Population Upkeep

A controllable parameter is used to decide how many fittest genes will proceed unchanged to the next generation. The best of a generation becomes a reference population, guaranteeing that the best fitness will never decrease. The offspring gets the best genes possible.

Selection & Crossover

Then random genes are selected from the gene pool two at a time as parents for remaining agents one by one. Then in the crossover process, there's a 50% chance that weights and biases are chosen from either of the parents. This ensures that the offspring is a combination of the weights and biases of the parents.

Mutation

Finally, keeping aside the population upkeep, each weight and bias of each gene is subjected to a mutation. This mutation is decided by the mutation rate parameter, which is 5% in this case. The mutation rate should not be too low such that the population converges to a local optimum solution. On the other hand, a very high mutation rate can help search for more optimal techniques but prevents converging. As mentioned in Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors [28], 5–20% mutation rate is best with the population size 16. Since the chosen population size, in this case, is 75, which is five-fold than the recommended, the lowest recommended mutation rate is best. In this situation, a decimal value in the range [– 1.0, 1.0] is added to the weight or bias. This ensures that there will be a specific change from generation to generation, which might even unlock a globally optimal solution.

The methodology is summarised and depicted in Fig. 7

Fitness Functions

The fitness score is a representation of the performance of an agent. It is a cumulative score where the higher scores depict the closeness to an optimal solution. Since the whole genetic algorithm depends on fitness scores, Choosing the correct fitness function of agents is crucial. The fitness function calculates the fitness score according to various weighted parameters. Four different fitness functions were tested, and the differences between them were noted. How the learning process differentiates is described in subsequent

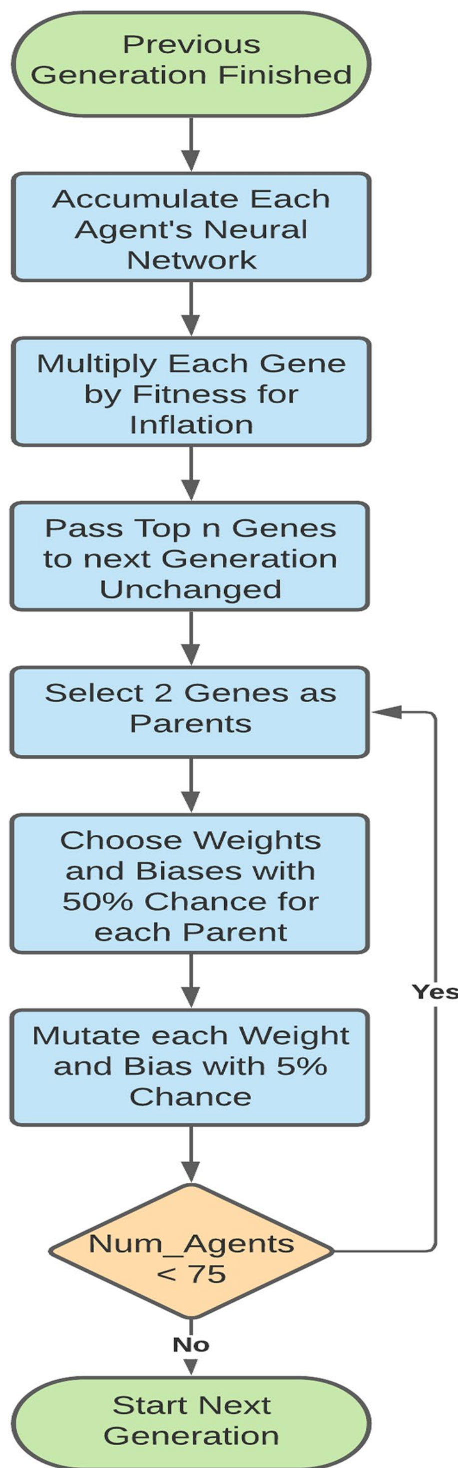


Fig. 7 Genetic Algorithm Methodology

subsections. Also, it is essential to note that we'll only list the parameters, and the final fitness function will be a linear combination of these parameters. This means $f(A) = ax + by$ where x, y are the parameters regarding the agent's state and a,b are the floating-point coefficients demonstrating the linear combination.

Displacement

In this, only the displacement of an agent from the initial position was made viable for the agent's fitness. The problem with this is that in the cases of turns, the change in displacement is minimal, and the agent directly hitting the side rails and taking a slight bend and then hitting the side rail had almost the same fitness and hence the chance in the gene pool. Thus, some incentive should be given to the agents to make even the most minor turns as the straight drive wasn't the issue from Generation 0.

Checkpoints

Invisible checkpoints were laid down on the roads, which had a default score of 1. This meant that as the agent crosses the checkpoint, the score will be added for that agent. In case of turns, the checkpoints could be set near the turns to be a roller coaster or a parabola of discrete scores ranging from 1 to 10 to 1. This way, the slightest turns had a significant influence on the fitness of any agent, making it evident that such turns are essential. The main problem with the above functions was the problem of the straight head-on collision. Most of the agents were just going for the fastest speed and were making a head-on collision with the front side rail. So even if the checkpoints incentive was able to teach the agents that it is better to turn. On average, most of the agents were not following that, and hence the average fitness was very low. This issue of speeding was solved with the following functions.

Checkpoints + Average Speed

If the speed also counts in fitness, agents could become better at controlling it instead of just racing for the first to crash. This fitness function is great because just before the front side rail, the cars learned to slow down very much, and this way, they were also better at taking turns because, at high speeds, the turn is difficult and discouraged. Another problem came up, which was an issue related to the stalemate. Also, to avoid a collision, the cars slowed down, but also, most of them just stopped or went into reverse gear. Even if the turns were better now, the learning curve was slow, and average fitness was still not good enough. This led to the following fitness function.

Checkpoints – Crash

Rather than average speed, a high penalty was given if an agent crashed inside rails. This prevented the agents from taking the speed down to zero, and they took better turns by slowing down. This also solved the problem of stalemate, and the average fitness kept on growing along with the best fitness of a generation. Most agents made good turns leaving only a handful behind, making it the best and hence the final fitness function for first training mode.

Checkpoints – Crash + 5000/Time

Once it was made sure that the agent can consistently learn and produce results, it was necessary to optimise them even more. To do so, another factor was added to the fitness function which is the inverse of time taken to reach a checkpoint in the middle of the track. This ensures that the agents which take less time will have more fitness and is a suitable fitness for the second training mode.

QoL Features

Quality of Life (QoL) features make it easy to track the performance and debug software. Implementation of such features usually increases workflow productivity, and a few of them were also added in this work, which is described in the following subsections.

Reproducibility

The maximum fitness of the generation decides the best neural network of any generation. To be able to reproduce the results, a way to save the neural network was needed. As mentioned before, the custom implementation of neural networks is made of lists of matrices, each matrix can be converted into a float array, and the list can be converted into an array of float arrays. With this approach, the neural network could be converted, or a better term would be serialised. For this, the Json.Net [29] library was used. After the conversions were done, it was possible to convert the neural network to a JSON file and save it locally. This way, whenever it is required to use one of the trained neural networks, it can always be done and compared with the neural network from other generations.

Replay System

It would be very tedious to stare at the screen for all the generations. It is still very much possible that some things are missed, something like the behaviour of an agent in a generation or the improvement from previous generations, etc. This problem was solved by implementing a replay system so that it is possible to jump back and view any generation's progress. However, it would be computationally expensive to reload and rerun neural networks. Moreover, this might decrease the performance which is observed in replays, and inconsistencies may also rise. Since the Neural network keeps track of the weights and biases and the inputs and calculated output, a complete list of outputs that is already there can be used.

So a "StateData" object can hold the generation information for terrain, the chart data, and an array of all the outputs for each generation. Since we'll be tracking generational performance, The XCharts [30] library was used to save the input data as chart data. This way, it is a matter of just translating those outputs into acceleration and steering and helping in escaping the inconsistencies and performance degradation.

Speaking of performance, outputs of multiple agents for hundreds of generations can take a lot of space and time for saving and loading back. Due to these constraints, JSON serialisation became a burden because only a hundred generations with seventy-five agents took 2.5 GB space, which is inefficient. The MessagePack [31] library was used to solve this problem, serialising an object into binary data. Though this format isn't directly readable by any text editor, It can save a lot of space and time while saving and loading. For six hundred generations and seventy-five agents in each generation, the final state file took less than 300MB space and was very efficient at the loading time.

Figure 8 depicts the comparisons for both MessagePack and JSON formats which includes Read Times (time to load the save data back into memory), Write Times (time to write the save data on disk), and File Sizes (total space took by the save data) for generations with gaps of 100. As shown in Fig. 8a, b displaying Read Times and Write Times, respectively, growth in the time taken for reading and writing is linear in the case of MessagePack but exponential in the case of JSON. Also by observing Fig. 8c, it is noticed that the file size of MessagePack serialisation is less than half of JSON serialisation. These comparisons prove how efficient and beneficial it will be to use MessagePack instead of JSON serialisation.

Test System

The trained agents should work in different terrains, and to test that, a test system is required visually. However, this turned out to be very easy because of my initial system of using a Procedural Generator (PG) system that can spit out random terrains every time requested. Since the reproducibility factor was already added to the neural network library, saving and loading neural networks is already done. So the test system was just linking these two up to test the best trained neural network on random terrains.

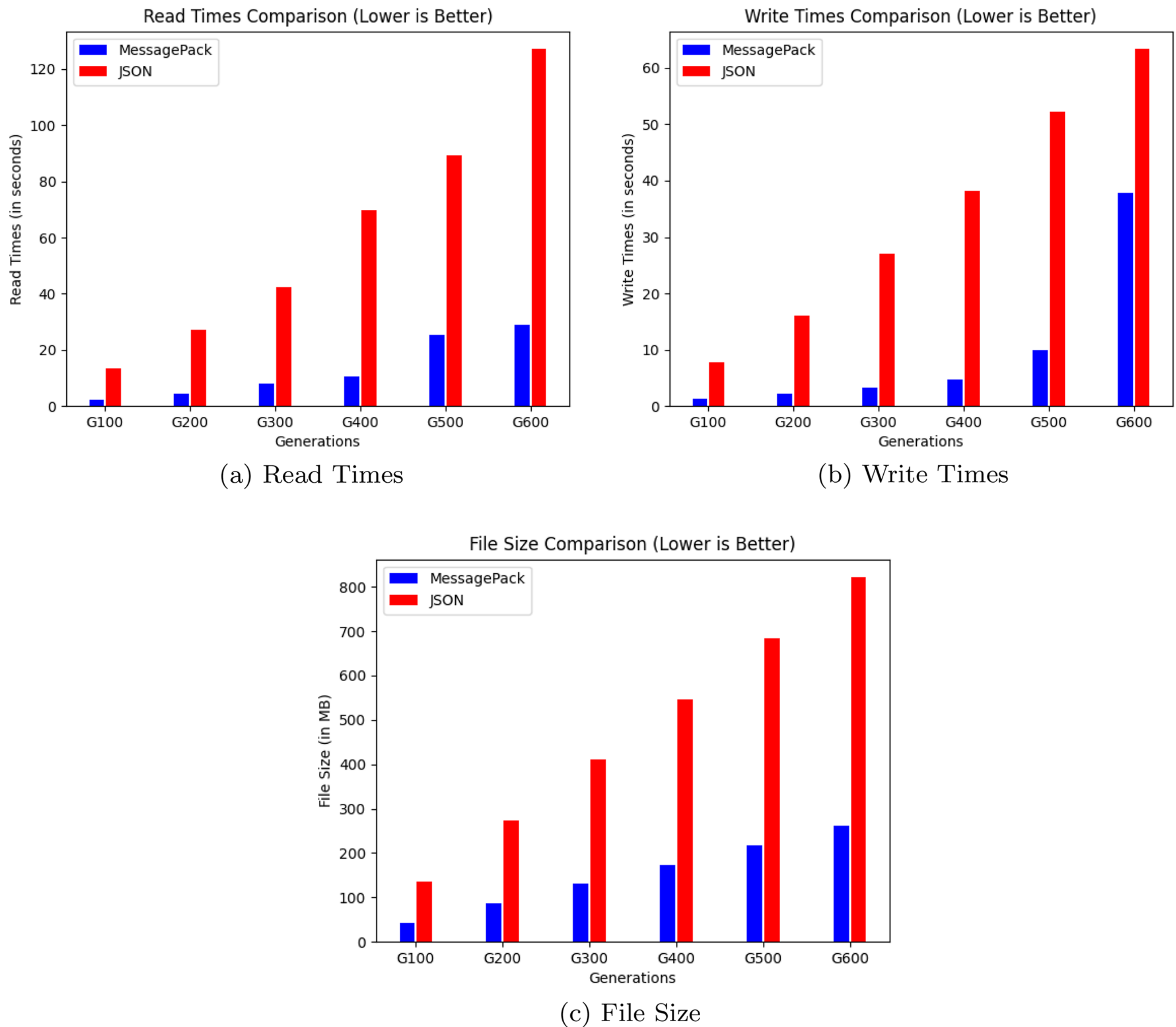


Fig. 8 Performance comparison of MessagePack and JSON

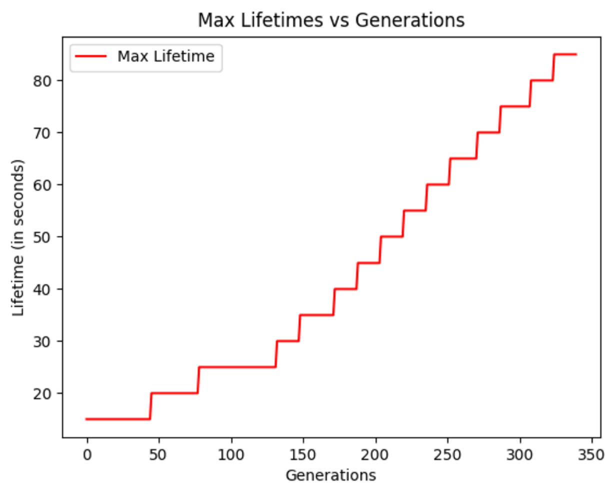
Results

Training Mode 1

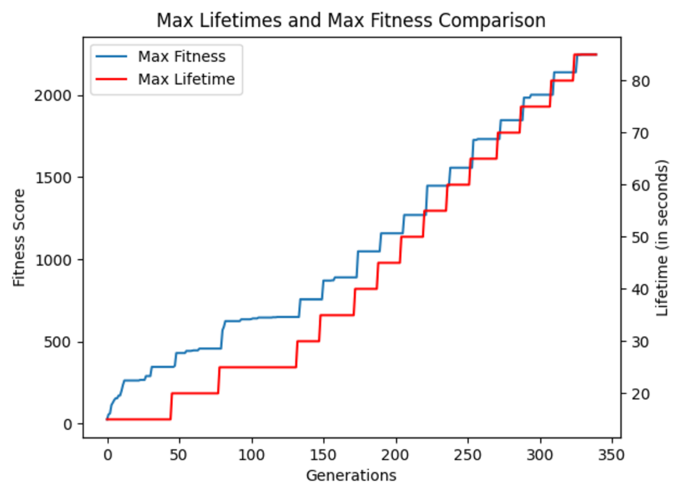
In total, agents were trained for 330 generations, and the results are depicted in the graph below. This is because the agents were able to complete the whole track at around 286 generations. Since they didn't show any quicker speed in solving the tracks, it was best to stop at 330 generations. This is because, at the next generation, they would get just an increase in a max lifetime by 5 more seconds which was utterly unnecessary. The blue line below depicts the "Maximum Fitness" in each generation, whereas the green line shows each generation's "average fitness". For maximum fitness, it is observed that after constant straight lines, there is a sudden improvement sometimes. This results from the "Generation Improvement" parameter, which increases the "max lifetime" by 5 s after 15 generations of constant fitness. Also, the average fitness is not far behind, which generally shows how much better each agent did for a generation.

In Fig. 9, the blue line denotes "Max Fitness" among all the (75) agents (cars). On the other hand, the green line indicates the "Average Fitness" of the agents. The red line shows the best fit line on the scattered points of average fitness to get an accumulated sense of an increase in it. This metric aggregates the entire performance of the agents on the terrain. This curve clearly shows that not only just one of the agents is doing well. Instead, in almost every generation, each agent is improving one way or the other. Thus,

Fig. 9 Generation 330 progress chart for Training Mode 1



(a) Max Lifetimes vs Generations



(b) Max Lifetimes & Max Fitness Comparison

Fig. 10 Max life times analysis for Training Mode 1

it is necessary to consider both these curves when the performance of the algorithms and the complete model itself is judged.

It can also be observed from the graph that the "Max Fitness" curve is looking similar to a step function which means that for few generations, it remained constant, and then a significant boost is observed. After that, it again approaches a continual shape, and the pattern repeats itself. The reason for this is "Max LifeTime," which would be the next point of inference.

Figure 10 is the analysis of maximum lifetimes, i.e., how many seconds the agents were allowed to be alive, as mentioned before. In Fig. 10a, It is visible that starting from 15 s of time, it gradually increased to 65 s of time. This happens when no improvement for successive 15 generations is observed, suggesting that agents have found the optimum solution for current time constraints.

This graph can be plotted side-by-side with the Max Fitness graph for better analysis, as shown in Fig. 10b. Since both have different scales, It is plotted on a Twin X-Axis graph such that there are 2 Y-Axes. On the left, there is "Fitness", and on the right, there is "Lifetime" (in seconds). Both of these share the common X-Axis with the generations. It is observed that whenever the fitness graph became constant for 15 generations, Max Lifetime increased by 5 s, and the fitness also starts climbing.

This allowed the agents to learn progressively instead of directly starting from a higher lifetime period, increasing the training time and probably being less efficient.

Training Mode 2

In the case of the second training mode, the focus is to improve the efficiency of the path chosen by the agents. Due to this reason, a checkpoint in the middle of the track was chosen to act as a finishing line for agents. The last fitness function is used in this case in which the inverse factor of time taken to reach the checkpoint is introduced.

As it is shown in Fig. 11, after generation 250, negligible improvements were done to the maximum fitness function. The change in the graphs shown by Figs. 11 and 12 as compared to the Figs. 9 and 10 is completely visible. This is due to the fact that there is a change in fitness function as well as the agents are destroyed on reaching the finish line for preparation of the next generation.

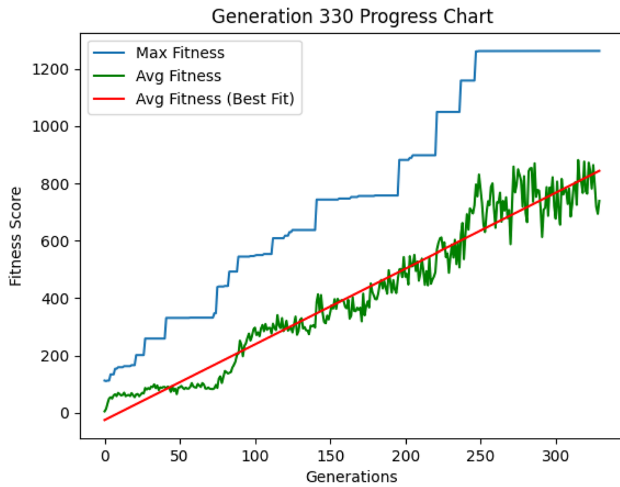
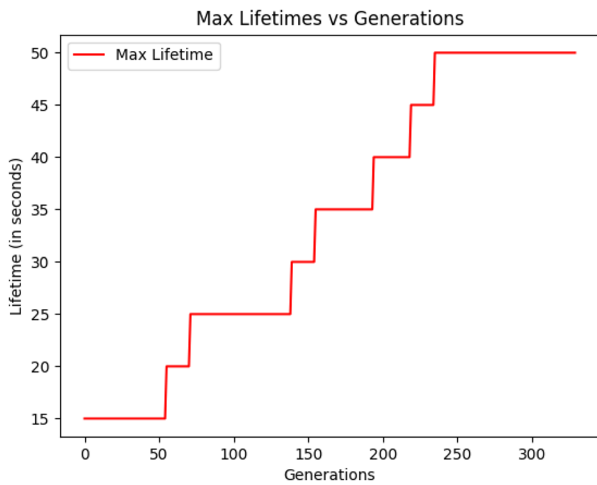
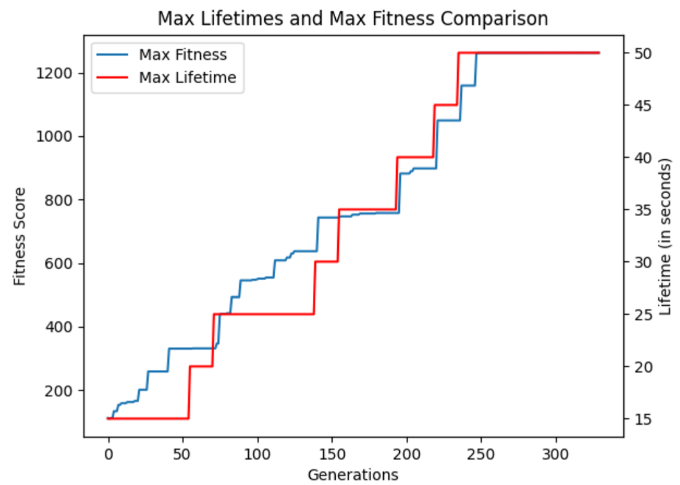


Fig. 11 Generation 330 Progress Chart for Training Mode 2



(a) Max Lifetimes vs Generations



(b) Max Lifetimes & Max Fitness Comparison

Fig. 12 Max life times analysis for Training Mode 2

However, Fig. 13 clearly depicts the improvements in the path chosen by agents over the generations which caused the decrease in time. The graph only focusses on the generations from 245 to 330. In the beginning, we can see a drastic improvement and it continued on to the further generations. The finishing time got a little stable from generation 290 onwards. Lack of improvement in the rest of the generations show that the most optimal path was chosen and when the same model is run for the complete track, it is able to reach the destination in the quickest possible way.

Future Work

The model presented in this work is still straightforward as compared to the more prominent companies. Genetic algorithms were used for the model's unsupervised training instead of backpropagation to update the weights, which isn't suited for such simulations and needs training data such as the BDD100K [32] and Level5 [33]. The reason was also discussed earlier that a simple model helps examine the fewer parameters of the model. This simple model, however, still traverses only on straight and turn roads. So for the future, more complexities like traffic signs, speed limits, and pedestrian crossings can be added. This way, the model will converge more towards the real-life scenario.

One more improvement can be made which is in the architecture. For this simple model, the neural network used was also simple. There is also the possibility that a different architecture converges to a more optimal solution. This can be solved by implementing the NEAT (NeuroEvolution of Augmenting Topologies) algorithm. An evolutionary algorithm is used to modify the weights and biases and the neural network's architecture, which could be more efficient.

Conclusion

This work aimed to simulate the real-world scenario of roads and cars and use neuro-evolution to train these cars for a driver-less approach. After comparing various parameters, the important ones are finalised through which the simple

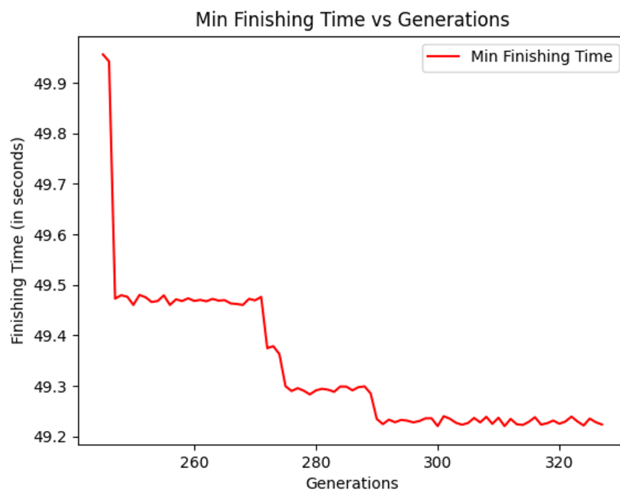


Fig. 13 Minimum Finishing Time vs Generations

model was working perfectly in the simulated environment. This work can also be extended and improved on by the methods discussed in the previous section. This model was also tested with a stochastic approach of random terrains, and the trained model worked perfectly in each of them.

Funding Not applicable

Declarations

Conflict of Interest / Competing interests On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Yao X. Evolutionary artificial neural networks. Int J Neural Syst. 1993;4(3):203–22. https://doi.org/10.1142/s0129065793000171 (**PMID: 8293227**).
2. Such FP, Madhavan V, Conti E, Lehman J, Stanley KO, Clune J. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. 2017. arXiv:1712.06567.
3. Man KF, Tang KS, Kwong S. Genetic algorithms: concepts and applications [in engineering design]. IEEE Trans Ind Electron. 1996;43(5):519–34. https://doi.org/10.1109/41.538609.
4. McCall J. Genetic algorithms for modelling and optimisation. J Comput Appl Math. 2005;184(1):205–22 (**ISSN 0377-0427**).
5. Zhang J, Jiacui. Research on congestion pricing to improve connected autonomous vehicles penetration rate. In: 2021 2nd International Conference on urban engineering and management science (ICUEMS). 2021. https://doi.org/10.1109/icuems52408.2021.00059.
6. Wang X, Shi S. Vehicle coupled bifurcation analysis of steering angle and driving torque. Proc Inst Mech Eng Part D J Automob Eng. 2021;235(7):1864–75. https://doi.org/10.1177/0954407020985405.
7. Lin Y, et al. Multiobjective environmentally sustainable optimal design of dedicated connected autonomous vehicle lanes. Sustainability. 2021;13(6):3454. https://doi.org/10.3390/su13063454.
8. Zhang, X, et al. Research on self driving customized bus lines based on users' real-time needs. 2020. https://doi.org/10.4271/2020-01-5140.
9. Fazeli SS, et al. Efficient algorithms for electric vehicles' min-max routing problem. arXiv:2008.03333 [Cs, Math], 2021.
10. Application of neuroevolution in autonomous cars. DeepAI, 2020. https://deepai.org/publication/application-of-neuroevolution-in-autonomous-cars. Accessed 15 Feb 2021.
11. Li M, et al. Development of the autonomous vehicle trajectory on the hilly road using approaches of eco-operating modes. In: ASCE, 2020; p. 725-33. https://doi.org/10.1061/9780784482902.085.
12. Wu H, Wei H, Liu Z, Xu J. A simplified dynamic model with driver's NMS characteristic for human-vehicle shared control of autonomous vehicle. Proc Inst Mech Eng Part D J Automob Eng. 2021. https://doi.org/10.1177/09544070211018944.
13. Gao J, Khandaker MR, Tariq F, Wong K-K, Khan RT. Deep neural network based resource allocation for V2X communications. In: 2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall), https://doi.org/10.1109/vtcfall.2019.8891446.
14. Li G, Li Y, Jha S, Tsai T, Sullivan M, Hari SK, Kalbarczyk Z, Iyer R. AV-FUZZER: finding safety violations in autonomous driving systems. In: 2020 IEEE 31st International Symposium on software reliability engineering (ISSRE), https://doi.org/10.1109/issre5003.2020.00012.
15. Du H, Leng S, Wu F, Zhou L. A communication scheme for delay sensitive perception tasks of autonomous vehicles. In: 2020 IEEE 20th International Conference on communication technology (ICCT), https://doi.org/10.1109/icct50939.2020.9295766.
16. Ai S, Song J, Cai G. Diagnosis of sensor faults in hypersonic vehicles using wavelet packet translation based support vector regressive classifier. IEEE Trans Reliab. 2021. https://doi.org/10.1109/tr.2021.3075234.
17. Al-Nima RR, Han T, Al-Sumaidaee SA, Chen T, Woo WL. Robustness and performance of deep reinforcement learning. Appl Soft Comput. 2021;105:107295. https://doi.org/10.1016/j.asoc.2021.107295.
18. Arrigoni S, Braghin F, Cheli F. MPC path-planner for autonomous driving solved by genetic algorithm technique. 2021. arXiv:arxiv.org/abs/2102.01211.
19. Saez Y, et al. Driving cars by means of genetic algorithms. In: Rudolph G, et al., editors. Parallel problem solving from nature—PPSN X, vol. 5199. Berlin: Springer Berlin Heidelberg; p. 1101–10. https://doi.org/10.1007/978-3-540-87700-4_109.
20. AbuZekry A, Sobh I, Hadhoud M, Fayek M. Comparative study of NeuroEvolution algorithms in reinforcement learning for self-driving cars. Eur J Eng Sci Technol. 2019;2(4):60–71. https://doi.org/10.33422/EJEST.2019.09.38.
21. Samuel CM. Self-driving cars using genetic algorithm. Int J Res Appl Sci Eng Technol. 2020;8(11):508–11. https://doi.org/10.22214/ijraset.2020.32200.
22. Xiang, Z. A comparison of genetic algorithm and reinforcement learning for autonomous driving (Dissertation). 2019. Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-261595. Accessed 15 Feb 2021.
23. Sainath G, Vignesh S, Siddarth S, Suganya G. Application of neuroevolution in autonomous cars. 2020. arXiv:2006.15175.
24. Unity Technologies. 2019.4.18f1. Unity3D Retrieved 2021; https://unity.com/. Accessed 8 Jan 2021.
25. Unity Technologies. 2019.4.18f1. Unity Asset Store. https://assetstore.unity.com/. Accessed 15 Feb 2021.
26. Saaarg. 2.0. Arcade car physics. 2019. https://assetstore.unity.com/packages/tools/physics/arcade-car-physics-119484.

27. Math.Net. Math.NET Numerics. 2021. https://numerics.mathdotnet.com/. Accessed 19 Mar 2021.
28. Haupt RL. Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. In: IEEE Antennas and Propagation Society International Symposium. Transmitting Waves of Progress to the Next Millennium. 2000 Digest. Held in Conjunction with: USNC/URSI National Radio Science Meeting (C, vol. 2, 2000, pp. 1034-37 vol. 2). IEEE Xplore, https://doi.org/10.1109/APS.2000.875398.
29. Newtonsoft. 2021. Json.NET. 2021. https://www.newtonsoft.com/json. Accessed 19 Mar 2021.
30. monitor1394. unity-ugui-XCharts. XCharts. 2021. https://github.com/monitor1394/unity-ugui-XCharts. Accessed 19 Mar 2021.
31. neuecc. MessagePack-CSharp. MessagePack for C# (.NET, .NET Core, Unity, Xamarin). 2021. https://github.com/neuecc/MessagePack-CSharp. Accessed 19 Mar 2021.
32. Fisher Y, Haofeng C, Xin W, Wenqi X, Yingying C, Fangchen L, Vashisht M, Trevor D. BDD100K: a diverse driving dataset for heterogeneous multitask learning. 2018. arXiv:1805.04687.
33. Houston J, Zuidhof G, Bergamini L, Ye Y, Chen L, Jain A, Omari S, Iglovikov V, Ondruska P. One thousand and one hours: self-driving motion prediction dataset. 2020. arXiv:2006.14480.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.