DEVELOPMENT OF A 3D SIMULATION TOOL LEVERAGING GENETIC ALGORITHMS FOR NEURAL NETWORK OPTIMIZATION IN AUTOMATED DRIVING SYSTEMS: AN APPLICATION FOR SCENARIO ANALYSIS

by
Darold Davis

A thesis submitted to Johns Hopkins University in conformity with the requirements for the degree of Master of Science

Baltimore, Maryland

March 2025

© 2025 Darold Davis
All rights reserved

Abstract

As vehicular automation progresses, designing and validating reliable yet proficient automated driving systems (ADS) remains a challenging imperative, as the current state relies on live driving scenarios and datasets. This research centers on advancing ADS capabilities by proposing a simulation framework and software tool for machine learning consisting of a genetic algorithm (GA) with selection methods and artificial neural network (ANN) architectures all configurable via a user interface (UI), which guides the training and optimization of ANNs, allowing ADS policies to learn and evolve rather than relying on existing datasets.

The Unity game engine provides the development environment for crafting the UI to configure driving agents, and for selecting from three different 3D simulation scenario environments of Pasadena, San Diego, and Tampa that are constructed from OpenStreetMap data. The research experiments aim to enhance ADS performance through simulation-based evaluation which aligns with the Intelligent Transportation Systems Joint Program Office (ITS JPO) technical activities, and the National Highway Traffic Safety Administration (NHTSA) vision for safety to accelerate the safe integration of autonomous vehicles into the transportation system.

Simulation experiments using different GA and ANN configurations were conducted in each of the 3D scenario environments. Analysis of the simulation results revealed several significant patterns of evolutionary performance of ADS agents across different parameter configurations, yielding varying levels of optimization success across generations in the simulations.

Primary Reader and Advisor: Larry D. Strawser PhD, Adjunct Faculty, Johns Hopkins University, Whiting School of Engineering

Secondary Reader: In-Kyu Lim, PhD, PE, GDL Manager/Highway Research Engineer, FHWA Office of Safety and Operations R&D, U.S. Department of Transportation

Acknowledgements

Gratitude is deeply extended to Dr. Jim Coolahan for imparting the foundations of Modeling and Simulation to me and its many applications in Systems Engineering, to Kevin Dopart at the ITS Joint Program Office of the U.S. Department of Transportation for his time and valuable insight into their automated vehicle research activities and simulation areas being explored, and to Dr. Larry Strawser for serving as my advisor throughout the entire process and who I had the pleasure of meeting at INCOSE IW 2019 in the same workshop.

I would like to thank my father, Harold T. Davis, for encouraging me to pursue this master's degree and his unwavering support, and to my supporting wife Kaori, daughter Mana, and son Jin who continue to motivate me and bring a sense of purpose into my life.

Contents

Abstract ii
Acknowledgements iv
List of Tables vii
List of Figures viii
Chapter 1 – Introduction 1
Automation and Safety 1
Bio-Inspired Artificial Systems 5
Genetic Algorithms 5
Artificial Neural Networks 6
Systems Simulation 7
Thesis Motivation 7
Chapter 2 – Methodology 9
Literature Review of Related Research 9
Genetic Algorithms and Neural Networks 9
Unity Engine-based Simulations 11
Research Hypotheses 17
Design Approach 18
Simulation Framework 20
Geo-Spatial Scenario Data 22
Chapter 3 – Experimental Design 24
Evolutionary Computation 24
Genetic Algorithm Design 25
Selection Methods 27
Tournament 28
Roulette Wheel 29
Elitist 29
Tournament + 20% Random 30
Neural Network Architecture 30
System Architecture 33
Car Physics Model 35
Sensors 38
Master 39
Simulator Manager 41

Chapter 4 – Implementation 45
Scenario Environment Generation 45
Agent Car Model 48
User Interface 51
Neural Network C# Class 52
Neuron Layer C# Class 55
Genetic Algorithm C# Class 58
Recombine and Mutate 60
Chapter 5 – Results and Analysis 63
Pasadena Scenario 63
San Diego Scenario 65
Tampa Scenario 68
Chapter 6 – Conclusions and Future Research 71
Conclusions 71
Future Research 73
Bibliography 75
Appendix 1: Select UML Class Diagrams 79
1-1. Neural Network Class 79
1-2. Genetic Algorithm Class 80
1-3. Simulation Manager Class 81
Appendix 2: 3D Modeling and Physics 82
2-1. Box Collider 82
2-2. Wheel Collider 83
Appendix 3: Neural Network and Genetic Selection 85
3-1. Neural Network FixedUpdate() Method 85
3-2. Tournament C# Class 87
3-3. Elitist C# Class 89
3-4. Tournament + 20% Random C# Class 90
3-5. Roulette Wheel C# Class 92
Appendix 4: Aggregate Analysis of All Three Scenario Simulation Results 94
4-1. Correlation Analysis 94
4-2. Regression Analysis 94

List of Tables

Table 1. Comparative simulation results of the Pasadena scenario 63
Table 2. Simulation results from the San Diego scenario 66
Table 3. Tampa simulation scenario results 68

List of Figures

Figure 1. ADS levels in SAE J3016 (SAE, 2016). Reprinted from Automated Driving Systems: A Vision for Safety (NHTSA, 2017, p. 4) 2
Figure 2. Safety Risk Management. From Preparing for The Future of Transportation: Automated Vehicles 3.0. (USGPO, 2018, p. 36) 4
Figure 3. Experimental design process 19
Figure 4. Workflow of the simulation tool 20
Figure 5. Development flow of assets in Unity Engine 21
Figure 6. Transport Map and layers selection menu panel 22
Figure 7. Hillis's sorting networks. From An Introduction to Genetic Algorithms (p. 18) by M. Mitchell, 1998, MIT Press. Copyright 1998. 24
Figure 8. Genetic Algorithm activity flow 26
Figure 9. Genetic Algorithm sequence flow 27
Figure 10. Tournament selection method 28
Figure 11. Roulette Wheel selection method 29
Figure 12. Elitist selection method 30
Figure 13. Feedforward Neural Network for this experiment 31
Figure 14. Artificial neural network flow of activity 32
Figure 15. Neural Network sequence of interactions 33
Figure 16. System architecture 34
Figure 17. DrivingSystem components 35
Figure 18. Sequence diagram of CarController interaction 36
Figure 19. Activity diagram of the CarController 37
Figure 20. Sequence diagram for the WallSensor 38
Figure 21. Master component logical flow of activity 39
Figure 22. Sequence diagram of Master and Manager 40
Figure 23. Main components of the simulation system 41
Figure 24. Activity flow for the simualtion Manager 42
Figure 25. State machine of the simulations system Manager 43
Figure 26. Sequence diagram of the user and simulation 44
Figure 27. Process flow of generating the scenario environments 45
Figure 28. Pasadena mesh model scenario environment 46
Figure 29. Overdraw visualization view 47
Figure 30. One-way box colliders to prevent counter flow driving 48
Figure 31. Sensors to detect obstructions in the environment 49
Figure 32. Magenta waypoint cubes are invisible during runtime 50
Figure 33. Splash start screen with Run Simulation, Options, and Load Data 51
Figure 34. Configuration UI of parameters for the simulation experiment 52
Figure 35. NeuralNetwork class 53
Figure 36. Switch logic statement for neuron layer 55
Figure 37. The NeuronLayer class structure 56
Figure 38. CalculateLayer method for propagating the neural network 57
Figure 39. GeneticAlagorithm class structure 58
Figure 40. Awake() and Start() methods 60

Figure 41. RecombineAndMutate method for crossover and mutation 61
Figure 42. Desirable driving emerges after more than 500 generations..... 64
Figure 43. Population of agent cars select between two routes 67
Figure 44. ADS agents exploring more of the environment achieve higher fitness 69

Figure 2-1. Box Collider applied to the agent car body..... 82
Figure 2-2. Wheel Colliders applied to the agent car tires..... 83
Figure 2-3. Physics settings for the Wheel Collider in Inspector panel..... 84
Figure 3-1. FixedUpdate() method of the NeuralNetwork class..... 86
Figure 3-2. Tournament genetic selection method 88
Figure 3-3. Elitist genetic selection method..... 89
Figure 3-4. Tournament + Random 20% genetic selection method 91
Figure 3-5. Roulette wheel genetic selection method..... 92
Figure 3-6. Selection() method for fitness 93
Figure 4-1. Correlation analysis of all three scenarios 94
Figure 4-2. Max Fitness vs Neuron Layers regression analysis 94
Figure 4-3. Max Fitness vs Neurons per Layer regression analysis..... 95
Figure 4-4. Max Fitness vs Mutation Possibility regression analysis..... 95
Figure 4-5. Max Fitness vs Neurons per Layer regression analysis (quadratic)..... 95

Chapter 1 – Introduction

Today's transportation technology is marking an autonomous future for society. The U.S. Department of Transportation has a role to play in building and shaping this future by developing a regulatory framework that encourages, rather than hampers, the safe development, testing and deployment of automated vehicle technology [1]. The National Highway Traffic Safety Administration (NHTSA) released *A Vision for Safety*, a set of guidelines to promote improvements in the safety of Automated Driving Systems (ADS). This voluntary guidance serves to assist designers of ADS technology to analyze, identify, and resolve safety considerations prior to deployment using their own, industry, and other best practices. Organizations are advised to use a comprehensive systems engineering methodology to design and validate ADSs, with the primary objective of ensuring they don't pose unacceptable safety hazards [1].

When evaluating the readiness of an ADS for public road use, organizations can employ multiple testing methods, including virtual simulations, closed-course testing facilities, and real-world road evaluations [1]. Aligned with this guidance, the research presented here takes the virtual simulation approach to validation, as it can guide organizations in evaluating how much simulation might be needed before conducting tests on public roads [1].

Automation and Safety

Automated Driving Systems utilize advanced sensor arrays and adaptive machine learning software algorithms that provide a holistic perception of environmental

surroundings enabling robust operation under diverse lighting, weather, and driving scenarios [2]. These systems demonstrate dynamic adaptation to novel situations through iterative software updates, which integrate insights derived from aggregated operational data and prior experiential learning. The Society of Automotive Engineers (SAE) classification system, shown in Figure 1, provides value in contextualizing the current and potential future state of automation in driving systems like those modeled and simulated in this experiment. Benchmarking against these standardized levels enables envisioning the evolutionary trajectory and objectives for these systems over time, charting progress in pursuit of the ultimate goal for human-like automated driving.

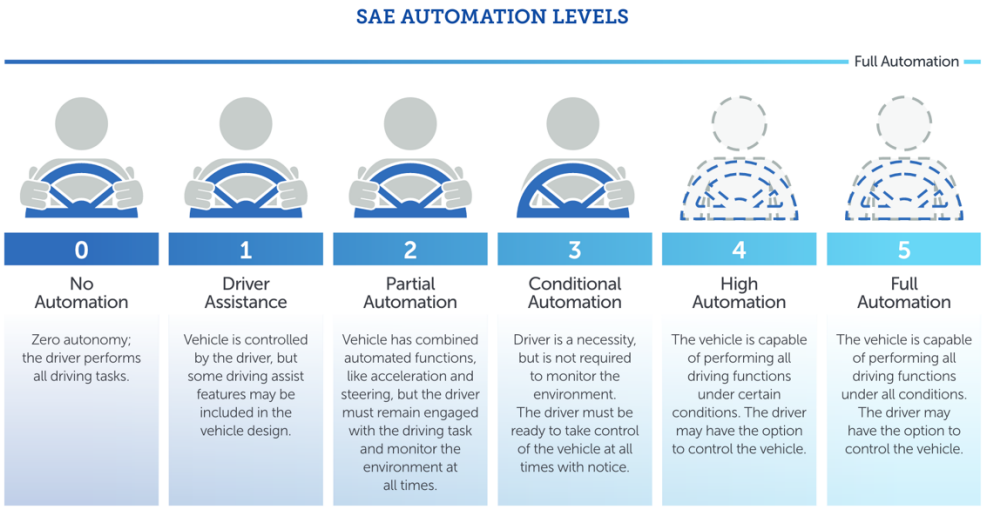


Figure 1. ADS levels in SAE J3016 (SAE, 2016). Reprinted from *Automated Driving Systems: A Vision for Safety* (NHTSA, 2017, p. 4)

Though autonomous vehicles demonstrate considerable latent capacity to augment productivity, mitigate collisions, traffic congestion, and reduce pollution, government oversight is vital in managing and ensuring the smooth incorporation of automated vehicles within the existing transportation infrastructure that will guarantee

compatibility with conventional vehicles and a wide range of road users [2], such as pedestrians, bicyclists, and motorcyclists. With such diversity of stakeholders and the expansive ecosystem of entities advancing ADS technologies, establishing a generalized conceptual framework is essential to elucidate the distinctions between developmental testing and full-scale deployment for public understanding. The framework in Figure 2 can serve to facilitate structured discourse on strategies to enhance safety protocols, mitigate risks, and maximize societal benefits arising from the integration of ADS innovations [2].

Overarchingly, contemporary determinations concerning if and how to sanction autonomous vehicle technologies will substantially impact their influence in transforming transportation ecosystems [3]. The nature and extent of the prospective utilities afforded by autonomous vehicle technologies will be contingent on the degree of automation attained. To exemplify, select safety enhancements conferred by automated functionalities such as automatic braking may be secured at modest levels of automation, whereas the putative land use and environmental payoffs likely necessitate more comprehensive automation reaching Level 4 status [3].

Studies have highlighted significant barriers to realizing fully self-driving Level 5 vehicles, including obtaining adequate on-road data capturing both optimal and imperfect driving conduct from humans [4]. To enable autonomous vehicles to independently handle end-to-end journey situations, driving datasets need to incorporate substantial quantities of real-world edge cases beyond ideal conditions so that automated skill can approach human adaptiveness and judgment. Hence procuring

ample heterogeneous data remains pivotal for advancing and validating versatile autonomous capabilities.

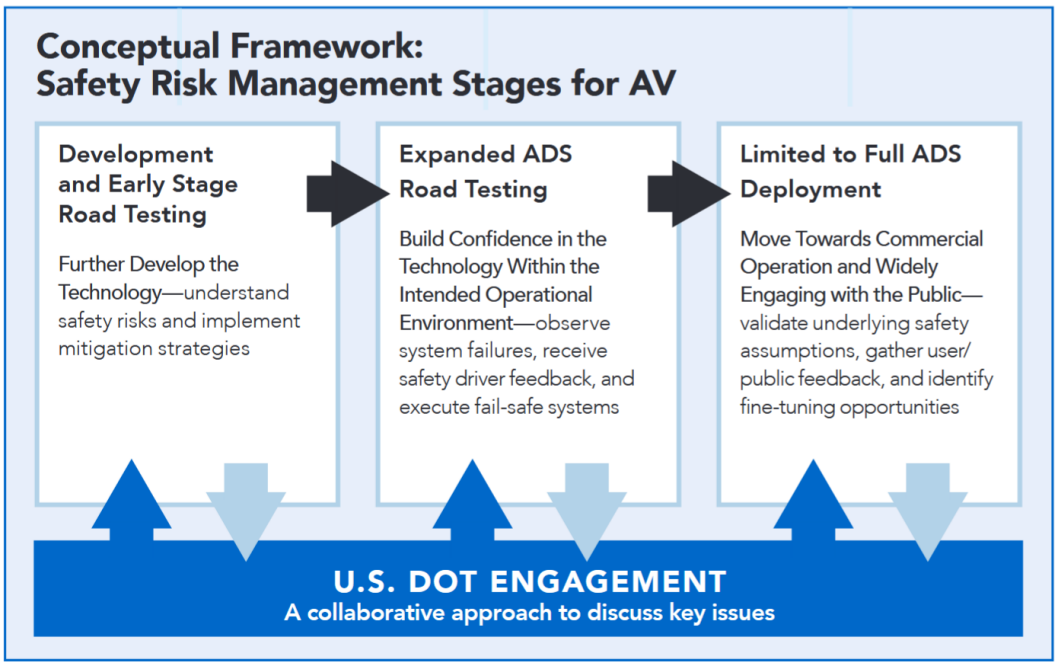


Figure 2. Safety Risk Management. From *Preparing for The Future of Transportation: Automated Vehicles 3.0.* (USGPO, 2018, p. 36)

Applying varied advanced algorithms through extensive research and development to enhance ADS technology remains necessary to further their evolution. Iteratively approaching the challenge from multiple angles can lead to enhanced capabilities, overcoming current limitations. Advancing the field will enable minimizing the risks of accidents, delivering a significant public impact from increased adoption of reliable automated driving systems.

This research centers on advancing the state-of-the-art by applying such algorithms for machine learning in the development of an application for simulating

automated driving systems at Level 4 and 5 automation by implementing a genetic algorithm, to train and optimize artificial neural networks without the dependence of existing driving datasets.

Bio-Inspired Artificial Systems

Genetic Algorithms

The precepts underpinning genetics and natural selection constitute the conceptual basis for a computational optimization and exploration method termed the genetic algorithm (GA), formulated by pioneering computer scientist John Holland through research conducted during the 1960s and 1970s [5]. This biological inspiration facilitates an adaptive search process wherein an initialized random assortment of solution candidates evolves over successive iterations guided by specified selection pressures toward maximal optimization of a fitness metric, analogous to the principle of natural selection, thereby effectively minimizing an objective cost function [5].

Genetic algorithms mimic the process of natural selection to find optimized solutions to complex problems that conventional mathematical optimization cannot handle. In particular, genetic algorithms use random mutation, crossover of solution components, and survival of the superior solutions to evolve generations of candidate solutions tailored to a target problem. Through this evolutionary technique, genetic algorithms provide an adaptive methodology to find very good, if not optimal, solutions. Moreover, genetic algorithms have proven ability to optimize solutions for complicated problems. An expanding trend connects this optimization capability with supplementary artificial intelligence methods like neural networks [5], which offer distinct benefits.

Combining the adaptive optimization of genetic algorithms with the pattern detection and predictive modeling of artificial neural networks creates a hybrid system that utilizes the strengths of both approaches for enhanced handling of intricacy and discovery of superior solutions. These genetic algorithms signify a unique strategy, especially when integrated with complementary machine learning procedures, for addressing multifarious problems intractable to traditional analytical approaches.

Artificial Neural Networks

Artificial Neural Networks (ANN) are computational frameworks designed to approximate unknown numerical values through the systematic derivation of patterns from observed data points, achieved by architecturally emulating the information processing mechanisms inherent in biological neural systems [5]. In this biomimetic paradigm, the biological neuron represents a processing unit acquiring numerous input signals. These incoming signals are subject to adjustable modulation by synaptic weights prior to aggregation. In essence, artificial neural networks seek to achieve versatile function approximation by simulating the multi-input integrative behaviors of organic neurons using tunable weighted connections. This neurobiological inspiration enables neural networks to conduct nonlinear statistical data modeling for tasks like machine learning classification, prediction and pattern recognition through exposure to representative training data. By leveraging organizational principles from neuroscience, artificial neural networks can extrapolate outputs for novel inputs after learning correlations from samples.

This hybrid framework aims to leverage the optimization power of genetic algorithms with the information processing strengths of artificial neural networks. The integrated framework of an artificial neural network guided by the genetic algorithm thereby seeks to capitalize on the unique merits of each method in order to engender solutions surpassing those from the individual techniques alone.

Systems Simulation

Simulation involves constructing a representative prototype of an actual system with which to perform exploratory experiments, enabling insight and comprehension of how the system functions, and assessment of diverse operational tactics for managing that system [6]. The simulation process allows for learning how a real-world system behaves and for optimizing approaches to its operation. The model is the core concept underlying simulations, encapsulating the key components, behaviors, and interactions that enable replicating how the actual system functions. The structure and logic captured in the simulation model give it the ability to imitate system functioning [7]. Models of genetic and neural machine learning processes have been created for this thesis research for conducting computational experiments through a simulation framework and software tool.

Thesis Motivation

A sustained intellectual curiosity in natural and biological systems, and their biomimetic potential for engineering innovation has informed both my professional trajectory and research aspirations. Previous participation in biomimicry organizations, coupled with professional experience in human-centered interface design for software

systems, has equipped me to address the challenges of developing simulation tools that enable researchers to explore biological systems within an engineering context.

This technical foundation, complemented by my *Unity Certified Associate Game Developer* certification, has crystallized my objective to synthesize these disciplines to advance the adaptability of simulation frameworks and accessibility of software tools, particularly those integrating biologically inspired computational models.

All the simulation experiments were conducted using a Dell Precision 7810 workstation with dual Xeon 3.0GHz CPUs, 128GB of system RAM, and Nvidia GeForce RTX 2080 GPU with 8GB of VRAM. Unity Engine/Editor and C# programming language were the software tools used to develop the simulator application.

This research objective is to evaluate the performance of genetic algorithms in optimizing artificial neural networks for automated driving systems in multiple environments with increasing complexity. Implemented in Unity Engine, the scenario environments are presented as 3D mesh models of U.S. cities using OpenStreetMap, and the genetic algorithm and artificial neural network is applied to a controller of an autonomous vehicle agent. The rest of the thesis is organized as follows: Chapter 2 introduces the methodology and experimental design used, Chapter 3 covers key aspects of the system architecture and design, Chapter 4 deals with implementation of the simulation environment, agent car models, genetic algorithms and artificial neural network, and in Chapter 5, the experimental simulation is carried out and the results of the experiments are analyzed. The thesis concludes in Chapter 6 with a summary of achieved results and plans for future development.

Chapter 2 – Methodology

Literature Review of Related Research

Several research activities using Unity Engine-based simulation, autonomous agents, genetic algorithms and artificial neural network optimization techniques exist in the technical literature that served as reference sources in informing aspects of this research and the experiments conducted.

Genetic Algorithms and Neural Networks

Ma, et al [8] uses neural networks optimized by genetic algorithms for logistics demand forecasting. Fundamentally, the study argues that this hybrid approach can improve forecasting accuracy to support intelligent logistics distribution systems. The authors summarize a proposed model combining neural networks and genetic algorithms for freight vehicle routing in complex urban settings. The primary point emphasized is that genetic algorithms help optimize neural network weights for higher precision forecasts. Testing results demonstrating over 96% accuracy are presented to validate the methodology's feasibility for logistics planning and optimization tasks.

The research work of H. Ma [9] explored optimizing logistics distribution routes using a hybrid genetic and neural network approach. The core proposition of the work asserts that combining neural networks for processing data with genetic algorithms for optimization provides effective route selection exceeding traditional methods. The researcher summarizes a proposed model using neural networks to establish an evaluation matrix for ranking route options based on cost, capacity, and traffic. Genetic

algorithms then operate on this matrix to identify optimal routes. This approach avoids limitations of conventional algorithms and testing validates lower complexity, faster convergence, and higher computational efficiency compared to baseline methods.

Feng and Zhu [10] research of using genetic algorithms to train neural networks for self-driving car vision and navigation is of most relevance to this experiment. Feng and Zhu's central thesis is that genetic algorithms can replace traditional feedback regulation in neural network training, enabling sample-efficient learning for tracking tasks. The researchers summarize an approach using genetic algorithms to evolve feedforward neural network weights over generations for visual tracking in a Unity simulation. The significance of the research is that this method does not require large datasets, and testing shows the trained network can guide an autonomous vehicle to avoid obstacles and reach targets. Though directly related to the research in this experiment, Feng and Zhu do not specify the ability to explore different configurations to identify the most optimal performing solutions.

In the work of Lu and Kuang, an injury prediction model to enhance automatic crash notification systems for faster emergency response was developed [11], which combined genetic algorithms and neural networks to improve prediction accuracy over other methods. The authors summarize an approach using National Highway Traffic Safety Administration data to train a genetic algorithm-optimized neural network model to categorize driver injuries from crashes. Analyzing correlations and using a hybrid GA-NN method appear to increase reliability and outperform SVM, LSTM, standard NN, and logistic regression models. Testing results demonstrating higher accuracy are presented

to validate the methodology's ability to optimize emergency medical dispatch through injured driver classification.

The main theme of [12] is reviewing and analyzing the use of genetic algorithms for railway system optimization. At the heart of the argument lies the claim that GAs have emerged as a popular technique for addressing the complex, multi-objective problems inherent in railway operations and design. The researchers reviewed highly cited and recent studies applying GAs and hybrid GA approaches to diverse railway challenges including scheduling, routing, forecasting, design, maintenance, and allocation. Their views are that GAs can effectively handle the nonlinear, time-varying nature of railway systems, and that hybridizing GAs with methods like particle swarm optimization, ant colony optimization, and neural networks enhances capabilities. A comprehensive bibliometric analysis of over 250 publications was presented to provide a roadmap identifying opportunities and research gaps in using evolutionary algorithms for railway optimization.

Unity Engine-based Simulations

In Proximal Policy Optimization Through a Deep Reinforcement Learning Framework for Multiple Autonomous Vehicles at a Non-Signalized Intersection, D. Quang et al uses advanced deep reinforcement learning to develop autonomous vehicle control models for improving traffic flow efficiency in mixed-autonomy environments [13]. They argue that leading autonomous vehicles, even at low penetration rates, can help optimize mobility by reducing delays and improving average speeds at intersections lacking traffic signals. Their research involved developing and testing a

proximal policy optimization reinforcement learning model to validate the benefits of autonomous vehicles for non-signalized intersection control. Using the SUMO tool the researchers demonstrated an approach that enables the reliable simulation of mixed-autonomy scenarios, and that the results demonstrate clear traffic improvements from integrating just a small percentage of autonomous vehicles.

Radwan, Sedky and Mahar research work uses reinforcement learning to train autonomous vehicles in simulated environments as an alternative to risky real-world data collection [14]. The foundational claim posits that Unity's ML-Agents toolkit enables evaluating different reinforcement learning algorithms for obstacle avoidance in self-driving cars. Their approach finds the optimal algorithm by testing proximal policy optimization (PPO) and soft actor-critic (SAC) on virtual driving scenarios using single and multi-agent training as well as camera and LiDAR sensors. They show that simulation training mitigates risks and labeling efforts compared to real-world data, and that the results demonstrate PPO and LiDAR outperform SAC and cameras in the experiments.

Improving the safety and efficiency of deep reinforcement learning for autonomous driving [15] is the central theme in the research of Cui et al. The central thesis is that a double-bias experience replay approach allows agents to choose their own driving learning tendency, enabling faster speeds while maintaining stability. This is demonstrated by proposing a new loss function and applying double-bias experience replay to DQN, DD-DQN, and QR-DQN algorithms thereby allowing agents to bias experience sampling from safe or risky driving improves performance over single

experience replay. Testing on a Unity-based simulator demonstrates the proposed approach enables agents to achieve higher rewards, increased speed, fewer lane changes, and more stability during training.

The research done in Evaluation of Proximal Policy Optimization with Extensions in Virtual Environments of Various Complexity focused on evaluating proximal policy optimization (PPO) reinforcement learning for control in a racing game scenario [16]. This combined PPO with imitation learning and curiosity-driven exploration to improve performance in completing racing circuits. Experiments were carried out in a custom Unity racing game using PPO alone and with extensions. The researchers found that hyperparameters tuning, learning curves, and inference metrics demonstrate PPO with imitation learning and curiosity modules surpasses base PPO in driving the agent.

Saez et al also present research of direct relevance in using genetic algorithms for automatic vehicle control, specifically focused on optimizing lap times in racing scenarios [17]. Their view is that evolutionary computation techniques like genetic algorithms can enable automated vehicle guidance and improve performance in domains like racing where aerodynamics, fuel efficiency, and power output are critical. This is shown by applying genetic algorithms to learn optimal driving policies that minimize lap times across different circuits. Central to the research are the claims that automated driving is an important capability with applications ranging from commercial safety to racing optimization, and that genetic algorithms present a promising approach for vehicle guidance learning in racing contexts where lap time is the primary objective.

Arrigoni et al presented work in real-time trajectory planning for autonomous vehicles using nonlinear model predictive control optimized by a genetic algorithm [18]. The primary assertion underpinning the analysis is that this approach enables effective obstacle avoidance and handling under various friction conditions with real-time performance. The author succinctly captures an NMPC implementation using a nonlinear single-track vehicle model and Pacejka tire formulas, with the NMPC problem solved through a proposed genetic algorithm strategy. The main assertions put forward are that simulations demonstrate robust performance under challenging conditions, and computational analysis proves real-time feasibility.

Naveen et al explored developing an automated driving assistance system for highways using deep reinforcement learning [19]. The overarching premise advanced is that deep RL can enable driver assistance features like cruise control and automatic braking to be extended for more complete vehicle autonomy in limited highway scenarios. The text summarizes a proposed approach where a machine learning agent leverages camera images and LiDAR data to make driving decisions through deep reinforcement learning algorithms. Among the core argument outlined is that this provides understanding of the surroundings to automate acceleration, braking, and lane changing while avoiding collisions. Testing in a simulated highway environment is presented to validate the methodology's ability to learn policies that deliver automated highway driving assistance.

Optimizing robustness and performance of deep reinforcement learning networks using genetic algorithms and neuron coverage [20] is featured in the research of Al-

Nima et al. Central to the author's position is the idea that a proposed Genetic Algorithm of Neuron Coverage (GANC) approach can optimize neuron coverage of DRL networks by generating augmented training inputs. In brief, the work recaps an application of GANC to self-driving car decision making, where reliability across diverse road views is critical. Testing on a driving dataset demonstrates maximized neuron coverage via the genetic algorithm produces superior driving accuracy over previous state-of-the-art results. The pivotal point articulated focus on the use of GANC which leverages genetic algorithms to increase neural coverage and augment training data, yielding more robust DRL policies.

Training autonomous vehicles to navigate environments using neural networks optimized by evolutionary algorithms was explored in the work by Samuel [21]. The driving hypothesis of the text suggests that neural networks can steer cars based on evolved weights from genetic algorithms. The analysis distills the essence of a 2D Unity simulation where cars learn driving policies by genetically evolving neural networks over generations. The principal idea presented revolve around standard feedforward neural networks being trained through genetic algorithms to control steering, and that this evolutionary approach enables cars to incrementally improve navigation skills. Testing in simplified environments with just lanes and static obstacles is presented to demonstrate the feasibility of the proposed technique. In essence, the text proposes and validates using genetic algorithms to optimize neural network weights for autonomous vehicle control, incrementally evolving driving proficiency over generations in a simulated environment.

Application of Neuroevolution in Autonomous Cars explores training autonomous electric vehicles through neuroevolution in simulated environments as an alternative to large real-world datasets [22]. This highly relevant research maintains that genetic algorithms like neuroevolution enable incremental optimization towards driving proficiency without reliance on massive data. The research consolidates key points about an approach using neuroevolution, a form of genetic algorithm, to evolve self-driving policies in a physics-based Unreal Engine simulation. Key takeaways from the analysis suggest that this evolutionary technique exploits serendipitous discoveries to develop driving skills from scratch, provides generalizable capabilities transferable to real-world applications, and offers a foundation for integrating other machine learning methods. Testing in the simulated environment is presented to demonstrate the feasibility of bootstrapping end-to-end autonomous driving through neuroevolution.

Muzahid et al employed reinforcement learning techniques like proximal policy optimization (PPO) and soft actor-critic (SAC) to enable autonomous vehicles to safely change lanes and avoid multiple vehicle collisions [23]. The work's principal contention revolves around both PPO and SAC successfully learning optimal driving policies for collision avoidance in simulated environments, with SAC demonstrating greater data efficiency but PPO providing more stable training. The discussion provides a condensed overview of an approach where agents are trained using Unity and ML-Agents to incrementally develop behaviors that allow smooth lane changing through sensing proximity of surrounding vehicles and adjusting speed accordingly. The foundational arguments advanced center on the 91-96% success rate of the trained agents in avoiding collisions which demonstrates that reinforcement learning enables agents to

acquire robust driving policies for critical autonomous vehicle functions like lane merging.

Research Hypotheses

In all of the previous research, there is a lack of UI tools for configuring different genetic selection methods and neural network architectures to explore simulation scenarios. It is here where this research contributes to advancing the simulation domain and autonomous systems research. To assess the safety of ADS technology in alignment with NHSTA objectives, this software tool will allow for testing the following hypotheses:

- Increasing the number of neurons and neuron layers leads to higher maximum navigational fitness for the cars.
- Higher mutation rates result in more exploration of the search space but lower average population fitness over generations.
- Using certain selection methods maintains higher genetic diversity compared to other selection methods.

Furthermore, the dataset-independence of the experiment has the advantage of also exploring scenarios not contained in existing datasets, thereby being able to adapt and learn from the environment. Such an approach regardless of positive or negative results will allow for the evaluation of virtual simulations prior to on-road testing.

Design Approach

Executing illuminating simulation-based systems research necessitates following a rigorous experimental research protocol encompassing model construction, simulation execution, and results analysis activities to reliably evaluate theoretical system behavior predictions. Virtual experiments using computer simulations to study natural or artificial system phenomena require clearly defining the simulation and modeling methodology to facilitate constructing an appropriate conceptual system representation. Running the simulation model experiment then produces informative results if the simulation process accurately captures the dominant components and interactions of real-world systems.

This research applies an experimental design approach illustrated in Figure 3 that will allow for evaluations of simulation scenarios to better understand the factors that influence a particular ADS configuration for safe drivability. The user interface (UI) of the simulator tool will enable researchers to identify and configure factors that impact simulation experiments. Following a systematic approach, the process begins with a screening design that incorporates key configurations likely to influence outcomes. By isolating the most critical factors, the system then transitions to an optimized experimental design phase for further refinement.

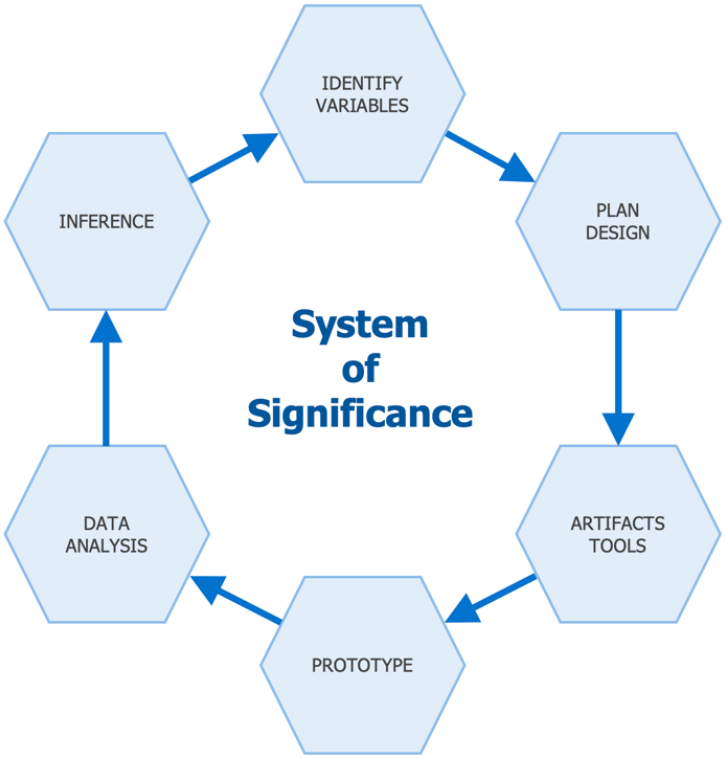


Figure 3. Experimental design process

The main objectives of this experimental research are as follows:

- Developing a genetic algorithm and neural network architecture for machine learning to enhance policy optimization of automated driving systems without dependence on large datasets.
- Creating a user interface (UI) for exploring different genetic selection methods and neural network architecture configurations.
- Investigating iterative improvement of autonomous vehicle controllers across generations via bio-inspired evolutionary algorithms for adaptive policy search in the machine learning training simulations.

Control variables are:

- Genetic Selection Methods.
- Neurons and Neuron Layers.
- Mutation Possibility and Rate.

Figure 4 illustrates the general flow of the simulation software tool. Accessible from the start screen is the configuration settings UI with options to adjust several parameters of the simulation.

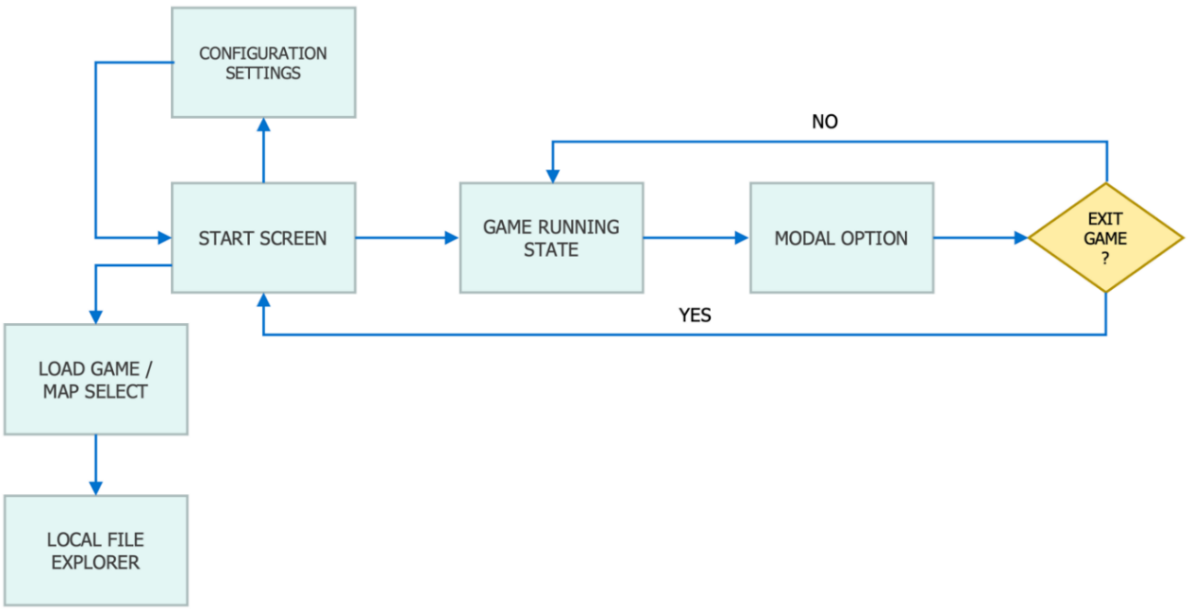


Figure 4. Workflow of the simulation tool

Simulation Framework

Unity Engine forms the core technology of the framework depicted in Figure 5 from which this simulation tool is built. In Unity, Scenes are asset bundles that compositionally aggregate GameObjects [24]. GameObjects are the fundamental

elements which encapsulate graphical and functional components [25], and through Scripts, GameObjects can respond to user input, drive physics simulations, render graphical effects, and implement custom character behavior [26]. Together, they provide the working canvas for game and application content by orchestrating collections of modular GameObjects to construct interactive environments using scripting logic. Specifically, the interplay between configurable GameObject components, Scene aggregation, and cross-cutting scripts constitutes the core mechanism for developing Unity gameplay and interactive simulations by linking interface appearance and underlying behavior across objects to enable manipulating environments, triggering events, producing graphics, and defining machine learning agents.

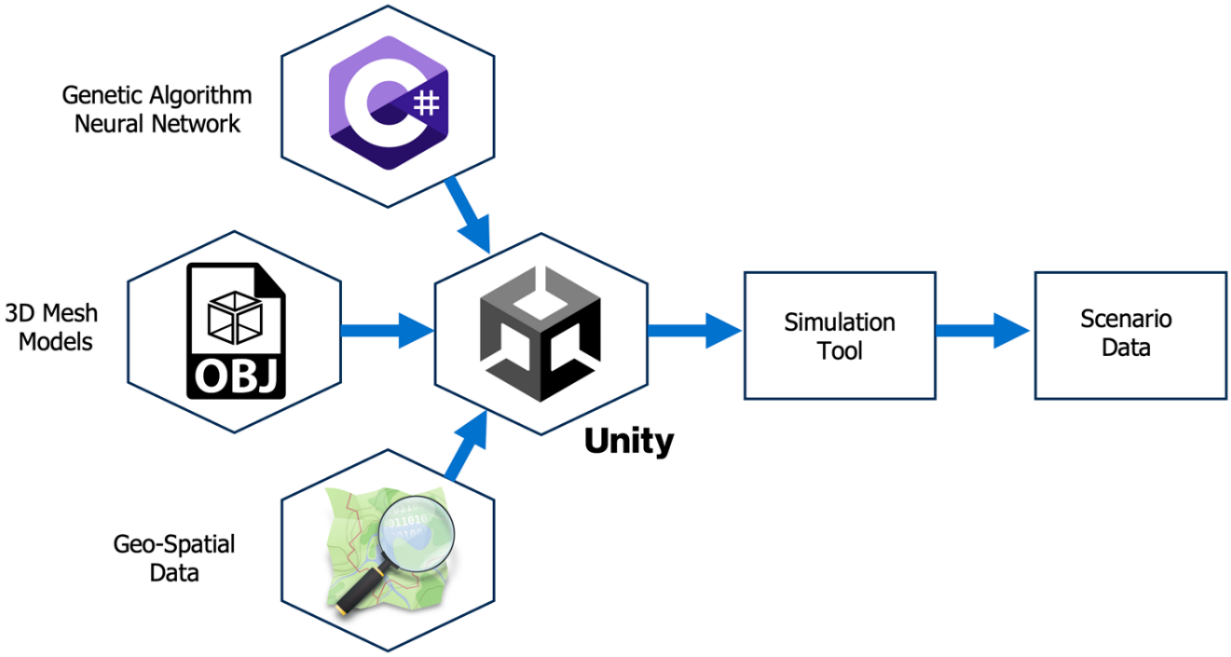


Figure 5. Development flow of assets in Unity Engine

Once the plans and designs are implemented in Unity, simulation experiments of various scenarios can be carried out and results analyzed.

Geo-Spatial Scenario Data

In order to create training environments for the machine learning agents in the simulations, OpenStreetMap (OSM) geospatial data was selected which uses tile map layers to display different features and annotations on top of the base geographic map. The map uses a topological data structure composed of Nodes, Ways, Closed Ways, Areas, and Relation all of which have tags [27]. OpenStreetMap Carto (Standard) is an open-source stylesheet that renders OpenStreetMap vector data into customizable raster map tiles [28], CyclOSM focuses on information relevant for bikers like trails and parking [29], while the Transport Map style as seen in Figure 6 spotlights global transit networks including railway lines and bus routes [30].

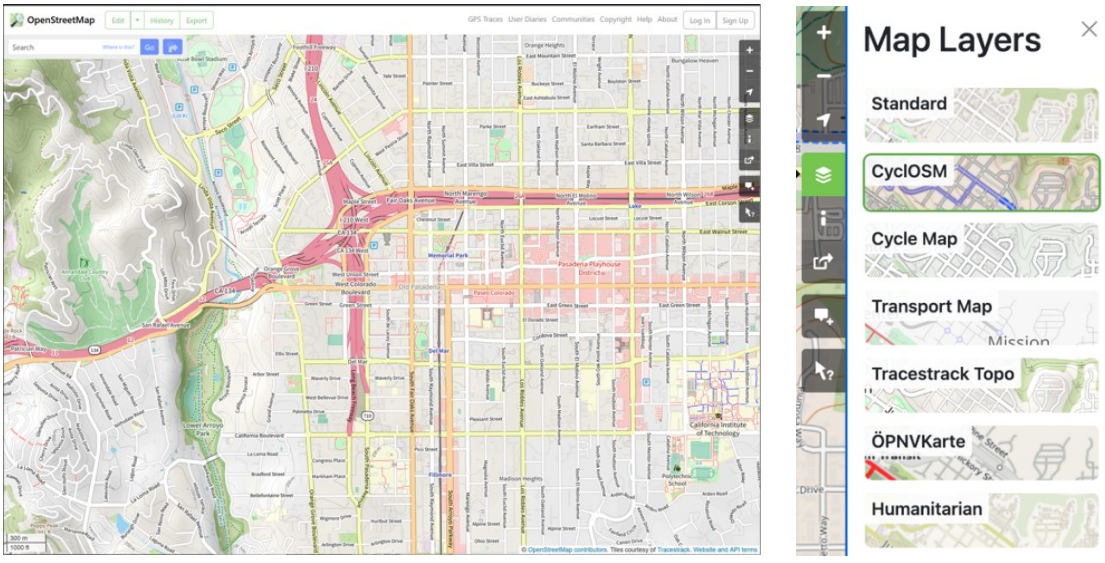


Figure 6. Transport Map and layers selection menu panel

ÖPNVKarte is a public transit-centered renderer underscoring routes and stops [31], and the Humanitarian map style highlights water sources, lighting, roads, social services and other assets useful for emergency response [32]. These alternative

renderers allow tuning map visualization for particular use cases while still leveraging the comprehensive content of the OpenStreetMap database.

Using a hard selection criterion of global coverage, traffic demands, reliability, and up-to-date data [33], Pasadena, San Diego, and Tampa metro areas were selected for the research experiments to construct simulated urban driving scenarios for investigating automated vehicle systems, with the bounding box of the target regions encoded as latitude/longitude coordinates in JSON objects to facilitate procedural generation in the Unity Engine. This OSM data was then imported into Unity by way of a special plugin tool that converts it into a 3D mesh model. Chapter 4 discusses the implementation details of this process.

Chapter 3 – Experimental Design

Evolutionary Computation

A number of computer scientists independently investigated evolutionary phenomena with the concept that the principles underpinning biological evolution could be harnessed as an optimization technique for solving complex engineering challenges [34]. The fundamental idea across these systems was to simulate the process of natural evolution to evolve a pool of candidate solutions for a target problem, employing computational analogues of genetic variation mechanisms and natural selection pressures seen in nature.

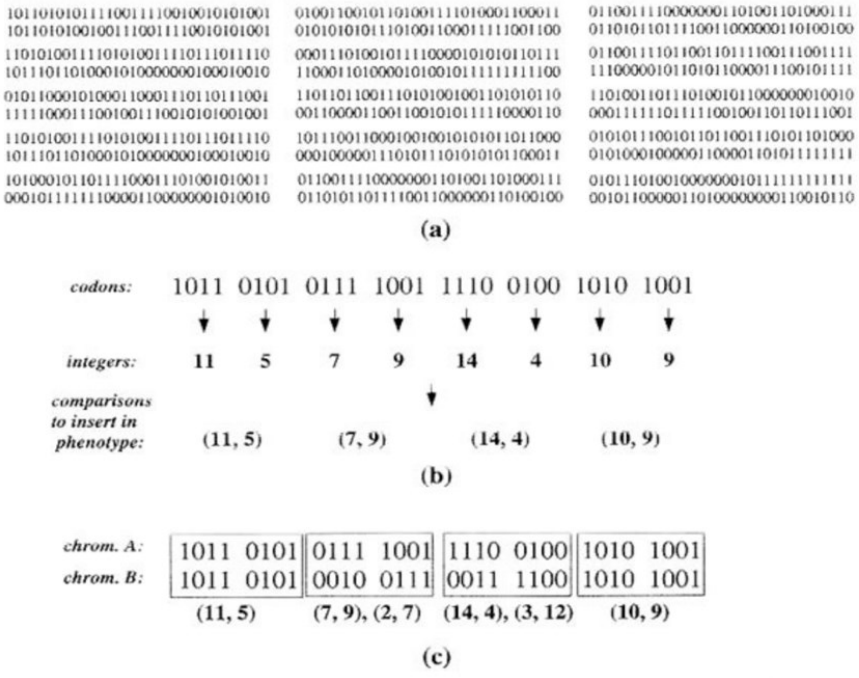


Figure 7. Hillis's sorting networks. From *An Introduction to Genetic Algorithms* (p. 18) by M. Mitchell, 1998, MIT Press. Copyright 1998.

Genetic Algorithm Design

In his seminal 1975 book *Adaptation in Natural and Artificial Systems*, Holland put forth genetic algorithms, which are modeled on an abstract depiction of biological evolution, as an adaptive optimization search method. This established the theoretical framework for a computational technique to evolve new populations of solution-encoding "chromosomes" (bit-strings) from an initial population using selection principles combined with genetics-inspired operators like crossover, mutation and inversion. Since then, many other researchers have applied genetic algorithms in their work, such as can be seen in Figure 7 which shows three example experiments in genotype representation of sorting networks by W. Daniel Hillis, who used a genetic algorithm in designing an optimal $n = 16$ sorting network.

In this experimental research, the genetic algorithm operations that will be used in order to evaluate and select the best solutions are as follows:

1. **Fitness function:** it evaluates the performance of each candidate.
2. **Selection:** it chooses the best individuals based on their fitness score.
3. **Recombination:** it replicates and recombines the individuals.
4. **Mutation:** this operator randomly flips some of the bits in a chromosome which can occur at each bit position in a string with some small probability.

Figure 8 illustrates an activity flow diagram model detailing the steps in the process of the genetic algorithm for these experiments.

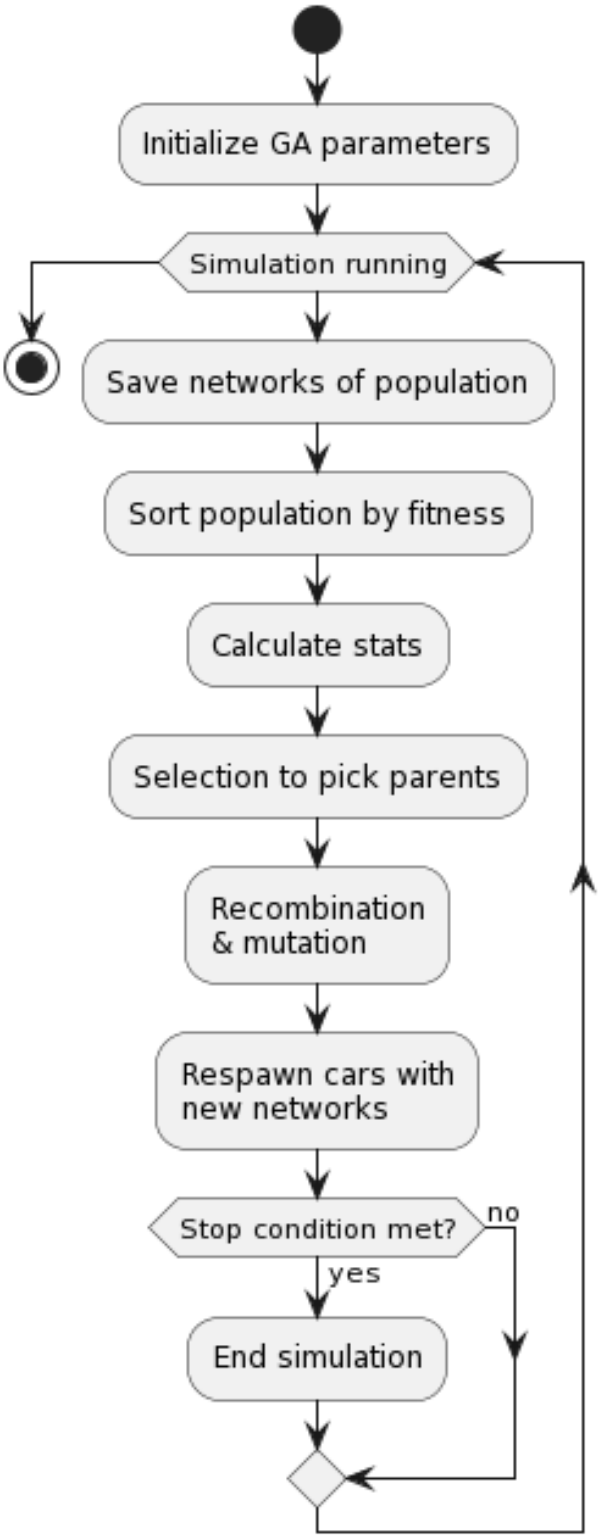


Figure 8. Genetic Algorithm activity flow

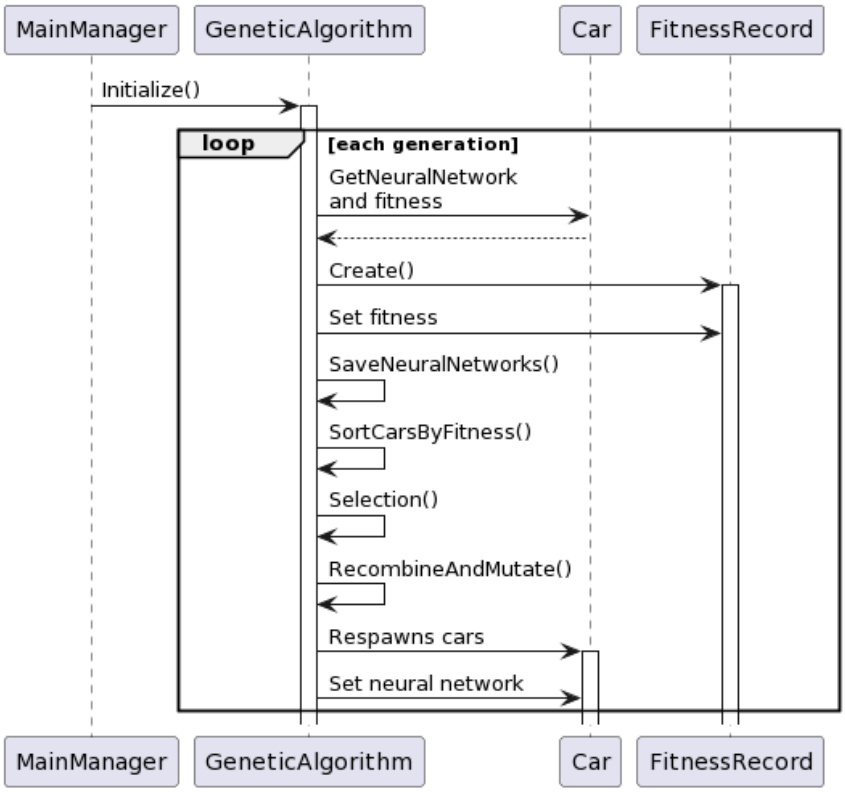


Figure 9. Genetic Algorithm sequence flow

In this system, the genetic algorithm supervises the evolutionary procedures of selection, crossover, and mutation. Figure 9 illustrates how it governs the probabilities of mutation and crossover, as well as maintaining records of all phenotypic traits for each vehicle-encoded solution across every iteration of the evolving population. By managing variation rates and cataloging attributes, the algorithm directs optimization towards incrementally improved subsequent generations.

Selection Methods

The objective of selection in genetic algorithms is to identify the fitter solution chromosome representations in a population and allow them to pass on features to subsequent generations, gradually improving offspring fitness over iterative evolution

[34]. The right equilibrium between exploiting the fittest solutions via selection and exploring through variation operators facilitates steady refinement rather than premature convergence or stagnation. Managing this tension allows fit genomes to propagate beneficial traits while maintaining enough diversity through mutation and recombination to continue optimization momentum. The four selections methods used in these experiments are detailed below:

Tournament

The Tournament selection method in genetic algorithms works by running "tournaments" among a few individual chromosomes randomly chosen from the population seen in Figure 10, and selecting the winner of each tournament to be a parent for the next generation.

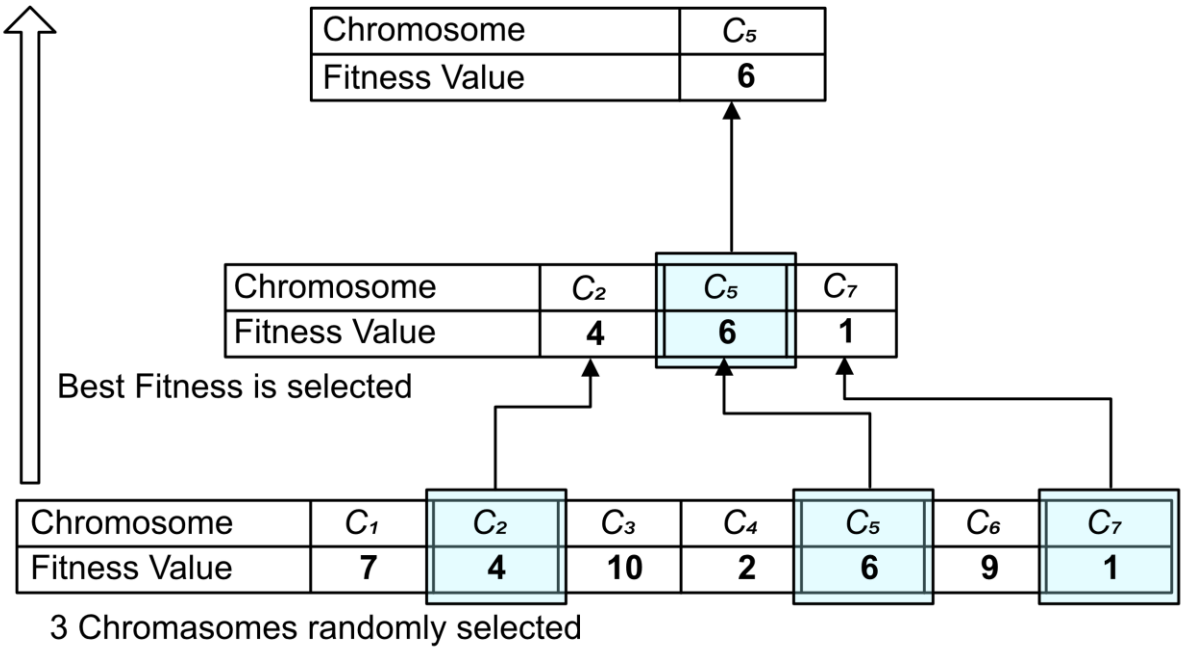
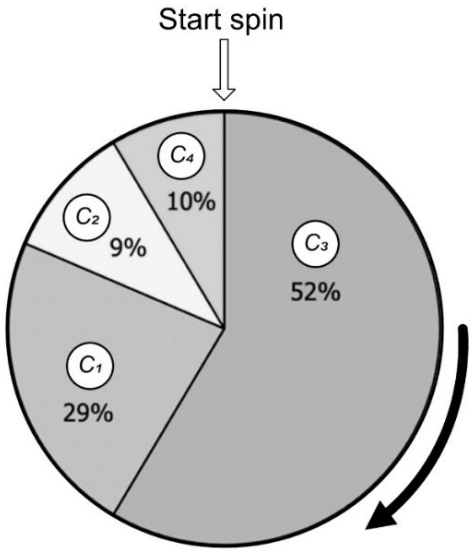


Figure 10. Tournament selection method

The tournament size controls the selection intensity - larger values make it more likely the fittest will get selected.

Roulette Wheel

The Roulette Wheel selection method in genetic algorithms works by assigning all individuals in the population a slice of a roulette wheel proportional to their fitness score as shown in Figure 11. Then the wheel is virtually "spun" multiple times to randomly pick individuals based on their slice size to be parents for producing the next generation.



Chromosome	Fitness Value
C_1	**.29**
C_2	**.09**
C_3	**.52**
C_4	**0.1**

Figure 11. Roulette Wheel selection method

Elitist

In Elitist, the top 50% supplements existing selection techniques in genetic algorithms by preserving several top-performing solutions per generation [34]. This

mechanism ensures the evolutionary search retains previously discovered peaks while seeking potentially superior candidates as depicted in Figure 12, balancing exploiting the current best with exploring uncharted landscapes. By safeguarding hard-won gains against destructive variation, elitist enhances traditional selection approaches for improved genetic algorithm reliability and efficiency.

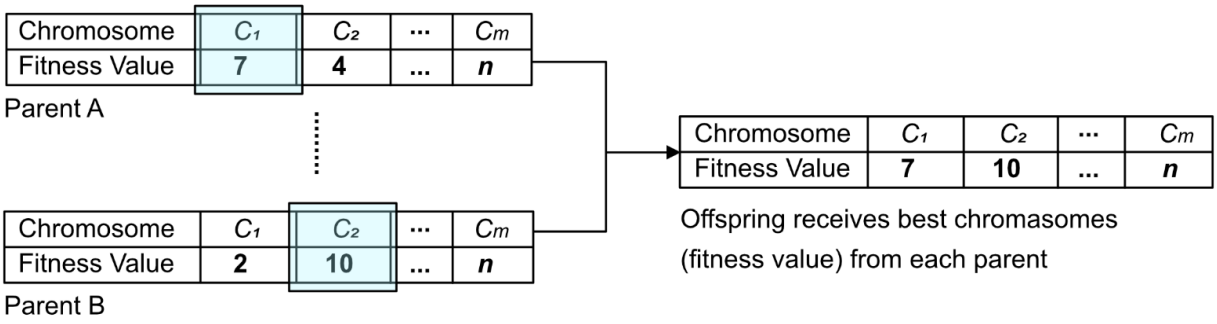


Figure 12. Elitist selection method

Tournament + 20% Random

Similar to the Tournament, this selection method uses tournament selection for pairing parents by fitness, along with mutation and recombination operators that clone the elite chromosomes, reset the bottom 20% portion, and crossover the middle chromosomes. The algorithm tries to balance retaining an elite specimen, recombining good chromosomes, and introducing randomness for continued evolution over generations.

Neural Network Architecture

A basic feedforward artificial neural network comprises interconnected neurons over weighted links that activate based on input signals propagating unidirectionally through the network layers, roughly imitating biological neural activation flows [4]. It

takes an input activation pattern that spreads across weighted connections toward the output layer, mimicking how neural networks in the brain process signals. The feedforward-only architecture without feedback loops differs from recurrent networks allowing two-way layer activation flows.

Each of the ADS agents in this simulation system utilizes a personalized neural network responsible for feeding sensor data forward in order to dictate steering and acceleration outputs as shown in Figure 13. This recursive tuning methodology, known as feedforward propagation, ingests perceptual information detailing proximities to environmental contours, then updating connection strengths through feedforward passes that accept sensory data and output driving controls.

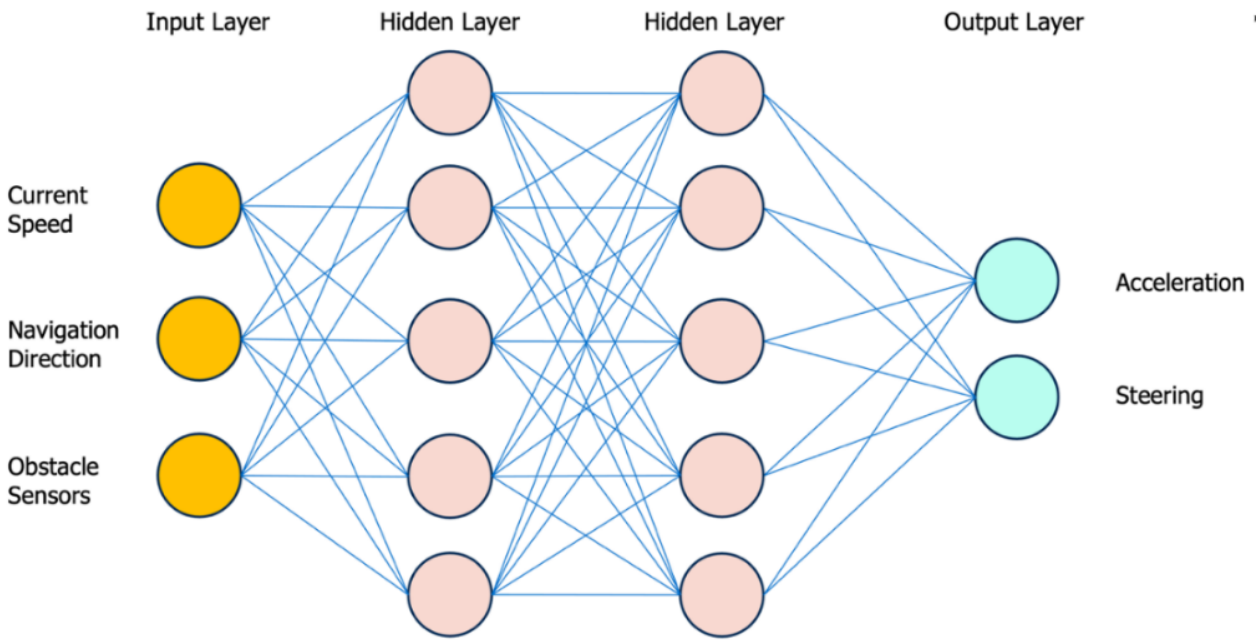


Figure 13. Feedforward Neural Network for this experiment

Each ADS agent learns implicit policy mapping scenarios to advantageous actions, bypassing hand-coded rules. For this particular system, the artificial neural

network architecture flow of activity is illustrated in Figure 14. Appendix 1 visually diagrams the structure and relationships of the NeuralNetwork class.

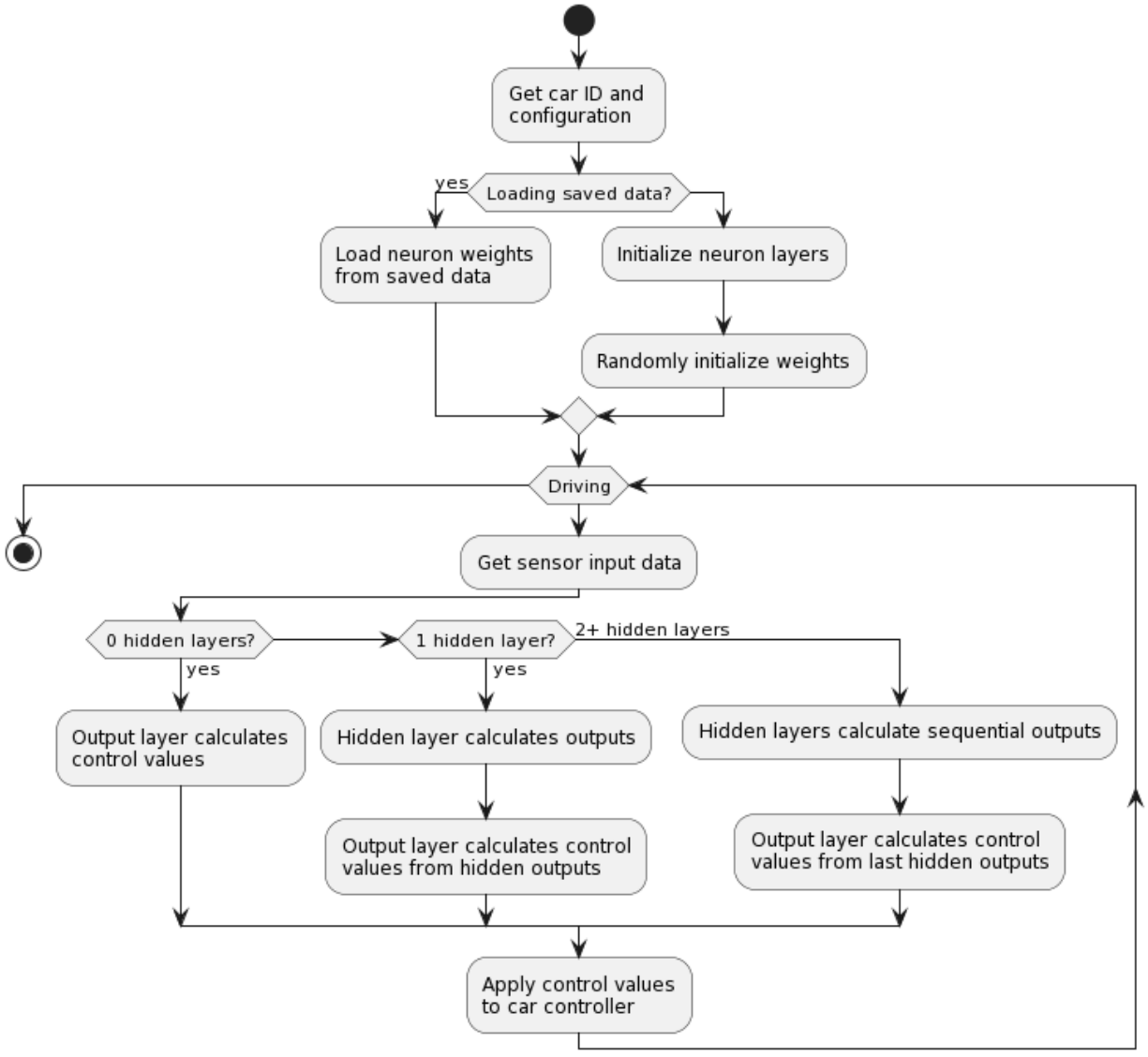


Figure 14. Artificial neural network flow of activity

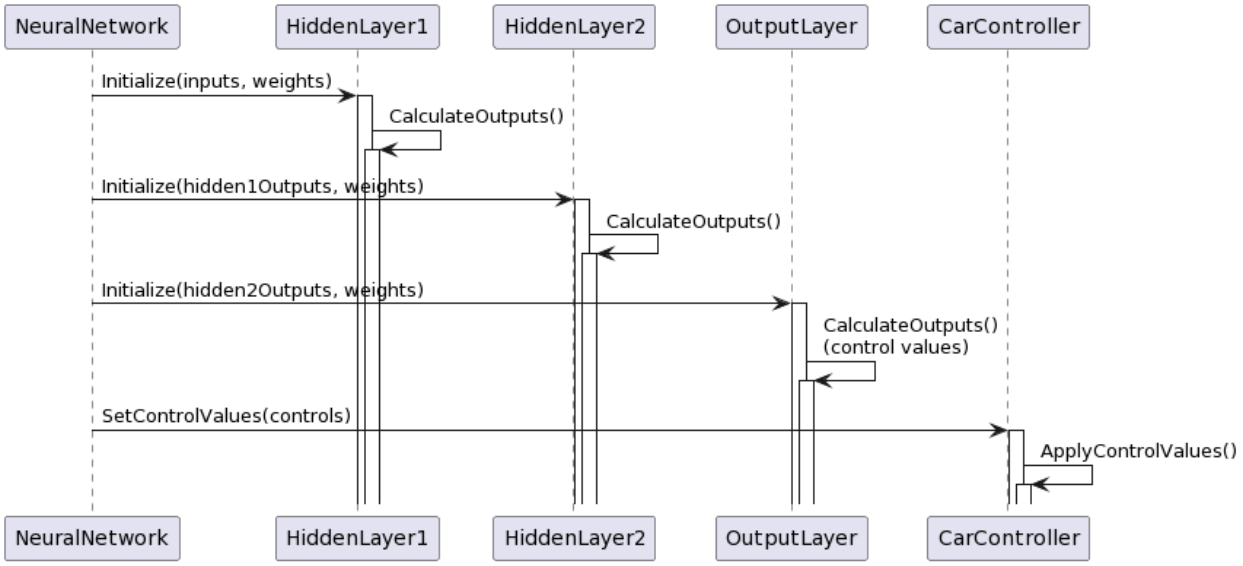


Figure 15. Neural Network sequence of interactions

Successive invocations will traverse the computational graph from input layer through hidden layers to the steering and acceleration output layer as the sequence model in Figure 15 illustrates. The system thereby avoids hand-authored control logic in favor of flexible data-driven controls mediated by optimized network weights. Weight inheritance and mutation is implemented between generations to enable online evolution of control policies with gradually accruing task proficiency. This neural network infrastructure offers a pathway toward sophisticated and adaptive autonomous systems capable of responding sensitively to complex scenarios.

System Architecture

Figure 16 focuses on defining the critical components both inside and external to the system boundary, as well as the key inputs and outputs that drive the simulation scenarios. It serves as a concise yet comprehensive architectural map of how the major pieces fit together.

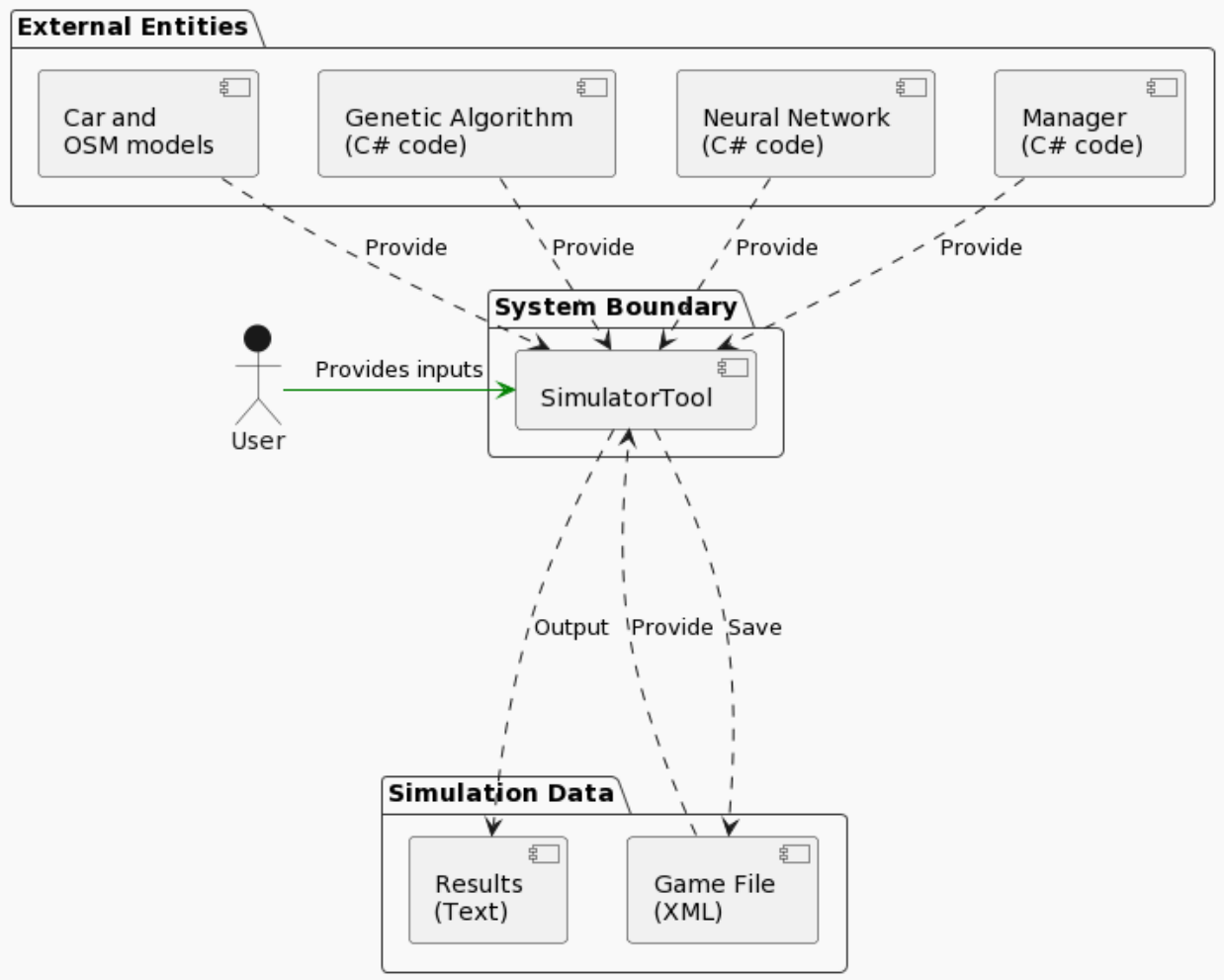


Figure 16. System architecture

The SimulatorTool sits within the system boundary and serves as the core simulation engine that runs the driving scenarios. External entities feed data and functionality into the SimulatorTool. This includes 3D visual assets like cars and environments, machine learning logic coded in C# for genetic algorithms and neural networks, a Manager module to coordinate the simulation, and a User that provides real-time inputs. Simulation data components exist outside the SimulatorTool to store the scenario configurations and output performance results. The Game File stores setup parameters in XML, while Results hold text summaries of the simulation runs.

Car Physics Model

Figure 17 illustrates the architecture and interactions of the DrivingSystem comprised of four key components - the CarController, Input, WheelColliders, and Rigidbody.

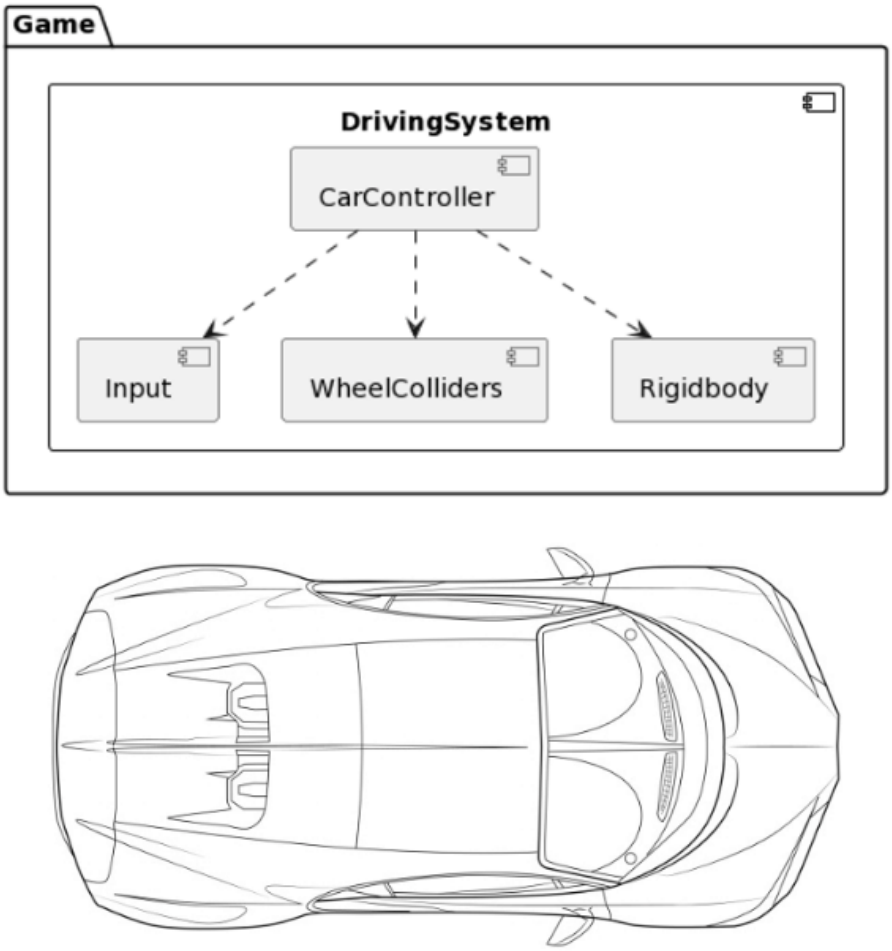


Figure 17. DrivingSystem components

The CarController is the main component responsible for controlling the car. Interactions in Figure 18 show how it relies on the other three components:

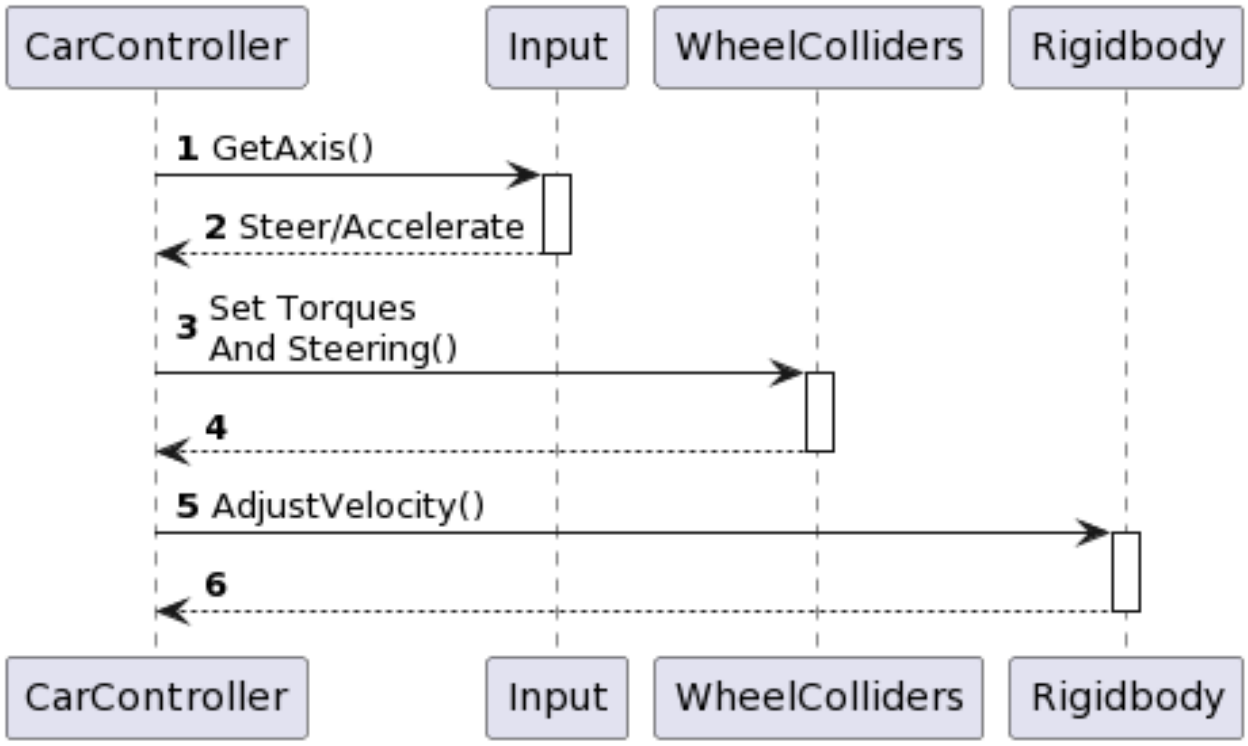


Figure 18. Sequence diagram of CarController interaction

- **Input**: component that handles all user input and control schemes, providing steering and acceleration values to the CarController.
- **WheelColliders**: contains the wheel collider physics models that handle traction, friction, and suspension. The CarController sets torques and steering.
- **Rigidbody**: the main physics component that handles momentum and collision response on the car. The CarController adjusts its velocity each frame.

Together, these components enable realistic car driving physics and handling within the simulation game. The CarController component orchestrates the rest -

retrieving user input, applying physics forces on wheel colliders and Rigidbodies, and adjusting velocity based on steering. Figure 19 shows that on Start(), the system components are initialized to get the car ready for driving. Then in each frame, user input is retrieved to determine values for steering and acceleration.

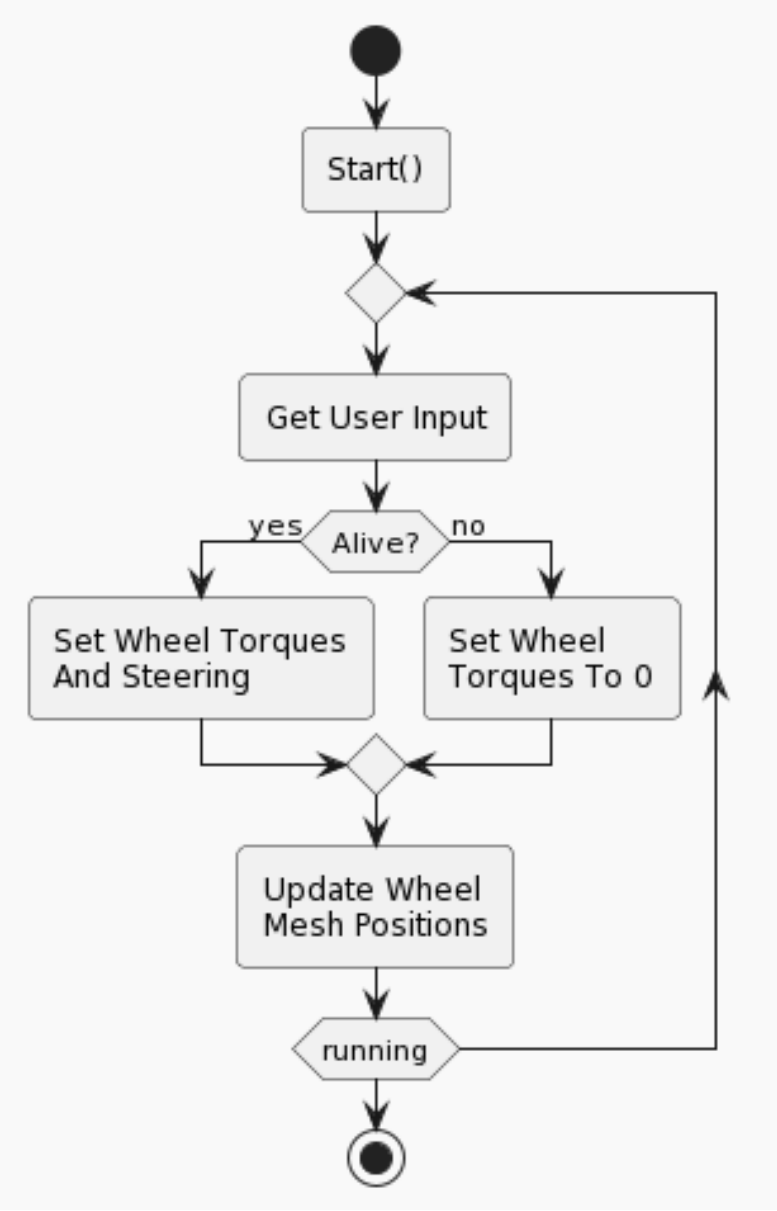


Figure 19. Activity diagram of the CarController

Sensors

Figure 20 shows the discrete sequence of operations that execute over the lifetime of a WallSensor. When Start() initializes the class, the FixedUpdate() method engages in a repetitive game loop. Rays are then cast out which senses the environment and distances recorded depending on collision detection results.

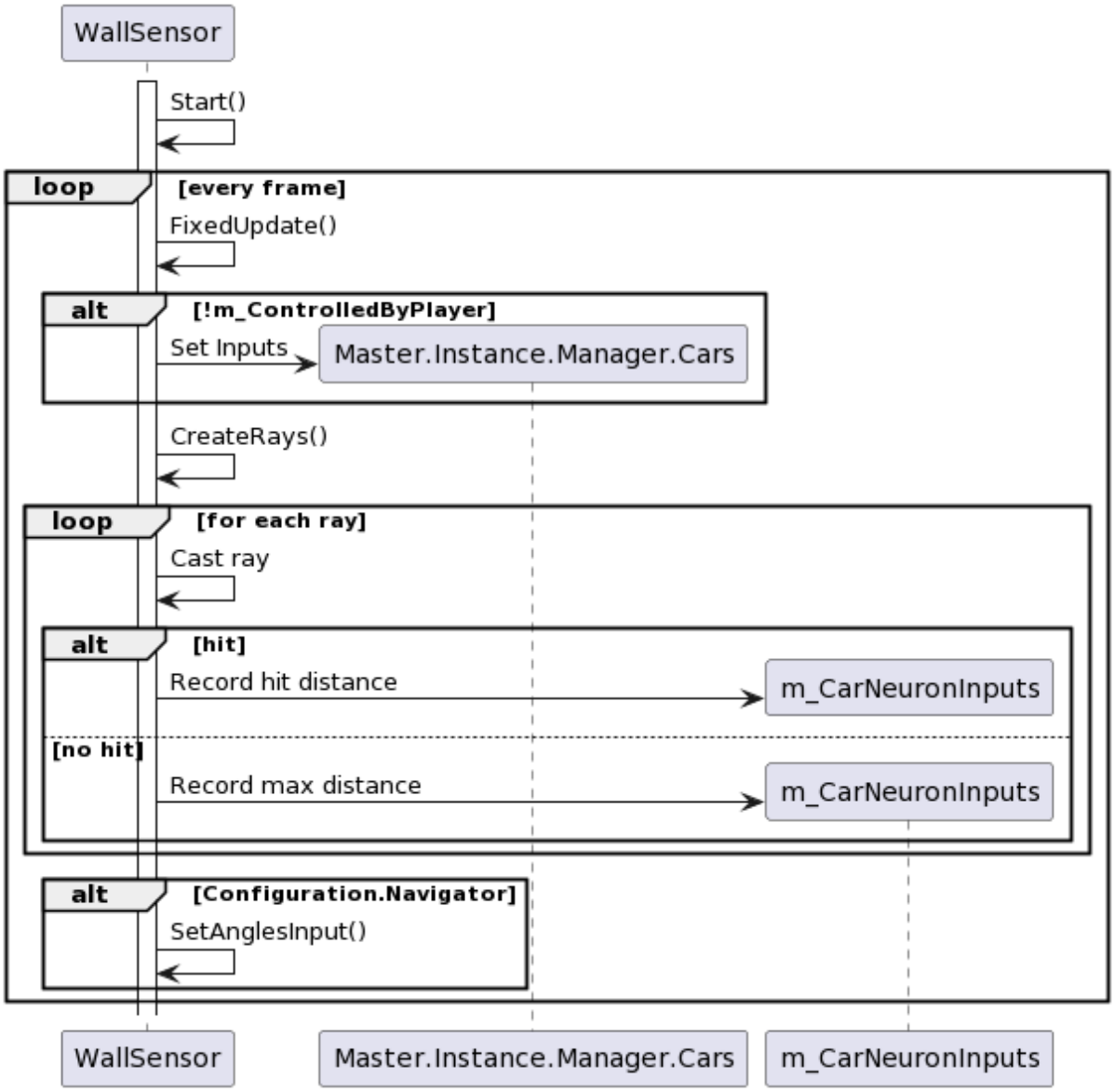


Figure 20. Sequence diagram for the WallSensor

Master

Master component controls the step-by-step procedural logic to setup, execute, evolve cars over generations and collect results from the complete simulation run. The activity flow diagram in Figure 21 shows how the system starts by initializing a new simulation run, then loading configuration parameters that govern how the simulation will proceed - number of cars, neural network topology, and selection pressures.

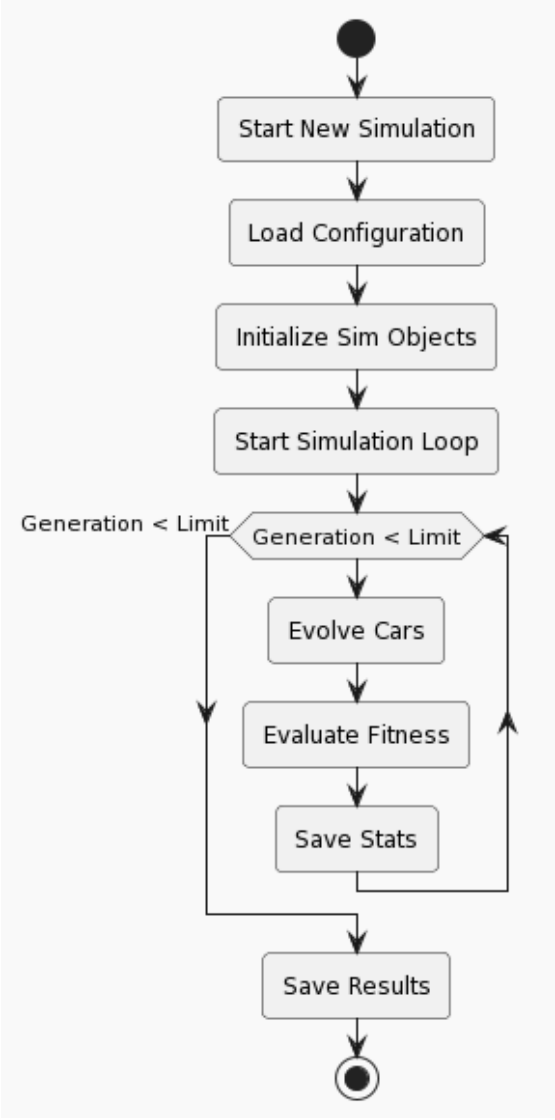


Figure 21. Master component logical flow of activity

Next, the main simulation loop begins, bounded by the condition - "Generation < Limit". Each iteration represents a generation in the evolution timeline. The cars compete and perform in the simulation, after which their fitness is evaluated based on criteria of distance covered, speed and collision avoidance. After configuration, the key simulation objects of the neural networked agent cars and the simulation environment are initialized. Based on fitness, the top performers are selected to seed the next generation. Selection pressures push the population towards agent cars with better performance over generations. The cars are evolved via crossover and mutation of their neural networks. Key stats get saved, Figure 22, after each generation to analyze simulation progress.

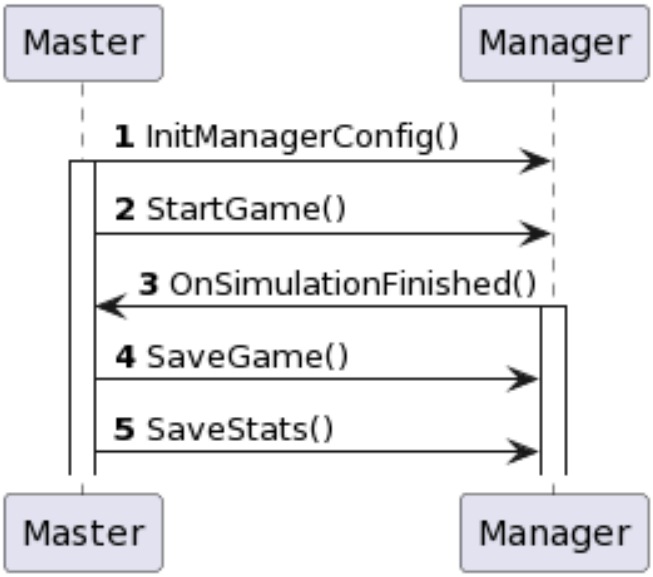


Figure 22. Sequence diagram of Master and Manager

The iterative evolution stops when the generation limit is hit. Finally, the model saves the results - the evolved cars and their brains, stats on the multi-generational progress and learnings. These can inform future simulations.

Simulator Manager

At the heart of the system is the Manager component, which acts as the core coordinator. It interacts with and relies on every other component. The Track represents the OSM data terrain which the Manager utilizes to setup and configure each scenario. Key to the core behavior are the agent cars which the Manager instantiates, each agent encapsulating the driving logic and integrated with a neural network that enables the machine learning capabilities. Behind the scenes, a genetic algorithm component handles the optimization and evolution of the agent car neural networks over generations to optimize performance. For visualization, the Manager directly coordinates with the Display to render the game simulation and provide user feedback.

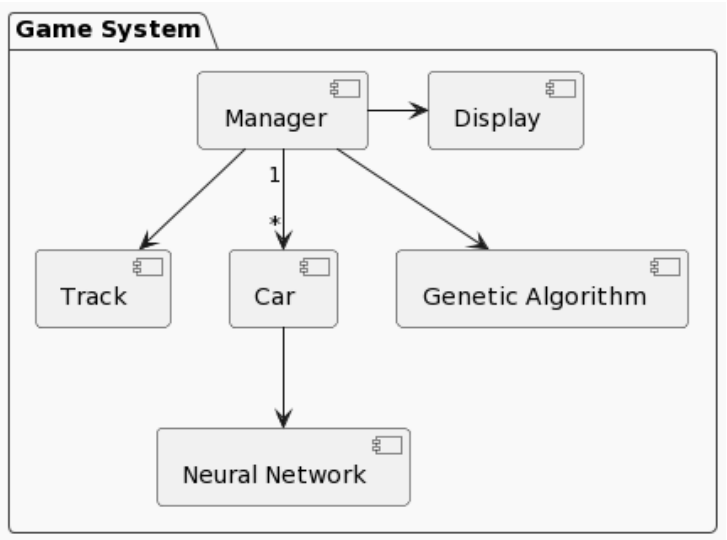


Figure 23. Main components of the simulation system

The unidirectional dependencies, with the Manager as the orchestrator, and the other components focused on specific functions as modeled in Figure 23 shows separation of concerns that provide flexibility to enhance different aspects like the

machine learning logic, tracks, and visualization independently. Figure 24 shows the process of the simulation manager initializing the game state, then checks if loading from a saved state. The agent cars are instantiated next before entering the game loop.

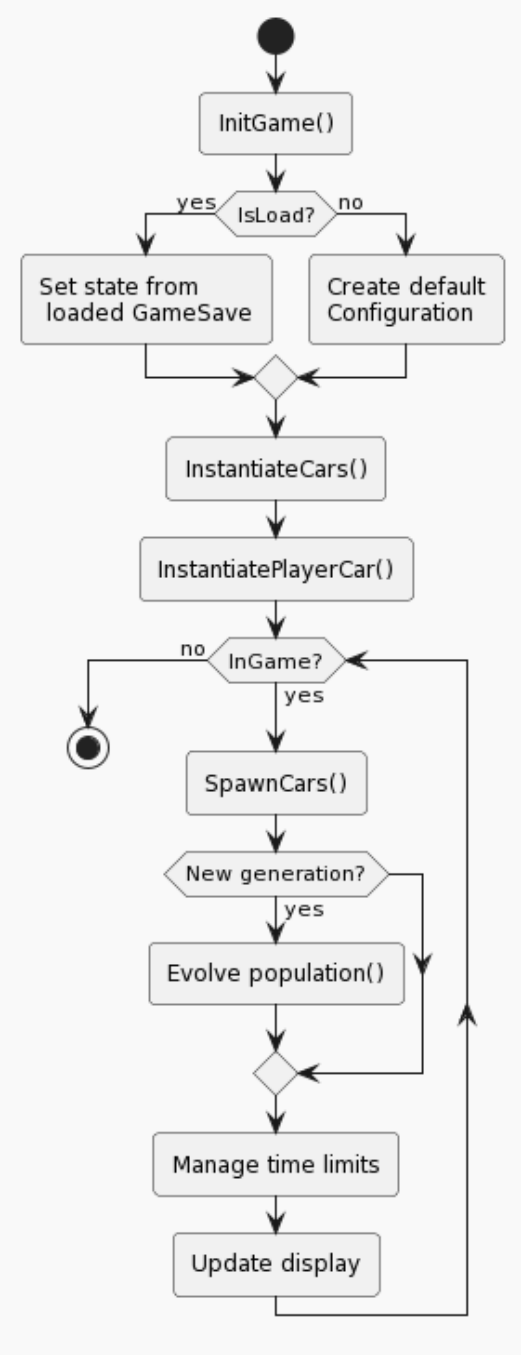


Figure 24. Activity flow for the simualtion Manager

The loop continues iterating as shown in the state machine of Figure 25, representing the running simulation, until the game over condition is reached based on various stopping criteria. After the loop, fitness and neural networks are saved, concluding the simulation.

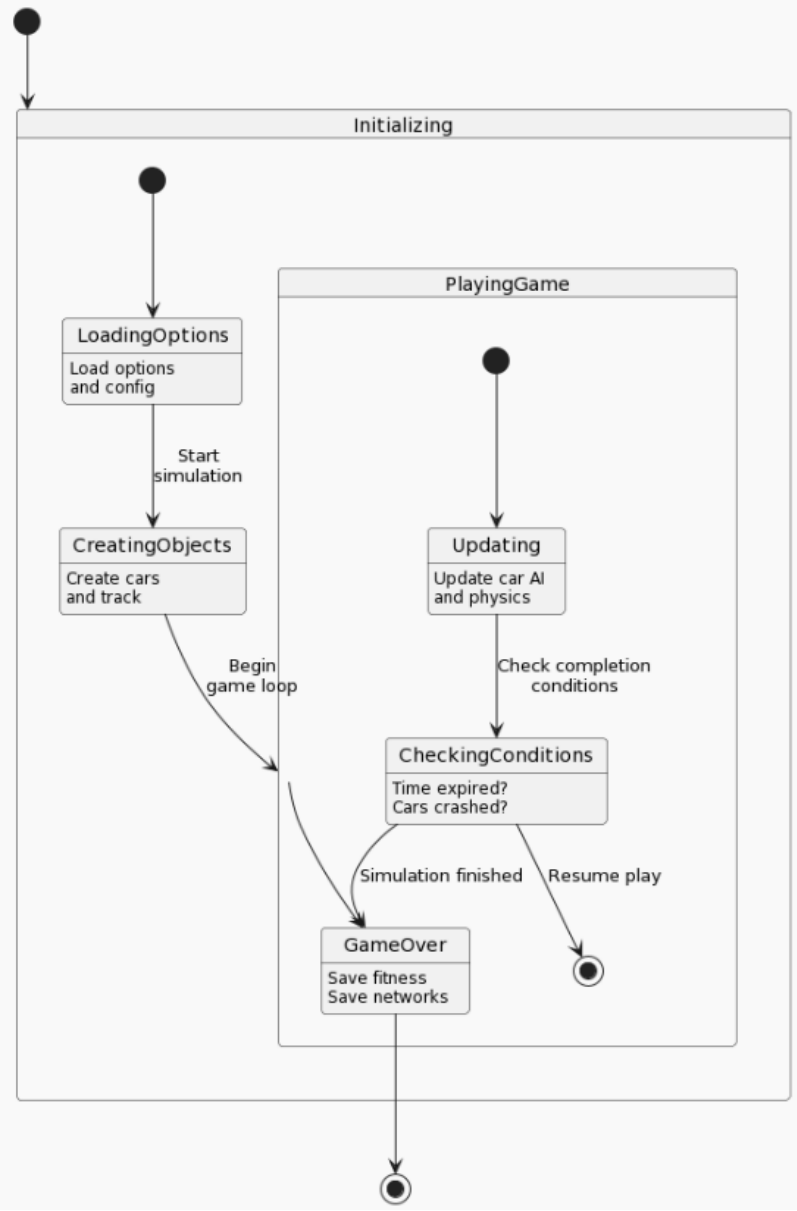


Figure 25. State machine of the simulations system Manager

The Manager coordinates the saving of game data into a GameSave object illustrated in Figure 26. Similarly, for LoadGame(), the Manager retrieves the saved state data from the GameSave providing it back to the Manager, which then updates its internal state, effectively loading the previous game state.

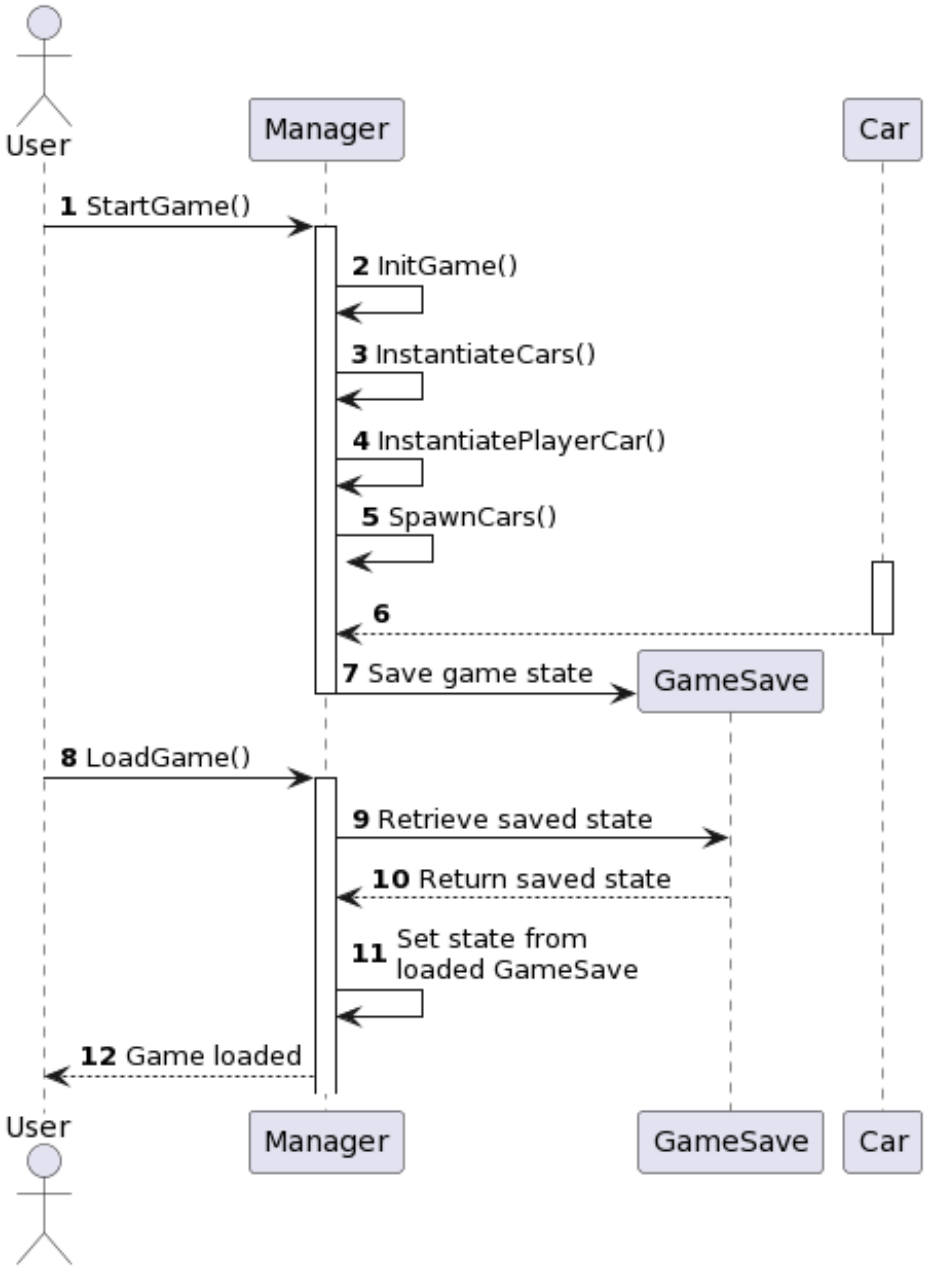


Figure 26. Sequence diagram of the user and simulation

Chapter 4 – Implementation

Scenario Environment Generation

OpenStreetMap (OSM) data is imported into Unity Engine using the *Global Roads & Traffic* plugin which connects to Overpass API. The Overpass API web database interface allows clients to send queries delineating map elements of interest based on constraints like location, feature type, tags, proximity and receive matching OSM entities per the defined selection criteria [35]. The plugin converts OSM data into optimized Unity meshes by downloading OSM map data for a specified region as a JSON file via a built-in editor tool [36] as shown in Figure 27, parses the JSON data and extracts relevant node, way and relation information about roads, intersections, bridges and other infrastructures, then adds height data, applies textures and materials to finalize mesh model segments as can be seen in Figure 28.

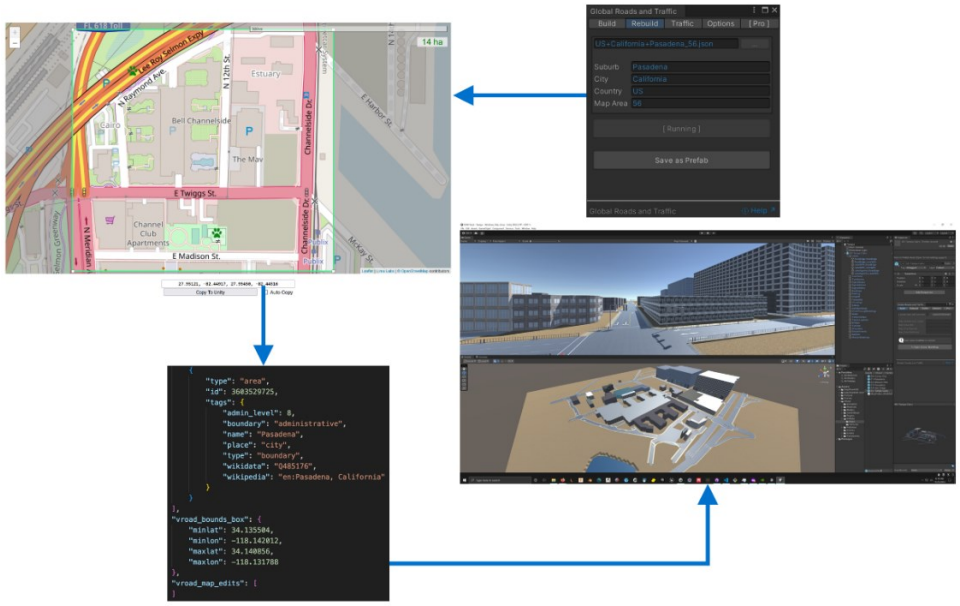


Figure 27. Process flow of generating the scenario environments

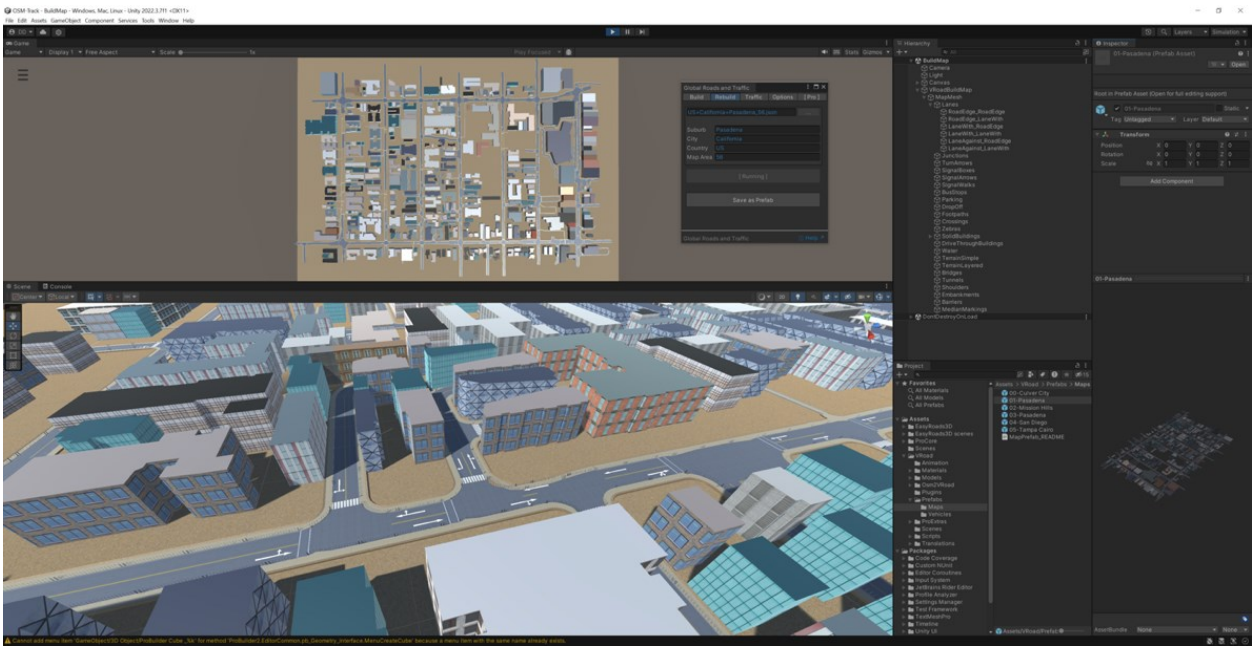


Figure 28. Pasadena mesh model scenario environment

The Overdraw shader mode in Unity shown in Figure 29 is used to visualize rendering performance by highlighting overlapping draws during runtime, or inefficient areas with too many redundant fragment shader executions. The Overdraw shader mode helped optimize performance of these complex urban city mesh models in Unity by identifying buildings/objects that are being drawn redundantly or unnecessarily. This guides rendering optimization efforts to improve graphics performance and efficiency. Scene entities like streetlamps and other traffic infrastructure assets can be repositioned if showing wasteful overlapping draws.

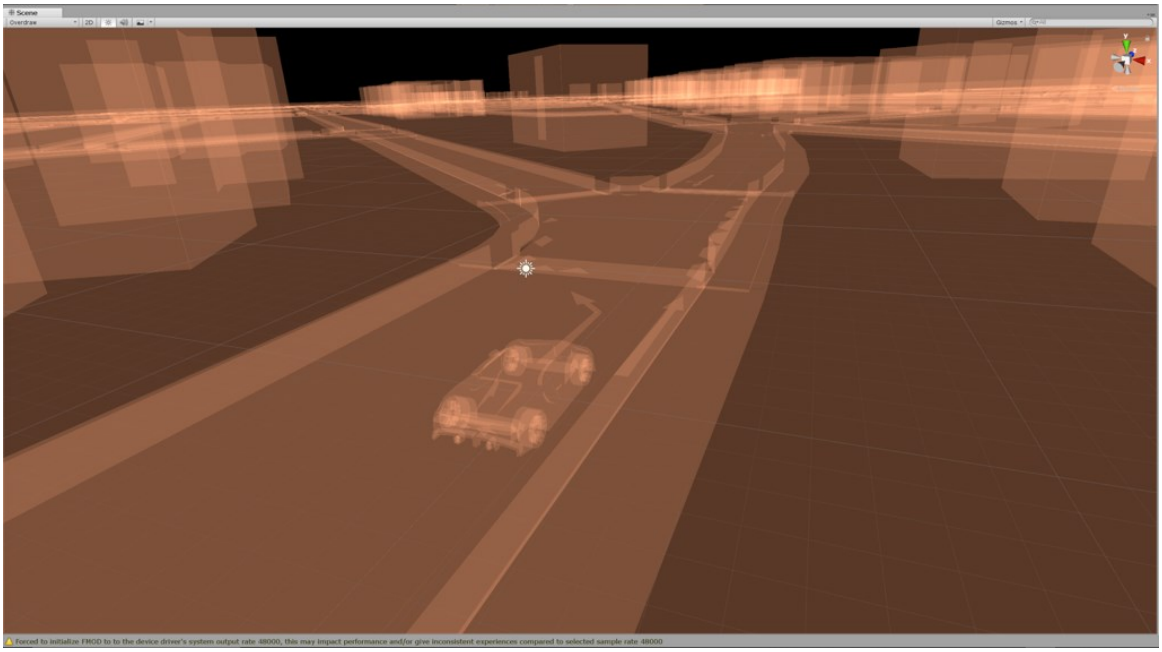


Figure 29. Overdraw visualization view

In this Unity-based driving simulation, collider objects play a key role in defining drivable road surfaces and navigation paths for ADS agent vehicles. Polygon colliders were used to precisely outline the shape and boundaries of roads, highways, and alleyways that are viable for vehicles to traverse. Box and mesh colliders were applied to the environment to approximate more complex irregular street geometry. The machine learning navigation system detects this collider network to identify possible driving paths. Vehicle controller scripts then Raycasts downwards against the detected ground colliders to discern the slope and physics material properties of roads to influence speed, gear changing and handling. The presence and contour of colliders essentially signifies drivable terrain.

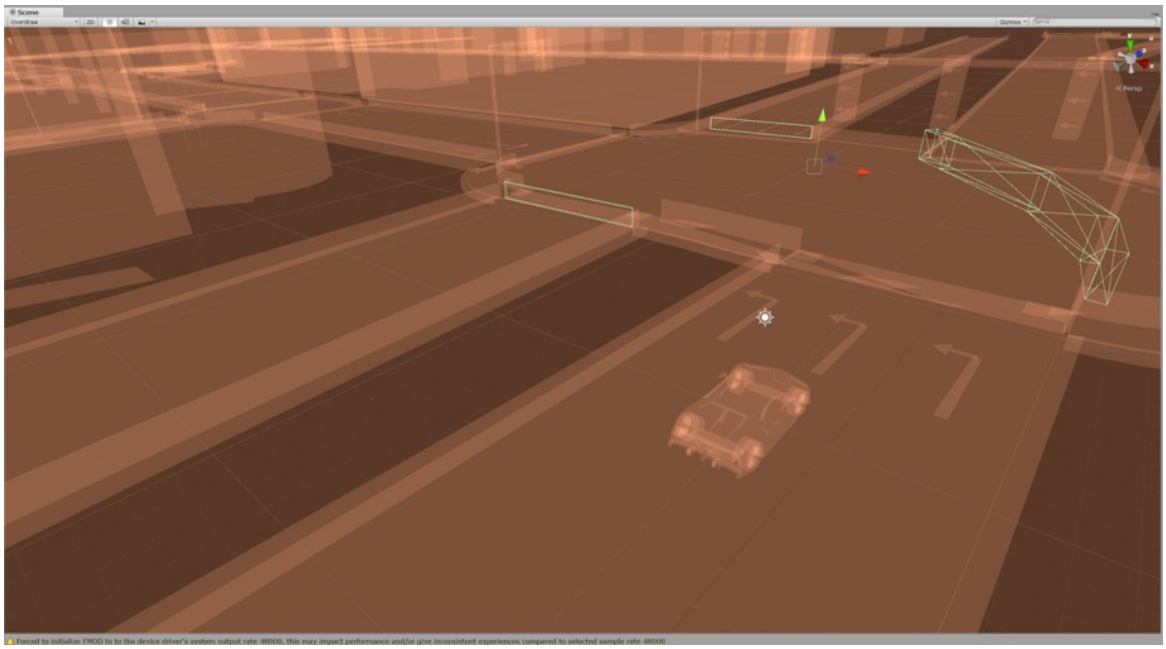


Figure 30. One-way box colliders to prevent counter flow driving

Trigger colliders around intersections, parking spots and garages allow ADS agent cars to slow down, stop or take turns appropriately. Road and lane directionality can also be encoded via one-way colliders preventing counter flow as shown in Fig 30. This collider setup provides key environmental affordances for guiding autonomous vehicle decision making and maneuvering for realistic road-constrained simulation driving, from navigation meshes for high-level pathing to physical materials and triggers for low-level control.

Agent Car Model

The ADS agent car is a GameObject that has several C# scripts attached to it that provide the required functionality to navigate, analyze and learn the environment. The main scripts are as follows:

- **CarController:** script handles the car's physics, taking input for steering and acceleration and applying forces to the wheel colliders.
- **WallSensor:** script is positioned at the front and shoots Raycasts out in front of the agent car, Figure 31, to detect the environment and provide distance input data to the neural network.
- **NeuralNetwork:** script which initializes the network layers and neuron counts based on user configuration settings. It takes the input data, runs it through the network calculations, and outputs steering/acceleration values to feed back to the CarController.
- **FitnessMeter:** script has a Transform variable assigned to track the agent car's position for fitness evaluation. On every frame, it calculates the car's distance to the current waypoint - magenta cubes in Fig 32 - on the road to generate a fitness score.

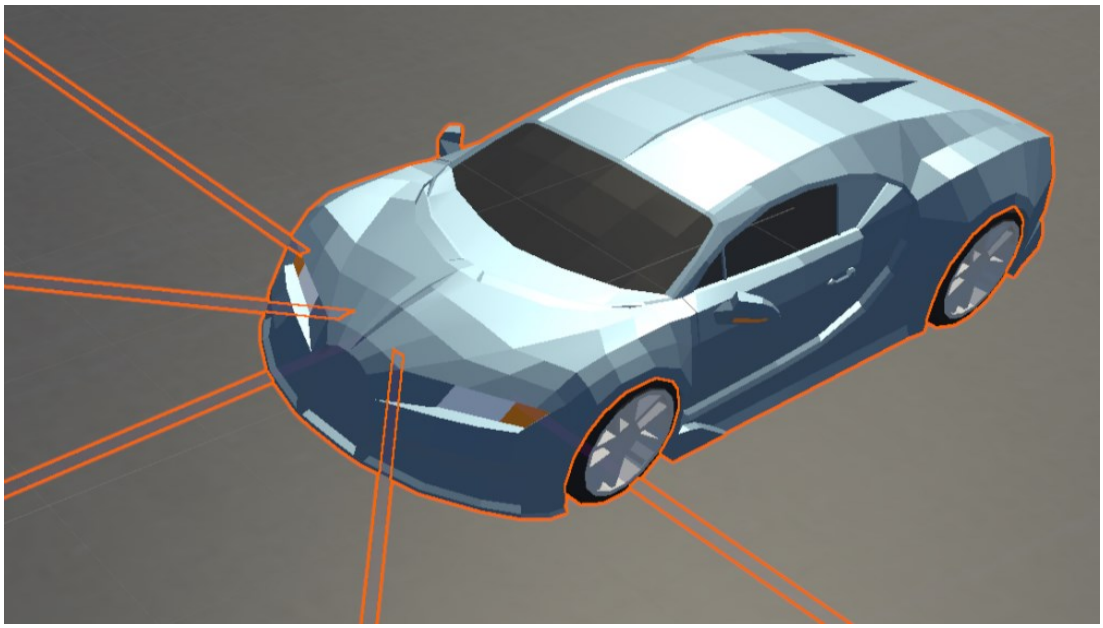


Figure 31. Sensors to detect obstructions in the environment

These scripts work together on each ADS agent car instance during simulations. The GeneticAlgorithm script is on a separate GameManager object, instantiated once. It contains evolution logic that operates on the NeuralNetwork components after each simulation, to produce the next generations.

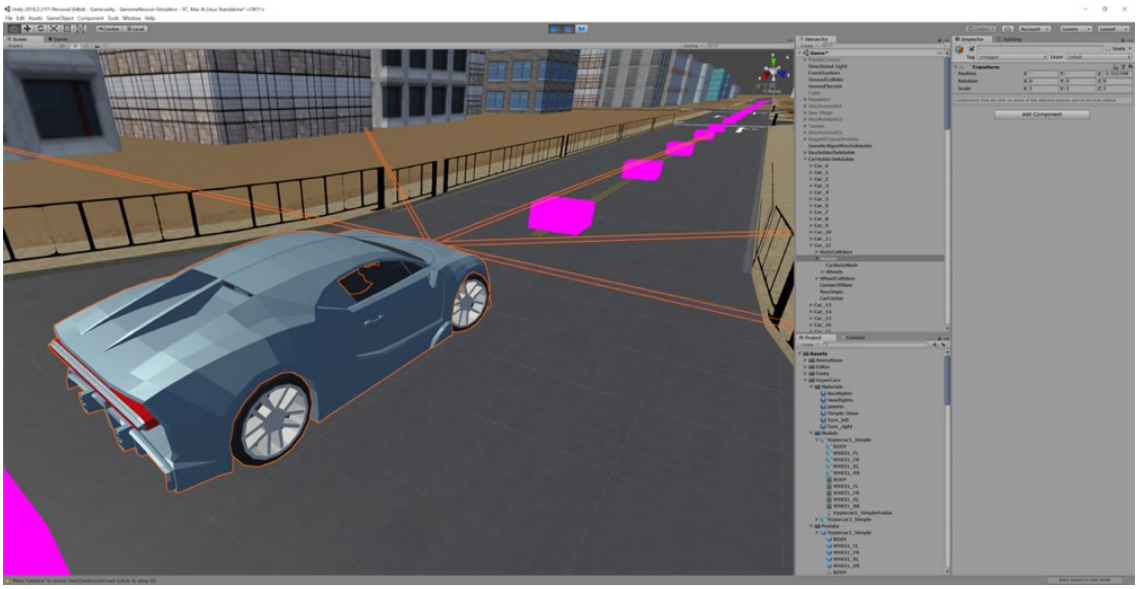


Figure 32. Magenta waypoint cubes are invisible during runtime

When the simulation ends, the GeneticAlgorithm takes the fitness scores for each agent car, sorts them, selects parents, and performs crossover and mutation based on the NeuralNetwork weights and biases. Then it respawns mutated cars to evolve optimized driving behavior over generations. The scripts allow the neural networks to control the car physics along with colliders detailed in Appendix 2, while evolution tunes the neural networks to improve based on the car's fitness in navigating the infrastructure and waypoints.

User Interface

Upon tool start up, a splash screen with a set of three choices are provided, as shown in Figure 33. *Run Simulation*, which has default settings, *Load Data* of previously saved and exported simulations to use in different scenarios, or configure a scenario in *Options*. The user interface for *Options* shown in Figure 34 is where the parameters for simulation scenarios can be configured. The main parameters are:

- **Genetic Algorithm:** quantity of agent cars and the selection rules.
- **Mutation Chance:** probability (30%-70%) and rate (2%-4%) of mutation.
- **Neural Network:** quantity of neuron layers and neurons per layer.
- **Select Level:** the three scenario environments for the simulations.



Figure 33. Splash start screen with Run Simulation, Options, and Load Data

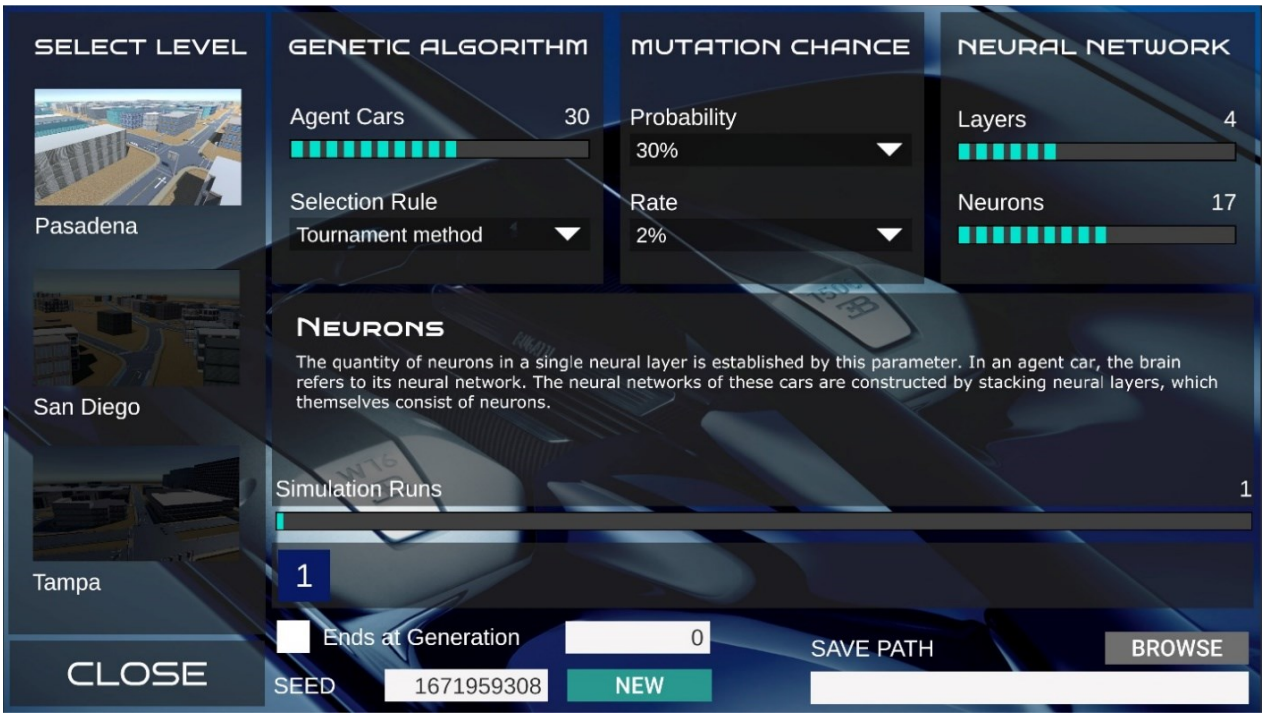


Figure 34. Configuration UI of parameters for the simulation experiment

Neural Network C# Class

The NeuralNetwork class defines the neural network that controls the car in the simulation. As illustrated in Figure 35, the key data members are the NeuronLayers array, which holds the layers of neurons; the *m_CarId* field which identifies which car this network controls; data members that store the number of hidden layers, total neurons, inputs, along with *m_TransferData* that holds the neuron outputs during forward propagation. Additionally, *m_CarInputs* stores the input states for the neural network that represents the current state of the car. The *m_Bias* field adds a bias value to the weighted inputs of each neuron, and *CarController* is a reference to the component that handles moving the car. The final output layer is defined in *m_Control* which holds the two key outputs: steering and acceleration control values that act as the

final outputs to physically drive the car in the simulation. This NeuralNetwork class encapsulates the layers, connections, and data of a neural network that takes input data about the car's state, passes that data through hidden neuron layers, calculates output control values used to drive the car via the *CarController* component reference.

```
using UnityEngine;

[System.Serializable]
public class NeuralNetwork : MonoBehaviour
{
    public NeuronLayer[] NeuronLayers { get; set; }

    private int m_CarId;
    private int m_HiddenLayerCount;
    private int m_NeuronCount;
    private int m_InputCount;
    private float[][] m_TransferData;
    private float[] m_CarInputs;
    private float m_Bias;

    public CarController CarController;

    // The car is controlled based on the array elements
    // control[0] = steering , control[1] = acceleration
    private float[] m_Control = new float[2];

    private void Start()
    {
        m_Bias = Master.Instance.Manager.Bias;
        m_CarId = this.gameObject.GetComponent<CarController>().Id;
        m_HiddenLayerCount = Master.Instance.Manager.Configuration.LayersCount;
        m_NeuronCount = Master.Instance.Manager.Configuration.NeuronPerLayerCount;

        if (Master.Instance.Manager.Configuration.Navigator)
        {
            m_InputCount = Master.Instance.Manager.CarSensorCount + 4;
        }
        else
        {
            m_InputCount = Master.Instance.Manager.CarSensorCount + 1;
        }

        m_TransferData = new float[m_HiddenLayerCount][];
        for (int i = 0; i < m_TransferData.Length; i++)
        {
            m_TransferData[i] = new float[m_NeuronCount];
        }

        // Hidden layers, +1 output layer
        NeuronLayers = new NeuronLayer[m_HiddenLayerCount + 1];
```

Figure 35. NeuralNetwork class

The Start() method initializes the neural network that will control the ADS agent car. It begins by setting the bias term to the global bias specified in Master and gets the ID of the car controller this neural network will control. It then sets the number of hidden layers and neurons per layer based on the global configuration in Master. Additional details regarding this process can be found in Appendix 3.

Next, it determines the input count - this will be the number of car sensors plus an additional input depending on whether navigation data is enabled. With the topology defined, the method allocates the transfer data array which will hold neuron outputs during forward propagation and initializes the NeuronLayers array which holds the different neural network layers.

The switch statement shown in Figure 36 constructs the correct layer architecture based on whether there are 0, 1 or 2+ hidden layers. If loading previously saved data, it iterates through each neuron and layer, setting the weights to values loaded from the Manager. Start() initializes the neural network topology, neuron layers, as well as weight values - configuring an appropriate neural network structure to control the ADS agent car based on global configuration parameters.

```
    switch (m_HiddenLayerCount)
    {
        // Initialize the hidden layers.
        // If zero hidden layer -> there is only the output layer
        case 0:
            NeuronLayers[0] = new NeuronLayerTanh(2, m_InputCount, m_Bias);
            break;

        // If one hidden layer -> first layer gets the input,
        // second layer is the output layer.
        case 1:
            NeuronLayers[0] = new NeuronLayerTanh(m_NeuronCount, m_InputCount, m_Bias);
            NeuronLayers[1] = new NeuronLayerTanh(2, m_NeuronCount, m_Bias);
            break;

        // If two or more hidden layers -> first layer gets the input,
        // the other ones get the output from the previous layer
        // and the last layer is the output layer.
        default:
            NeuronLayers[0] = new NeuronLayerTanh(m_NeuronCount, m_InputCount, m_Bias);
            for (int i = 1; i < NeuronLayers.Length - 1; i++)
            {
                NeuronLayers[i] = new NeuronLayerTanh(m_NeuronCount, m_NeuronCount, m_Bias);
            }
            NeuronLayers[NeuronLayers.Length - 1] = new NeuronLayerTanh(2, m_NeuronCount, m_Bias);
            break;
    }

    if (!Master.Instance.Manager.IsLoad) return;

    for (int i = 0; i < NeuronLayers.Length; i++)
    {
        for (int j = 0; j < NeuronLayers[i].NeuronWeights.Length; j++)
        {
            for (int k = 0; k < NeuronLayers[i].NeuronWeights[j].Length; k++)
            {
                NeuronLayers[i].NeuronWeights[j][k] = Master.Instance.Manager.Save.SavedCarNetworks[m_CarId][i][j][k];
            }
        }
    }
}
```

Figure 36. Switch logic statement for neuron layer

Neuron Layer C# Class

The NeuronLayer class defines the base elements and functionality for a layer of neurons within the neural network. It will be further extended by concrete neuron layer implementations. The key properties and fields shown in Figure 37 define the number of neurons in this layer, number of input values each neuron receives, a bias value, and most importantly the neuron weights matrix. The constructor method initializes these

fields and allocates a two-dimensional array to represent the weights matrix. Each neuron in the layer has a row, and each row has a weight value per input, plus one extra weight for the bias term. This base class constructor calls InitWeights() which will be defined in subclasses to initialize the starting weights values.

```
[System.Serializable]
public abstract class NeuronLayer
{
    public float[][] NeuronWeights { get; set; }

    protected int NeuronCount;
    protected int InputCount;
    protected float Bias;

    /// <summary>
    /// Creates a neuron layer.
    /// </summary>
    /// <param name="neuronCount"> The number of neurons in the layer.</param>
    /// <param name="inputCount"> The number of inputs the neuron layer gets (without the bias).</param>
    /// <param name="bias"> The bias value for the layer.</param>
    protected NeuronLayer(int neuronCount, int inputCount, float bias)
    {
        this.NeuronCount = neuronCount;
        this.InputCount = inputCount;
        this.Bias = bias;

        NeuronWeights = new float[neuronCount][];
        for (int i = 0; i < NeuronWeights.Length; i++)
        {
            // +1 input is the bias
            NeuronWeights[i] = new float[inputCount + 1];
        }

        InitWeights();
    }

    /// <summary>
    /// Initializes the weights in the layer for all neurons to a random number
    /// between -1 and 1.
    /// </summary>
    protected void InitWeights()
    {
        for (int i = 0; i < NeuronWeights.Length; i++)
        {
            for (int j = 0; j < NeuronWeights[i].Length; j++)
            {
                // Get a random number between -1 and 1
                NeuronWeights[i][j] = RandomHelper.NextFloat(-1f, 1f);
            }
        }
    }
```

Figure 37. The NeuronLayer class structure

The InitWeights() method initializes the weights matrix for the layer with random values between -1 and 1. It iterates through each neuron row, and within each row sets every input weight including the bias weight to a random number in that range. Getting good initial weight values is important for the efficient training of neural networks.

The CalculateLayer() method performs the main computational logic during neural network propagation. As shown in Figure 38, for each neuron, it calculates the weighted sum by multiplying the inputs by their corresponding weights in that neuron's row. It adds the bias term multiplied by its weight as well. This weighted sum is then passed to the layer's activation function, defined in the abstract Activate() method. The activations from each neuron are stored in the layer output array, which is finally returned.

```
    public float[] CalculateLayer(float[] inputs)
    {
        float[] layerOutput = new float[NeuronCount];

        // for each neuron -
        for (int i = 0; i < NeuronCount; i++)
        {
            float weightedSum = 0;
            // - calculate the output
            for (int j = 0; j < InputCount; j++)
            {
                weightedSum += inputs[j] * NeuronWeights[i][j];
            }
            weightedSum += Bias * NeuronWeights[i][InputCount];

            layerOutput[i] = Activate(weightedSum);
        }
        return layerOutput;
    }

    protected abstract float Activate(float x);
}
```

Figure 38. CalculateLayer method for propagating the neural network

Together these methods define the learnable parameters and functional logic of a neural network layer. Concrete child classes specialize the activation function and optionally the weight initialization. By encapsulating these basic behaviors, NeuronLayer provides an extensible baseline to implement different layer types.

Genetic Algorithm C# Class

The abstract GeneticAlgorithm class handles the main evolution operations of the population over each generation by selecting fitter individuals, recombining them to

```
public abstract class GeneticAlgorithm : MonoBehaviour
{
    protected struct FitnessRecord : IComparable
    {
        public int Id;
        public float Fitness;

        public int CompareTo(object obj)
        {
            if (obj == null) return 1;  // this is greater than obj

            if (!(obj is FitnessRecord)) throw new ArgumentException("Object is not a FitnessRecord!");
            FitnessRecord other = (FitnessRecord)obj;
            return Fitness.CompareTo(other.Fitness);
        }
    }

    public int PopulationSize { get; set; }

    protected FitnessRecord[] FitnessRecords;

    public float MutationChance { get; set; }
    public float MutationRate { get; set; }

    public int GenerationCount;
    protected NeuralNetwork[] CarNetworks;

    // In this array it stores the pairs created during selection
    // The first index indicates the sequence number of the created pair (as many new cars as needed)
    // The second index indicates the left (0) and right (1) parents
    protected int[][] CarPairs;

    // It stores the values of all cars' neural networks in a 4D array,
    // because it has to work with the original values during recombination
    public float[][][][] SavedCarNetworks;

    private Manager m_Manager;
```

Figure 39. GeneticAlagorithm class structure

produce offspring, introducing random mutation, and replacing less fit individuals as specified in Figure 39. This iterative process optimizes the population to find improved solutions to the target problem.

The Awake() method in Figure 40 initializes key parameters like population size, and mutation settings from the simulation configuration. The Start() method allocates the data structures to store the population - CarNetworks array stores the neural network models for the population, FitnessRecord struct stores fitness information for each agent car in the population to track performance, and CarPairs stores parent pairs selected for breeding new ADS agent cars each generation. The FixedUpdate loop runs when the current generation finishes (all cars frozen). It begins by saving the neural networks of the current population then sorts them by fitness. It calculates population statistics and invokes the selection process to pick optimal parents to breed the next generation. This continues for multiple generations, evolving better performing agent cars over time. C# code details for each of the selection methods used in the genetic algorithm can be found in Appendix 3.

```
private void Awake()
{
    m_Manager = Master.Instance.Manager;
    PopulationSize = m_Manager.Configuration.CarCount;
    MutationChance = m_Manager.Configuration.MutationChance; // 30-70 int %
    MutationRate = m_Manager.Configuration.MutationRate;   // 2-4 float %
}

private void Start()
{
    CarNetworks = new NeuralNetwork[PopulationSize];
    CarPairs = new int[PopulationSize][];
    FitnessRecords = new FitnessRecord[PopulationSize];

    for (int i = 0; i < CarPairs.Length; i++)
    {
        CarPairs[i] = new int[2];
    }

    for (int i = 0; i < FitnessRecords.Length; i++)
    {
        FitnessRecords[i] = new FitnessRecord();
    }
}
```

Figure 40. Awake() and Start() methods

The SavedCarNetworks loops through the entire 4D array of neural network weights for the population and copies the current values from the car networks into it, saving their state for the next generation. This preserves the "genetic material" - the weight parameters - before modification. It acts as a gene pool for crossover. This method ensures the structure is initialized properly in the first generation.

Recombine and Mutate

The RecombineAndMutate() method implements the core genetic operators of crossover and mutation to produce new neural networks for the next generation. Figure 41 shows how It loops through all the weights of each network, and for the top fitness

individual, copies its weights unchanged to preserve the current best. For others, it randomly picks one parent index from the breeding pairs, and has a MutationChance probability of mutating those weights by a random factor within range. Otherwise, the weights get directly copied. This mixes parental genetics to produce new variety in neural networks and explore the solution space of new high-performing configurations.

```
protected virtual void RecombineAndMutate()
{
    float mutationRateMinimum = (100 - MutationRate) / 100;
    float mutationRateMaximum = (100 + MutationRate) / 100;

    for (int i = 0; i < SavedCarNetworks.Length; i++)    // which car
    {
        for (int j = 0; j < SavedCarNetworks[i].Length; j++) // which neuron layer
        {
            for (int k = 0; k < SavedCarNetworks[i][j].Length; k++) // which neuron
            {
                for (int l = 0; l < SavedCarNetworks[i][j][k].Length; l++) // which weight
                {
                    if (i == FitnessRecords[0].Id)
                    {
                        CarNetworks[i].NeuronLayers[j].NeuronWeights[k][l] =
                            SavedCarNetworks[i][j][k][l];
                    }
                    else
                    {
                        float mutation = RandomHelper.NextFloat(mutationRateMinimum, mutationRateMaximum);

                        // 50% chance of inheriting from one parent
                        // carPairs[i] contains indices of both parents
                        int index = CarPairs[i][RandomHelper.NextInt(0, 1)];

                        // The probability of mutation varies with the MutationChance value
                        if (RandomHelper.NextInt(0, 100) <= MutationChance)
                        {
                            CarNetworks[i].NeuronLayers[j].NeuronWeights[k][l] =
                                SavedCarNetworks[index][j][k][l] * mutation;
                        }
                        else
                        {
                            CarNetworks[i].NeuronLayers[j].NeuronWeights[k][l] =
                                SavedCarNetworks[index][j][k][l];
                        }
```

Figure 41. RecombineAndMutate method for crossover and mutation

In essence, crossover propagates components from fit parents, while mutation introduces new traits. Recombination without mutation would reduce the search space rapidly, while only mutation without crossover loses good solutions discovered so far. Using both allows balanced exploitation of current peaks versus exploration of new possibilities. The interplay allows continual improvement of neural networks over generations to maximize the fitness metric. Tracking stats monitors evolution direction, while saving top performers passes on beneficial genetics unchanged. This evolves neural networks tailored to the problem.

Chapter 5 – Results and Analysis

The simulation results present a comparative analysis of 16 distinct neural network architectures with varied genetic selection methods, across three scenarios, each exhibiting varying configurations and training parameters. The data reveals notable variations in performance and structural characteristics across these configurations.

Pasadena Scenario

PSim1 demonstrated superior performance as shown in Table 1, with the highest maximum fitness value of 179, though its median fitness of 33 suggests considerable variance in outcomes. This configuration utilized an 8-layer neural network architecture with 6 neurons per layer and implemented Tournament selection for genetic progression. After 500 generations, desirable driving behavior emerged in more ADS agent cars as shown in Figure 42.

Pasadena Simulation

20 Cars Evolving Over 500 Generations

Performance Level Range

2 179.23

	Simulation	Maximum Fitness	Median Fitness	Mutation Possibility	Mutation Rate	Number of Layers	Neuron per Layer	Genetic Selection
1	PSim1	179	33	50%	3%	8	6	Tournament
2	PSim2	23	14	40%	3%	6	12	Elitist
3	PSim3	17	5	50%	4%	8	20	Tournament+ 20% Random
4	PSim4	66	10	40%	3%	3	6	Elitist
5	PSim5	67	8	70%	2%	2	2	Roulette Wheel

Created with Datawrapper

Table 1. Comparative simulation results of the Pasadena scenario

In contrast, PSim2 and PSim3 exhibited relatively modest performance metrics, with maximum fitness values of 23 and 17 respectively. Despite PSim3's more complex neural network architecture (8 layers, 20 neurons per layer) and hybrid selection strategy (Tournament+ 20% Random), it achieved the lowest median fitness of 5.

PSim4 and PSim5 showed intermediate performance levels, with maximum fitness values of 66 and 67 respectively. Notably, PSim5 employed the simplest neural network architecture (2 layers, 2 neurons per layer) while maintaining the highest mutation possibility at 70%, suggesting that architectural complexity may not directly correlate with performance optimization in this context.

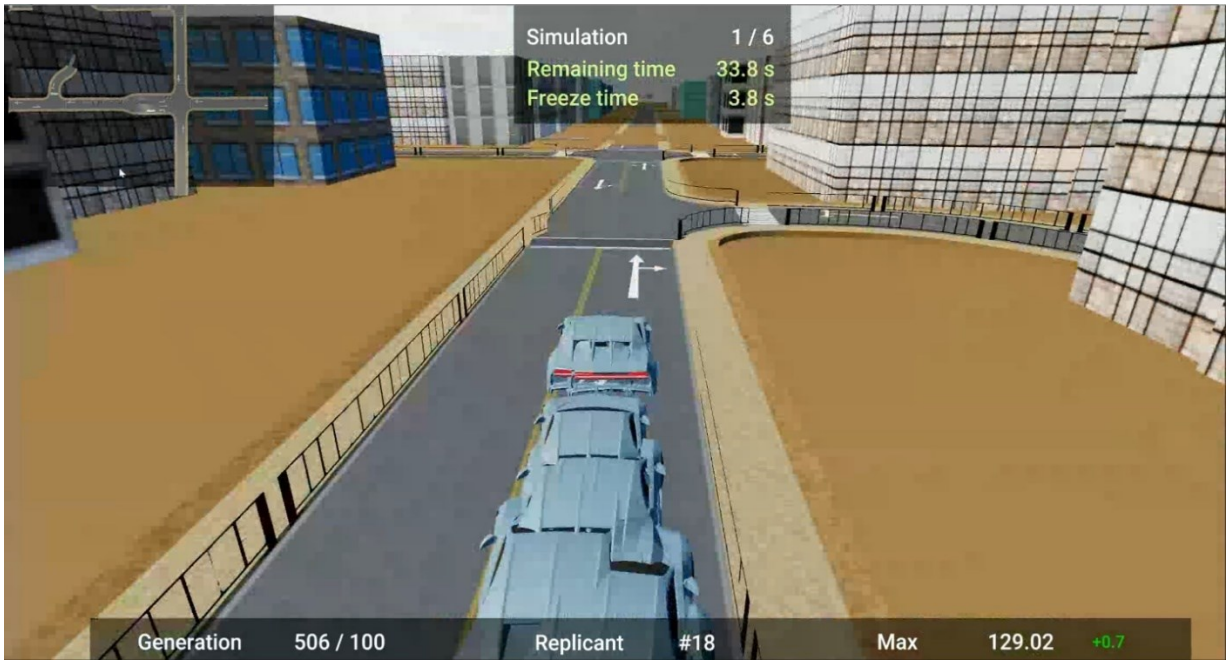


Figure 42. Desirable driving emerges after more than 500 generations

The mutation rates across all simulations remained relatively consistent (2-4%), indicating a controlled approach to genetic variation. The diversity in genetic selection

methods (Tournament, Elitist, Roulette Wheel) provides valuable comparative data on selection strategy efficacy in evolutionary algorithms.

These results indicate that architectural complexity does not directly correlate with performance effectiveness, as evidenced by the superior performance of simpler configurations in some instances. The data suggests that the interplay between neural network architecture, mutation parameters, and fitness outcomes is highly non-linear and merits further investigation to determine optimal configuration strategies that would significantly influence evolutionary outcomes in this simulation context.

San Diego Scenario

The San Diego Scenario simulation results also offer valuable insights into the interplay between neural network architecture, mutation parameters, and genetic algorithm selection strategies.

Looking at the performance metrics in Table 2, SDSim3 achieved the highest maximum fitness of 144, employing a moderately complex architecture with 7 layers and 5 neurons per layer. What makes this configuration particularly interesting is its use of a Roulette Wheel selection method, combined with balanced mutation parameters (60% possibility, 3% rate). This suggests that probabilistic selection methods can effectively guide evolutionary optimization when paired with appropriate neural network structures. Behavior such as half the population of agent cars selecting different driving routes seen in Figure 43 was a notably observation, demonstrating exploration of optimal navigation paths over each generation.

San Diego Scenario

20 Cars Evolving Over 300 Generations

Performance Level Range

< 29.7 | 29.7–58.4 | 58.4–87.09 | 87.09–115.79 | ≥ 115.79

	Simulation	Maximum Fitness	Median Fitness	Mutation Possibility	Mutation Rate	Neuron Layers	Neurons/ Layer	Genetic Selection
1	SDSim1	111	28	50	2	6	3	Tournament
2	SDSim2	140	29	70	4	2	4	Elitist
3	SDSim3	144	29	60	3	7	5	Roulette Wheel
4	SDSim4	131	5	70	4	1	1	Tournament+ 20% Random
5	SDSim5	39	28	40	2	3	9	Tournament
6	SDSim6	143	28	30	4	10	6	Elitist
7	SDSim7	142	27	50	3	3	6	Elitist

Created with Datawrapper

Table 2. Simulation results from the San Diego scenario

Close behind in performance are SDSim6 and SDSim7, with maximum fitness values of 143 and 142 respectively. Despite their similar performance outcomes, these configurations showcase dramatically different architectural approaches. SDSim6 implements the most complex structure with 10 layers and 6 neurons per layer, while SDSim7 opts for a much simpler design with just 3 layers and 6 neurons per layer. Both utilize the Elitist selection strategy, though with different mutation parameters, suggesting that this selection method can be effective across varying neural network architecture complexities.

A particularly intriguing case is SDSim4, which employs a unique hybrid selection approach combining Tournament selection with 20% random inclusion. Despite having the simplest neural network architecture (1 layer, 1 neuron per layer), it achieved a respectable maximum fitness of 131. However, its notably low median fitness of 5 indicates high performance volatility, possibly due to the increased randomness in its selection method.

SDSim5 stands as an outlier in terms of performance, achieving only a maximum fitness of 39 despite having a moderate architecture (3 layers, 9 neurons per layer). This configuration uses traditional Tournament selection with conservative mutation parameters (40% possibility, 2% rate), suggesting that certain combinations of selection methods and mutation rates may lead to suboptimal outcomes.

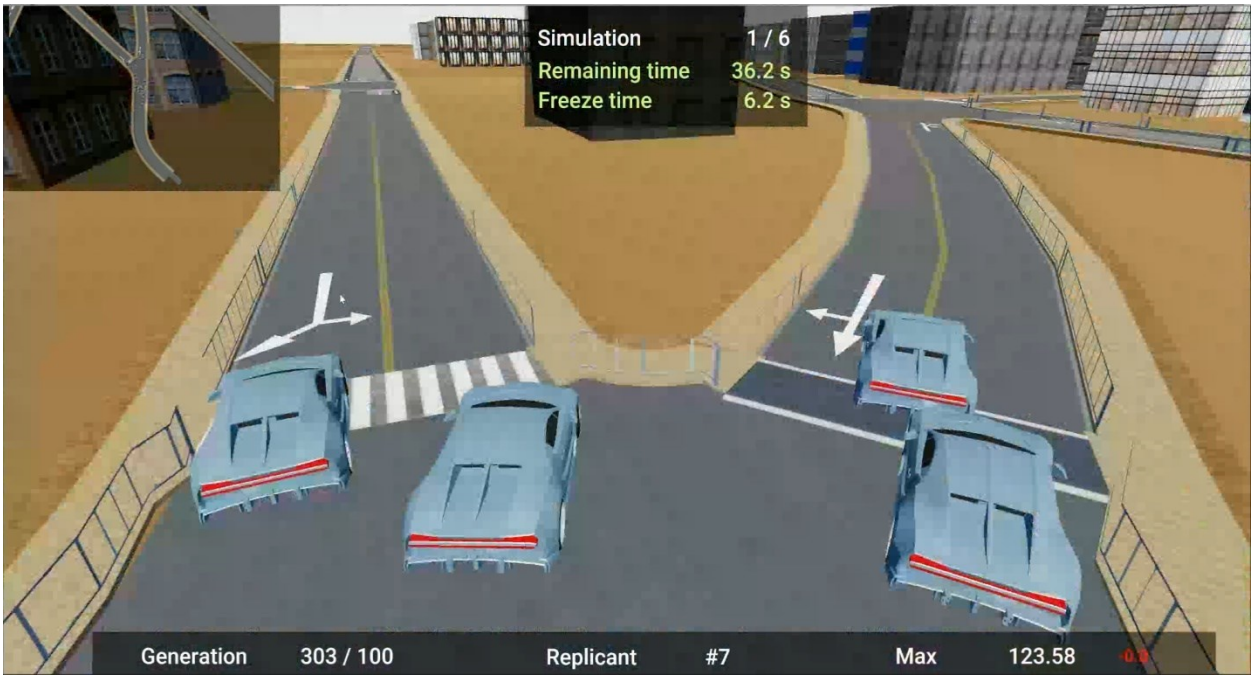


Figure 43. Population of agent cars select between two routes

The relationship between architectural complexity and performance shows no clear linear correlation across these simulations. Instead, the data suggests that the effectiveness of a configuration depends on the harmonious interaction between neural network architecture, mutation parameters, and genetic selection strategy. These findings challenge conventional assumptions about neural network design and

emphasize the importance of considering selection methods as a crucial component in evolutionary neural network optimization.

These results provide valuable insights for future research directions, particularly in understanding how different genetic selection strategies might be optimally paired with specific neural network architectures and mutation parameters to achieve desired performance outcomes.

Tampa Scenario

In the Tampa Scenario, TSim2 emerged as the most effective configuration, achieving a remarkable maximum fitness value of 269 while maintaining a median fitness of 26. Table 3 shows this superior performance was achieved with a relatively simple neural network architecture (2 layers, 3 neurons per layer) and Tournament selection method, operating at a 60% mutation possibility and 2% mutation rate. This high fitness score can be attributed to agent cars successfully exploring more of the environment as Figure 44 shows a few driving further in the distance.

Tampa Simulation

20 Cars Evolving Over 300 Generations

Performance Level Range

2 269.23

	Simulation	Maximum Fitness	Median Fitness	Mutation Possibility	Mutation Rate	Number of Layers	Neuron per Layer	**Genetic Selection**
1	TSim1	114	27	50%	3%	3	6	Elitist
2	TSim2	269	26	60%	2%	2	3	Tournament
3	TSim3	187	14	50%	4%	4	12	Tournament+ 20% Random
4	TSim4	50	28	60%	4%	6	8	Roulette Wheel

Created with Datawrapper

Table 3. Tampa simulation scenario results

TSim3 demonstrated the second-highest performance with a maximum fitness of 187, though its median fitness of 14 was the lowest among all configurations. This simulation employed a more complex neural network architecture (4 layers, 12 neurons per layer) and a hybrid selection strategy of Tournament with 20% Random.

TSim1 achieved moderate success with a maximum fitness of 114 and the second-highest median fitness of 27. Its configuration utilized a neural network with 3 layers and 6 neurons per layer, implementing an Elitist selection strategy.

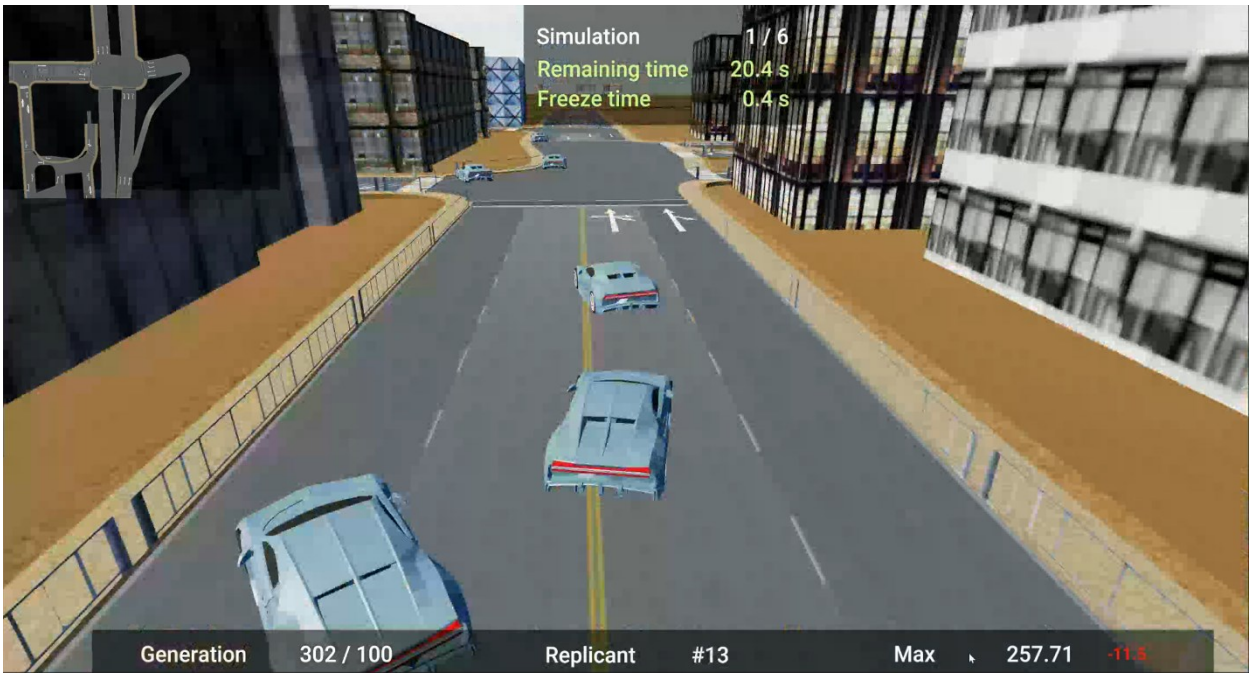


Figure 44. ADS agents exploring more of the environment achieve higher fitness

TSim4, despite having the most complex neural network architecture (6 layers, 8 neurons per layer), recorded the lowest maximum fitness of 50, though it maintained the

highest median fitness of 28. This configuration employed a Roulette Wheel selection method with a 60% mutation possibility.

The results suggest that architectural complexity may not be directly proportional to performance optimization in this context. Furthermore, the data indicates that simpler neural network structures, when paired with appropriate selection methods and mutation parameters, can yield superior results in evolutionary algorithms. The variation in median fitness values across configurations (ranging from 14 to 28) provides valuable insights into the stability and consistency of different evolutionary strategies.

Chapter 6 – Conclusions and Future Research

Conclusions

This thesis research in machine learning and systems simulations applied genetic algorithms to refine neural network architectures for automated driving system agents to maximize driving fitness performance over generations. Simulation experiments were run with differing parameters to study their impact, with results demonstrating that evolutionary algorithms can optimize policies to improve driving performance without large datasets, though consistency varies.

A software tool was developed with a user interface for the configuration of the simulations, which include specifying the quantity of neurons and neuron layers, establishing mutation probability percentages, and selection methods for genetic variation to train the navigation and pathfinding capabilities of automated driving systems. Three scenarios are also selectable from the user interface, providing the option to experiment with trained automated driving systems in different environments for adaptability testing.

Comprehensive analysis of the simulation data results from Pasadena, San Diego, and Tampa scenarios revealed several significant patterns regarding the evolutionary performance of the ADS agents across different parameters and selection methods. Different genetic selection methods yield varying levels of success across simulations. Tournament selection, while producing the highest individual fitness score of 179 in PSim1, shows inconsistent performance across different scenarios. The Elitist

selection method, however, demonstrates more consistent high-performance outcomes, particularly evident in the San Diego simulations where it produced multiple instances of fitness scores above 140 (SDSim2, SDSim6, and SDSim7).

The neural network architectures, defined by the number of layers and neurons per layer, appears to have a complex relationship with performance. The highest performing simulations did not necessarily correlate with more complex neural network architectures. For instance, TSim2, which achieved the highest overall fitness score of 269, utilized a relatively simple architecture of 2 layers with 3 neurons per layer. This suggests that simpler neural network structures might be more effective for certain scenarios, possibly due to better generalization capabilities.

Mutation parameters also play a crucial role in the evolutionary process. The data indicates that moderate mutation possibilities (50-60%) combined with lower mutation rates (2-3%) tend to produce better results across all three scenarios. This is exemplified in TSim2 and several high-performing San Diego simulations, suggesting that this balance allows for sufficient exploration of the solution space while maintaining stable evolution.

The median fitness scores across all simulations, detailed in Appendix 4, remained relatively low compared to their maximum fitness values, indicating significant performance variation within individual populations. This suggests that while the evolutionary process can produce highly capable individuals, maintaining consistent performance across the entire population remains challenging.

Optimal performance in ADS evolution depends on a delicate balance of parameters rather than extreme values in any single dimension. The results advocate for simpler neural network architectures, moderate mutation rates, and selective pressure that maintains diversity while promoting improvement. These findings could inform future approaches to evolutionary algorithm design for autonomous vehicle development and similar complex optimization problems in other domains.

In this work, some methodological aspects of gene selection and system evolution through algorithms and some practical aspects related to their implementation in neural networks and different simulation environments have been addressed. From the methodological point of view, the results extend to the design of a genetic algorithm, for its simplicity and low computational power requirements, and the design of a feedforward artificial neural network, for its accuracy in predicting the state of the system. Both bio-inspired models that have been used for the design of these algorithms are well-known in the technical literature and are derived from organic systems found in nature.

Future Research

While these neuroevolutionary approaches significantly advanced policies, the simulations operated within simplified environments. Future plans include expanding the simulation environment to capture more real-world complexities and dynamics that include adding varied road types, traffic controls, weather effects, and pedestrian agents. Simulating more of these intricate environments will help further validate and enhance these evolved driving policies. Additional research aims include a focus on

efficiency and smoothness of control, exploring multi-objective optimization targets beyond the scalar driving fitness metric used, and consideration of other vehicles to promote safer, more well-rounded driving policies.

Expanding beyond feedforward neural network architectures to more complex neural models like convolutional neural networks (CNN) and recurrent neural networks (RNN) are also planned as these may better capture visual, sequential, and time-series data patterns relevant for driving scenarios.

Bibliography

[1] National Highway Traffic Safety Administration, *Automated Driving Systems: A Vision for Safety*. USGPO, 2017.

[2] U.S Department of Transportation, *Preparing for The Future of Transportation: Automated Vehicles 3.0*. USGPO, 2018.

[3] J. M. Anderson, K. Nidhi, K. D. Stanley, P. Sorensen, C. Samaras, and O. A. Oluwatola, *Autonomous vehicle Technology: A Guide for Policymakers*. Rand Corporation, p. 29-30, 2014.

[4] S. Varun, T. Rohan, S. Abhishek and S. Rajan, 2020, “Self-Driving Car Simulation Using Genetic Algorithm,” International Journal for Research in Applied Science & Engineering Technology (IJRASET)., Vol. 8, No. 4, pp. 514-519.

[5] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, 2. ed. in Wiley-Interscience. Hoboken, NJ: Wiley, p. 22, 179, 2004.

[6] R. E. Shannon, “Introduction to the art and science of simulation,” in *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*, Washington, DC, USA: IEEE, 1998, pp. 7–14. doi: 10.1109/WSC.1998.744892.

[7] Y. Kubera, P. Mathieu, and S. Picault, “IODA: an interaction-oriented approach for multi-agent based simulations,” *Auton Agent Multi-Agent Syst*, vol. 23, no. 3, pp. 303–343, Nov. 2011, doi: 10.1007/s10458-010-9164-z.

[8] L. Ma, Z. Ou, Y. Zhang, J. Luo and R. Tian, "Design of Decision Model of Intelligent Distribution System of Telecom Logistics Based on Neural Network Optimization Genetic Algorithm," 2023 2nd International Conference on Artificial Intelligence and Autonomous Robot Systems (AIARS), Bristol, United Kingdom, 2023, pp. 214-218, doi: 10.1109/AIARS59518.2023.00050.

[9] H. Ma, "Simulation of Logistics Route Optimization Model Based on Genetic Algorithm Optimization Neural Network," 2023 2nd International Conference on Artificial Intelligence and Autonomous Robot Systems (AIARS), Bristol, United Kingdom, 2023, pp. 105-109, doi: 10.1109/AIARS59518.2023.00028.

[10] X. Feng and Y. Zhu, "Trace AI Simulation of Feedforward Neural Network Visualization Optimized by Genetic Algorithm Based on Unity3D," *2021 China Automation Congress (CAC)*, Beijing, China, 2021, pp. 4934-4938, doi: 10.1109/CAC53003.2021.9727304.

[11] Y. Lu and R. Kuang, "A Driver Injury Prediction Model based on Genetic Algorithm and BP Neural Network," 2023 7th International Conference on Transportation

Information and Safety (ICTIS), Xi'an, China, 2023, pp. 1984-1989, doi: 10.1109/ICTIS60134.2023.10243825.

[12] H. J. Kaleybar, M. Davoodi, M. Brenna and D. Zaninelli, "Applications of Genetic Algorithm and Its Variants in Rail Vehicle Systems: A Bibliometric Analysis and Comprehensive Review," in IEEE Access, vol. 11, pp. 68972-68993, 2023, doi: 10.1109/ACCESS.2023.3292790.

[13] D. Quang Tran and S.-H. Bae, “Proximal Policy Optimization Through a Deep Reinforcement Learning Framework for Multiple Autonomous Vehicles at a Non-Signalized Intersection,” *Applied Sciences*, vol. 10, no. 16, p. 5722, Aug. 2020, doi: 10.3390/app10165722.

[14] M. O. Radwan, A. A. H. Sedky, and K. M. Mahar, “Obstacles Avoidance of Self-driving Vehicle using Deep Reinforcement Learning,” in *2021 31st International Conference on Computer Theory and Applications (ICCTA)*, Alexandria, Egypt: IEEE, Dec. 2021, pp. 215–222. doi: 10.1109/ICCTA54562.2021.9916640.

[15] J. Cui, L. Yuan, L. He, W. Xiao, T. Ran, and J. Zhang, “Multi-Input Autonomous Driving Based on Deep Reinforcement Learning With Double Bias Experience Replay,” *IEEE Sensors J.*, vol. 23, no. 11, pp. 11253–11261, Jun. 2023, doi: 10.1109/JSEN.2023.3237206.

[16] R. Rauch, S. Korecko, and J. Gazda, “Evaluation of Proximal Policy Optimization with Extensions in Virtual Environments of Various Complexity,” in *2022 32nd International Conference Radioelektronika (RADIOELEKTRONIKA)*, Kosice, Slovakia: IEEE, Apr. 2022, pp. 1–5. doi: 10.1109/RADIOELEKTRONIKA54537.2022.9764924.

[17] Y. Saez, D. Perez, O. Sanjuan, and P. Isasi, “Driving Cars by Means of Genetic Algorithms,” in *Parallel Problem Solving from Nature – PPSN X*, vol. 5199, G. Rudolph, T. Jansen, N. Beume, S. Lucas, and C. Poloni, Eds., in Lecture Notes in Computer Science, vol. 5199. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1101–1110.

[18] S. Arrigoni, F. Braghin, and F. Cheli, “MPC path-planner for autonomous driving solved by genetic algorithm technique,” *Vehicle System Dynamics*, vol. 60, no. 12, pp. 4118–4143, Dec. 2022.

[19] A. M. Naveen, R. Ravish, and S. Ranga Swamy, “Distributional Reinforcement Learning For Automated Driving Vehicle,” in *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*, Mysuru, India: IEEE, Oct. 2022, pp. 1–6.

[20] R. R. O. Al-Nima, T. Han, S. a. M. Al-Sumaidaee, T. Chen, and W. L. Woo, “Robustness and performance of Deep Reinforcement Learning,” *Applied Soft Computing*, vol. 105, p. 107295, Jul. 2021

[21] C. M. Samuel, “Self-Driving Cars using Genetic Algorithm,” *IJRASET*, vol. 8, no. 11, pp. 508–511, Nov. 2020.

[22] S. G, V. S, S. S, and G. Suganya, “Application of Neuroevolution in Autonomous Cars.” arXiv, Jun. 26, 2020. Accessed: Nov. 17, 2023. [Online]. Available: http://arxiv.org/abs/2006.15175

[23] A. J. M. Muzahid, S. F. Kamarulzaman, and M. A. Rahman, “Comparison of PPO and SAC Algorithms Towards Decision Making Strategies for Collision Avoidance Among Multiple Autonomous Vehicles,” in *2021 International Conference on Software Engineering & Computer Systems and 4th International Conference on Computational Science and Information Management (ICSECS-ICOCSIM)*, Pekan, Malaysia: IEEE, Aug. 2021, pp. 200–205. doi: 10.1109/ICSECS52883.2021.00043.

[24] *Unity – Manual: Introduction to scenes*, Unity Technologies, San Francisco, CA, https://docs.unity3d.com/Manual/CreatingScenes.html (accessed December 1, 2023).

[25] *Unity – Manual: Introduction to GameObjects*, Unity Technologies, San Francisco, CA, https://docs.unity3d.com/Manual/GameObjects.html (accessed December 1, 2023).

[26] *Unity – Manual: Scripting*, Unity Technologies, San Francisco, CA, https://docs.unity3d.com/Manual/ScriptingSection.html (accessed December 1, 2023).

[27] OpenStreetMap Wiki contributors, "Beginners Guide 1.3," *OpenStreetMap Wiki,* https://wiki.openstreetmap.org/w/index.php?title=Beginners_Guide_1.3&oldid=2582746 (accessed December 16, 2023).

[28] OpenStreetMap Wiki contributors, "OpenStreetMap Carto," *OpenStreetMap Wiki,* https://wiki.openstreetmap.org/w/index.php?title=OpenStreetMap_Carto&oldid=2622328 (accessed December 16, 2023).

[29] OpenStreetMap Wiki contributors, "CyclOSM," *OpenStreetMap Wiki,* https://wiki.openstreetmap.org/w/index.php?title=CyclOSM&oldid=2569286 (accessed December 16, 2023).

[30] OpenStreetMap Wiki contributors, "Transport Map," *OpenStreetMap Wiki,* https://wiki.openstreetmap.org/w/index.php?title=Transport_Map&oldid=2328782 (accessed December 16, 2023).

[31] OpenStreetMap Wiki contributors, "ÖPNVKarte," *OpenStreetMap Wiki,* https://wiki.openstreetmap.org/w/index.php?title=%C3%96PNVKarte&oldid=2613206 (accessed December 16, 2023).

[32] OpenStreetMap Wiki contributors, "HOT style," *OpenStreetMap Wiki ,* https://wiki.openstreetmap.org/w/index.php?title=HOT_style&oldid=2437938 (accessed December 16, 2023).

[33] OSMF Operations Working Group, “Criteria for possible inclusion - A proposed tile layer must satisfy the following hard criteria:,” *New Tile Layers Policy,* https://operations.osmfoundation.org/policies/new-tile-layers/ (accessed December 16, 2023).

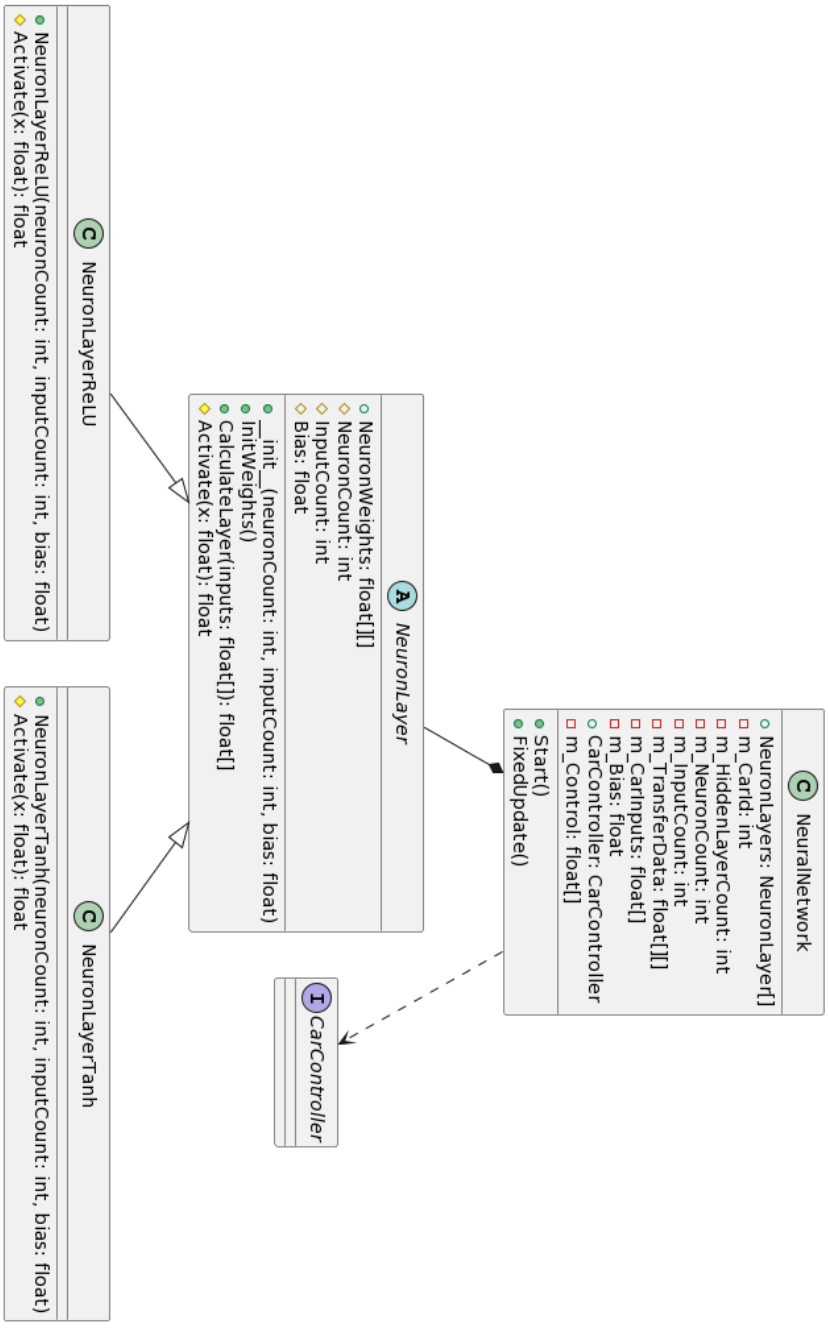
[34] M. Mitchell, 1998. *An Introduction to Genetic Algorithms.* Cambridge, MA, USA: MIT Press, 1998.

[35] OpenStreetMap Wiki contributors, "Overpass API," *OpenStreetMap Wiki,* https://wiki.openstreetmap.org/w/index.php?title=Overpass_API&oldid=2624841 (accessed December 16, 2023).

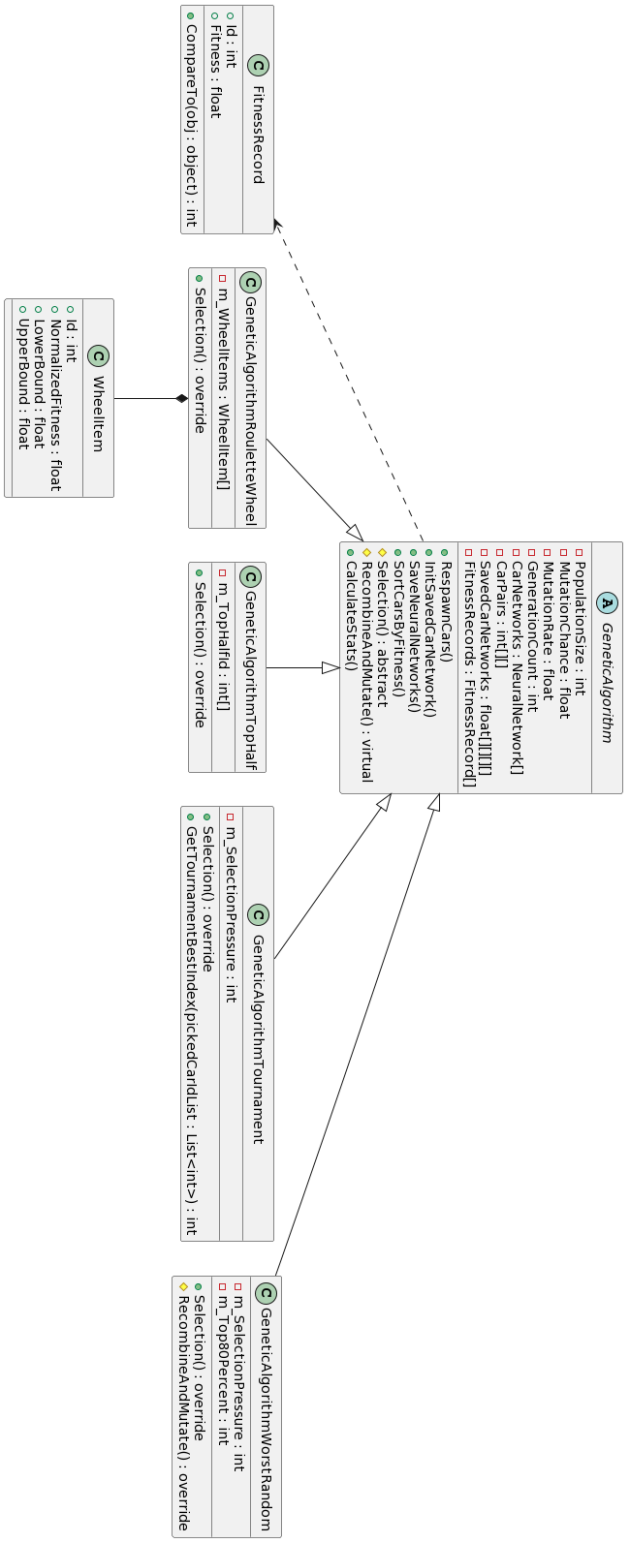
[36] *Global Roads & Traffic-User Manual*, Virtual Road, Edinburgh, UK, Accessed: December 16, 2023. [Online]. Available: https://vroad.uk/doc/installation/GlobalRoadsAndTraffic_Manual.pdf

Appendix 1: Select UML Class Diagrams

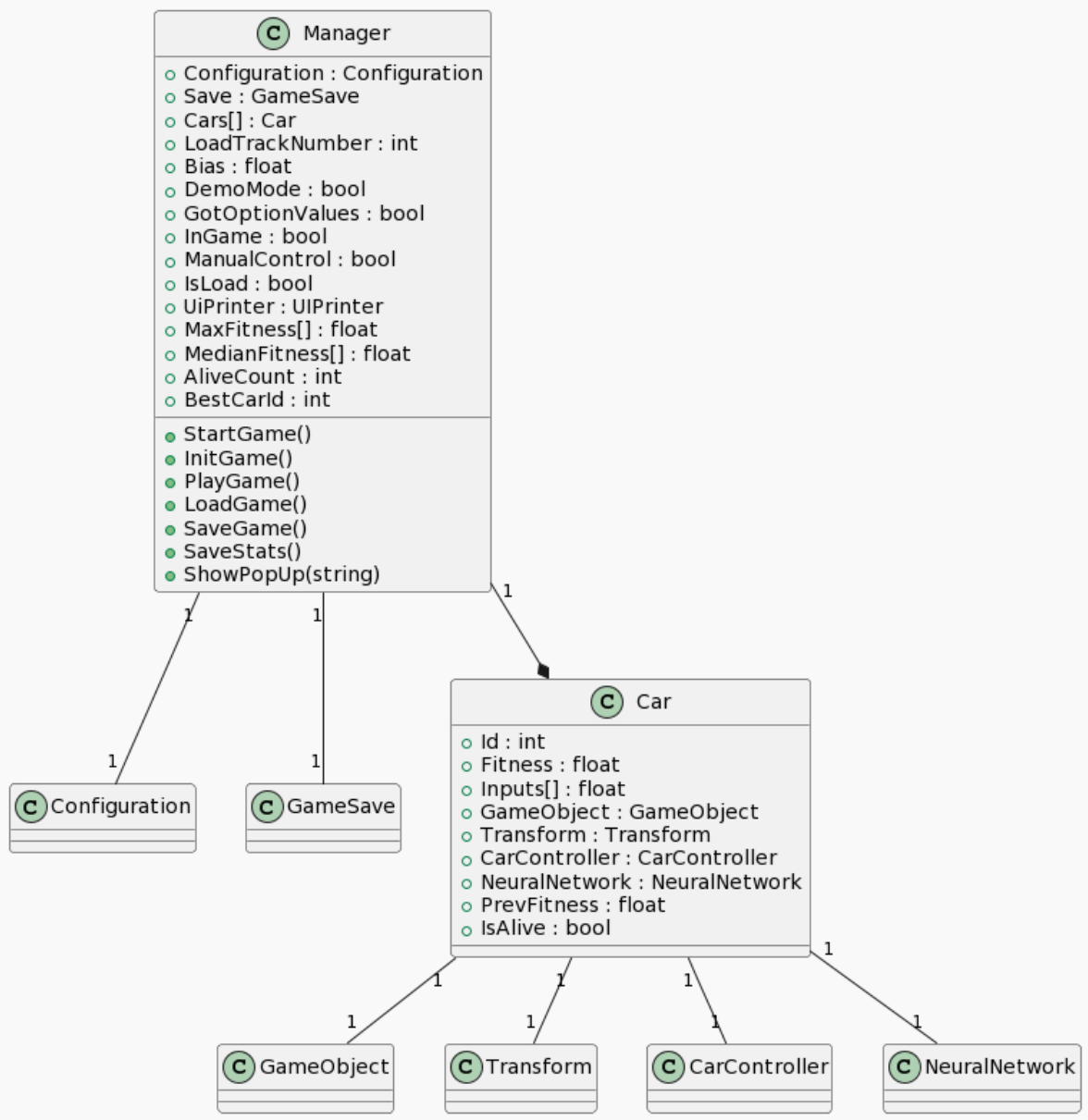
1-1. Neural Network Class



1-2. Genetic Algorithm Class



1-3. Simulation Manager Class



Appendix 2: 3D Modeling and Physics

2-1. Box Collider

The Box Collider acts as the main collider for detecting collisions of the overall agent car object. It is added as a component to the car GameObject and sized to match the car's body as shown in Figure 2-1.

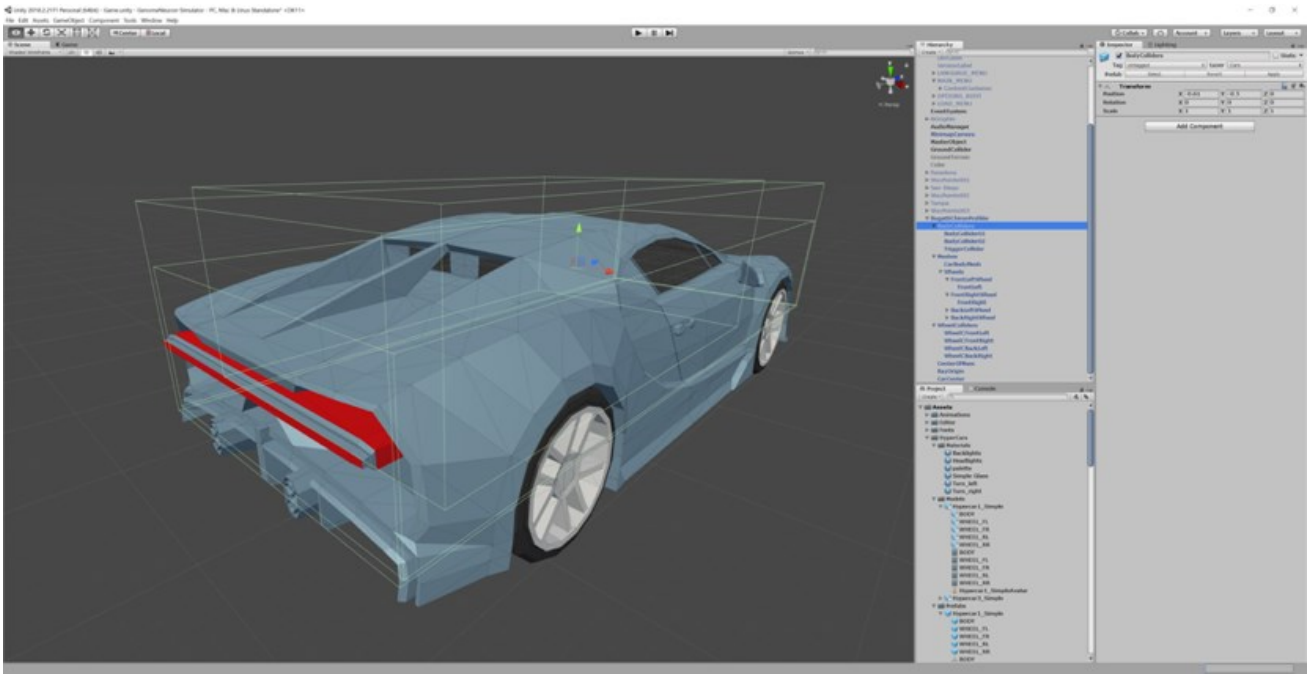


Figure 2-1. Box Collider applied to the agent car body

2-2. Wheel Collider

The Wheel Collider component specifically handles the physics and motion of the wheels. Four Wheel Collider components are added as children of the car GameObject, positioned to match the actual wheel transforms as shown in Figure 2-2.

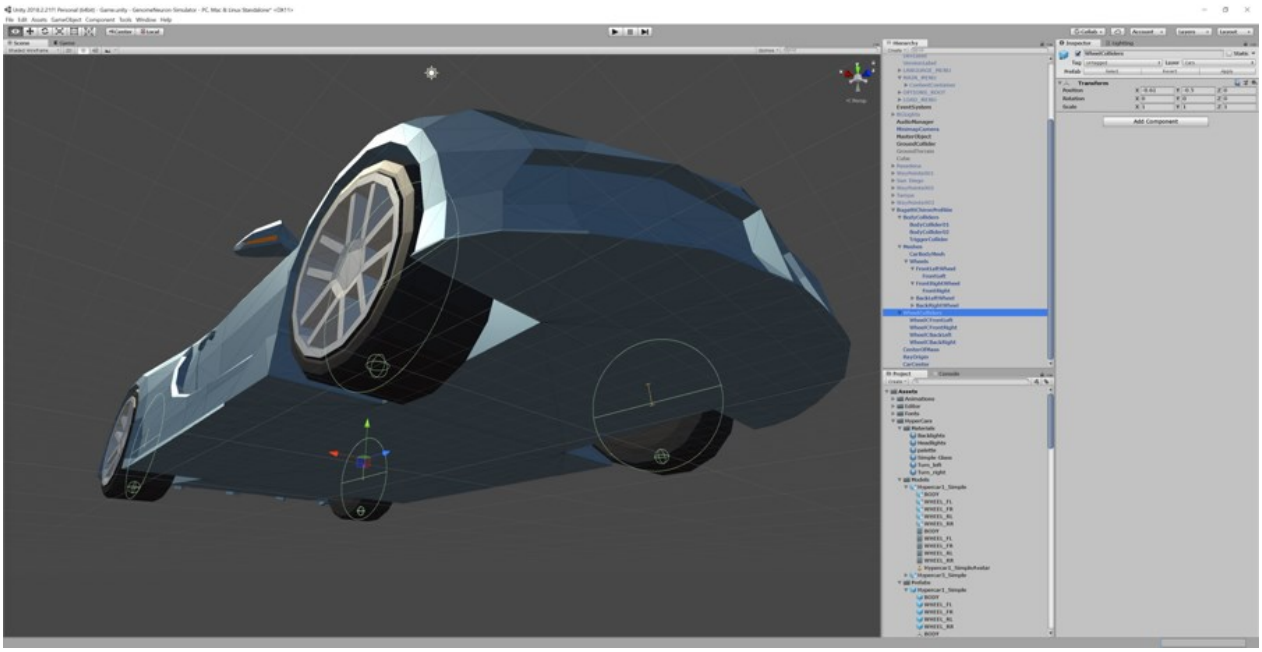


Figure 2-2. Wheel Colliders applied to the agent car tires

The Wheel Colliders simulate real wheel physics. The CarController script accesses them to apply motor/brake torques and steering angles. Wheel Colliders are invisible - the visible 3D wheel models as children are positioned to match the colliders via the UpdateMeshes() method.

Key Wheel Collider properties as seen in Figure 2-3 are:

- Mass - The wheel's mass affects physics

- Radius/Width - Dimensions matching the wheel model
- Suspension Distance - How much the wheel can compress
- Spring/Damper - Suspension responsiveness
- Steer Angle - Maximum turn of the steering wheel
- Motor Torque - Torque applied to accelerate the wheel
- Brake Torque - Torque applied to slow the wheel
- Friction Curve - Lateral and longitudinal friction

This setup with a Box Collider for the car body and Wheel Colliders for the wheels provides realistic physics, tire friction, suspension, and controllable steering/acceleration.

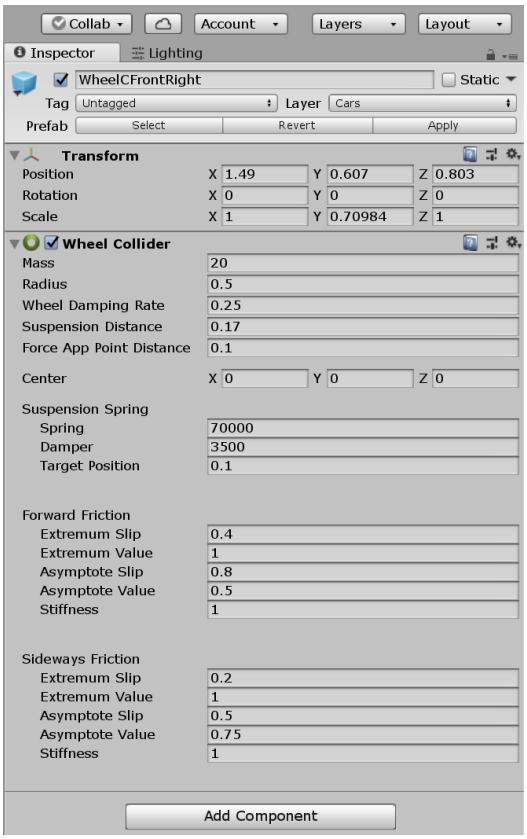


Figure 2-3. Physics settings for the Wheel Collider in Inspector panel

Appendix 3: Neural Network and Genetic Selection

3-1. Neural Network FixedUpdate() Method

The FixedUpdate() method runs each frame to utilize the neural network to control the agent car. First, it gets the latest array of input sensor data for this car from the Manager. Then, a switch statement, shown in Figure 3-1, handles propagating the data through the neural network based on the number of hidden layers. For zero hidden layers, the output layer calculates directly from the inputs. For one hidden layer, the first hidden layer calculates outputs from the inputs which are fed into the output layer. For two or more hidden layers, the data flows sequentially from input, through each hidden layer, with the output from one layer becoming inputs to the next, until finally reaching the output layer.

The resulting output layer activations are stored in *m_Control,* with the first value controlling steering and the second acceleration. Finally, these neural network output control values are applied to the CarController to physically steer and accelerate the car every frame in the simulation.

```
    private void FixedUpdate()
    {
        // The inputs array contains the car's sensor datas and it's current speed.
        m_CarInputs = Master.Instance.Manager.Cars[m_CarId].Inputs;

        switch (m_HiddenLayerCount)
        {
            // If zero hidden layer -> there is only the output layer
            case 0:
                m_Control = NeuronLayers[0].CalculateLayer(m_CarInputs);
                break;

            // If one hidden layer -> first layer gets the input,
            // second layer is the output layer.
            case 1:
                m_Control = NeuronLayers[1].CalculateLayer(
                      NeuronLayers[0].CalculateLayer(m_CarInputs));
                break;

            // If two or more hidden layers -> first layer gets the input,
            // the other ones get the output from the previous layer
            // and the last layer is the output layer.
            default:
                m_TransferData[0] = NeuronLayers[0].CalculateLayer(m_CarInputs);
                for (int i = 1; i < m_TransferData.Length; i++)
                {
                    m_TransferData[i] = NeuronLayers[i].CalculateLayer(m_TransferData[i - 1]);
                }
                m_Control = NeuronLayers[NeuronLayers.Length - 1].CalculateLayer(m_TransferData[m_TransferData.Length - 1]);
                break;
        }

        CarController.Steer = m_Control[0];
        CarController.Accelerate = m_Control[1];
    }
}
```

Figure 3-1. FixedUpdate() method of the NeuralNetwork class

3-2. Tournament C# Class

In the *GeneticAlgorithmTournament* class, the tournament size is defined by the *SelectionPressure* parameter shown in Figure 3-2. In each tournament, individuals compete and the one with the highest fitness gets selected as a parent, the rest are eliminated. This repeats, forming parent pairs for crossover. By default, one parent of each pair is randomly chosen. The second goes through the tournament process. Tournaments continue until all the breeding pairs for the next generation are filled. Multiple tournaments are held across the population.

The methods implement the tournament logic through random sampling without replacement, fitness-based sorting within groups, and pairing winners over iterations. The advantage of tournament selection is that fitter individuals have a higher chance of being selected across multiple simulation runs, but selection is still stochastic, so genetic diversity is maintained. The smaller the tournaments, the higher the selection pressure rewarding top fitness.

```
public class GeneticAlgorithmTournament : GeneticAlgorithm
{
    // During tournament selection this many cars "compete" with each other at once
    private int m_SelectionPressure = 3;

    protected override void Selection()
    {
        // Stores the selected car IDs for one round
        List<int> pickedCarIdList = new List<int>();

        int paired = 0;

        // The first parent of each pair is fully random
        for (int i = 0; i < PopulationSize; i++)
        {
            CarPairs[i][0] = RandomHelper.NextInt(0, PopulationSize - 1);
        }

        // While not all pairs are ready, new tournament
        while (paired < PopulationSize)
        {
            #region Jelenlegi tournament inicializálása
            List<int> tournament = new List<int>();
            for (int i = 0; i < PopulationSize; i++)
            {
                tournament.Add(i);
            }
            #endregion

            // As long as there are enough competitors in the tournament (and more pairs needed), select competitors
            while (tournament.Count >= m_SelectionPressure && paired < PopulationSize)
            {
                // Emptying the selected
                pickedCarIdList.Clear();

                // Until each competitor is selected
                while (pickedCarIdList.Count != m_SelectionPressure)
                {
                    int current = tournament[RandomHelper.NextInt(0, tournament.Count - 1)];
                    if (!pickedCarIdList.Contains(current))
                    {
                        pickedCarIdList.Add(current);
                        tournament.Remove(current);
                    }
                }

                // Pairing
                CarPairs[paired][1] = GetTournamentBestIndex(pickedCarIdList);
                paired++;
            }
```

Figure 3-2. Tournament genetic selection method

3-3. Elitist C# Class

GeneticAlgorithmTopHalf class selects parents from the Elitist fittest individuals of the population. As shown in Figure 3-3, it first determines which are the top half neural networks by sorting the FitnessRecords, then IDs of these top performers are stored. Then for every breeding pair, it randomly samples two parents from this top half set, ensuring they are distinct. If the same ID is picked twice, re-sampling occurs. Crossover only happens between the elite of that generation. By restricting the gene pool to above average candidates, it focuses evolution on propagating beneficial genetics of neural networks already partially optimized. However, since pairing is still random, it maintains enough diversity for continued improvement across generations.

```
public class GeneticAlgorithmTopHalf : GeneticAlgorithm
{
    private int[] m_TopHalfId;

    protected override void Selection()
    {
        m_TopHalfId = new int[PopulationSize / 2];

        for (int i = 0; i < m_TopHalfId.Length; i++)
        {
            m_TopHalfId[i] = FitnessRecords[i].Id;
        }

        for (int i = 0; i < PopulationSize; i++)
        {
            // Randomly selects one from the top 50%, it will be the left parent
            int random = RandomHelper.NextInt(0, m_TopHalfId.Length - 1);
            CarPairs[i][0] = m_TopHalfId[random];

            do
            {
                // Randomly selects one from the top 50%, it will be the right parent
                // If it is the same as the left parent, randomly selects a new one.
                random = RandomHelper.NextInt(0, m_TopHalfId.Length - 1);
                CarPairs[i][1] = m_TopHalfId[random];
            } while (CarPairs[i][0] == CarPairs[i][1]);
        }
```

Figure 3-3. Elitist genetic selection method

3-4. Tournament + 20% Random C# Class

GeneticAlgorithmWorstRandom class combines tournament selection with random resetting of the worst neural networks each generation. It runs tournaments - *while* loops in Figure 3-4 - to pick good parents for crossover. This ensures above average candidates pass on genetics. Additionally, the bottom 20% lowest performing neural networks have their weights randomly reinitialized instead of undergoing recombination. This introduces new genetic diversity and the possibility of better configurations, avoiding stagnation. Over successive generations, the threshold dividing fit and unfit neural networks rises, so more get reset as the average fitness increases from those benefiting from crossover.

```
public class GeneticAlgorithmWorstRandom : GeneticAlgorithm
{
        // During tournament selection this many cars "compete" with each other at once
        private int m_SelectionPressure = 3;
        private int m_Top80Percent;

        protected override void Selection()
        {
                // Stores the selected car IDs for one round
                List<int> pickedCarIdList = new List<int>();

                int paired = 0;

                // The first parent of each pair is fully random
                for (int i = 0; i < PopulationSize; i++)
                {
                        CarPairs[i][0] = RandomHelper.NextInt(0, PopulationSize - 1);
                }


                // While not all pairs are ready, new tournament
                while (paired < PopulationSize)
                {
                        #region Current tournament initialization
                        List<int> tournament = new List<int>();
                        for (int i = 0; i < PopulationSize; i++)
                        {
                                tournament.Add(i);
                        }
                        #endregion

                        // As long as there are enough competitors in the tournament (and more pairs needed), select competitors
                        while (tournament.Count >= m_SelectionPressure && paired < PopulationSize)
                        {
                                // Emptying the selected
                                pickedCarIdList.Clear();

                                // Until each competitor is selected
                                while (pickedCarIdList.Count != m_SelectionPressure)
                                {
                                        int current = tournament[RandomHelper.NextInt(0, tournament.Count - 1)];
                                        if (!pickedCarIdList.Contains(current))
                                        {
                                                pickedCarIdList.Add(current);
                                                tournament.Remove(current);
                                        }
                                }

                                // Pairing
                                CarPairs[paired][1] = GetTournamentBestIndex(pickedCarIdList);
                                paired++;
                        }
```

Figure 3-4. Tournament + Random 20% genetic selection method

3-5. Roulette Wheel C# Class

GeneticAlgorithmRouletteWheel class randomly picks two numbers between 0 and 1 to select parents. The numbers fall into segments which map to a specific neural network's arc. Start(), initializes the *CarNetworks* array in Figure 3-5 to hold the neural networks and creates a 2D array *CarPairs* to store the breeding pairs. It then allocates a *FitnessRecord* array to contain fitness data and allocates the *WheelItem* array with size equal to population, to store information for roulette wheel segments.

```
public class GeneticAlgorithmRouletteWheel : GeneticAlgorithm
{
    private struct WheelItem
    {
        public int Id;
        public float NormalizedFitness;
        public float LowerBound;
        public float UpperBound;
    }

    private WheelItem[] m_WheelItems;

    private void Start()
    {
        CarNetworks = new NeuralNetwork[PopulationSize];

        CarPairs = new int[PopulationSize][];
        for (int i = 0; i < CarPairs.Length; i++)
        {
            CarPairs[i] = new int[2];
        }

        FitnessRecords = new FitnessRecord[PopulationSize];
        for (int i = 0; i < FitnessRecords.Length; i++)
        {
            FitnessRecords[i] = new FitnessRecord();
        }

        m_WheelItems = new WheelItem[PopulationSize];
    }
```

Figure 3-5. Roulette wheel genetic selection method

The Selection() method normalizes the fitness scores into a proportional range as shown in Figure 3-6, so that if a neural network has a fitness twice that of another, its range also doubles. These form segments of a wheel with higher fitness scores equating to larger arcs on the wheel. Spinning a random number across generations allows periodic selection of even, low fitness neural networks to maintain diversity. Over successive generations, as fitness increases, the wheel gets divided into a finer resolution, and subsequently, fitness differentials between candidates become more pronounced. Tradeoffs are done to optimize genetic propagation across generations.

```
protected override void Selection()
{
    float minFitness = float.MaxValue;
    float maxFitness = float.MinValue;

    // Searches for the minimum and maximum fitnesses
    foreach (var stat in FitnessRecords)
    {
        if (stat.Fitness < minFitness)
            minFitness = stat.Fitness;
        if (stat.Fitness > maxFitness)
            maxFitness = stat.Fitness;
    }

    float sum = 0;

    // Normalizes the fitness values
    for (int i = FitnessRecords.Length - 1; i >= 0; i--)
    {
        m_WheelItems[i].Id = FitnessRecords[i].Id;
        m_WheelItems[i].NormalizedFitness = FitnessRecords[i].Fitness - minFitness;
        sum += m_WheelItems[i].NormalizedFitness;
    }

    float wheelUnit = 1.0f / sum;
    float current = 0;

    // Calculates the lower and upper bounds
    for (int i = 0; i < m_WheelItems.Length; i++)
    {
        m_WheelItems[i].LowerBound = current;
        m_WheelItems[i].NormalizedFitness = m_WheelItems[i].NormalizedFitness * wheelUnit;
        current += m_WheelItems[i].NormalizedFitness;
        m_WheelItems[i].UpperBound = current;
    }
```

Figure 3-6. Selection() method for fitness

Appendix 4: Aggregate Analysis of All Three Scenario Simulation Results

4-1. Correlation Analysis



Figure 4-1. Correlation analysis of all three scenarios

4-2. Regression Analysis

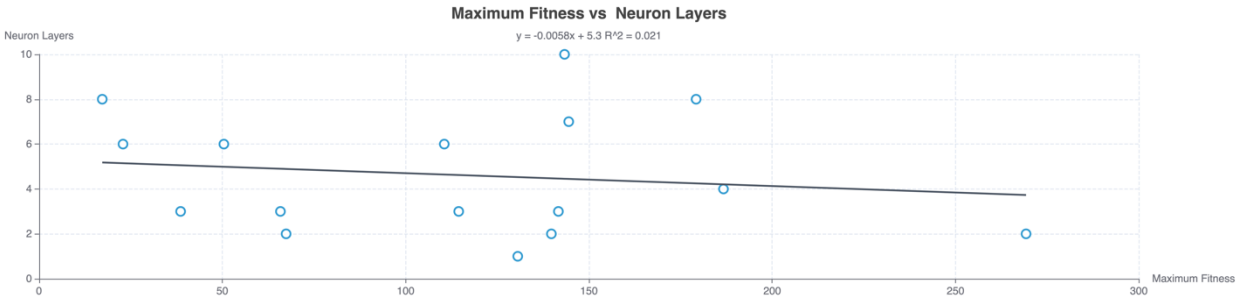


Figure 4-2. Max Fitness vs Neuron Layers regression analysis

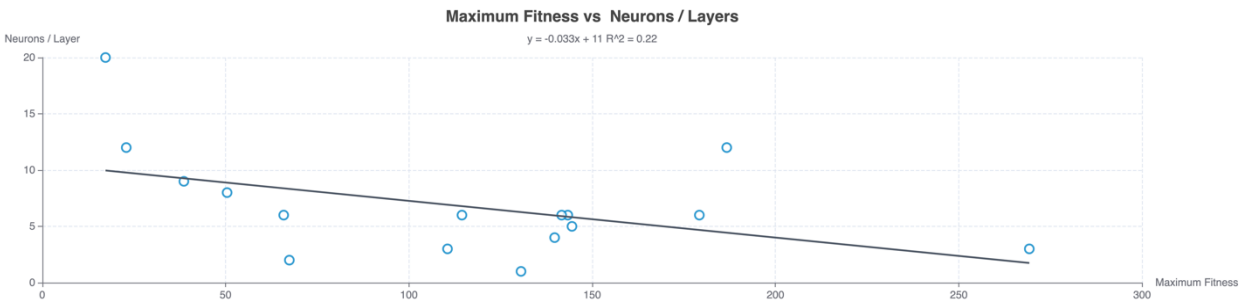


Figure 4-3. Max Fitness vs Neurons per Layer regression analysis

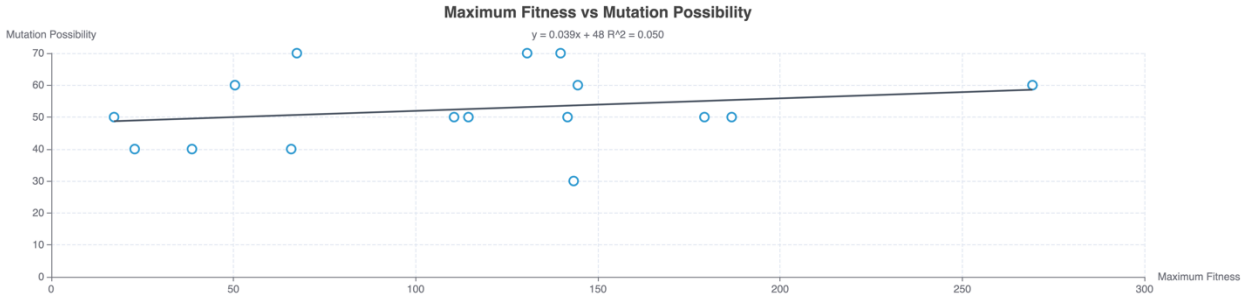


Figure 4-4. Max Fitness vs Mutation Possibility regression analysis

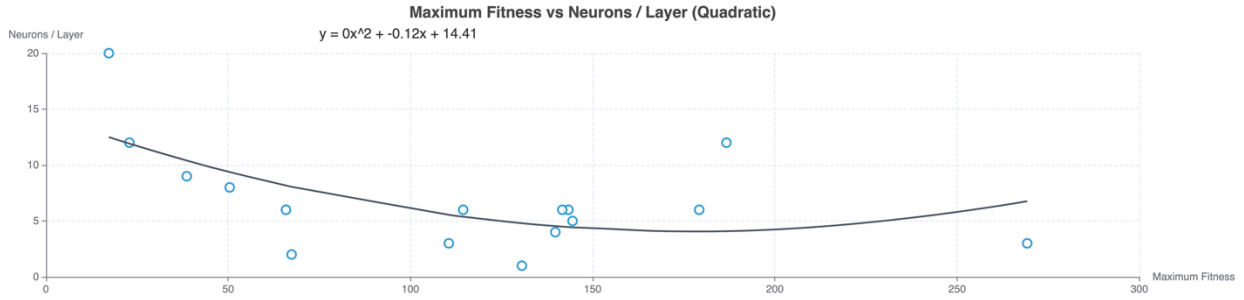


Figure 4-5. Max Fitness vs Neurons per Layer regression analysis (quadratic)