Neural network-based vehicle control in simulated environments using real-coded genetic algorithms

Denis Shabalin[1*], and *Vladimir* Stanovov[2]

[1]Siberian Federal University, 79, Svobodny Ave., Krasnoyarsk, 660041, Russia
[2]Reshetnev Siberian State University of Science and Technology, 31, Krasnoyarsky Rabochy Ave, Krasnoyarsk, 660037, Russia

Abstract. This research explores a method of optimizing neural networks for vehicle control in a simulation environment using a real-coded genetic algorithm (RCGA). The study focuses on applying RCGA in conjunction with multiple genetic operators, including simulated binary crossover (SBX), power mutation (PM), and tournament selection, to evolve neural network weights and biases, enhancing control performance for simulated vehicles. By utilizing RCGA to adjust neural network parameters, the approach enables adaptive and efficient vehicle control. The experiments demonstrate that combining sensor data with neuroevolutionary optimization in a simulation leads to a highly reliable control system, achieving performance metrics comparable to human operators. These findings suggest that RCGA-based optimization methods can be effectively applied to complex dynamic systems in various technical fields.

1 Introduction

Neuroevolutionary algorithms for solving vehicle control problems in a simulation environment hold significant meaning and value in modern realities due to the rapid advancements in autonomous driving, artificial intelligence (AI), and intelligent transportation systems. The challenges faced in developing robust, adaptable, and efficient vehicle control systems in dynamic environments make the application of neuroevolution particularly timely and relevant.

The purpose of this work is to conduct research on the method of optimizing neural networks using a real-coded genetic algorithm for the task of controlling a car in a simulation environment.

In the process of conducting this research, the following steps have been taken:

- Development of a genetic algorithm for tuning the parameters of neural networks used for vehicle control;
- Creation of a simulation environment for testing and evaluating the quality of neural network control solutions;

* Corresponding author: d.s.shabalin@yandex.ru

- Implementation and integration of neural networks and the genetic algorithm into the simulation environment;
- Conducting a series of experiments using the developed optimization method to assess its effectiveness under various parameters;
- Analysis and interpretation of experimental results.

The application of neuroevolutionary algorithms to control problems is a highly sought-after area of research and is gaining more and more support over time. For example, neuroevolution can be applied to adaptive traffic light control [1]. A fairly large number of research papers are devoted to the study of neuroevolutionary algorithms in vehicle control. For example, this problem of controlling of vehicle was solved using computer vision evolutionary strategies with deep neural networks [2]. Other work used the NEAT algorithm [3] to solve car control problems in Simulated Car Racing [4]. Similar problems have been solved in the real world, for example, controlling a robot using neuroevolution [5]. All these works demonstrate the effectiveness of combining neural networks and evolutionary algorithms to handle dynamic and complex scenarios.

This work examined the application of neuroevolution in a Unity Real-Time Development Platform [6]. The Real Coding Genetic Algorithm (RCGA) [7] is used, which develops neural networks by encoding the network's weights.

2 Description of the system

2.1 Application of neuroevolution to the problem of driving a car

In the system combining genetic algorithms and neural networks, the controlled models of vehicles serve as dynamic objects. Each vehicle model is equipped with sensors to detect the distance to the nearest objects, enabling them to interact with the environment and other objects in the simulation. The main goal of the system is to develop and test effective control algorithms for these vehicles using the evolution of neural network weights.

Figure 1 shows the 3D model used in the simulation environment.

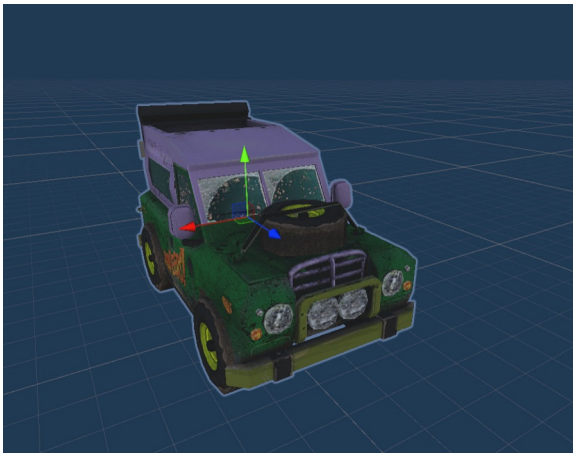

Fig. 1. Vehicle Model.

The control system for each vehicle is implemented using a neural network. The neural network receives input from the vehicle's sensors and, based on this data, makes decisions about the direction and speed of movement. The neural network's structure includes several

input neurons corresponding to the number of sensors, one or more hidden layers, and output neurons that determine the control commands for the vehicle.

Each vehicle in the system represents an individual in a population. The population encodes the weights and biases of each vehicle's neural network as chromosomes, which evolve over time using genetic algorithms. The learning process is iterative and can be described by the following key stages:

1. Initialization of the population: An initial population of vehicles is created with randomly generated neural network weights and biases.
2. Chromosome decoding: This crucial stage plays a key role in the interaction between the neural network and the genetic algorithm. At this stage, the neural network's parameters are filled with the population data, with each gene mapped to its corresponding weight or bias. To achieve this, the genes of the entire population, initially presented as a single data array before decoding, are split into five separate arrays. Each new data array is strictly ordered and responsible for its own domain in the neural network's parameters. The first three arrays are responsible for the neural network's weights, and the remaining two for biases.
3. Simulation of one generation: The population is run in the simulation. A virtual environment is generated. For testing, new virtual models of the population are created, representing individuals in the virtual environment. Each model is mapped to its own neural network, coordinates, and rotation angle in space.
4. Fitness evaluation of the population: Each individual receives a fitness score, allowing the evaluation of both the specific individual and the population as a whole.
5. Application of genetic operators: A new population is formed based on the fitness data.
6. Population update: New individuals replace the old ones, and the process repeats until a specified stopping criterion is reached, such as a maximum number of generations or a certain performance level.

The learning process can be presented in more detail with a flowchart in Figure 2.

From this flowchart, it is evident that tournament selection [8] is used as the selection operator, simulated binary crossover (SBX) [9] is applied as the crossover operator, and the mutation operator is implemented using the Power mutation method [10].

2.2 Artificial neural networks structure

The neural network is quite simple in structure, utilizing a fully connected neural network with the following parameters:

- The activation function is the hyperbolic tangent (tanh).
- The input consists of 3 neurons, responsible for sensor data from the individual.
- It is possible to set the required number of hidden layers and neurons in each layer.
- The output of the neural network has 2 neurons. The implementation of the forward pass involves matrix multiplication and addition.

2.3 Vehicle control

Each vehicle model in the simulation is equipped with several sensors, which are used to detect the distance to nearby objects. These sensors play a key role in gathering information about the environment, providing the data needed for the control system's decision-making process. Each sensor represents a vector of a given length, and when an obstacle is detected, it indicates the distance as a percentage of its length, ranging from 0 to 1, where 0 means an obstacle at the beginning of the sensor vector and 1 at the end or more. The sensors are

positioned on the front and sides of the vehicle, allowing it to detect obstacles in its path. Sensor data is updated in real time and is used as input signals for the neural network.

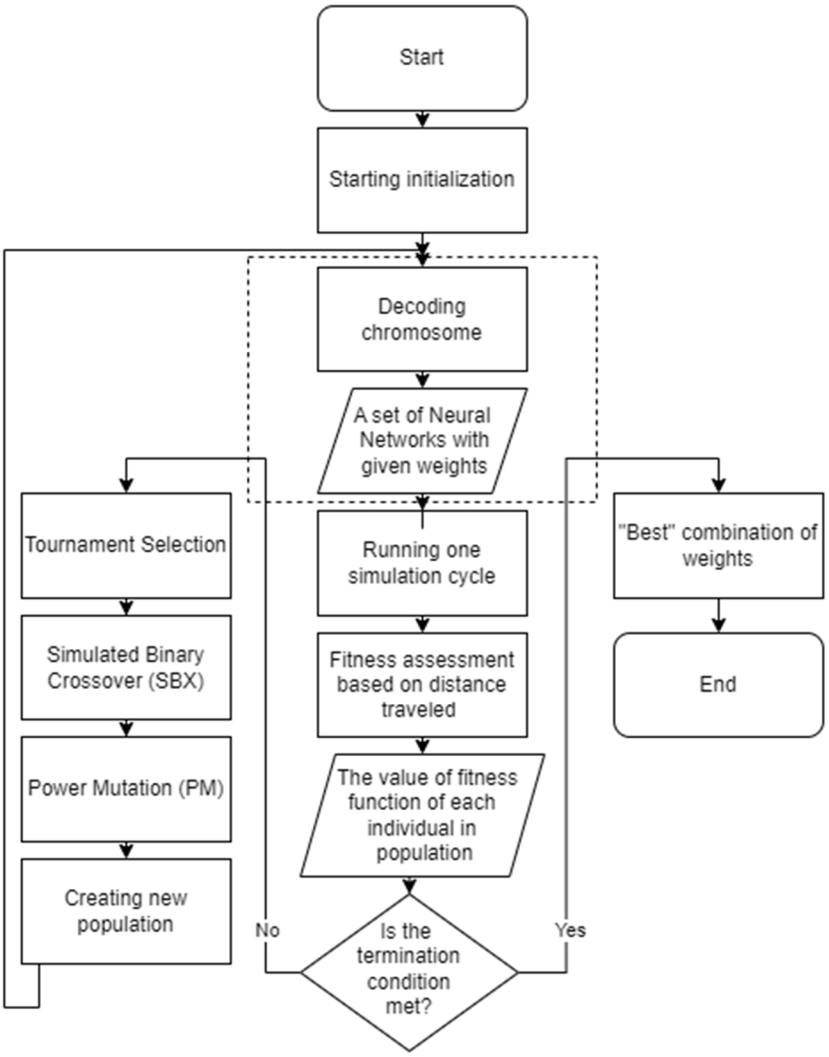

Fig. 2. Flowchart of the Neuroevolutionary Approach.

Each vehicle model in the simulation is equipped with several sensors, which are used to detect the distance to nearby objects. These sensors play a key role in gathering information about the environment, providing the data needed for the control system's decision-making process. Each sensor represents a vector of a given length, and when an obstacle is detected, it indicates the distance as a percentage of its length, ranging from 0 to 1, where 0 means an obstacle at the beginning of the sensor vector and 1 at the end or more. The sensors are positioned on the front and sides of the vehicle, allowing it to detect obstacles in its path. Sensor data is updated in real time and is used as input signals for the neural network.

Figure 3 describes what sensors that read distance look like.

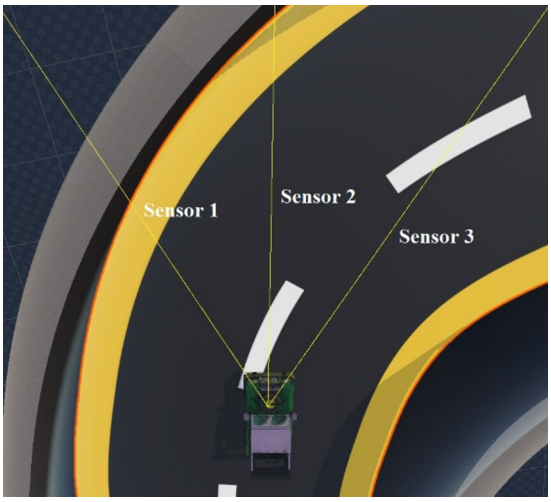

Fig. 3. Sensor Layout Diagram.

Once the neural network processes the input data, the results of the neural network's forward pass are used to control the dynamic object. The result consists of two real numbers, both of which range from minus one to plus one. The first output value of the neural network simulates steering control and is responsible for adjusting the angle of the front wheels. The second output controls speed adjustment—negative values decrease speed, while positive values increase it.

Thus, vehicle control in the simulation allows the system to achieve capabilities similar to those of a human driver, ensuring high precision and reliability under various dynamic conditions.

2.4 Fitness Evaluation

Fitness evaluation is a key aspect of training the entire system of dynamic objects. The direction of development and adaptation of the entire system depends on how accurately the fitness function is defined in accordance with the task.

The fitness function is based on how efficiently each vehicle completes the track within a given time, i.e., how much distance it can cover. At each frame of the simulation, the performance of the vehicle is assessed based on how well it accomplishes its task. Thus, the system indirectly analyzes the vehicle's ability to avoid collisions and maintain high speed along the route.

Figure 4 describes a simulation track for testing population fitness

The fitness function is calculated in several stages. First, the nearest point on the track is determined. To find the nearest position, all points on the track are compared, and the point with the smallest squared of the distance is chosen. The squared of the distance is calculated as the difference between the position of the dynamic object and the point on the track:

$$\sum_{j=1}^{n} min_i \quad \left(a_i - b_j\right)^2 \tag{1}$$

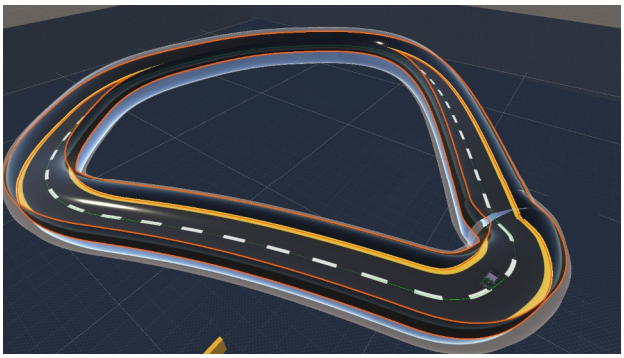

Fig. 4. Track used for fitness evaluation.

Where a is the set of points on the track, and b is the current point.

The track has a looped structure, so at some point, the movement begins again from the starting point. The shape of the track can be seen below in Figure 5.

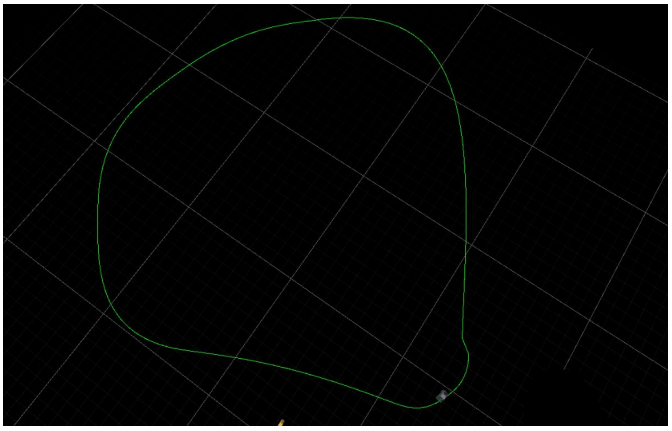

Fig. 5. Track shape.

After the nearest point is found, the percentage from the current point on the path to the starting point is calculated, full lap of the track corresponding to a fitness value of 1.0. To ensure that the fitness values do not reset when the dynamic object crosses the track's boundary and starts a new lap, each new value is compared to the previous one. If the change is less than half the track length, the number of completed laps is added. Otherwise, depending on the direction of the object's movement, the lap count is either incremented or decremented.

2.5 Description of the software environment used

To implement and test the proposed approach, a powerful and flexible software environment was used, which includes various tools and technologies that ensure high performance and development efficiency. This section provides a detailed overview of the components of the software environment used during the project.

The foundation for simulation and visualization of results was the **Unity engine**, version 2023.2.18f1. Unity is one of the most popular and powerful game engines, offering a wide range of tools for creating interactive applications and simulations. Due to its cross-platform

support and user-friendly interface, Unity allows developers to efficiently develop and test complex systems.

The genetic algorithm code was written in **C++**, as it is suitable for computationally intensive tasks. The use of C++ made it possible to efficiently implement selection, crossover, and mutation operations, ensuring high execution speed and optimal resource utilization.

The remaining project code was written in **C#**, the standard programming language for Unity. In this project, C# was used to implement simulation control logic, user interaction, and the integration of various system components.

For working with neural networks, the Unity Sentis library was used. This library supports various neural network architectures and training methods, allowing machine learning to be easily integrated into game and simulation projects.

To optimize the software product's performance, the Unity Data-Oriented Technology Stack (DOTS) was utilized. DOTS significantly improves application performance through the use of multithreading and memory optimization. The main components of DOTS include:

- Entity Component System (ECS): ECS provides efficient data organization and simplifies state management of objects in the scene.
- Jobs System: The Jobs System effectively distributes computational tasks across multiple threads, significantly speeding up the execution of complex operations.
- Burst Compiler: The Burst Compiler optimizes code execution at the machine instruction level, ensuring maximum performance.

3 Performing experiments

For the experiments, a neural network with two hidden layers and two neurons per layer was used. The sensor range was 33.3 meters, and the time for one generation was set to 25 seconds.

The system was tested on a special virtual track. The track is a closed-loop course with walls on the sides and a transparent surface above. The generation time was chosen to ensure that the system could complete the required distance on the test track.

During testing, three parameters were varied. All three parameters relate to genetic operations. The results of changing each parameter were compared in terms of how quickly the population learns and how high the final learning result is. The first parameter to be changed is eta, which relates to the crossover operator and is responsible for the exponential distribution. The second parameter for testing was mutation strength, which determines how much individuals will change. Finally, the third parameter for testing was tournament size, which is involved in tournament selection and determines how many individuals participate in the tournament.

3.1 Analysis of the obtained experimental results

To analyze the results, the final learning goal was set to determine when the algorithm would be able to control the dynamic object better than a human user. For this, the system’s training results were compared with the best user-controlled performance. The user performance was rated as one unit of fitness.

During the research, results were obtained for various values of the eta parameter, with mutation strength set to 3 and tournament size also set to 3.

Figures 7 and 8 describe the best fitness of individuals in population during experiments for various parameters of genetic operators.

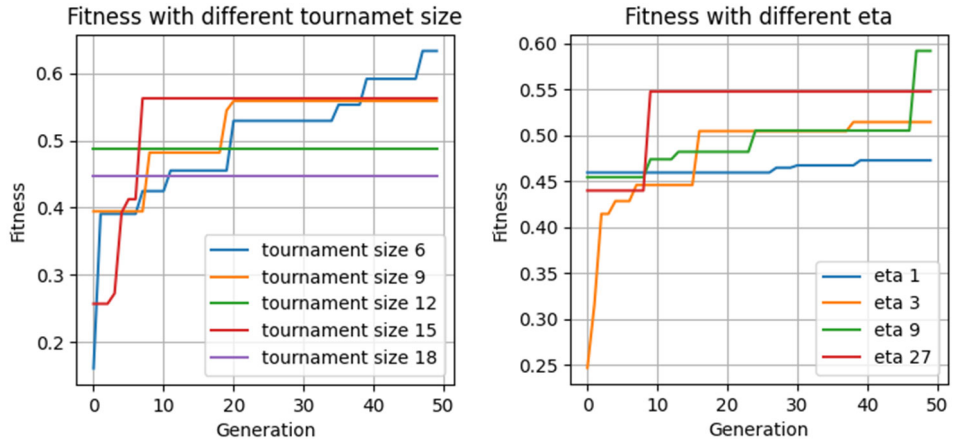

Fig. 7. Fitness graph for different eta parameters and tournament size.

Based on the graph, it can be concluded that the highest fitness is achieved when the eta parameter is set to 9. With eta set to 27, training gets stuck in a local minimum. It's also worth noting that with eta set to 1, a slow but steady genetic progress of the population is observed. Next, it was decided to fix the eta parameter at 9 and analyze how fitness behaves with an increased tournament size. From the data, it can be concluded that increasing the tournament size can improve the result. For instance, with a tournament size of 6, higher fitness is achieved, but further increases lead only to worse results and, in the long term, to an inability for the population to learn if the tournament size becomes too large.

The next step was to vary the **population size** and the **mutation strength**. It was expected that small mutation strengths would result in less disruption in decision-making, while larger values would increase diversity. The tournament size was set to 3, eta to 1, and mutation strength to 1.

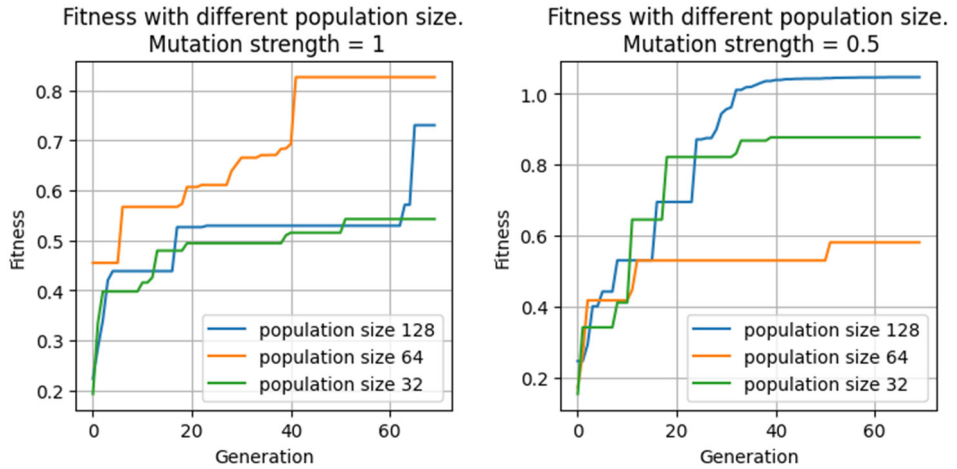

Fig. 8. Fitness graph for different population sizes and mutation strength.

It is clear that changing the mutation strength parameter significantly affects the fitness result. A similar study was conducted, but this time the mutation strength was set to 0.5.

Training with a larger number of individuals and appropriately chosen mutation parameters resulted in a fitness score that exceeded the user-controlled result by 4.6%.

TABLE 1. The results of all experiments conducted with different parameter values. Each experiment took approximately 29 minutes.

Population size	Tournament size	eta	Power mutation	Fitness
64	3	1	3	0.472
64	3	3	3	0.514
64	3	9	3	0.591
64	3	27	3	0.547
64	6	9	3	0.633
64	9	9	3	0.558
64	12	9	3	0.487
64	15	9	3	0.562
64	18	9	3	0.446
128	3	1	1	0.730
64	3	1	1	0.826
32	3	1	1	0.542
128	3	1	0.5	1.046
64	3	1	0.5	0.580
32	3	1	0.5	0.876
64	3	1	3	0.472
64	3	3	3	0.514
64	3	9	3	0.591
64	3	27	3	0.547
64	6	9	3	0.633
64	9	9	3	0.558

As shown in Table 1, the highest fitness result was 1.046, achieved with a population size of 128 individuals, a tournament size of 3, eta set to 1, and a mutation strength of 0.5. The second-best fitness result was 0.876, with a population size of 32, tournament size of 3, eta set to 1, and mutation strength of 0.5. The third-best result was 0.826, achieved with a population size of 64, tournament size of 3, eta set to 1, and mutation strength of 1.

4 Conclusion

A combined approach using genetic algorithms and neural networks was investigated and implemented to create effective control systems for dynamic objects, exemplified by vehicle models in a simulation built on the Unity 2023.2.18f1 engine. The results of the study demonstrated that neuroevolutionary methods enable the automatic generation of control algorithms.

The experiments revealed that using sensor data to gather environmental information and optimizing neural network weights through genetic algorithms significantly improved control quality. The vehicles in the simulation, treated as individuals within a population, successfully adapted and optimized their parameters through the processes of selection, crossover, and mutation.

Thus, the conducted research demonstrates that controlling a vehicle in a simulation using neural networks and genetic algorithms can achieve a level of performance and reliability comparable to human control. This opens up new possibilities for applying combined approaches in various technical and scientific fields, offering substantial potential for further research and development.

This work was supported by the Ministry of Science and Higher Education of the Russian Federation (grant No. 075-15-2022-1121).

References

1. M. Bernas, B. Płaczek, J. Smyła, *Sensors* **19(8)**, 1776 (2019)
2. A. AbuZekry, European Journal of Engineering Science and Technology **2(4),** 60-71 (2019)
3. K. Stanley, R. Miikkulainen, Evolutionary Computation **10(2),** 99-127 (2002)
4. L. Cardamone, D. Loiacono, PL. Lanzi, Evolving competitive car controllers for racing games with neuroevolution in: *Proceedings of the 11th Annual Genetic and Evolutionary Computation Conference, GECCO-2009*, 1179-1186 (2009)
5. C. Cáceres Flórez, J. Rosário, D. Amaya, Neural Computing and Applications **32 (20),** 15771-15784 (2020)
6. Unity Technologies. Unity. [Online]. Available from: https://unity.com/.
7. L. Eshelman, J. Schaffer, Foundations of Genetic Algorithms **2**, 187-202 (1993)
8. B. Miller, D. Goldberg, Complex Systems **9,** 193-212 (1995)
9. K. Deb, HG. Beyer, Evolutionary computation **9** 197-221 (2001)
10. K. Deep, M. Thakur, Applied Mathematics and Computation **193** 211-230 (2007)