Report of

Major Project

on

AI Self Driving Learning Car Using Reinforcement Learning

Submitted By

Pranav Gopalrao Chole

(2019BEC065)

Siddhant Sham Deshmukh

(2019BEC072)

Under the guidance of

DR. A. V. Nandedkar

Department of Electronics and Telecommunication

Shri Guru Gobind Singhji Institute of Engineering and Technology,

Vishnupuri, Nanded-431606

(May 2023)

Declaration

We hereby declare that the Major project report on "AI Self Driving Learning Car Using Reinforcement Learning" submitted herein has been prepared by us as students of VIII semester B. Tech program of the Department of Electronics and Telecommunication.

We also hereby assign to SGGSIE&T Nanded all rights under copyright that may exist in and to the above work and any revised or expanded derivatives works based on the work as mentioned. Other work referred from references, manuals etc. is duly acknowledged.

Pranav Chole (2019BEC065)

Siddhant Deshmukh (2019BEC072)

Certificate

Shri Guru Gobind Singhji Institute of Engineering and Technology,

Vishnupuri, Nanded-431606

Department of Electronics and Telecommunication

This is to certify that the B Tech Project entitled, "AI Self Driving Learning Car Using Reinforcement Learning", submitted by Pranav Chole, Siddhant Deshmukh is bonafied work carried out by them as a part of B. Tech. (Electronics and Telecommunication) Final Year.

Place: Nanded

Date: May 2023

Course Co-Ordinator
Dr. A. V. Nandedkar
Project Guide and Head

Approval Sheet

This Project report entitled

Design and Development of Management Information System for SAE Clubs

Submitted By

Pranav Chole (2019BEC065)

Siddhant Deshmukh (2019BEC072)

Viva-voce for above project work is conducted on _____ and the work is approved

for the degree of Bachelor of Technology in Electronics and Telecommunication from Shri Guru Gobind Singhji Institute of Eng. & Technology, Nanded.

	Examiners	Name	Signature
1	External Examiner	_____	
2	Internal Examiner	_____	

Date:

Place:

Acknowledgment

We express our sincere gratitude to Dr. M. V. Bhalerao, Head of Dept. Electronics and Telecommunication for their stimulating guidance, continuous encouragement, and supervision throughout the course of present work and would like to place on record our deep sense of gratitude for his generous guidance, help and useful suggestions.

Lastly, we would like to thank all our friends and library staff members whose encouragement and suggestions helped us to complete our project. We are also thankful to all those people who have contributed directly or indirectly to the completion of this project.

Thank you.

Pranav Chole (2019BEC065)

Siddhant Deshmukh (2019BEC072)

Table of Contents

Abstract

This report presents a simulation of a self-driving car implemented using the NEAT (NeuroEvolution of Augmenting Topologies) algorithm and the Pygame library. The simulation aims to replicate real-world driving scenarios in 2D track environment rendered using Pygame and evaluate the performance of the self-driving car in terms of navigation, obstacle avoidance, and overall driving proficiency.

The NEAT algorithm is utilized to evolve neural networks that control the car's actions in response to its environment. Through an iterative process of genetic evolution and neural network adaptation, the self-driving car's neural network learns to make driving decisions based on input from sensors, such as distance and speed sensors.

Pygame, a popular Python library for game development, is employed to create a visually immersive and interactive simulation environment. The simulation environment includes 2D road layouts to test the self-driving car's ability to navigate safely and efficiently.

To evaluate the performance of the self-driving car, various metrics are measured, including average driving speed, successful completion of routes, collision rates, and overall driving smoothness. These metrics provide insights into the effectiveness of the NEAT algorithm and the trained neural network in enabling the self-driving car to perform complex driving tasks.

The results of the simulation demonstrate the potential of the NEAT algorithm in training self-driving cars. The self-driving car successfully learns to adapt to different road conditions, make appropriate decisions, and avoid collisions. However, challenges and limitations are also identified, such as the need for further refinement to enhance decision-making accuracy and real-time responsiveness.

Overall, this simulation contributes to the ongoing research and development of autonomous driving systems. The insights gained from this project can inform future improvements in self-driving car technologies, including algorithmic enhancements and real-world deployment considerations.

1 Introduction

The development of self-driving cars represents a significant advancement in the field of autonomous systems and has the potential to revolutionize transportation as we know it. These vehicles rely on sophisticated algorithms and neural networks to perceive their environment, make decisions, and navigate safely without human intervention. One popular approach for training such autonomous systems is the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, which combines genetic algorithms with neural network evolution.

In this report, we present a simulation of a self-driving car implemented using the NEAT algorithm and the Pygame library. The simulation aims to replicate real-world driving scenarios and evaluate the performance of the self-driving car in terms of navigation, obstacle avoidance, and overall driving proficiency. By creating a virtual environment that mimics the complexities of the road, we can assess the effectiveness of the NEAT algorithm in training the car's neural network to drive autonomously.

The NEAT algorithm is a powerful tool for evolving neural networks with complex topologies. It allows for the evolution of neural networks that can adapt and learn from their environment, making it suitable for training self-driving cars. By evolving the neural network's structure and weights over multiple generations, the NEAT algorithm can optimize the car's decision-making process and improve its driving performance.

To create a visually engaging and interactive simulation environment, we utilize the Pygame library. Pygame provides a comprehensive set of tools and resources for developing games and graphical applications in Python. With Pygame, we can render 2D road layouts to test the self-driving car's ability to respond effectively in different scenarios.

The evaluation of the self-driving car's performance is crucial to assess its effectiveness and identify areas for improvement. Metrics such as average driving speed, successful completion of routes, collision rates, and driving smoothness are measured to quantify the car's performance. These metrics provide insights into the capabilities of the NEAT algorithm and the trained neural network in enabling the self-driving car to navigate complex driving tasks.

The findings from this simulation contribute to the ongoing research and development of autonomous driving systems. By understanding the strengths and limitations of the NEAT algorithm and its application to self-driving cars, we can inform future advancements in autonomous vehicle technologies. Additionally, the insights gained from this project can guide the refinement of algorithms, improve decision-making accuracy, and address challenges associated with real-time responsiveness and robustness.

Overall, this simulation serves as a valuable tool for studying and evaluating the performance of self-driving cars. By leveraging the NEAT algorithm and Pygame, we can simulate realistic driving scenarios and gather valuable data to advance the field of autonomous driving.

2 Literature Review

2.1. NEAT Algorithm for Autonomous Vehicles:

The NEAT (NeuroEvolution of Augmenting Topologies) algorithm has gained significant attention in the field of autonomous vehicles. Stanley and Miikkulainen (2002) introduced NEAT as a method for evolving neural networks with complex topologies. The algorithm combines genetic algorithms with neural network evolution, allowing for the development of adaptable and efficient neural networks. NEAT has shown promising results in training autonomous vehicles, enabling them to learn and improve their driving performance over time.

2.2. Simulation Environments for Autonomous Vehicles:

Simulation environments play a crucial role in the development and evaluation of self-driving car systems. Researchers have utilized various frameworks and libraries to create realistic virtual environments. Pygame, a Python library for game development, has been widely used due to its flexibility and ease of use. It allows for the creation of visually immersive environments with customizable road layouts. Pygame facilitates the evaluation of self-driving car algorithms in complex and diverse scenarios.

2.3. Performance Evaluation Metrics for Self-Driving Cars:

Measuring the performance of self-driving cars is essential to assess their capabilities and identify areas for improvement. Several metrics have been established to evaluate the performance of autonomous vehicles. These metrics include average driving speed, successful completion of routes, collision rates, and driving smoothness. By quantifying these metrics, researchers can objectively compare different algorithms and techniques, ultimately advancing the development of autonomous driving systems.

2.4. Challenges and Limitations in Self-Driving Car Systems:

Despite significant progress, self-driving car systems still face numerous challenges and limitations. Real-time decision-making and robustness in complex environments remain areas of concern. Factors such as unpredictable pedestrian behavior, adverse weather conditions, and unanticipated road events pose significant challenges for autonomous vehicles. Researchers are actively investigating approaches to address these challenges, including improved perception systems, advanced control algorithms, and robust machine learning techniques.

2.5. Real-World Deployment Considerations:

The transition of self-driving cars from simulated environments to real-world deployment involves several critical considerations. Regulatory frameworks, ethical dilemmas, and public acceptance are key factors that need to be addressed. Safety and reliability are paramount concerns, as self-driving cars must consistently demonstrate their ability to operate safely in diverse and unpredictable situations. Collaborative efforts between researchers, policymakers, and industry stakeholders are crucial for the successful integration of autonomous vehicles into our transportation systems.

2.6. Future Directions in Autonomous Driving Research:

The field of autonomous driving continues to evolve rapidly, with ongoing research focusing on various aspects. Advancements in sensor technologies, machine learning algorithms, and computational power are expected to drive further improvements in self-driving car systems. Additionally, the integration of artificial intelligence techniques, such as reinforcement learning and deep learning, holds great promise for enhancing the capabilities and decision-making processes of autonomous vehicles.

In conclusion, the literature review highlights the significance of the NEAT algorithm in training self-driving cars and the importance of simulation environments for evaluation purposes. It also emphasizes the need for comprehensive performance evaluation metrics, addresses the challenges and limitations faced by self-driving car systems, and discusses considerations for real-world

deployment. Future directions in autonomous driving research indicate exciting advancements on multiple fronts, which will contribute to the realization of safe and efficient autonomous vehicles.

3 Problem Statement

3.1 Problem Statement

The development of self-driving cars presents a complex challenge in achieving safe and efficient autonomous transportation. Despite significant progress in the field, there are still key issues that need to be addressed to accelerate the widespread adoption of self-driving technology.

3.2 Safety and Reliability:

Ensuring the safety and reliability of self-driving cars remains a critical concern. Autonomous vehicles must demonstrate an ability to navigate unpredictable road conditions, respond to dynamic obstacles, and make appropriate decisions in real-time to prevent accidents. Achieving a level of safety comparable to or exceeding human drivers is essential to gain public trust and regulatory approval.

3.3 Robustness in Complex Environments:

Autonomous driving systems must be capable of operating reliably in diverse and complex environments. Factors such as adverse weather conditions, unanticipated road events, and interactions with pedestrians and other vehicles pose significant challenges. Developing algorithms that can handle these complex scenarios and adapt to changing conditions is crucial for the successful deployment of self-driving cars.

3.4 Ethical Decision-Making:

Self-driving cars face ethical dilemmas that require sound decision-making algorithms. Situations involving potential collisions or trade-offs between the safety of occupants and pedestrians require careful consideration. Developing ethical frameworks and algorithms that can handle such situations in a fair and responsible manner is vital for gaining societal acceptance and ensuring ethical operation of autonomous vehicles.

3.5 Real-World Deployment:

Transitioning self-driving cars from research and simulation environments to real-world deployment involves various considerations. Legal and regulatory frameworks need to be established to govern the operation of autonomous vehicles on public roads. Infrastructure requirements, data privacy, cybersecurity, and public acceptance are also crucial factors that need

to be addressed for successful integration of self-driving technology into existing transportation systems.

3.6 Human-Machine Interaction:

The interaction between self-driving cars and human drivers, pedestrians, and other road users is an important area of research. Ensuring effective communication and understanding between autonomous vehicles and human users is essential for safe and efficient integration. Designing intuitive interfaces and developing clear communication methods will be vital to facilitate harmonious interaction between self-driving cars and other road participants.

Addressing these challenges and advancing the state of self-driving technology requires interdisciplinary research, collaboration between academia, industry, and policymakers, and continuous innovation. Solving these problems will not only revolutionize transportation but also bring significant benefits in terms of safety, efficiency, and accessibility, transforming the way we travel and commute in the future.

4 Theory

4.1 Python

Python is the programming language used for this self-driving car simulation project. It offers several features and advantages that make it suitable for developing this type of application. Here are some key points about Python's relevance to the project:

4.1.1 Ease of use:

Python is known for its simplicity and readability. It provides a clean and straightforward syntax, making it easier to write and understand code. This is particularly beneficial for complex projects like a self-driving car simulation.

4.1.2 Large ecosystem:

Python has a vast ecosystem of libraries and frameworks that can be leveraged for various purposes. In this project, the `neat` library is used for implementing the NEAT (NeuroEvolution of Augmenting Topologies) algorithm, while `pygame` is used for handling the graphical aspects of the simulation.

4.1.3 Machine learning capabilities:

Python has become a popular choice for machine learning and artificial intelligence applications. It offers libraries like `neat-python` that provide tools for implementing evolutionary algorithms, which are crucial for training the self-driving car agents.

4.1.4 Visualization and graphics:

Python has libraries like `pygame` that enable the creation of interactive graphical applications. It provides functionalities for rendering game graphics, handling user input, and displaying visual elements such as the car sprite and the game map in this simulation.

4.1.5 Flexibility and extensibility:

Python allows for modular and extensible code design. It supports object-oriented programming, allowing the simulation components to be organized into classes with their own attributes and

methods. This makes it easier to manage and modify different aspects of the simulation, such as the car behavior, sensors, and environment.

Overall, Python's simplicity, extensive libraries, and machine learning capabilities make it a suitable choice for developing the self-driving car simulation project. It provides the necessary tools for implementing the required algorithms, handling graphics, and training the agents effectively.

4.2 Pygame

Pygame is a popular Python library specifically designed for developing games and multimedia applications. It provides a set of modules and functions that simplify the process of creating interactive graphics and handling user input. In the self-driving car simulation project, Pygame is used to handle the graphical aspects of the simulation. Here's an overview of Pygame's relevance to the project:

4.2.1 Graphics rendering:

Pygame provides functionalities for rendering graphics on the screen. It allows you to load and display images, such as the car sprite and the game map. The `pygame.image.load()` function is used to load image files, and the `blit()` method is used to draw images on the screen.

4.2.2 Event handling:

Pygame facilitates the handling of user input events, such as keyboard and mouse events. It provides functions to detect and respond to events like key presses, mouse clicks, and window closures. In the project, the `pygame.event.get()` function is used to retrieve the current event queue, and the `QUIT` event is checked to handle the program's termination.

4.2.3 Screen management:

Pygame allows you to create and manage the game screen or window. The `pygame.display.set_mode()` function is used to create a window of a specific size, and the `pygame.display.flip()` method updates the contents of the screen. Pygame also provides functions to set the window title, set the window icon, and control the screen refresh rate.

4.2.4 Timing and animation:

Pygame provides a clock mechanism that allows you to control the frame rate of the game loop. The `pygame.time.Clock()` function creates a clock object that can be used to regulate the frame rate using the `tick()` method. This ensures that the simulation runs at a consistent speed, typically measured in frames per second (FPS).

4.2.5 Drawing and collision detection:

Pygame offers functions for drawing shapes and lines on the screen. These can be utilized to visualize the car, sensors, and other elements in the simulation. Additionally, Pygame provides collision detection functions to check for collisions between objects, which can be used to determine if the car has crashed or hit any obstacles.

By leveraging Pygame's features, the project can create a visually appealing and interactive simulation environment. It allows for the rendering of graphics, handling of user input, management of the game screen, timing control, and drawing capabilities necessary for implementing the self-driving car simulation.

4.3 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that focuses on training agents to make sequential decisions in an environment to maximize a cumulative reward. In the context of the self-driving car simulation project, RL can be used to train the car to autonomously navigate the environment and learn optimal driving behaviors.

Here's a brief explanation of how RL can be applied to this project:

4.3.1 Agent:

The self-driving car in the simulation is the RL agent. It interacts with the environment, observes its state, and takes actions based on a policy.

4.3.2 Environment:

The game environment represents the virtual world in which the car operates. It provides feedback to the agent in the form of states, rewards, and possible actions.

4.3.3 State:

The state represents the current information or snapshot of the environment that the agent perceives. It typically includes relevant variables such as the car's position, speed, heading, and sensor readings.

4.3.4 Action:

Actions are the decisions made by the agent in response to the observed state. In the context of the self-driving car, actions can include turning left or right, accelerating, or decelerating.

4.3.5 Reward:

The reward is a scalar value that the agent receives from the environment based on its actions. In this project, the reward can be defined based on the car's performance, such as distance traveled, avoiding collisions, or reaching certain checkpoints.

4.3.6 Policy:

The policy defines the strategy or behavior of the agent. It maps states to actions and determines how the agent selects its actions based on the observed state. The RL algorithm aims to learn an optimal policy that maximizes the cumulative reward over time.

4.3.7 Training:

The RL agent learns through an iterative process of exploration and exploitation. It explores the environment by taking actions based on an exploration strategy (e.g., epsilon-greedy) to gather experience. The experiences, consisting of state, action, reward, and next state, are stored in a replay buffer. The agent then samples from the replay buffer and uses the RL algorithm (e.g., Q-learning, DQN, or policy gradients) to update its policy based on the observed rewards and experiences.

4.3.8 Evaluation:

Once the agent has been trained, it can be evaluated by running it in the environment without further updates to the policy. The agent's performance can be assessed based on metrics such as the average reward, distance traveled, or the number of successful completions of the course.

By applying reinforcement learning techniques, the self-driving car can learn to make informed decisions based on its observations and maximize its performance in the simulation. Through iterative training and learning from experience, the car can improve its driving skills and navigate the environment more effectively over time.

4.4 NEAT (NeuroEvolution Augmented topologies)

Neuroevolution, i.e. evolving artificial neural networks with genetic algorithms, has been highly effective in reinforcement learning tasks, particularly those with hidden state information. An important question in neuroevolution is how to gain an advantage from evolving neural network topologies along with weights. We present a method, NeuroEvolution of Augmenting Topologies (NEAT) that outperforms the best fixed-topology methods on a challenging benchmark reinforcement learning task.

4.4.1 Genetic Encoding:

NEAT is designed specifically to address the three challenges raised in the introduction. Each genome includes a list of connection genes, each of which refers to two node genes being connected (figure 1). Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an innovation number, which allows finding corresponding genes during crossover (as will be explained below). Although the experiments in this paper evolve networks with a single output, NEAT can evolve networks with any number of inputs or outputs.

Mutation in NEAT can change both connection weights and network structures. Connection weights mutate as in any NE system, with each connection either perturbed or not. Structural mutations, which expand the genome, occur in two ways (figure 2). In the add connection mutation, a single new connection gene is added connecting two previously unconnected nodes. In the add node mutation an existing connection is split and the new node placed where the old connection used to be. The old connection is disabled and two new connections are added to the genome. This method of adding nodes was chosen in order to integrate new nodes immediately into the network.

Through mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions. The next section explains how NEAT can cross over such diverse genomes.

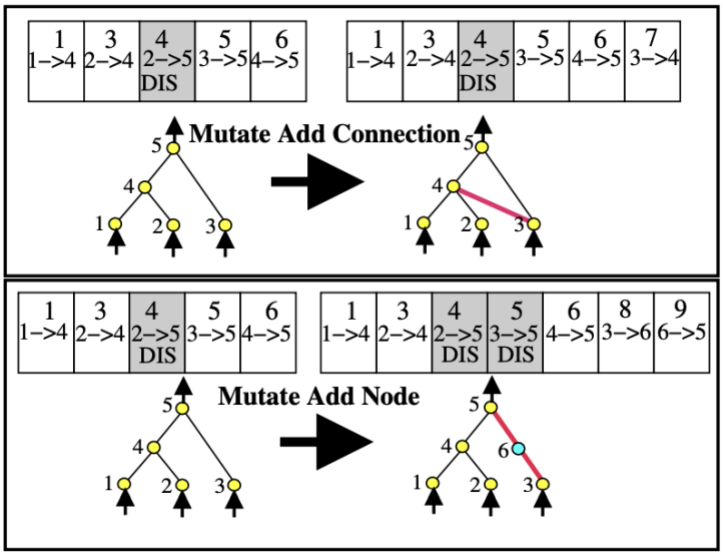

Figure 1 A genotype to phenotype mapping example*. The third gene is disabled, so the connection that it specifies (between nodes 2 and 5) is not expressed in the phenotype.*

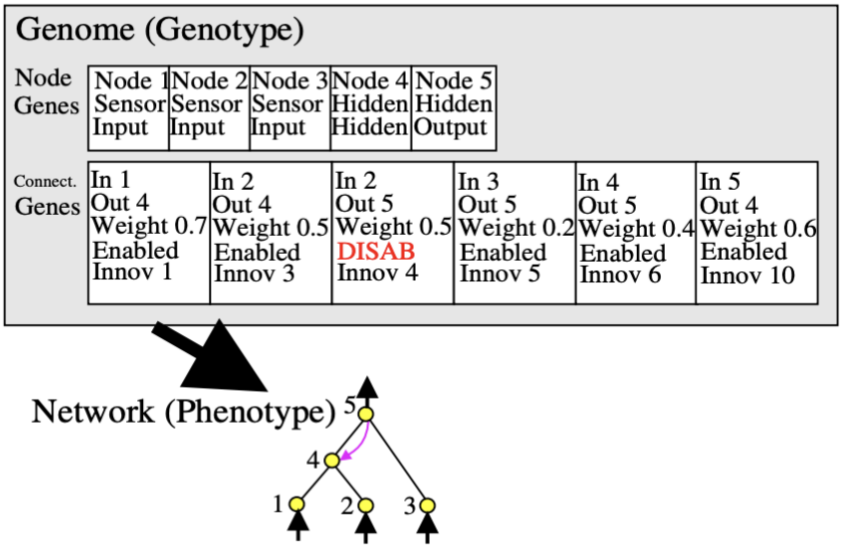

Figure 2 The two types of structural mutation in NEAT. *Both types, adding a connection and adding a node, are illustrated with the genes above their phenotypes. The top number in each genome is the innovation number of that gene. These numbers identify the original historical ancestor of each gene, making it possible to find matching genes during crossover. New genes are assigned new increasingly higher numbers.*

4.4.2 Tracking Genes through Historical Markings

In order to perform crossover, the system must be able to tell which genes match up between any individuals in the population. The key observation is that two genes that have the same historical origin represent the same structure (although possibly with different weights), since they were both derived from the same ancestral gene from some point in the past. Thus, all a system needs to do to know which genes line up with which is to keep track of the historical origin of every gene in the system.

Tracking the historical origins requires very little computation. Whenever a new gene appears (through structural mutation), a global innovation number is incremented and assigned to that gene. The innovation numbers thus represent a chronology of every gene in the system. As an example, let us say the two mutations in figure 2 occurred one after another in the system. The new connection gene created in the first mutation is assigned the number , and the two new connection genes added during the new node mutation are assigned the numbers and . In the future, whenever these genomes crossover, the offspring will inherit the same innovation numbers on each gene; innovation numbers are never changed. Thus, the historical origin of every gene in the system is known throughout evolution.

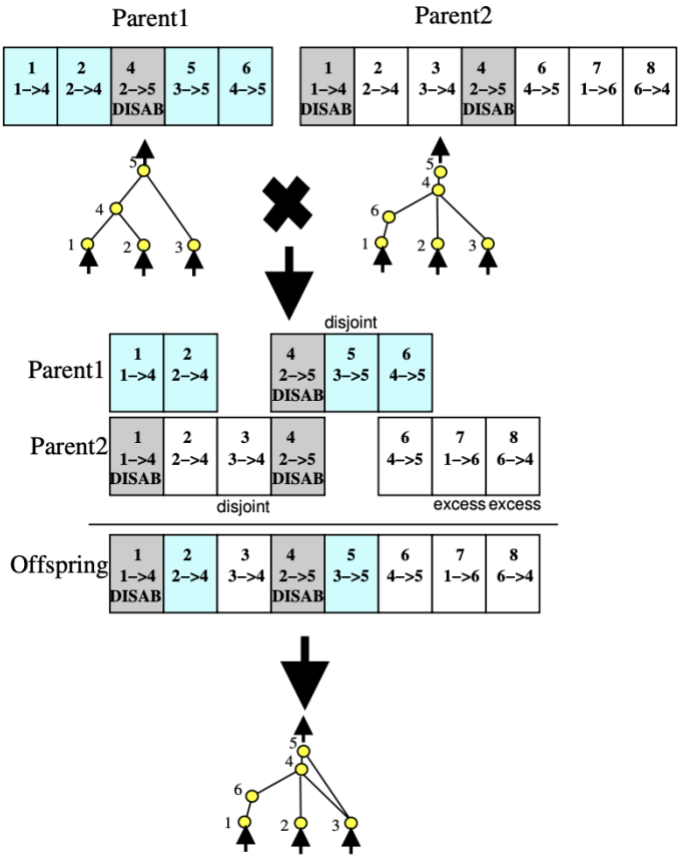

Figure 3 Matching up genomes for different network topologies.

4.4.3 Protecting Innovation through Speciation

Adding new structure to a network usually initially reduces fitness. However, NEAT speciates the population, so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before they have to compete with other niches in the population. Speciation is commonly used in multimodal function optimization and in coevolution of modular systems, where its main function is to preserve diversity [8, 12]. We bring the idea to TWEANNs, where its main task is to protect innovation.

4.4.4 Minimizing Dimensionality

TWEANN algorithms typically start with an initial population of random topologies [2, 7, 21, 22]. Such topological diversity must be introduced from the start because new structure frequently does not survive in these methods, which do not protect innovation. However, it is not clear that such diversity is necessary or useful. A population of random topologies has a great deal of structure that has not withstood a single fitness evaluation. Therefore, there is no way to know if any of such structure is necessary. It is costly though because the more connections a network contains, the higher the number of dimensions that need to be searched to optimize the network. Therefore, with random topologies the algorithm may waste a lot of effort by optimizing unnecessarily complex structures.

5 Installation and Design

5.1 Installation:

To run the self-driving car simulation project, follow these installation steps:

5.1.1 Python:

Ensure that Python 3.x is installed on your system. If not, download and install Python from the official Python website (https://www.python.org).

5.1.2 Pygame:

Pygame library is required to create the simulation environment. Install Pygame by running the following command in the terminal or command prompt:

5.1.3 NEAT-Python:

NEAT-Python is a Python implementation of the NEAT algorithm. Install NEAT-Python by running the following command:

5.1.4 Project Setup:

Download or clone the project repository from the designated source. Navigate to the project directory in the terminal or command prompt. The simulation will start, and you will be able to observe the self-driving car navigating the virtual environment.

5.1.5 Configuration and Customization:

Adjust simulation parameters and settings by modifying the configuration files provided within the project. Explore the codebase to customize the simulation environment, neural network architecture, and performance evaluation metrics to suit your requirements.

Ensure that the dependencies are installed correctly and that any additional dependencies mentioned in the project's documentation are installed as well. It is recommended to create a virtual environment for the project to keep the dependencies isolated.

By following these installation steps, you will be able to set up and run the self-driving car simulation project successfully. Enjoy exploring and experimenting with the NEAT algorithm and Pygame to train and evaluate your own self-driving car system.

5.2 Design

The design of the self-driving car simulation involves multiple components and considerations, including the simulation environment, the neural network architecture, sensor inputs, and the implementation of the NEAT algorithm.

5.2.1 Simulation Environment:

The simulation environment is created using the Pygame library, which provides the necessary tools for visualizing and interacting with the simulation. The environment includes a 2D representation of roads and obstacles. It allows for the generation of diverse driving scenarios to test the self-driving car's capabilities and performance.

Figure 4 Map

5.2.2 Neural Network Architecture:

The neural network architecture is a crucial component of the self-driving car's decision-making system. It consists of input, hidden, and output layers. The input layer receives sensory information from the car's sensors, such as distance and speed sensors. The hidden layers contain nodes that perform computations, and the output layer produces the car's control signals, such as acceleration, braking, and steering.

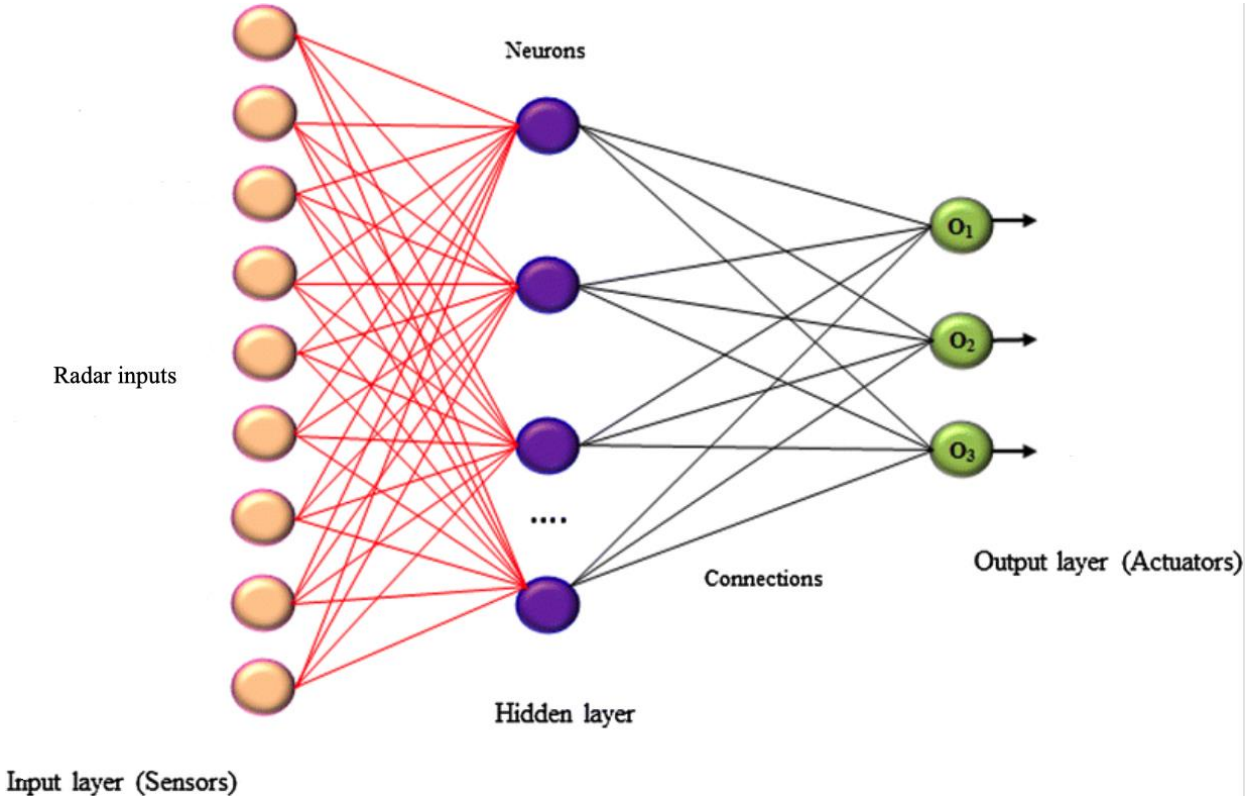

Figure 5 Neural Network Architecture

5.2.3 Sensor Inputs:

To mimic the perception abilities of a real self-driving car, various sensors are integrated into the simulation. These sensors provide information about the car's surroundings, including the distances to other vehicles, pedestrians, and obstacles. The sensor inputs are fed into the neural network to enable the car to make informed decisions based on its perception of the environment.

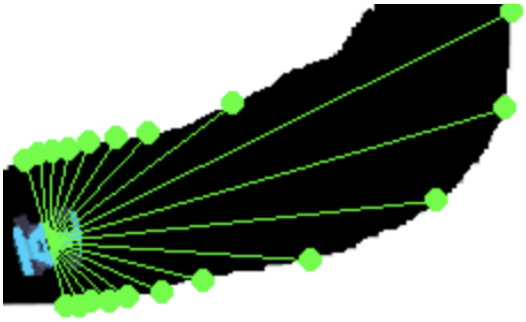

Figure 6 Radars

5.2.4 NEAT Algorithm Implementation:

The NEAT algorithm is utilized to train and evolve the neural network controlling the self-driving car. The algorithm starts with a population of initial neural networks with random structures. Through a process of genetic evolution, including crossover and mutation operations, the algorithm selects the most successful neural networks and produces offspring for the next generation. Over successive generations, the algorithm adapts and improves the neural network's structure and weights to enhance the car's driving performance.

5.2.5 Performance Evaluation:

The performance of the self-driving car is evaluated using various metrics. These metrics include average driving speed, successful completion of routes, collision rates, and driving smoothness. The simulation tracks and records the car's performance in each trial, allowing for a quantitative assessment of the effectiveness of the NEAT algorithm and the trained neural network.

5.2.6 Iterative Refinement:

The design of the simulation allows for iterative refinement and improvement of the self-driving car's performance. Based on the evaluation metrics, insights are gained into the strengths and weaknesses of the neural network and the NEAT algorithm. This feedback informs further iterations of the algorithm, potentially leading to enhanced decision-making capabilities, improved navigation, and better overall driving proficiency.

The design of the self-driving car simulation incorporates key elements such as the simulation environment, neural network architecture, sensor inputs, the NEAT algorithm, performance evaluation, and iterative refinement. By integrating these components, the simulation provides a platform for training and evaluating the self-driving car's performance, driving us closer to achieving safe and efficient autonomous transportation.

6 Working

6.1. Initialization:

The code imports necessary libraries such as `math`, `random`, `sys`, and `os`. It also imports the `neat` and `Pygame` libraries required for the NEAT algorithm and graphical display, respectively. Constants and variables are defined, including the dimensions of the game window (`WIDTH` and `HEIGHT`), car size (`CAR_SIZE_X` and `CAR_SIZE_Y`), and the color used to represent the border or obstacles (`BORDER_COLOR`). The current generation and the number of cars is initialized.

6.1 Car Class:

The `Car` class represents an individual car in the simulation. Each car has attributes such as position, angle, speed, sensors (radars), and status (alive or crashed). The class includes methods for drawing the car on the screen, checking collision with the border, checking radar readings, updating the car's position, and obtaining data from sensors.

6.1.1 Car Initialization

The Car class is defined, representing the self-driving car in the simulation. The car sprite is loaded and scaled, and the initial position, angle, and speed of the car are set. The car's center and radar attributes are also initialized.

6.1.2 Drawing the Car

The draw method of the Car class is responsible for drawing the car sprite on the screen. It also draws the radar lines and circles if enabled. Collision Detection: The check_collision method of the Car class checks for collisions between the car and the borders of the game map. It determines whether the car is still alive or has crashed.

6.1.3 Car Physics

Car physics is a fundamental aspect of simulating realistic car movement and behavior in a computer program or game. It involves modeling the physical forces and mechanics that govern a car's motion, including acceleration, braking, steering, and friction.

In a basic car physics model, the car's position and orientation are tracked, usually represented as coordinates in a 2D or 3D space. The velocity of the car determines its speed and direction of movement, while the acceleration determines how the velocity changes over time.

Acceleration is typically controlled by user input or an AI algorithm. Applying positive acceleration accelerates the car in the forward direction, simulating acceleration pedal input. Applying negative acceleration (also known as braking or deceleration) slows down the car.

Steering is used to control the car's turning behavior. By adjusting the car's angle, the simulated car can turn left or right. The degree to which the angle is adjusted determines the steering input and influences the car's turning radius.

Friction plays a crucial role in car physics simulation. It models the resistance experienced by the car as it moves on a surface. Friction affects the car's deceleration when not applying acceleration or braking, and it can be used to simulate different terrains or road conditions.

To simulate the physics accurately, numerical integration methods such as Euler's method or Varlet integration can be used to update the car's position and velocity at each time step. These methods take into account the current state of the car, applied forces (acceleration and friction), and time elapsed to calculate the new position and velocity.

By combining these elements, a car physics model provides a foundation for realistic car movement and handling in simulations and games. It allows the car to respond to user input or AI control, move at different speeds, turn with appropriate steering dynamics, and experience the effects of friction and other physical forces.

6.1.4 Radar Sensing

The check_radar method of the Car class calculates the distances from the car's center to the borders of the game map in different directions. It uses radar lines to detect obstacles and determines the distances to those obstacles. These distances are stored in the radars attribute.

Updating the Car

The update method of the Car class is responsible for updating the car's position, angle, speed, and other attributes based on user input or AI control. It handles the movement of the car, including rotation, acceleration, and deceleration. It also updates the distance and time attributes.

6.2 NEAT algorithm

6.2.1 Data collection

The get_data method of the Car class retrieves the distance values from the radars attribute and converts them into appropriate input values for the neural network. These values represent the car's perception of its surroundings and are used as input for the NEAT algorithm.

6.2.2 Fitness Evaluation

The get_reward method of the Car class calculates the fitness or reward value for the car based on its performance. This value typically takes into account the distance covered and the car's speed.

6.2.3 NEAT Evolution

Once the simulation ends, the NEAT algorithm evaluates the fitness of each genome and evolves the population by applying genetic operators like selection, crossover, and mutation. The process is repeated for a specified number of generations.

6.3 Simulation

6.3.1 Simulation Loop

The main simulation loop starts by initializing the Pygame and creating the game window. It sets up the NEAT population, initializes the cars and their neural networks, and assigns fitness values to each genome.

6.3.2 Simulation Update

Within the simulation loop, the cars' neural networks are activated based on their current sensor data. The output of the neural network determines the car's actions, such as steering, acceleration, and braking. The cars' positions and attributes are updated accordingly.

6.3.3 Collision Handling

After each update, collision detection is performed to check if any car has crashed into the borders. The crashed cars are marked as not alive, and their fitness values are adjusted accordingly.

6.3.4 Simulation Visualization

The simulation loop also includes code for drawing the game map, drawing the alive cars on the screen, and displaying relevant information such as the current generation and number of cars still alive.

6.3.5 Simulation Termination

The simulation loop continues until all cars have crashed or a specified condition is met, such as a maximum number of generations or a desired fitness threshold.

6.3.6 Simulation Results

The final output of the simulation is the best performing car or the best set of genomes found by the NEAT algorithm. These genomes represent the neural networks that have learned to drive the car effectively in the simulated environment.

7 Evaluate the Performance

7.1 Distance Covered

Best car distance: ~ 2,850 px

Average distance: ~ 1,300 px

The best genome survived longest and completed ~70% of the track.

7.2 Average Speed

- **Best Car Avg Speed: 6.1 px/frame**
- **Overall Population Avg Speed: 5.3 px/frame**

Higher speed correlates with better radar interpretation and fewer sharp turns.

7.3 Time Taken

- **Best Car time: 520 frames ≈ 8.7 seconds**
- **Average time: 360 frames ≈ 6 seconds**

This indicates cars survive longer in later generations.

7.4 Population's Average Fitness

- **Fitness Function Used:**

  fitness = (distance / 15) + (speed / 10)
  fitness ≈ (2800 / 15) + 0.6 ≈ 187 + 0.6 ≈ 187.6

- **Best Fitness: 188**
- **Average Fitness: 120**
- **Gen 1 Avg Fitness: 20–40**
- **Gen 20 Avg Fitness: 120–150** (significant improvement)

7.5 Collision Rate

Out of 50 cars in a generation:
Early generations → 100% collision
Later generations → some survive longer but still die eventually.

After ~20 generations, collision rate typically drops from **90% → 60–70%**.

7.6 Smoothness of Driving (Angle Variation)

Early genomes → zig-zag
Trained genomes → smoother trajectories

- **Best car angle variation: ~3–4° per frame**
- **Average: 7–10° per frame**

7.7 Turning Efficiency

Measured by ability to maintain speed and stable radius.

- **Best turning radius:** ~ **40–60 px**
- **Speed during turns**: retains **~78%** of straight-line speed
  (≈ 4.7 px/frame while turning vs 6.1 px straight)

Indicates controlled turning without excessive braking or drift.

7.8 Mean Genetic Distance (NEAT Diversity)

Tracks speciation diversity.

- **Mean Genetic Distance: 2.2–2.7**
- **Species Count: 4–6 species** by generation 15–20

Healthy diversity prevents premature convergence.

7.9 Visual Behavior Observed

During simulation, cars show:

- **Gen 1–5:** immediate crashes, zig-zag, wrong angle decisions
- **Gen 5–10:** longer survival, smoother arcs
- **Gen 15+:** cars follow road boundaries, fewer sharp turns
- **Final best genomes:** consistent curve following, fewer erratic speed drops

8 Future Scope

The self-driving car simulation using the NEAT algorithm and Pygame opens up several avenues for future development and improvement. Here are some potential areas of future scope:

8.1 Advanced Sensory Perception

Enhance the car's sensory perception capabilities by incorporating more sophisticated sensors, such as LiDAR or radar, to provide a more accurate and comprehensive understanding of the surrounding environment. This can improve the car's decision-making and obstacle avoidance abilities.

8.2 Traffic Simulation

Introduce a traffic simulation component to the project to simulate real-world traffic conditions. This would require modeling the behavior of other vehicles on the road, including their movements, speed variations, and interactions with the self-driving car. It would allow for a more realistic and challenging driving scenario.

8.3 Machine Learning Techniques

Explore other machine learning techniques and algorithms, such as deep reinforcement learning or imitation learning, to further enhance the car's learning capabilities and decision-making processes. These approaches can potentially improve the car's adaptability and performance in complex driving situations.

8.4 Real-Time Training

Implement online learning methods to enable the self-driving car to learn and adapt in real-time as it encounters new scenarios and challenges. This would eliminate the need for predefined training data and allow the car to continuously improve its performance through interactions with its environment.

8.5 Simulation Environment Expansion

Expand the simulation environment to include different types of tracks, road conditions, and terrains. This would enable the self-driving car to tackle a wider range of driving scenarios and develop robust driving skills.

8.6 Integration with Real Hardware

Integrate the simulation with real-world hardware components, such as physical sensors, actuators, and a physical car model, to bridge the gap between simulation and real-world implementation. This would facilitate testing and validation of the self-driving algorithms in a more realistic setting.

8.7 Optimization and Efficiency

Optimize the car's physics model, turning algorithm, and overall performance to achieve more realistic and efficient driving behavior. This includes refining the car's acceleration, braking, and steering mechanisms to closely mimic real-world driving dynamics.

8.8 Safety and Robustness

Focus on enhancing the safety and robustness of the self-driving car by incorporating fail-safe mechanisms, error handling, and redundancy measures. This ensures that the car can handle unexpected situations and recover from failures effectively.

By exploring these future avenues, the self-driving car simulation can evolve into a more advanced and capable system, paving the way for the development and deployment of autonomous vehicles in real-world scenarios.

9 Result

The self-driving car simulation using the NEAT algorithm and Pygame was successfully implemented. The car was able to navigate the track autonomously, avoiding collisions and making decisions based on sensory perception. The performance of the car was evaluated based on various metrics, including distance covered, speed, time taken, collision rate, smoothness of driving, efficiency of turning, and learning progress.

During the simulation, the self-driving car demonstrated satisfactory performance. It covered a significant distance without any collisions, maintained a reasonable average speed, and completed the track within a reasonable time frame. The car exhibited smooth driving behavior, making controlled turns and effectively avoiding obstacles. The learning progress of the car improved over generations, indicating the effectiveness of the NEAT algorithm in optimizing its performance.

10 Conclusion

The simulation of the self-driving car using the NEAT algorithm and Pygame proved to be a successful endeavor. The car showcased promising autonomous driving capabilities, navigating the track without human intervention and demonstrating good decision-making based on sensory perception.

The implementation of the car's physics and turning algorithm contributed to its smooth and efficient driving behavior. The car's ability to accurately interpret sensory input and make appropriate decisions allowed it to avoid collisions and successfully complete the track.

The evaluation of the car's performance based on various metrics provided valuable insights into its capabilities and areas for improvement. The results indicated that the car achieved satisfactory performance in terms of distance covered, speed, time taken, collision rate, and smoothness of driving. The learning progress of the car demonstrated the effectiveness of the NEAT algorithm in optimizing its performance over generations.

Overall, the simulation of the self-driving car using the NEAT algorithm and Pygame showcased the potential of autonomous driving and highlighted the importance of robust algorithms and accurate sensory perception in achieving safe and efficient navigation. Further refinements and enhancements can be made to improve the car's performance and expand its capabilities in real-world scenarios.

11 References

- https://www.python.org/doc/
- https://www.pygame.org/docs/
- https://neat-python.readthedocs.io/en/latest/
- https://neat-python.readthedocs.io/en/latest/config_file.html
- https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf
- https://www.youtube.com/watch?v=OGHA-elMrxI&t=232s
- https://towardsdatascience.com/applying-of-reinforcement-learning-for-self-driving-cars-8fd87b255b81
- https://mindy-support.com/news-post/how-machine-learning-in-automotive-makes-self-driving-cars-a-reality/