Development of the Intelligent Control System of an Unmanned Car

Yuri Kravchenko[1], Hennadii Dakhno[1], Olga Leshchenko[1], Andriy Dudnik[1,2], Andriy Miroshnyk[1]

[1] *Taras Shevchenko National University of Kyiv, Volodymyrs'ka str. 60, Kyiv, 01601, Ukraine*
[2]*Interregional Academy of Personnel Management, 2 Frometivska str., Kyiv, 03039, Ukraine*

Abstract

This paper discusses the development of an intelligent control system for an using machine learning. To achieve this goal, the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, implemented in the Python programming language, is used. NEAT allows for the evolutionary improvement of artificial neural networks, as well as the structure of these networks, with the goal of teaching a self-driving car to control itself in different conditions. The study's findings highlight the potential of NEAT to automate the operation of self-driving cars, ensuring their ability to adapt to and respond effectively to different driving situations.unmanned vehicle

Keywords

Neural networks, evolutionary learning, genetic algorithms, NEAT algorithm, NEAT learning efficiency, neural network architecture.

1. Introduction

The rapid development of information technology is quickly changing our lives and affecting all areas of human activity [1,2]. Information technologies allow automating various processes, increasing productivity, and reducing costs.

Such areas as networking, cloud technologies, Internet of Things technologies, and artificial intelligence are actively developing [3, 4].

The rapid development of technology has led to significant advances in the field of unmanned vehicles, both ground and airborne [5, 6]. One of the most promising areas in this field is the development of intelligent control systems for ground-based unmanned vehicles, also known as self-driving cars.

The development of an intelligent control system for self-driving cars is an endeavor that encompasses a variety of advanced software development technologies. It involves the combination of artificial intelligence, machine learning, computer vision, and advanced algorithms to create a system that allows the vehicle to perceive information from the outside and make decisions in real time [7]. By using a multitude of sensors, these systems collect and process huge amounts of data.

In this paper, we propose to create a simple system for intelligent control of an unmanned vehicle in the Python programming language using the NEAT (NeuroEvolution of Augmenting Topologies) algorithm. In the context of self-driving vehicles, a simple model system developed using NEAT in Python has great prospects. Such a system makes it possible to understand the fundamental concepts of intelligent driving.

[1]*14th International Scientific and Practical Conference from Programming UkrPROG'2024, May 14-15, 2024, Kyiv, Ukraine*
* Corresponding author.
† These authors contributed equally.
yurii.kravchenko@knu.ua (Y. Kravchenko); gennadiy.dakhno@gmail.com (H. Dakhno); olga.leshchenko@knu.ua (O. Leshchenko); andrii.dudnik@knu.ua (A. Dudnik); andriimiroshnyk@knu.ua (A. Miroshnyk)
0000-0002-0281-4396 (Y. Kravchenko); 0000-0002-3997-2785 (O. Leshchenko); 0000-0003-1339-7820 (A. Dudnik)

2. Main part

Intelligent driverless vehicle control systems use advanced Machine Learning algorithms [8]. Through a process known as Deep Learning, these algorithms analyze large data sets to discover patterns, allowing the vehicle to improve its decision-making capabilities over time. This iterative learning process enables the system to adapt to new scenarios, optimize its performance, and ultimately improve the safety and reliability of automated vehicles.

The NEAT (NeuroEvolution of Augmenting Topologies) technology was used in this work. NEAT is an evolutionary algorithm that creates artificial neural networks. It combines neural networks and genetic algorithms to create intelligent control systems.

2.1. Algorithm description

It is necessary to describe the genetic coding algorithm used in NEAT. NEAT's genetic coding strategy is designed to facilitate gene alignment during mating when two genomes overlap. Genomes serve as linear representations of network connections, and each genome contains a list of connecting genes that refer to pairs of genes of connected nodes [9].

In NEAT, mutations can affect both the weights of connections and the structure of the network. The weights of connections change, as in other neuroevolutionary systems. At the same time, each connection is potentially broken or remains constant in each generation [10].

Each mutation leads to the expansion of the genome by introducing new genes. In the "linkage addition" mutation, a new linkage gene with a random weight is added. In the "node addition" mutation, the existing linkage is split and a new node is placed. The linkage leading to the new node is assigned a weight of 1. This approach to adding nodes was chosen to minimize the direct impact of mutation [11]. Although the new nonlinearity in the connection slightly changes the function, new nodes can be quickly integrated.

As a result of mutations, genomes in NEAT gradually increase in size, resulting in genomes of different lengths. Allowing genomes to grow unlimitedly inevitably leads to a more complex form of the "controlling convention problem" with different topologies and combinations of weights [12].

In the course of evolution, there is information that accurately reveals the correspondence of genes between individuals. This information is the historical origin of each gene. Genes with the same historical origin necessarily represent the same structural component, although potentially with different weights. Therefore, all that the system needs to establish genetic alignments is to store records of the historical origin of each gene in the system.

Keeping track of these historical journals requires minimal computing resources. Each time a new gene appears, the innovation global number is incremented and assigned to that gene. These innovation numbers essentially reflect the appearance of each gene in the system [13].

These historical markers provide NEAT with powerful capabilities [14]. Now the system has an accurate knowledge of which genes correspond to each other. During the crossing process, genes from both genomes with matching innovation numbers are paired and called matching genes. Genes that do not match are classified as redundant. These redundant genes represent structural components that are not present in the rest of the genome [15]. In the generation of offspring, genes are randomly selected from either parent, while all redundant genes are included sequentially from the parent with the best fit. Therefore, historical markers allow NEAT to perform crosses using linear genomes.

By introducing new genes into the population and combining genomes with different structures, the system can create a population with a diverse topology. However, it is obvious that this population alone is not enough to preserve topological innovation. Smaller structures tend to be optimized faster than larger ones, and adding nodes and connections usually initially reduces the fitness of the network. The newly added structures have a limited probability of surviving longer than one generation, even though the innovations they represent may be vital to solving problems [16].

The degree of compatibility between a pair of genomes is naturally determined by the number of redundant genes. The greater the divergence between two genomes, the less evolutionary history they share. Therefore, in NEAT, we can quantify the compatibility distance, denoted as δ, for different structures through a direct linear combination of several factors, including the number of redundant genes (E), the number of non-overlapping genes (D), and the average weight differences between the corresponding genes (W), which includes even non-functional genes. The formula for calculating the compatibility distance is as follows:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}.$$

The coefficients c_1, c_2, c_3 provide a means of adjusting the relative importance of the three factors in calculating the compatibility distance. The Factor N, which represents the number of genes in the largest genome, is used to normalize the genome size. The threshold of compatibility, called δ_t, is used in conjunction with the distance measure δ, to determine how genomes should be classified into species. An ordered list of species is maintained to organize the population into species.

Each existing species is characterized by a randomly selected genome of the next generation, which serves as a representative genome. To assign a given gene, called g, from the current generation to a species, it is placed in the first species, for which g is considered compatible with the representative genome of that species. This method ensures that the species do not overlap and do not share any common members [17]. If the genome g is incompatible with any of the existing species, a new species is created, where g is a representative genome.

The NEAT reproduction mechanism uses explicit fitness sharing, which means that organisms belonging to the same species must collectively share the fitness of their niche. This arrangement ensures that a species cannot grow excessively, even if a significant number of its members are doing very well. Therefore, it is unlikely that one species will dominate the entire population. To calculate the adjusted suitability, denoted as f_i', for a given organism i its distance δ from all other organisms j in the population is taken into account:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh\left(\delta\left(i,j\right)\right)}.$$

Sharing function sh is set to 0 when the distance $\delta(i,j)$ above the threshold value δ_t, under the other condition $sh(\delta(i,j))$ will be equal to 1. Thus, $\sum_{j=1}^{n} sh(\delta(i,j))$ reduced to the number of organisms of the same species as organism i. This decrease is natural, as species are already clustered by compatibility with the threshold δ_t. Each species is assigned a potentially different number of offspring. Then the species are multiplied, first excluding the members with the lowest efficiency from the population [18]. Then the entire population is replaced by the offspring of the remaining organisms in each species.

2.2. Comparison of Tesla's AI Algorithms and the NEAT AI Algorithm

Tesla's autonomous driving technology leverages deep learning, particularly convolutional neural networks (CNNs), to process sensory data from a suite of cameras, radar, and ultrasonic sensors. The Full Self-Driving (FSD) computer, a custom-built hardware platform, manages these complex neural networks, enabling real-time decision-making essential for safe and efficient driving.

The architecture of Tesla's CNNs includes several key components. The input layer processes high-definition images from the vehicle's cameras, capturing a 360-degree view of the environment. Convolutional layers apply filters to these images to extract features such as edges, textures, and patterns, which are essential for identifying objects, lane markings, and other critical elements of the driving environment. Pooling layers follow, reducing the spatial dimensions of the feature maps to streamline computation and enhance the network's efficiency [19].

Fully connected layers aggregate these features, transforming them into predictions about the driving environment, such as object classifications, lane positions, and traffic sign identifications. The final output layer provides these predictions, guiding the vehicle's decisions.

Tesla trains its CNNs using supervised learning techniques. The training process involves adjusting the network's weights through backpropagation and gradient descent algorithms to minimize the error between predicted outputs and true labels. This process relies on vast datasets collected from Tesla's fleet, which continuously gather data from real-world driving scenarios. The ability to deploy over-the-air updates allows Tesla to improve its models continually, ensuring that the system benefits from the latest advancements in neural network research and real-world data.

Applications of Tesla's CNNs in autonomous driving include object detection, where the system identifies vehicles, pedestrians, cyclists, and road obstacles; lane detection, which involves recognizing and tracking lane markers; and traffic sign recognition, crucial for obeying traffic rules and making informed driving decisions. The strengths of Tesla's approach lie in its high accuracy and scalability, made possible by robust feature extraction capabilities and continuous data-driven improvements. However, these systems require large labeled datasets and significant computational resources, which can be a limitation.

The NEAT (Neuro Evolution of Augmenting Topologies) algorithm, developed by Kenneth Stanley, represents a fundamentally different approach to neural network design and training. Unlike traditional methods that rely on fixed architectures, NEAT evolves both the structure and weights of neural networks using genetic algorithms. This evolutionary process starts with simple networks and progressively complexifies them by adding nodes and connections.

NEAT's training methodology revolves around reinforcement learning, where the performance of neural networks is evaluated based on a fitness function. The algorithm maintains a population of neural networks and evolves them over generations. The best-performing networks are selected to create offspring, incorporating mutations and crossovers to introduce new structures and improve performance. This iterative process continues until an optimal solution is found.

The architecture of NEAT begins with an initial population of simple networks with minimal nodes and connections. Through the processes of mutation (adding new nodes and connections) and crossover (combining parts of different networks), NEAT evolves the networks' topologies. A key feature of NEAT is speciation, which groups networks into species to protect innovation and ensure diverse solutions. This approach allows NEAT to adapt the network structure dynamically to the complexity of the task at hand.

NEAT has been applied successfully in various domains, such as game playing and robotics. For example, it has been used to evolve strategies for playing games like Pac-Man and to develop control strategies for robotic systems. The strengths of NEAT lie in its ability to adapt topologies and explore novel architectures, making it particularly useful for tasks where the optimal network structure is not known in advance. However, NEAT's evolutionary process can be computationally intensive, and its stochastic nature can lead to variable performance.

Tesla's CNNs and the NEAT algorithm represent two distinct paradigms in neural network training and architecture design. Tesla's approach utilizes supervised learning with labeled datasets, employing fixed architectures defined by convolutional, pooling, and fully connected layers. This method is highly effective for tasks with clear hierarchical structures, such as image recognition in autonomous driving. The training process, while requiring significant computational resources, is robust and scalable, benefiting from continuous data collection and over-the-air updates.

In contrast, NEAT employs an evolutionary algorithm to dynamically evolve both the structure and weights of neural networks. This method is particularly advantageous in reinforcement learning scenarios where the optimal network topology is not predefined. NEAT's ability to adapt and explore diverse network architectures allows it to find novel solutions that might be missed by traditional fixed-architecture approaches. However, the computational demands of evolving large populations and the inherent variability of the evolutionary process can be challenging.

2.3. Description and development of the system

For the development, we used the Python programming language together with the PyGame visualization package. The Python programming language is best suited for the development of neural networks, and the PyGame package is suitable for visualizing the system's operation [20].

For the successful implementation of the project, it is necessary to choose the optimal configuration of the NEAT algorithm (Figure 1).

config.txt

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 100000000
pop_size              = 30
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.01
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.01
aggregation_options     = sum
```

Figure 1: NEAT algorithm configuration.

The NEAT section defines parameters specific to the system:

- fitness_criterion: used to calculate the completion criterion based on the genome fitness set.
- fitness_threshold: when the value of the fitness function calculated with fitness_criterion, reaches the threshold specified in the code, the evolution process is completed.
- pop_size: number of individuals in each generation.
- reset_on_extinction: this parameter is set to True. When all species die out simultaneously due to stagnation, a new random population will be created.

The DefaultGenome section defines the parameters for the built-in DefaultGenome class. This section is necessary for the implementation of the genome when creating an instance of the Config:

- activation_default: attribute of the default activation function assigned to new nodes.
- activation_mutate_rate: the probability that a mutation will replace the node's activation function with a randomly determined activation_options parameter.
- activation_options: activation functions that can be used by nodes.
- aggregation_default: attribute of the default aggregation function assigned to new nodes.
- aggregation_mutate_rate: the probability that a mutation will replace the node's aggregation function with a randomly determined aggregation_options parameter.
- aggregation_options: aggregation functions that can be used by nodes. New aggregation functions can be defined in the same way as new activation functions.

A description of the car driving on the race track is shown in Figure 2. The program code describes:

- Uploading a car image, setting car dimensions.
- Determining the starting position and speed of the car.
- Create a list where data from the car sensors will be recorded.
- Setting the initial parameter to check if the car has not crashed.
- Setting the initial parameters of the distance and time traveled.

```
class Car:

    def __init__(self):
        # Load Car Sprite and Rotate
        self.sprite = pygame.image.load('car.png').convert()
        self.sprite = pygame.transform.scale(self.sprite, (CAR_SIZE_X, CAR_SIZE_Y))
        self.rotated_sprite = self.sprite

        self.position = [830, 920] # Starting Position
        self.angle = 0
        self.speed = 0

        self.speed_set = False # Flag For Default Speed Later on

        self.center = [self.position[0] + CAR_SIZE_X / 2, self.position[1] + CAR_SIZE_Y / 2] # Calculate Center

        self.radars = [] # List For Sensors / Radars
        self.drawing_radars = [] # Radars To Be Drawn

        self.alive = True # Boolean To Check If Car is Crashed

        self.distance = 0 # Distance Driven
        self.time = 0 # Time Passed
```

Figure 2: Description of the race car.

After that, the system aims to autonomously navigate race tracks of varying complexity by evolving the neural network architecture through an iterative evolutionary process [21]. Our model is a neural network with five input neurons - sensors - and four output neurons - actions it can perform between them (Figure 3).

Figure 3: Input and output neurons.

Next, we added some hidden layers with additional neurons. These layers increase the complexity and sophistication of our model, but they also increase the training time and the likelihood of neurons "overfitting" (Figure 4).

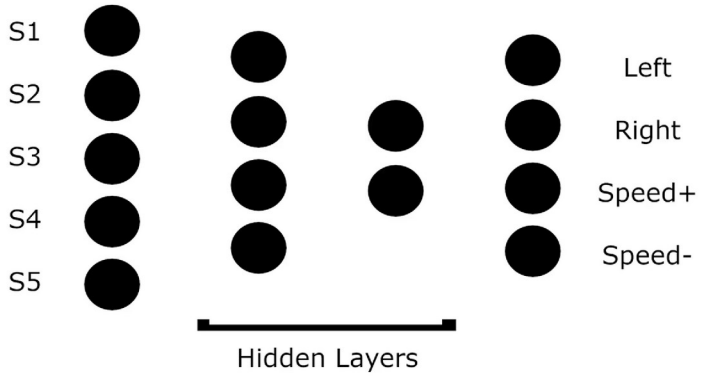

Figure 4: Hidden layers.

All neurons are interconnected, the connections between neurons have a certain weight depending on all those values to which the model will react in a certain way based on the input data (Figure 5).

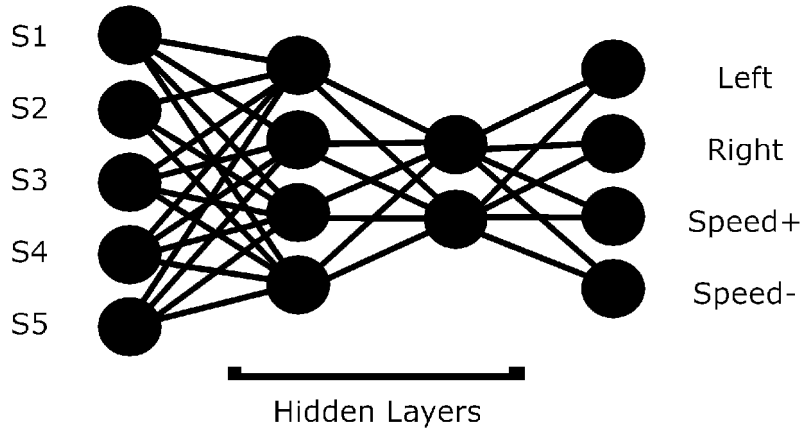

Figure 5: Connections between neurons.

Initially, sensor values and actions are completely random, but over time, cars learn to perform actions more rationally and efficiently. However, for each action that cars perform, they will receive a reward or a penalty, and this is realized by using a so-called fitness function. In our simple model, the fitness function of a car increases depending on the distance it travels without accidents. After each generation, the cars improve - the cars with the highest fitness function are likely to survive and reproduce, while cars that do not perform as well will disappear after a while. When a car is reproduced, it won't simply copy its parent's properties. It'll be similar but not identical, potentially

improving performance on the track and avoiding accidents. Thus, cars that are very similar to each other form their own species. If a species does not improve within a fixed number of generations, it becomes extinct. Taking all these principles into account, an environment was created in which the best cars survive and reproduce, while the worst disappear. The basic principle is that what works is likely to survive and be reproduced.

2.4. Testing the system

First, the system was tested for a simple oval track. The simulation start function is shown in Figure 6:

```
def run_simulation(genomes, config):

    # Empty Collections For Nets and Cars
    nets = []
    cars = []

    # Initialize PyGame And The Display
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT), pygame.FULLSCREEN)

    # For All Genomes Passed Create A New Neural Network
    for i, g in genomes:
        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        cars.append(Car())

    # Clock Settings
    # Font Settings & Loading Map
    clock = pygame.time.Clock()
    generation_font = pygame.font.SysFont("Arial", 48)
    alive_font = pygame.font.SysFont("Arial", 36)
    time_font = pygame.font.SysFont("Arial", 36)
    game_map = pygame.image.load('map2.png').convert()
```

Figure 6: The function of starting a simulation.

The self-driving car successfully overcomes the track during the first few generations. With each new generation, the car increases its speed and completes the track faster. After the fourth generation, there are several cars that successfully pass the track. Further, with the following generations, the number of cars and their speeds are constantly increasing.

The implementation of creating populations and running a simulation with 1000 generations is shown in Figure 7.

```
if __name__ == "__main__":

    # Load Config
    config_path = "./config.txt"
    config = neat.config.Config(neat.DefaultGenome,
                                neat.DefaultReproduction,
                                neat.DefaultSpeciesSet,
                                neat.DefaultStagnation,
                                config_path)

    # Create Population And Add Reporters
    population = neat.Population(config)
    population.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    population.add_reporter(stats)

    # Run Simulation For A Maximum of 1000 Generations
    population.run(run_simulation, 1000)
```

Figure 7: Code snippet for creating populations and starting a simulation.

We can also see how the cars were divided into two groups, forming two types (Figures 8 – 10).

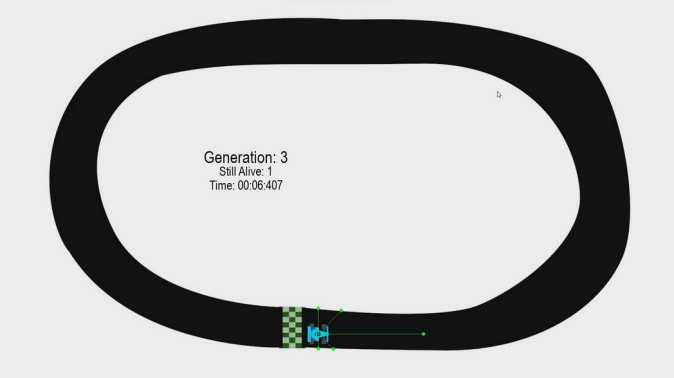

Figure 8: Third generation on a simple oval track.

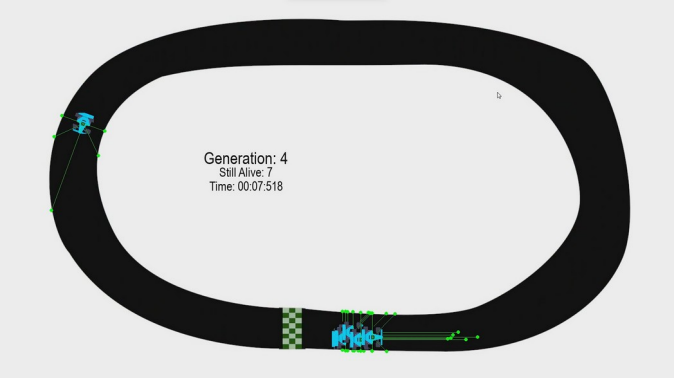

Figure 9: Fourth generation on a simple oval track

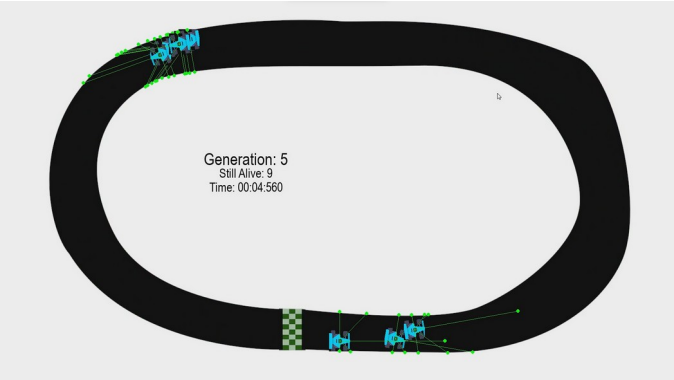

Figure 10: The fifth generation on a simple oval track.

Then the system was tested on a more complex track with many turns. Until the fourth generation, the cars could not drive through the first turn. But the model gradually develops, trains, and therefore, after the fourth generation, it can easily overcome this turn. Then there are problems with other turns. With each subsequent generation, the model improves and covers a greater distance of the track. More and more cars successfully complete the track before crashing. And finally, after twenty-four generations, one car successfully completes the track without an accident. With each new generation, there are more cars and they gradually increase their speed to drive faster around the track. In this example, we can clearly see how the NEAT neural network learns and develops (Figures 11-13).

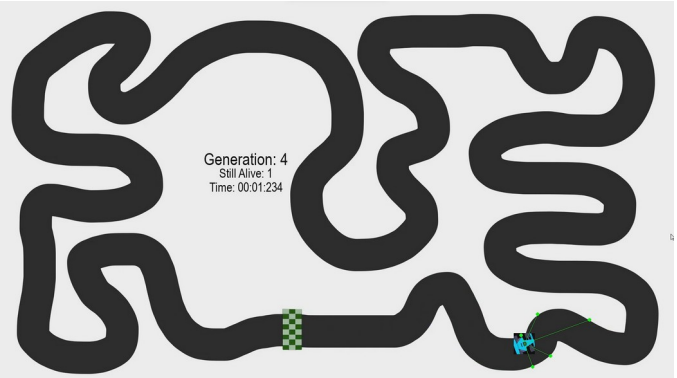

Figure 11: Fourth generation on a challenging track.

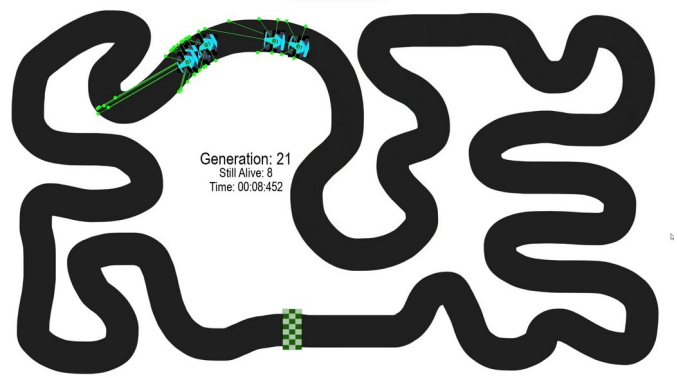

Figure 12: Twenty-first generation on a challenging track.

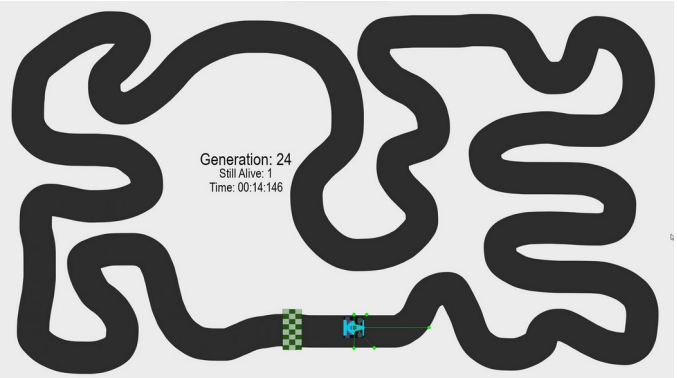

Figure 13: The twenty-fourth generation on a difficult track.

Testing the system on different tracks confirmed the system's functionality.

3. Conclusions

Thanks to the integration of NEAT, Python, and Pygame, a system capable of autonomously navigating race tracks of varying complexity was successfully created.

Throughout the development process, the capabilities of NEAT, a neuro-evolutionary algorithm, were used to develop a neural network architecture that demonstrates intelligent behavior. NEAT

facilitated the automatic generation and modification of neural network structures, allowing the system to adapt and learn based on interaction with the environment.

Python, with its simplicity and large library ecosystem, was the ideal programming language for developing an intelligent control system. Pygame, in particular, provided us with the necessary tools for graphics rendering, collision detection, and time control, which allowed us to create a visually interactive environment for the system.

Iterative cycles of training and evaluation, as well as system refinement and optimization, have played a crucial role in improving the system's performance and adaptability. Thanks to several generations of evolution and continuous improvement, neural networks have learned to navigate race tracks, maintain a high average speed, avoid collisions, and adapt to various racing challenges.

This project demonstrated the potential of NEAT and its integration with Python and Pygame in the development of intelligent control systems for self-driving cars. The developed system serves as a basis for further development of self-driving cars, as it can be extended to handle more complex scenarios and integrate with real-world hardware.

In general, the successful development of an intelligent control system for an unmanned vehicle based on NEAT demonstrates the effectiveness of evolutionary algorithms in solving complex control problems. It opens the door for future research and development in the field of unmanned driving, paving the way for safer and more efficient transportation systems.

References

[1] Iatsyshyn, A., Iatsyshyn, A., Kovach, V., Zinovieva, I., Artemchuk, V., Popov, O., Turevych, A. (2020). Application of open and specialized geoinformation systems for computer modelling studying by students and PhD students. Paper presented at the CEUR Workshop Proceedings, , 2732 893-908.

[2] Hubanova, T., Shchokin, R., Hubanov, O., Antonov, V., Slobodianiuk, P., & Podolyaka, S. (2021). Information technologies in improving crime prevention mechanisms in the border regions of southern Ukraine. Journal of Information Technology Management, 13, 75-90. doi:10.22059/JITM.2021.80738

[3] A. Dudnik, Y. Kravchenko, O. Trush, O. Leshchenko, N. Dakhno and V. Rakytskyi, "Study of the Features of Ensuring Quality Indicators in Multiservice Networks of the Wi-Fi Standard," 2021 IEEE 3rd International Conference on Advanced Trends in Information Theory (ATIT), Kyiv, Ukraine, 2021, pp. 93-98, doi: 10.1109/ATIT54053.2021.9678691.

[4] A. Dudnik, Y. Kravchenko, O. Trush, O. Leshchenko, N. Dahno and Y. Ryabokin, "Routing Method in Wireless IoT Sensor Networks," 2022 IEEE 3rd International Conference on System Analysis & Intelligent Computing (SAIC), Kyiv, Ukraine, 2022, pp. 1-6, doi:10.1109/SAIC57818.2022.9922998.

[5] N. Dakhno, Y. Kravchenko, L. Lunin, O. Pliushch, O. Trush and H. Shevchenko, "Analysis of VANET Connection Quality in Dynamic Environment," 2022 IEEE 4th International Conference on Advanced Trends in Information Theory (ATIT), Kyiv, Ukraine, 2022, pp. 255-258, doi: 10.1109/ATIT58178.2022.10024203.

[6] N. Dakhno, O. Barabash, H. Shevchenko, O. Leshchenko and A. Dudnik, "Integro-differential Models with a K-symmetric Operator for Controlling Unmanned Aerial Vehicles Using a Improved Gradient Method," 2021 IEEE 6th International Conference on Actual Problems of Unmanned Aerial Vehicles Development (APUAVD), Kyiv, Ukraine, 2021, pp. 61-65, doi: 10.1109/APUAVD53804.2021.9615431.

[7] S. Bhaggiaraj, M. Priyadharsini, K. Karuppasamy and R. Snegha, "Deep Learning Based Self Driving Cars Using Computer Vision," 2023 International Conference on Networking and Communications (ICNWC), Chennai, India, 2023, pp. 1-9, doi: 10.1109/ICNWC57852.2023.10127448.

[8] S. Bhaggiaraj, M. Priyadharsini, K. Karuppasamy and R. Snegha, "Deep Learning Based Self Driving Cars Using Computer Vision," 2023 International Conference on Networking and

Communications (ICNWC), Chennai, India, 2023, pp. 1-9, doi: 10.1109/ICNWC57852.2023.10127448.
[9] K. O. Stanley, Efficient Evolution of Neural Networks through Complexification, 2004, URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e158baaec08a54313c74ea0e1af1b72a6863406c.
[10] K. O. Stanley, R. Miikkulainen, Evolving Neural Networks through Augmenting Topologies, 2002. URL: https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf.
[11] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. "Evolving Neural Network Agents in the NERO Video Game". In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05). Piscataway, NJ: IEEE.
[12] K. O. Stanley, R. Miikkulainen. "Efficient reinforcement learning through evolving neural network topologies". In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO2002). San Francisco: Kaufmann.
[13] S. Whiteson, K. O. Stanley, and R. Miikkulainen. "Automatic feature selection in neuroevolution". In GECCO 2004: Proceedings of the Genetic and Evolutionary Computation Conference Workshop on Self-Organization, July 2004.
[14] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl. "Automatic Feature Selection in Neuroevolution". In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 05). pp. 1225-1232, Washington, D.C., June 2005.
[15] X. Yao. Evolving artificial neural networks. Proceedings of the IEEE, 87(9):1423–1447, 1999.
[16] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. "Evolving adaptive neural networks with and without adaptive synapses". In Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC-2003). Canberra, Australia: IEEE Press, 2003.
[17] K. O. Stanley, and R. Miikkulainen. "Competitive coevolution through evolutionary complexification". Journal of Artificial Intelligence Research 21: 63-100, 2004.
[18] K. O. Stanley, and R. Miikkulainen. "Continual coevolution through complexification". In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002). San Francisco, CA: Morgan Kaufmann, 2002.
[19] Tesla AI Day 2021. Full Self-Driving Presentation. URL: https://www.youtube.com/watch?v=j0z4FweCy4M.
[20] NEAT-Python 0.92 documentation. Welcome to NEAT-Python documentation! – NEAT-Python 0.92 documentation, 2022. URL: https://neat-python.readthedocs.io/en/latest/neat_overview.html.
[21] M. E. Timin. The robot auto racing simulator, 1995. URL: http://rars.sourceforge.net.