Zone-Sensitive Self Driving Bus

Anna Granberg[1], Erik Dahlström[2], Filip Hamrelius [3]

Abstract

This project presents the development of a self-driving bus capable of navigating a track with speed-sensitive zones using the NeuroEvolution of Augmenting Topologies (NEAT) library within an environment developed with Python and the library PyGame. The NEAT algorithm is driven by a fitness function to evaluate and reproduce effective generations. This project employs a fitness function similar to those used in Q-learning approaches, but with NEAT enabling the dynamic evolution of both the structure and weights of neural networks, allowing the bus to adapt to complex environmental conditions. The bus is trained to obey speed regulations in school zones, avoid collisions, and pass through checkpoints. The implementation focuses on optimizing the reward function to balance various goals such as speed, distance travelled, and safety. This work demonstrates the potential of NEAT for autonomous driving systems, though challenges remain, including the balancing of the reward function and inefficiencies in training. In conclusion, the implementation successfully solves the task, with the bus navigating the track and slowing down in school zones. However, a drawback is that the bus also drives slowly on non-school-straight-zone sections of the track.

Source code: https://github.com/hamreliusfilip/Self$_{D}riving_{B}us$

Video: https://youtu.be/9WJq2jezN80

Authors
[1] *Media Technology Student at Linköping University, anngr950@student.liu.se*
[2] *Media Technology Student at Linköping University, erida600@student.liu.se*
[3] *Media Technology Student at Linköping University, filha243@student.liu.se*

Keywords: Neural Network — Neuro Evolution– Fitness – Rewards — NEAT

Contents

1. Introduction

Methods for developing simple self-navigating agents in simulations or games have seen significant advancements in recent years. Machine learning libraries have gained considerable traction and are becoming increasingly accessible. Implementing these methods using a forward-driving vehicle is a straightforward and common approach. These methods offer substantial flexibility, providing a wide range of possibilities for what can be achieved.

The most common solution for this problem is to use some kind of reinforcement learning, often opting for Q-learning or Deep Q-learning. Since this has been done a lot of times for similar environments as this one, this project has opted for a different solution - this project has also tried to make the environment more advanced than simply driving around a track as fast as possible. This report will discuss the implementation and workflow of a self driving agent in the form of a bus, capable of navigating a simple track with specific speed zones.

2. Theory

This project is based around the Python library: NeuroEvolution of Augmenting Topologies, also called NEAT. An implementation with NEAT falls under the broader category of genetic algorithms, specifically focusing on evolving neural networks. In short an algorithm that evolves both the structure and weights of neural networks by encoding them as genomes, which are subject to crossover, mutation, and selection based on their fitness in solving a given task [1].

To evolve a solution to a problem, the user provides a fitness function that assigns a score to each genome, reflecting its ability to solve the problem. Higher scores indicate better performance. The algorithm runs for a specified number of generations, where each generation is created by reproducing and mutating the fittest individuals from the previous generation. Through reproduction and mutation, genomes can become more complex by adding nodes and connections. The algorithm stops either when the set number of generations is reached, or when a genome exceeds a specified fitness threshold. NEAT begins with simple networks, typically without any hidden layers or with a minimal predefined structure. During the training process, NEAT can dynamically add hidden layers and nodes as needed. This allows it to evolve the network topology, making it more complex over generations, as the task demands [2].

The only influence the implementation has over the specifics of the neural network is based on a configuration file. The file can be specified with a number of predetermined parameters for the network to base its implementation around. There are a lot of potential for optimizing and changing the implementation with the configuration files. This implementation focused on a few parameters: population size, number of inputs, number of outputs, number of hidden layers, initial connections and amount of genomes brought between generations [2].

The fitness model has to be implemented from scratch depending on the environment and task at hand. For this project the fitness function is based on rewards that depend on how the bus interacts with the environment. Fundamentally, the bus receives negative rewards for crashing into a wall, positive rewards relative to the distance travelled, and positive rewards for passing a checkpoint. The bus also interacts with different zones in the environment, such as a slow zone: a school zone. In these zones, the bus is expected to drive slowly and is penalized for driving too fast while within the zone. The inspiration for this kind of rewards structure comes from Q-learning within the Reinforcement Learning category of Machine Learning. In these cases is very common to use basic inputs for a reward structure [2] [3].

3. Method

The following chapter will present the project methodology and implementation of the self-driving bus and its corresponding neural-network, defining what actions to perform, how the reward function was defined and how it was trained.

3.1 NEAT

A central part of NEAT is how it structures and manages the networks, which can be divided into three main components: input layer, hidden layer, and output layer.

3.1.1 Input Layer

The input layer is the first layer in the neural network. It serves as a receiver for the information that the network needs to make its calculations and decisions [4]. In our project, we have defined a specific number of inputs, which represent various sensor readings or states that the bus needs to perceive. These include:

- 5 sensors with distance vectors from the car to the track boundaries.
- A Boolean variable to track whether the bus is in a school zone or not.

Each neuron in the input layer is connected to a specific sensor or parameter and sends these values on to the hidden layer for further processing.

3.1.2 Hidden Layer

NEAT allows for hidden layers to grow and evolve over time, providing great flexibility and adaptability. Hidden layers can contain a varying number of neurons, and each neuron in this layer can receive signals from both the input layer and other neurons in the hidden layer [5]. These neurons use activation functions, such as *sigmoid* and *tanh*, to process the input signals and create new signals that represent more abstract and complex patterns. Hidden layers are crucial for capturing the non-linear relationships in the data, enabling the bus to learn more advanced behaviours over time.

3.1.3 Output Layer

The output layer is the last layer in the neural network and is responsible for producing the actions that the bus should perform based on the information processed in the input and hidden layers. Each neuron in the output layer corresponds to a specific action or decision, and they are:

- Speed (Accelerate or brake).
- Steering (Turn left or right).

The outputs from these neurons are interpreted and used to control the bus movements in the simulation. The outputs can be either continuous values like speed or discrete decisions such as which direction the bus should turn. All of this is illustrated in the following Figure 1

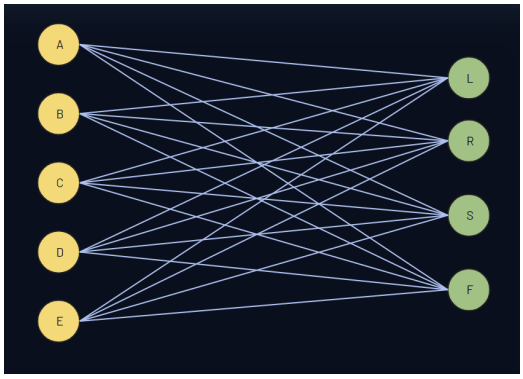

Figure 1. Input- and output-layer.

3.2 Environment

The environment that the bus interacts with is constructed in Python with the library PyGame. A common library that comes with all tools needed to handle graphics, game loops, interactions and interactive applications in general.

The track that the bus drives on is a simple PNG created in a drawing program. It consists of simple turns, left and right, in order for the bus to navigate around the circuit. The bus is never allowed to cross the boundaries of the track otherwise the bus will reset at the start/finish line. In order to make sure that the bus knows where the in- and outside lines are at all time from the png track, the program gathers this information from using the function *Canny*. The Canny algorithm is used as an image processing technique for detecting edges in an image. It operates in several stages to identify the points where there is a significant change in the intensity of pixels, which correspond to the tracks boundaries in this case. These represent the key boundaries that define where the vehicle can or cannot travel on the track, allowing further steps like collision detection. The end result from this algorithm can be shown in Figure 2.

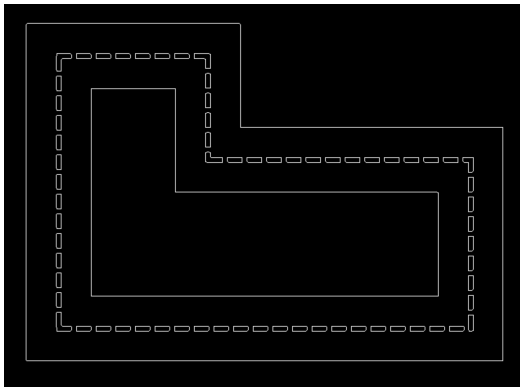

Figure 2. Resulting image of track after using Canny algorithm.

After this we created the tracks *checkpoints* and *school zones*, which play a critical role in shaping the simulation's dynamics. The checkpoints are placed manually at specific coordinates. These don't directly interact with the neural network, but they serve an important purpose. They influence the fitness of each generation as we evolve the system. On the other hand, school zones are more sophisticated. They are also manually placed, but they do interact directly with the neural network, creating an environment where the bus must adapt its behaviour, ensuring it recognizes and responds to these zones. I.e the variable for school zone is an input to the neural network. The checkpoints can be seen i Figure 3. The school zones do not have a graphical representation, they are instead defined with an array to assist the checkpoints, meaning each checkpoint has a boolean value of 0 or 1, school zone after the checkpoint or not.

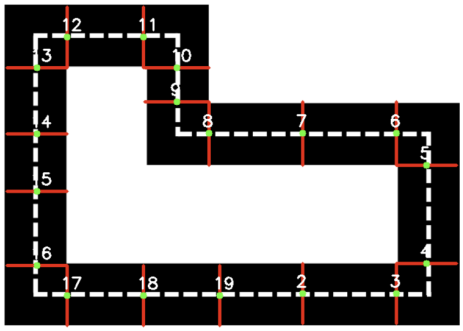

Figure 3. Track with checkpoints used.

3.3 Sensors

In order for the bus to navigate around the circuit it uses the sensors that were mentioned in section 3.1.1. The sensor integration is crucial for enabling the bus to perceive its environment and make intelligent decisions. The bus is equipped with a radar system consisting of five radar rays positioned and distributed across a range of 180 degrees, from -90 to 90 degrees relative to the bus's forward direction, allowing the bus to detect obstacles in front, as well as to its left and right.

Each radar ray has a defined length, which determines the maximum distance it can measure. When a radar ray is cast, it checks for intersections with the environment's barriers. This is done using a function that calculates the nearest intersection between the ray and the polygonal contours representing obstacles or road boundaries. The bus continually updates the distances detected by these rays using the method *get_radar_distances()*, which computes the distance from the bus to the nearest barrier for each ray.

The output of the radar system is a vector of radar distances, which are passed as inputs to the NEAT neural network. These distances serve as the bus's primary means of perceiving its surroundings [6]. For instance, if a barrier is detected close to the bus by one or more rays, the neural network should generate an output that prompts the bus to adjust its trajectory, such as steering away from the obstacle or slowing down.

Conversely, if no obstacles are detected, the bus may decide to accelerate forward. An illustration of the sensors can be seen in Figure 4.

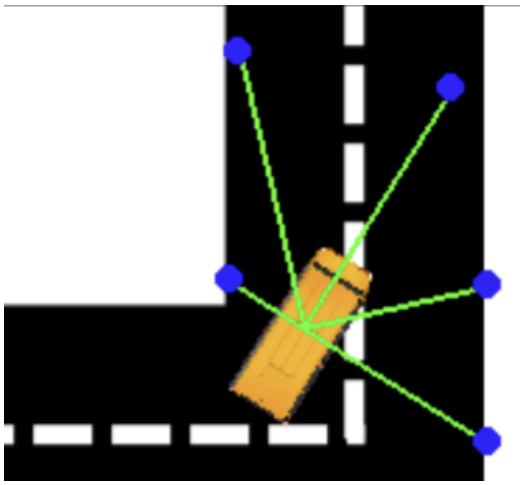

Figure 4. Sensors used from the bus.

3.4 Reward function

The reward function plays a critical role in shaping the behaviour of the agent. It evaluates the bus's performance by assigning a score based on various aspects of its navigation, encouraging efficient movement, obstacle avoidance, and compliance with contextual rules e.g. speed limits in school zones. The NEAT algorithm uses this reward function to evolve neural networks, selecting those that demonstrate improved performance over generations. The total fitness score is the cumulative sum of these rewards, computed at each time step. The fitness score is derived from several key factors, including:

- **Distance Moved**: The bus receives rewards based on the distance travelled, encouraging forward progress.
- **Speed Regulation in School Zones**: Penalties are applied when the bus exceeds the speed limit in designated school zones, while rewards are given for maintaining appropriate speeds.
- **Collision Avoidance**: Penalties are incurred when the bus collides with or comes too close to obstacles, promoting safer navigation.
- **Checkpoint Rewards**: Reaching predefined checkpoints provides rewards, encouraging the agent to follow the correct route and complete laps.
- **Survival Reward**: A small constant reward is given for continuous movement, motivating the bus to avoid stalling.

This score reflects the overall effectiveness of the agent in navigating the environment, balancing progress, safety, and rule adherence. The general form of the equation can be represented as follows:

$$F_{\text{total}} = \sum_{t=1}^{T} (d_t + s_t + z_t + c_t + p_t) \tag{1}$$

- d_t: Distance covered at time step t
- s_t: Survival at time step t
- z_t: School zone penalty or reward at time step t
- c_t: Collision penalty at time step t
- p_t: Checkpoint reward at time step t

3.5 Training

The training process is centred around two key parameters: *genomes* and *generations*. Which govern how the control strategies evolve over time. These parameters can be adjusted by modifying the population size, which determines how many genomes are evaluated within each generation, and by controlling the number of generations or evolutionary cycles that the system undergoes.

3.5.1 Genome

A larger population introduces greater diversity in each generation, encouraging the bus to explore a wide range of possible routes and try new strategies for navigation. However, it also requires more computational resources. On the other hand, a smaller population focuses the bus's exploration on fewer paths, potentially leading to more refined and efficient routes, but limiting the variety of strategies tested [7].

3.5.2 Generations

Running more generations gives the algorithm more opportunities to evolve better strategies. As the bus completes each generation, its performance is assessed through the *fitness evaluation* process, which scores each genome based on how well it navigates the environment. The best-performing genomes are selected to reproduce, passing on successful traits to the next generation. This continuous process helps the system learn and improve its navigation capabilities over time [7].

3.5.3 Balancing Exploration and Exploitation

Our NEAT algorithm uses genetic variation mutation and crossover to create new genomes each generation, striking a balance between *exploration* (trying new strategies) and *exploitation* (refining successful ones) [4]. While running more generations can lead to better optimization, there's a risk of *over-fitting* where the system becomes too specialized for current used training environment and less adaptable to new situations.

4. Result

This chapter presents the final parameters and fitness score from a completed lap, along with a visual representation of the simulation's outcome.

The final training session did not have a strict limit; it was initially set to 200 generations, which would continue for a couple of hours. However, the training was ultimately terminated when a sufficient result was achieved, specifically when the model completed the track and slowed down at the correct parts of the map.

The final run of the bus can be viewed with the link in the beginning of the report. The specifics of that run can be viewed in Table 1 below. The final environment and bus can be viewed in Figure 5.

Generations	Genomes per Generation	Successful genomes	Training Time (hours)
27	30	808	1

Table 1. Training data for the self-driving bus project, including final score and training time.

Figure 5. Final view of the environment.

5. Discussion

The following chapter presents a reflection of the results and ideas of how to improve the performance of the agent.

5.1 Choice of algorithm

Using NEAT in the early stages of the project made it easy to get the system up and running, particularly in refining how the bus was driving and setting up the environment. However, the real challenge emerged during the training phase, where fine-tuning the system became more complex. The primary difficulty was in adjusting the configuration parameters for NEAT. With so many parameters that can be modified, each influencing the others, optimizing the system became a tedious task. Initially, an exclusion method was used to test individual variables and their effects. Unfortunately, this approach proved unreliable, as the complex interactions between the variables meant that trial and error was insufficient for producing optimal results.

A better approach, instead of using NEAT, could have been to start with a more conventional learning method, such as deep neural networks or reinforcement learning algorithms like Deep Q-Learning (DQN). These methods allow for more straightforward control over the architecture and tuning process. Alternatively, model-based methods could have been explored, where a model of the environment is built to simulate different outcomes, allowing for more efficient planning and decision-making. This would have reduced the complexity and unpredictability associated with NEAT.

5.2 Optimizing the reward function

The reward function presented some challenges, and the final version was developed through a trial-and-error approach. As previously discussed, the final reward function included rewards for passing checkpoints and maintaining the correct speed within designated zones. The bus was penalized for crashing into walls or driving at the wrong speed in school zones.

In the end, the car learned to drive as intended, except for one issue: it tended to drive slowly on straight sections of the track that did not include a school zone. It is possible the neural network learned to drive slowly on straight parts and faster in curves. This was anticipated during the implementation, and a countermeasure was introduced by rewarding the car for driving fast outside of school zones, not just for driving slowly. However, it is possible the reward system was unbalanced, which could have caused this behaviour. No solution was found for the slow driving on the straight parts of the track, but the sole purpose of the bus was achieved - driving slow in school zones and faster outside.

5.3 Training

The training phase of the projects was similar to optimizing the reward function, a lot of trial and error. After some modification the training cycle would start over, which lead to some dead time while waiting for it to complete. This was not ideal and most of the time the modifications didn't improve the result.

When training the reward function took everything in to consideration i.e. learning to turn, speeding correctly in the different zones and avoiding collision. This could have been improved if the training sessions was split up into different part and built up neural network with previous training sessions. As a result issues with the system could have been quicker since working in sections.

5.4 Similar Projects

During this project, extensive research was conducted to find similar projects. Many of them used NEAT and genetic algorithms to train simple agents in 2D games, such as games

where the character jumps up and down. Some projects involving self-driving cars were also found, but they typically featured less complex environments. While it is certainly possible to introduce more complex environments, this method may not be ideal. As discussed earlier, relying on a fitness function based on basic rewards works well for simple tasks where right and wrong are clear-cut. However, using a fitness function and reward structure in an environment with many nuances of right and wrong becomes cumbersome. The rewards can interfere with each other, and the fitness score may never converge.

A preferred solution or method could involve combining different approaches. While a genetic algorithm might work well for simple survival tasks, other machine learning algorithms may be better suited for handling specific rules. A possible solution to this would have been to do the setup for a neural network ourself or use a library which lets the user have more control. Similar projects have used CARLA, TensorFlow, PyTorch all which would have been good contenders.

6. Conclusion

In conclusion, the project successfully demonstrated the implementation of a self-driving bus using NEAT to navigate a track with speed- zones. The project combined key concepts of genetic algorithms and neural network training, enabling the bus to dynamically adjust its behaviour based on environment conditions. While the bus achieved its primary goal some challenges were encountered. Particularity with optimizing the reward function and improving performance in certain track sections.

Despite these issues the results show the potential of using NEAT algorithms for self-driving application, with room for further refinement through more focused training strategies and reward function adjustments. As for future work, this could involve more segmented training phases or deeper exploration into balancing speed and accuracy to address the slower performances in the straight sections. Overall, the project provides a strong foundation for developing navigations systems using NEAT.

References

[1] Pierangelo Dellacqua. Genetic algorithms, 2024. Lecture 11, Linköpngs University, September 2024, Artificial Intelligence for Interactive Media, TNM114.

[2] CodeReclaimers. Neat overview — NEAT-Python documentation, 2024. Accessed: 2024-10-15.

[3] Pierangelo Dellacqua. Reinforcement learning, 2024. Lecture 9, Linköpngs University, September 2024, Artificial Intelligence for Interactive Media, TNM114.

[4] Robert MacWha. Evolving ais using a neat algorithm, 2021.

[5] Austin Osborne. Science: Neuroevolution is neat!, 2018. Accessed: 2024-10-15.

[6] Marios Skevofylakas. Deep reinforcement learning using unity ml- agents — part ii, 2021. Accessed: 2024-10-16.

[7] Trevor Burton-McCreadie. The neat algorithm: Evolving neural networks, 2024. Accessed: 2024-10-23.