UNIVERSIDADE LUSÓFONA

MSc, Artificial Intelligence for Games

Machine Learning Fundamentals Final Project

NeuroEvolution of Augmenting Topologies Concepts Applied to a Racing Video Game

Rafael Manjua

12th December 2025

Abstract

This report presents the design, implementation, and evaluation of a racing video game which uses concepts from NeuroEvolution of Augmenting Topologies (NEAT) [1] to train AI agents. The system combines simple, dense neural networks with a genetic algorithm inspired by NEAT to evolve driving behaviours without explicit programming.

The developed process involves a population of AI-controlled cars which learn to race through iterative evolution. Each AI controller perceives its environment through distance sensors that detect the boundaries of the racing track. The neural network outputs control signals for accelerating, braking, and turning the car. A rich fitness function rewards progress through checkpoints, lap completion, and safe driving behaviour while penalising inefficient movement.

Additionally, the system includes real-time visualisation of the training process, both via histograms, heat maps, and other graphs and a rendered viewport of the process itself.

Experimental results and their statistical analysis demonstrate that the evolutionary approach successfully produces AI controllers capable of completing multiple laps on tracks of varying complexity. After sufficient generations, the evolved genomes learn optimal racing lines and recovery from near-collisions.

Keywords: NEAT, neuroevolution, genetic algorithms, autonomous agents, neural networks, video game AI

1 Introduction

1.1 Context

Machine learning offers an alternative solution to traditional approaches to game AI, which often rely in handcrafted rules and state-machines.

Racing video games provide an ideal test environment for studying autonomous game agents: they require real-time decisions, continuous controls, and the ability to learn optimal decisions. Furthermore, a video game with low requirements (such as a low complexity of inputs) is ideal for the academic nature of this project.

It is important to note that this report focuses on the machine learning aspect of the project, skipping over details of the game's implementation.

1.2 Objectives

This project addresses the challenge of training neural network-controlled cars to navigate racing tracks autonomously. The system must:

– Learn to perceive track boundaries through distance sensors;
– Balance its weights and biases in to translate sensor inputs into driving actions;
– Evolve increasingly effective behaviours over successive generations;
– Generalise the learnt behaviours to navigate tracks of varying complexities;
– Achieve competitive performance against human players.

2 Background

2.1 Artificial Neural Networks

An artificial neural network (ANN) is a computational model inspired by the human brain. It consists of interconnected nodes, the neurons, organised into multiple layers. Each connection carries a weight which controls how much of an influence the first neuron's output has over the second neuron's input.

The basic architecture of a neural network consists of an input layer, which receives environmental inputs, one or more hidden layers, which perform intermediate computations, and an output later which produces actions or predictions. [2]

Each neuron computes a weighted sum of its inputs, adds a bias term, and applies an activation function. Common activation functions [3] include:

$$\text{Sigmoid,} \quad \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\text{Hyperbolic tangent,} \quad \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\text{Rectified Linear Unit,} \quad \text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

2.2 Genetic Algorithms

Genetic algorithms are optimisation methods inspired by Darwinian evolution. A population of individuals evolves over generations through selection, crossover, and mutation. Each individual is evaluated using a fitness function that dictactes the quality of that individual's performance. Higher-fitness individuals are more likely to be selected as the next generation's parents. [4]

Common methods for selection include tournament selection, where a subset of the population compete, usually in pairs, and the fittest one wins, and elitism, which preserves the best individuals across generations, preventing the loss of good solutions.

Crossover combines the genetic material from two parents in order to create a new genome. Arithmetic crossover computes weighted averages of parent values, while uniform crossover randomly selects genes from either parent.

Mutation introduces random variations to maintain population diversity and escape local optima. The mutation rate and magnitude control how often and how significant the mutations are. Traditionally, mutations can be applied to weights, biases, and activation functions.

2.3 NeuroEvolution of Augmenting Topologies (NEAT)

NEAT, introduced by Stanleu and Miikkulainen in 2002, is a neuroevolution algorithm that evolves both the weights and the structure of neural networks. NEAT begins with minimal networks and incrementally adds complexity through structural mutations that insert new nodes and connections.

Historical markings, a key aspect of NEAT, track gene origins, enabling more meaningful crossover between networks with different topologies. Speciation, another key concept in NEAT, groups similar networks into species that compete within themselves. [1]

While this project implements a significantly simplified variant of NEAT, it draws inspiration from its principles.

3 Methodology

3.1 System Overview

The system is composed by several interconnected components: a physic simulation for cars, a track system with collision detection, neural network controllers for AI cars, a genetic algorithm for evolution, and visualisation tools for monitoring progress. This architecture follows a design that separates concerns, leaving the genetic algorithm agnostic to the game it is tied to.

During training, a population of neural network-controlled cars attempts to navigate a racing track. Each car perceives its environment through distance sensors and outputs signals that are processed by the physics engine and turned into actions that control the cars. After all the cars have either crashed or reached the time limit, final fitness scores are calculated and the genetic algorithm produces a new generation. The cycle repeats itself until training is stopped by the user.

3.2 Neural Network Architecture

3.2.1 Input Layer

Each car is equipped with distance sensors that emit rays at fixed angles (e.g., -60°, -30°, 0°, 30°, 60°) relative to the car's own angle. These sensors detect the distance to the nearest track boundary in their respective directions, which provides the AI controller with spatial awareness of its surroundings.

Sensor readings are normalised to the range $[0, 1]$ by dividing by the maximum sensor range. A reading of 1.0, then, indicates maximum distance (no obstacle detected within the range), while values approaching 0.0 indicate a possible collision.

3.2.2 Hidden Layers

The number and size of hidden layers is configurable by the user. Each hidden layer consists of fully connected neurons with an activation function. The topology is defined by a list of layer sizes (e.g., $[4, 6]$ for two hidden layers with 4 and 6 neurons respectively). The implementation supports Sigmoid, Tanh, and ReLU activation functions.

3.2.3 Output Layer

The output layer produces four values corresponding to four actions: accelerate, brake, turn left, and turn right. These outputs use sigmoid activation to produce values in the range $[0, 1]$, indicating how likely an action is to be correct.

This activation function was chosen due to the nature of the game's control scheme; a car can, for instance, accelerate and turn left at the same time. The AI controller should not choose only one of four actions in every case.

Acceleration occurs if the acceleration output exceeds the brake output, and vice versa. Turning direction is determined by comparing left and right turn outputs against a threshold (in this case, 0.5).

3.3 Genetic Algorithm Design

3.3.1 Genome

Each genome represents an entire neural network as a collection of weight matrices and bias vectors. Additionally, a genome stores data including the network topology and activation functions for each layer. Any neural network can be (re)created with this data.

3.3.2 Selection

Parent selection uses tournament selection with a configurable tournament size. In each tournament, genomes are paired off, and the fittest of each pair is declared the winner. The same genome can participate in multiple tournaments, which leads to superior genomes potentially being picked multiple times.

Elitism is used to preserve the top performers across generations. A configurable number of elite individuals are separated from the rest and copied unchanged to the next generation. Additional copies of these elite individuals can still participate in the tournaments, however.

3.3.3 Crossover

Crossover combines two parent genomes to produce offspring. The implementation uses arithmetic crossover for weights: for each weight of the network, a genome inherits a weighted sum of the parents' weights. The child genome's activation function is copied from the fittest parent.

After tournament selection, another tournament is run in order to determine the parents of each genome needed for the next generation. Parent A is the name given to the fittest parent. The other parent is named Parent B.

This process is restricted to genomes with the same topology. This varies from NEAT principles for the sake of simplicity and time. If two genomes with differing topologies are picked as parents, the child genome is simply a copy of Parent A.

3.3.4 Mutation

Mutation introduces random variations to genome weights, activation function, and/or topology. Each of these components has a probability of being modified after crossover.

Weight mutation involves adding noise to weight values. The range of the noise is configurable. Activation mutation involves picking a random activation function for the genome, including its current activation function. Finally, topology mutation involves adding or removing hidden layers or changing the size of existing hidden layers. When a topology mutation occurs, weights and biases are reinitialised.

3.4 Fitness Function

As opposed to traditional neural networks, neuroevolution does not use backpropagation. As such, gradients must be computed in another way.

The fitness function guides evolution by quantifying the performance of each genome. A well-designed fitness function balances multiple objectives in order to provide meaningful gradients.

The implemented fitness function combines the following metrics:

1. **Distance score:** rewards total distance travelled along a track while moving forward. This provides a way to measure progress between each training checkpoint.
2. **Checkpoint score:** provides rewards for passing through each checkpoint in the correct order. Additionally, it penalises passing through checkpoints in the wrong order. This

shapes the car's behaviour toward completing the circuit in the intended direction.

3. **Lap score:** provides substantial rewards for completing full laps.
4. **Safety score:** rewards maintaining a certain distance from the track boundaries based on average sensor readings. This discourages riskier racing lines.
5. **Velocity score:** rewards faster, consistent speeds to promote fast racing rather than slow, methodical driving.
6. **Time penalty:** applies a penalty for each second a car survives in order to discourage aimless driving.

The final fitness is computed as the sum of all these components.

3.5 Physics and Simulation

3.5.1 Car Dynamics

The car's physics model simulates acceleration, braking, turning, and fiction. Position and velocity are updated using a fixed-timestep inspired by modern engines such as Unity and Unreal Engine that ensures deterministic behaviour regardless of frame rate variations.

Key configurable parameters include acceleration and brake strengths, turn speed, and friction coefficient. This allows for wildly different driving experiences, from arcade-like controls to more realistic dynamics.

3.5.2 Track Representation

Tracks are created using the Tiled map editor, which exports files to the TMX format. Each track consists of three layers:

1. **Background layer:** contains the track's background image.
2. **Objects layer:** contains starting positions for the cars, training checkpoints, and the finish line.
3. **Bounds layer:** contains two polygons that represent the track's outer and inner bounds.

The track boundary is computed as the difference between the outer and inner bound polygons, creating a ring-shaped circuit. Shortcuts, figure-eights, and other similar features are not supported.

3.5.3 Collision Detection

Collision detection operates on two levels.

For track boundaries, all the vertices of each car's main shape (a triangle) are considered. If any vertex's position is outside the valid track area, a collision is registered.

For checkpoints and finish line detection, bounding boxes provide with a faster collision detection, as it does not need to be nearly as precise.

4 Implementation

4.1 Technologies Used

The system is implemented in Python 3.13, which was chosen due to its extensive and important role in machine learning. Key libraries include:

- **Pygame:** provides graphics rendering, input handling, and window management.
- **PyTMX:** allows for the manipulation of maps from Tiled.
- **NumPy:** efficient array/matrix operations.
- **Shapely:** handles complex geometric operations with polygons. Instrumental for collision detection.
- **Matplotlib and Seaborn:** used for the real-time visualisation of plots and graphs.

4.2 Project Architecture

The codebase follows a modular architecture with clear separation of concerns:

- **config/:** contains all the configuration files.
- **data/:** stores files such as track data and saved genomes.
- **src/algorithm/:** contains all the code pertaining to the implementation of the agnostic genetic algorithm, including neural networks and genomes.
- **src/core/:** contains core simulation components such as cars, tracks, and events.
- **src/game/:** implements the gameplay loop.
- **src/io/:** contains useful methods to save and load genomes.
- **src/training/:** implements the training loop and the AI controllers.
- **src/ui/:** provides user interfaces for menus and plotting.

4.3 Key Classes

Below are classes which are essential to the neuroevolution process.

4.3.1 Neural Network Classes

The `NeuralNetwork` class performs forward propagation through layers. It is constructed from a `Genome` object, which stores the network's weights, biases, and topology. The `DenseLayer` class represents an individual, fully-connected layer of the neural network.

4.3.2 Genetic Algorithm Classes

The `Genome` class is the genetic representation of a neural network, storing all the parameters needed to instantiate one. It provides methods for mutation.

The `GeneticAlgorithm` class is responsible for the evolutionary process. It keeps track of the population of genomes, implements selection and crossover, and tracks generation statistics. The `next_generation()` method accepts a fitness function and produces offspring based on the fitness of genomes.

4.3.3 AI Controller Class

The `AIController` class acts as a bridge between genomes and cars. Each controller owns a `Car` instance and a `NeuralNetwork` created for its genome. Additionally, it contains the distance sensors that guide its controlled car through the racing track.

4.3.4 Training Loop Class

The `TrainingLoop` class manages the entire training process from beginning to end. It handles population initialisation, the execution of the training simulation, and generation transitions.

It supports two display modes: a minimal console mode for maximum performance and a visual mode for observing car behaviours in real-time.

4.4 Visualisation Tools

The training loop provides a dashboard with six plots which update every generation:

- **Fitness Over Generations:** tracks the best, average, and worst fitness across all generations.
- **Fitness Distribution:** histogram showing the spread of fitness values in the current generation.
- **Checkpoints Reached:** histogram showing the distribution of checkpoint progress for cars that have not completed a lap in the current generation.
- **Laps Completed:** histogram of lap completion counts in the current generation.
- **Survival Times:** histogram showing the distribution of how long cars survived before crashing in the current generation.
- **Death Heatmap:** displays where cars have crashed most frequently across all generations.

5 Experiments and Results

5.1 Experimental Setup

Experiments were conducted to evaluate the system's ability to evolve competent racing agents. The following parameters were used:

Table 1: `GenomeConfig` Parameters

Parameter	**Value**
`GENOME.MIN_LAYERS`	1
`GENOME.MAX_LAYERS`	4
`GENOME.MIN_NEURONS`	5
`GENOME.MAX_NEURONS`	10
`GENOME.WEIGHTS_STD`	0.1

Table 2: `MutationConfig` Parameters

Parameter	**Value**
`MUTATION.CHANCE_WEIGHT`	0.05
`MUTATION.CHANCE_TOPOLOGY`	0.01
`MUTATION.CHANCE_ACTIVATION`	0.05
`MUTATION.NOISE_LIMIT`	0.1
`MUTATION.RESIZE_LIMIT`	4

Table 3: `GeneticConfig` Parameters

Parameter	**Value**
`GENETIC.ELITISM_CUTOFF`	5
`GENETIC.TOURNAMENT_SIZE`	1

Table 4: `GameConfig` Parameters

Parameter	**Value**
`GAME.SCREEN_WIDTH`	1280
`GAME.SCREEN_HEIGHT`	720
`GAME.FPS`	60
`GAME.FIXED_DT`	0.0167

Table 5: `InputConfig` Parameters

Parameter	**Value**
`INPUT.KEY_ACCELERATE`	`pygame.K_w`
`INPUT.KEY_BRAKE`	`pygame.K_s`
`INPUT.KEY_TURN_LEFT`	`pygame.K_a`
`INPUT.KEY_TURN_RIGHT`	`pygame.K_d`
`INPUT.KEY_CHECKPOINTS`	`pygame.K_0`
`INPUT.KEY_SENSORS`	`pygame.K_1`

Table 6: `CarConfig` Parameters

Parameter	**Value**
`CAR.SIZE`	20
`CAR.ACCELERATION`	500
`CAR.BRAKE_STRENGTH`	600
`CAR.TURN_SPEED`	400
`CAR.FRICTION`	0.98
`CAR.SLIDING_FRICTION`	0.8

Table 7: `RNG` Parameters

Parameter	**Value**
`RANDOM_SEED`	42

Table 8: `ControllerConfig` Parameters

Parameter	**Value**
`CONTROLLER.SENSORS`	[-45, -30, -15, 0, 15, 30, 45]
`CONTROLLER.SENSOR_RANGE`	100

Table 9: `TrainingConfig` Parameters

Parameter	**Value**
`TRAINING.POPULATION_SIZE`	50
`TRAINING.MAX_GENERATION_TIME`	30.0
`TRAINING.SPEED`	20
`TRAINING.INTERVAL`	4
`TRAINING.SAVE_AMOUNT`	10
`TRAINING.AUTOSAVE_INTERVAL`	25

Table 10: `FitnessConfig` Parameters

Parameter	**Value**
`FITNESS.REWARD_DISTANCE`	2.0
`FITNESS.REWARD_SAFETY`	50.0
`FITNESS.REWARD_VELOCITY`	0.1
`FITNESS.REWARD_CHECKPOINT`	1000.0
`FITNESS.REWARD_LAP`	10000.0
`FITNESS.PENALTY_WRONG_CHECKPOINT`	-10000.0
`FITNESS.PENALTY_TIME`	-200.0

Training was conducted on `track_0`, a track with a relatively high complexity and several sharp tuns. The experiment lasted for 1000 generations.

5.2 Training Progression

Fitness curves showed evolutionary progress across generations. Initial generations showed slow improvement as the random weights were not at all adequate for the task. A period of steady improvement followed as the weights were tuned, leading to a more sensible interpretation of each sensor. Following a significant spike in maximum fitness, the average fitness steadily rose as agents converged.

The observed training progression was:

1. **Generations 1–45:** cars learn to accelerate (in either direction). Most crash immediately into the first wall they find. Most cars survive around 1.75 seconds.
2. **Generations 45–80:** a few select cars learn basic steering and navigate a couple turns. The first training checkpoint is crossed by those cars.
3. **Generation ∼80:** a couple cars master steering. This allows them to complete at least a full lap, as, once they can navigate tight turns, the rest of the track becomes trivial. However, they still have trouble with multiple laps as the start position of each lap differs slightly and requires different manoeuvres.
4. **Generation ∼85:** a couple cars complete multiple laps and manage to survive the full 30 seconds of testing. A lot of others cars are surviving for around 5–10 seconds.
5. **Generation 86 Onward:** the average fitness rises as the other cars catch up and learn how to run full laps.

Interestingly, as the average and maximum fitness values rise, the minimum fitness values become wildly inconsistent as some cars who master steering get confused and drive multiple laps in the wrong direction.

5.3 Performance Metrics

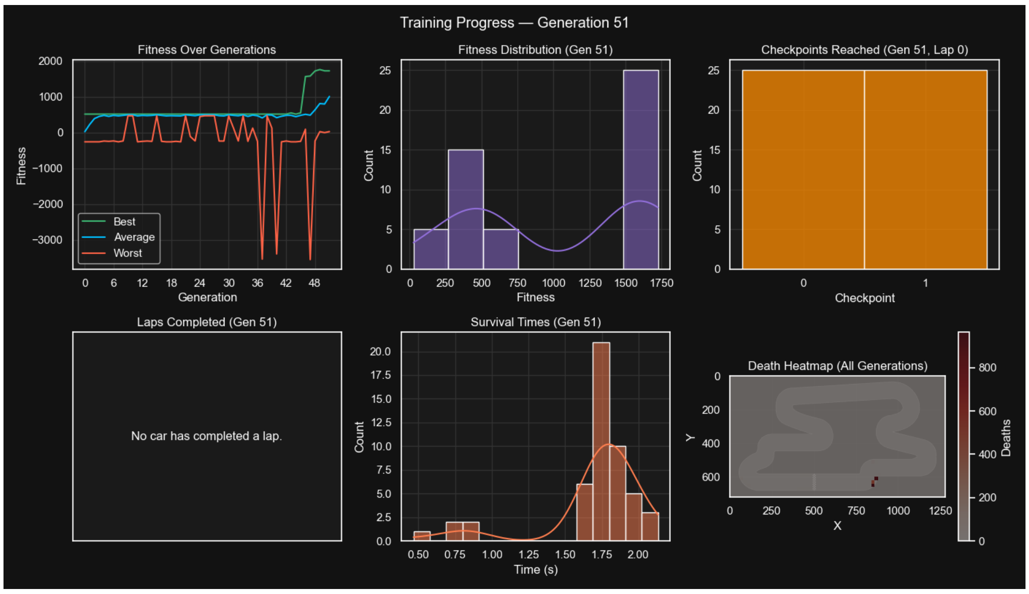

Figure 1: After 50 generations, most cars still cannot get past the first turn.

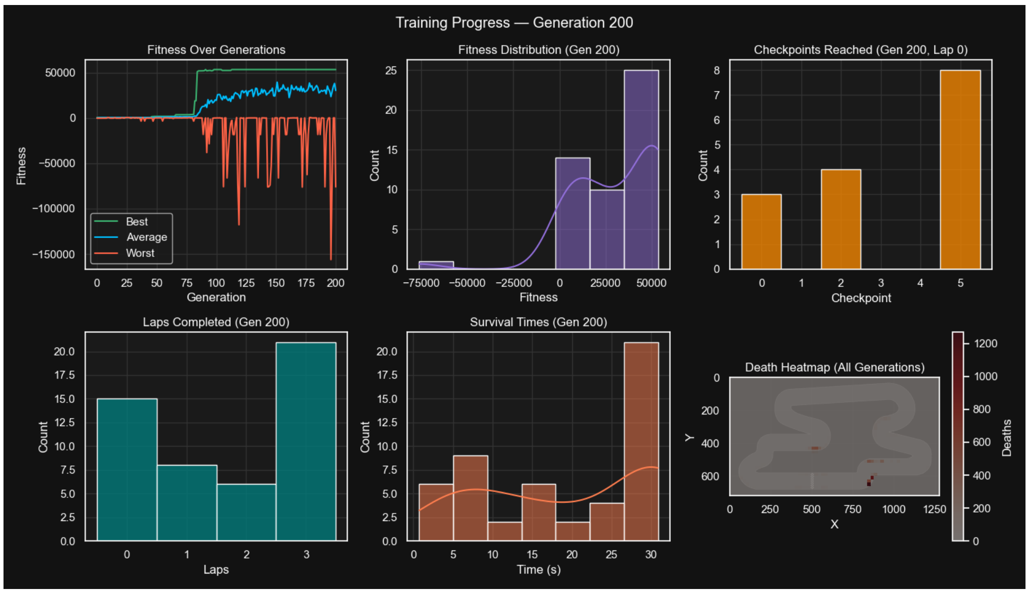

Figure 2: By the 200th generation, max fitness has plateaued and average fitness rises steadily.

Table 11: Performance Metrics after 400 Generations

Parameter	**Value**
Best Fitness Achieved	53423.25
Average Fitness (Final Generation)	39708.00
Cars With 2+ Laps (Final Generation)	29

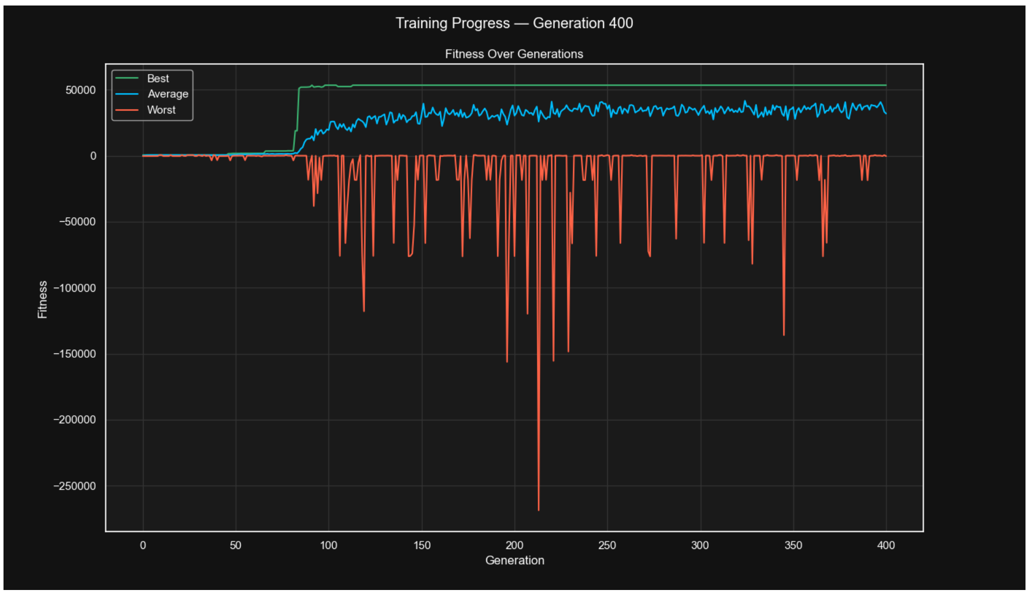

Figure 3: Fitness evolution across 400 generations.

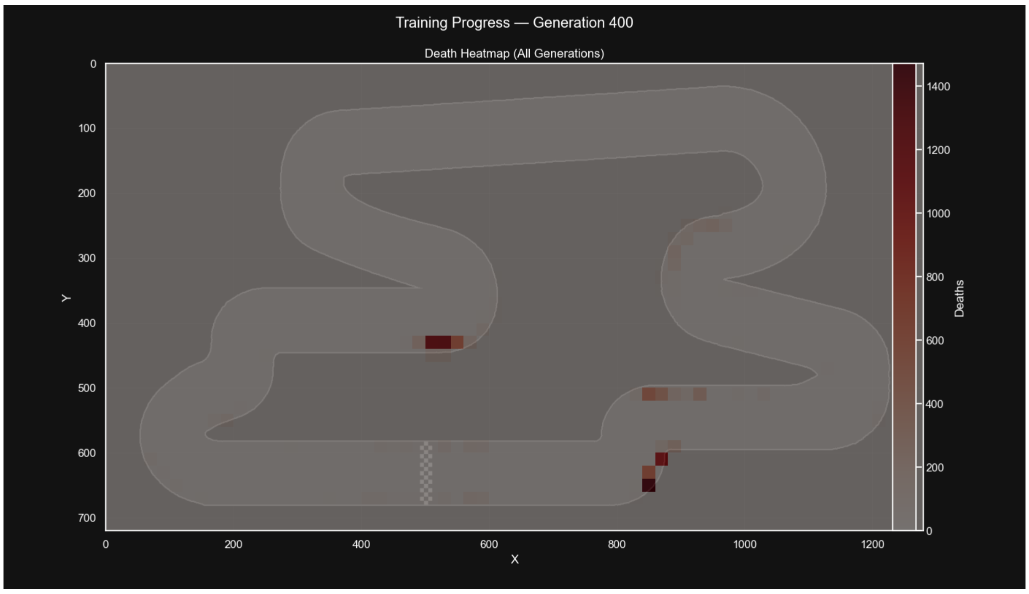

Figure 4: Death heatmap after 400 generations.

5.4 Cross-Track Generalisation

Agents trained on `track_0` performed well in three other tracks. This suggests that learnt behaviours are not track-specific and are well generalised. The weights are based purely on what the sensors report, and there's no dependence on any specific piece of geometry.

6 Discussion

6.1 Interpretation of Results

The experimental results demonstrate that neuroevolution can successfully produce competent racing agents without explicit rules or human intervention. The evolutionary process discovered effective sensor-to-action mappings through fitness-guided search.

The tight, S-shaped curve of the maximum fitness in the fitness progression plot paints an accurate picture of the racing problem. Initial improvements are slow as the AI learns how to control the car. Once cars manage to bypass a couple turns, progression accelerates exponentially as all that's needed to complete full laps is slight tuning of the weights. Turns aren't inherently any different from each other; some are tighter than others, but all require the same kind of movement. After the controls are mastered, top performances plateau as there's nothing else to achieve.

Curiously, the racing lines taken by the most advanced agents were so optimised that none of them were ever required to brake. Braking is useful to human players as reaction times are not perfect. However, as the AI was trained to have perfect reaction times, braking is rarely—if ever—necessary. Some curves are taken as close as possible to the boundary, despite the safety reward. This highlights the power that evolutionary algorithms have to discover non-obvious solutions.

However, it is clear that there is a low variety of racing lines being taken. The algorithm always searches for the most optimal solution, which leads to low variance in AI racers. When a human player races against them, they might notice the lack of width in the AI racers.

6.2 Strengths of the Approach

- The system learns entirely from fitness signals, rather than labelled data.
- Multiple objectives are easily combined through fitness summation.
- The simple feed-forward architecture is easier to analyse than deeper architectures.
- The evolved networks achieve satisfactory performance.
- Agents trained on one track perform well on other tracks with similar features.

6.3 Limitations of the Approach

- Training involves a large number of generations and evaluations.
- Unlike full NEAT, crossover only happens when genomes share a topology.
- There's little variety in racing lines.
- Agents do not learn how to interact with or avoid other cars.

6.4 Challenges Encountered

Even though the final algorithm and training loop are relatively simple, several challenges were faced throughout the development of the project.

Finding an adequate fitness function was particularly challenging. Early attempts rewarded survival time and penalised death, resulting in agents who drove in circles rather than progression. Additionally, finding a way to accurately track progress along the intended direction of racing was taxing. Initial attempts at this relied on external knowledge, such as checkpoint positioning and distance to the finish line. This resulted in agents who could not follow the intended path as, often, the absolute Euclidean distance to checkpoints does not account for track curvature. On sharp turns, agents following the intended racing line may temporarily increase their straight-line distance to the next checkpoint and the finish line. This created misleading fitness signals that rewarded incorrect behaviour. Finally, defining very open and general objectives led to agents getting stuck in local optima as they found the solution to be "good enough," with no incentive to explore any further.

Tuning the genetic algorithm parameters proved to be complex, as well. Many combinations of parameters were tried apart from the one reported in previous sections. With selection, crossover, and mutation, it was far too easy to select parameters that led to no progress after hundreds of generations. For instance, mutating the weights too often leads to genomes that forget useful weights from previous generations. Not mutating them often enough leads to genomes who never experiment and learn. Mutating the topology too often leads to little crossover, as parents do not have matching topologies.

As the game was developed alongside the genetic algorithm, it was sometimes tough to separate the game logic from the AI logic. For instance, the `Car` class initially stored sensor-related information. This, ultimately, broke the separation of concerns this project aimed for, as the car should be agnostic as to how it is being controlled. Sensor-related data is now stored in the `AIController` class, which needs that information to perform, similarly to how a human player would use their eyes to make decisions when controlling a car.

Another notorious part of the development was the responsiveness and performance of the software. With hundreds of physics operations per frame and a logging of several metrics, especially at higher training speeds, Pygame cannot keep up with the drawing operations it must perform. Pre-computed masks provided a performance boost, and limiting the amount of physics operations per frame improved UI responsiveness.

Finally, the displaying of plots was lagging significantly. The solution was to create an entire separate process that runs parallelly to the main training loop.

7 Conclusion

7.1 Summary

This project successfully developed a neuroevolution system for training autonomous racing agents. The implementation, based on concepts from NEAT, demonstrates that even simple genetic algorithms can evolve neural networks capable of navigating racing tracks.

The implementation illustrates how complex behaviours can emerge from simple evolutionary bases. Without any explicit instructions on how to drive, the algorithm discovered effective strategies which most human players cannot execute without extensive practice.

7.2 Future Work

Due to the academic nature of this project and the time constraints associated with it, its scope is ultimately limited. There are several directions in which to extend this project:

- Adding full racing mechanics for the gameplay portion of the project, such as tracking each racer's place, having a 3-lap limit, and showing a countdown before the race begins;
- Implementing full NEAT, with speciation, historical markings, and deep crossover;
- Training AI controllers to avoid other racers;
- Improving the variety of racing lines taken, perhaps with different fitness functions that allow of easier/harder opponents;
- Implementing more game-related variables, such as power-ups, shortcuts, and different types of vehicles;
- Using cross-validation to find optimal combinations of configuration parameters for the genetic algorithm. Alternatively, evolving the parameters themselves while training;
- Focusing on optimisation, performance, and responsiveness, perhaps by exploring alternative solutions to Pygame.

References

[1] K. O. Stanley and R. Miikkulainen. “Evolving Neural Networks Through Augmenting Topologies”. In: *Evolutionary Computation* (2002). URL: `https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf`.

[2] C. Gershenson. “Artificial Neural Networks for Beginners”. In: *CoRR* cs.NE/0308031 (2003). URL: `http://arxiv.org/abs/cs/0308031`.

[3] *Activation Functions*. ML Cheatsheet. URL: `https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html`.

[4] *Genetic Algorithm (GA)*. EBSCO Research Starters. URL: `https://www.ebsco.com/research-starters/computer-science/genetic-algorithm-ga`.

[5] *Python Documentation*. Python Software Foundation. 2024. URL: `https://docs.python.org/3/`.

[6] *Pygame Documentation*. Pygame Development Team. 2024. URL: `https://www.pygame.org/docs/`.

[7] *Matplotlib Documentation*. Matplotlib Development Team. 2024. URL: `https://matplotlib.org/stable/index.html`.

[8] *Seaborn Documentation*. Seaborn Development Team. 2024. URL: `https://seaborn.pydata.org/`.