

Project aegis

*Abhyuday Roychowdhury*

*School: Reading School*

*Candidate no. 7169*

AEGIS: Artificial Emergency Guidance and Intelligent System

Applying Neuroevolution to create a self-learning artificially intelligent collision avoidance system.

Table of Contents

[Introduction 2](#_Toc720285)

[Analysis 3](#_Toc720286)

[Project Outline 3](#_Toc720287)

[Research 4](#_Toc720288)

[Machine Learning 4](#_Toc720289)

[Autonomous Vehicles 8](#_Toc720290)

[Current Systems and Problems 9](#_Toc720291)

[Potential Solution 10](#_Toc720292)

[Applications 10](#_Toc720293)

[Potential Clients 11](#_Toc720294)

[Objectives 11](#_Toc720295)

[User Interface 11](#_Toc720296)

[Environment 11](#_Toc720297)

[Neural network 11](#_Toc720298)

[Genetic Algorithm 12](#_Toc720299)

[Extensions 12](#_Toc720300)

[Potential Programming Languages 13](#_Toc720301)

[Design 14](#_Toc720302)

[Overall system flow diagram 14](#_Toc720303)

[Artificial Neural Network 15](#_Toc720304)

[FeedForward method: 16](#_Toc720305)

[Sigmoid function 16](#_Toc720306)

[Environment 18](#_Toc720307)

[Environment pseudocode 20](#_Toc720308)

[Genetic Algorithm 21](#_Toc720309)

[Initialization 21](#_Toc720310)

[Fitness Function 22](#_Toc720311)

[Selection 22](#_Toc720312)

[Genetic operators 24](#_Toc720313)

[Limitations 24](#_Toc720314)

[Techniques demonstrated 25](#_Toc720315)

[List operations 25](#_Toc720316)

[Tree generation and traversal 25](#_Toc720317)

[Complex Scientific model 25](#_Toc720318)

[File saving 25](#_Toc720319)

[Complex user-defined **O**bject-**O**riented **P**rogramming 25](#_Toc720320)

[Coding practices 26](#_Toc720321)

[Functional Programming 26](#_Toc720322)

[Implementation of AI/machine learning 26](#_Toc720323)

[Use of version management 26](#_Toc720324)

[Further Considerations 28](#_Toc720325)

[Output 28](#_Toc720326)

[Input 29](#_Toc720327)

[Training time 29](#_Toc720328)

[Generation pool size 29](#_Toc720329)

[Technical Solution 30](#_Toc720330)

[constants.py 30](#_Toc720331)

[environment.py 31](#_Toc720332)

[Class CircleObstacle 31](#_Toc720333)

[Class Environment 32](#_Toc720334)

[car.py 37](#_Toc720335)

[Class Car 38](#_Toc720336)

[neural\_network.py 43](#_Toc720337)

[Class Connection 43](#_Toc720338)

[Class Node 43](#_Toc720339)

[Class Network 44](#_Toc720340)

[genetic\_algorithm.py 46](#_Toc720341)

[Class Code 46](#_Toc720342)

[ML\_controller.py 48](#_Toc720343)

[Main training function 49](#_Toc720344)

[Testing 52](#_Toc720345)

[A: User Interface 52](#_Toc720346)

[Test 1,3,4: HUD Rendering 53](#_Toc720347)

[Test 2: Real-time graph 54](#_Toc720348)

[B: Neural network 54](#_Toc720349)

[Test 1: Initialisation 55](#_Toc720350)

[Test 2: Updating the network 56](#_Toc720351)

[Test 3: Weight generation: 57](#_Toc720352)

[Test 4: Output evaluation: 58](#_Toc720353)

[C: Genetic algorithm 58](#_Toc720354)

[Test 1: Initialisation 59](#_Toc720355)

[Test 2: Member marking 59](#_Toc720356)

[Test 3: Crossover and mutation 60](#_Toc720357)

[D: Environment 60](#_Toc720358)

[Test 1: Initialisation: 62](#_Toc720359)

[Test 2: Collision detection 64](#_Toc720360)

[Test 3: Sensor data 65](#_Toc720361)

[E: Evidence of Machine learning/Artificial Intelligence 65](#_Toc720362)

[Test 1,2: Effectiveness of machine learning 66](#_Toc720363)

[Video evidence 67](#_Toc720364)

[Evaluation 68](#_Toc720365)

[User Interface 68](#_Toc720366)

[Neural Network 69](#_Toc720367)

[Genetic Algorithm 69](#_Toc720368)

[Environment 70](#_Toc720369)

[Conclusion 70](#_Toc720370)

[References 72](#_Toc720371)

# Introduction

Over the last year, I have been researching and learning about machine intelligence and the numerous fields it can be applied to. I think it is a revolutionary concept and I have been fascinated by how simple but effective it can be at accomplishing complicated tasks by demonstrating human-like intelligence. It is a simple technique that can be used to solve complicated problems which are impossible through a traditional approach which involves hardcoding algorithms for a task.

I intend to design an AI that can accompany driving systems, whether human or autonomous. The main goal of the AI is to assist the driver by avoiding any collisions while the vehicle is moving. The AI will be given data about the immediate proximity of the vehicle. I will use machine learning to train the AI to establish patterns from the distances returned by the proximity sensors. By doing this, I will be able to create a general driving AI which is not specific to one track or course but is able to work simply from the data about its surroundings. Once designed and trained in a software environment, it should be able to perform similarly in a real environment.

# Analysis

## Project Outline

I will create an Artificial intelligence that is able to assist a moving vehicle to **avoid obstacles** in its immediate proximity. It will use data from its surroundings like distance from **proximity sensors** and use a **neural network** to compute a **decision** from this data in real-time. I will use an **evolutionary algorithm** to train this AI by producing an ideal neural network for **collision avoidance**. This training will be done in a **virtual environment** so that the AI can later be exported and used in real vehicles any environment.

I have divided the project into 3 main components:

First, I will have to design an **environment** for my whole model to train and test in. This will be designed appropriately so that it can represent realistic surroundings so once trained, the AI can perform similarly in the real world.

Next, I will design a **neural network** which takes data from this virtual environment as inputs. This neural network will be used to make a decision and manoeuvre the vehicle.

Finally, to train the neural network, I will use an evolutionary or **genetic algorithm**. This will generate a population of neural networks and use natural selection to refine and obtain the best neural network for the task, hence finding the solution.

## Research

First, I will investigate the field of autonomous cars and machine learning to gain a better understanding of the scope of problem I am trying to solve and set the objectives my solution should meet.

### Machine Learning

Computers are evolving at an astounding rate and are replacing humans in many industries as they are much better than us at certain tasks. Cashiers, clerks and assembly workers have already been replaced across the world. This is because computers are much more efficient at performing these procedural tasks. There are, however, many fields in which computers cannot yet compete with humans. These fields are where intuition, creativity or emotions are required. This, however, is changing with the rapid development of machine learning algorithms. This is exactly what my system is trying to solve. My system will attempt to mimic and improve upon the intuition we rely on for driving.

In most types of programming, programmers must explicitly program the software to do anything. This type of programming involves static programming of instructions for a task. However, this is not easily applied to an AI driver as there are **too many variables** in a driving environment to design a static formula.

Instead of creating an algorithm specific to one environment or vehicle, I can use machine learning to develop a model which once trained, is able to react in any environment, on any vehicle provided the data required is supplied in real-time (proximity distance sensors in this case).

The AI identifies **patterns** in data and uses these patterns to learn similarly to humans. Understanding correlation between data allows the AI to react to any future data. This is how artificial intelligence works. Machine learning relies on **predictive modelling** where the output is only as good as the input provided, hence useful and accurate sample data will be required to produce the best intelligence. I conducted most of my research on machine learning through Wikipedia (Anon., 2018).

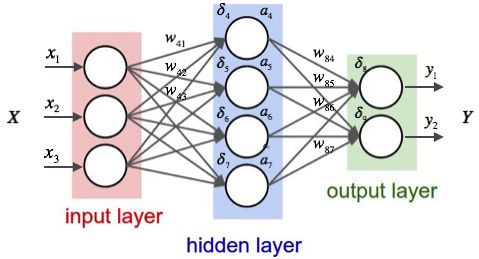


Figure 1-Neural network structure from Medium.com

The data structure I intend to use to store and process the data from the car’s proximity is a **neural network**. A neural network simulates the architecture of a human brain with nodes, each of them interconnected to other nodes through connections called weights. This allows the network to recognize patterns, learn and make decisions in a similar way to a human. Modelling artificial neural networks in a similar way to our brains is the reason it is able to replicate near human-like intelligence.

The principle behind it is that when provided with an input, it will produce an output, which is specific to that input. I will provide inputs to the input layer, which will allow the AI to essentially, ‘see’ the environment. These nodes are arranged in a series of layers where each node is connected to layers on either side (Figure 1).

When referring to a network **[L1, L2, L3 … Ln]**, the value of L1 represents the number of nodes on the first layer, L2, the number of nodes on the second layer and so on. The value n is the cardinal number representing the number of layers in the structure and could be any integer greater than 0.

The first layer of nodes is called the **input layer (X)** where each node (x1, x2, x3) receives a signal. Other nodes (y1, y2) sit on the opposite side of the network and signal how it responds to the information it has learned; these nodes make up the **output layer (Y)**. Between these lie one or more layers of **hidden nodes (a4, a5, a6, a7)**, which usually make up most of the network. These hidden nodes act very similarly to the input and output nodes except that their input is from the previous layer. By propagating the values through the network, we can obtain values in the output layer. This is the decision computed by the neural network.

Note: Occasionally, a **bias** node will be added to some layers. This is a node that is **always activated** and adds flexibility to the network. They can be thought of as **constants** as in the larger equation. e.g. the constant c in y = 2x + c which will output c even if the node (2x) is zero.

Due to the way neural networks process and learn from data, it is impossible to determine whether the algorithm is working as intended from the code as all the decisions and calculations are in the structure of the network. Since the network acts like a black box, it is not possible to debug through traditional means such as trace tables or stepping through the code. Therefore, I must have another way to judge the success of my algorithm. One way to do this is to monitor the actual results achieved by the algorithm. I will plot a graph of performance against time trained to visualize the progress made by the algorithm to evaluate the success of learning.

In summary, machine learning relies on neural networks to handle the data. Data is passed into starting nodes. The connections of these nodes with the rest of the nodes allows the data to pass through the network and exit through output nodes to produce a solution. I will expand on the specifics of the implementation in the design section.

#### Neural Network Model/ NEAT Algorithm

First, I had to consider the type of neural network I would use to handle the data. Through my research, I learned a lot about how AI learns and how different algorithms affect the efficiency and accuracy.

One algorithm I considered was NEAT (Neuroevolution of augmenting topologies). Essentially, this algorithm creates a neural network where not only are the weights and biases (i.e. the connection parameters) of the neural network optimised, but the whole structure is changed, and nodes are repositioned depending on whichever achieves better results. (Stanley, 2002)

This is excellent for networks where the overall difficulty is not easily determined but I decided against this as its complexity was greater than its effectivity. The **difficulty** of a neural network can be defined as the **complexity of the structure** required to achieve results. The more nodes and connections, the more complex the structure is.

Through trial and error on a prototype model, I decided to use a static structure with 3 layers. The network will have **5 input and 2 output nodes** with a specified number of hidden nodes in the middle layer. This will provide enough complexity for the problem I am attempting to solve. By using a simpler network, it will take less time to train it for a model as fewer connection values have to be optimised. This can, however, be changed as required by the user.

#### Choice of learning algorithm

Secondly, I had to consider how I would find the perfect combination of weights for the neural network. This is what is meant by training a neural network. Therefore, I will refer to the ideal combination of weights required to be able to successfully avoid collisions as the **solution** from now on.

Standard neural networks rely on **backpropagation** in which case, the AI compares the output to the input and calculates a difference between the expected and actual value. This is calculated by a cost function. Training is achieved by minimizing the **cost function**. The cost is spread across the network so that the difference on the next attempt is lower, thus lowering the cost function. This is useful where there is a large sample of known solutions. In other words, to modify the algorithm so that the actual value is close to the expected value. This way, it will be able to produce solutions to similar problems with previously unknown solutions. (David Rumelhart, 1986 )

This type of learning is known as **supervised learning** where a human is able to provide the expected solution. For example, we can specify whether an image contains an apple and have the algorithm modify the weights itself, so it is able to recognise similar images as apples. However, in this case, there is no known solution to compare the produced values to. Therefore, I will have to use a **classic reinforcement-learning** (Figure 2 - learning algorithms) algorithm which learns by repeating the task and working to **maximize a reward/fitness function** by comparing the result each time to its previous results. It repeats this process until an acceptable solution is found. (Błażej Osiński, 2018)

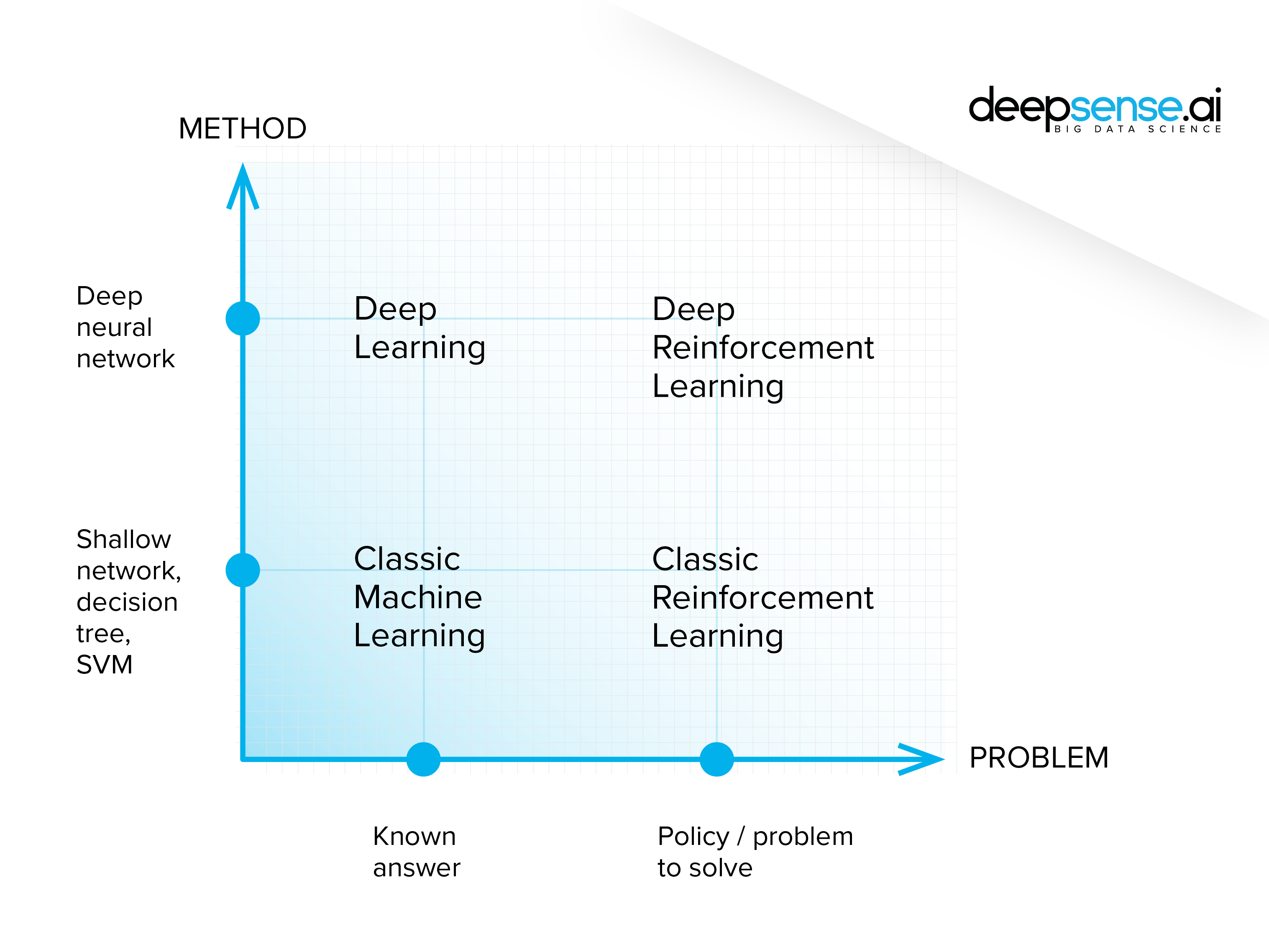


Figure 2 - learning algorithms

One type of reinforced learning algorithm I could use is an evolutionary algorithm. I will create multiple neural network instances in an environment, each controlling their own virtual car. I will then use an algorithm to repeatedly improve the neural networks until a satisfactory solution has been found. I will expand on the specifics of the algorithm’s design in the design section.

### Autonomous Vehicles

Vehicles have been used for decades and are still used on a day-to-day basis by almost everyone. Vehicles require constant input from its driver to perform correctly and safely. As I have mentioned before, with the introduction of computers into almost every field, they are even being installed into cars, to create something known as an autonomous car. An autonomous car is a vehicle that is capable of sensing its environment and navigating without human input. Predefined algorithms are implemented which allow it to drive without collisions. (Fridman, 2018).

Through my research, I found out a lot about how these cars operate from an article on self-driving cars (Gates, 2017). First, the car is equipped with several sensors to provide the on-board computer with all the data it needs. Alongside basic data like speed and GPS location, more data required for autonomous driving is relayed to the onboard computer. **High-quality cameras** are equipped usually at the top of the car to **recognize key objects** like traffic lights, moving objects (pedestrians and vehicles) and signs. Multiple images are captured continuously and parallax from these different images are used to calculate distances to various objects. A **LIDAR** (Light Detection and Ranging) unit uses laser beams to generate a 360-degree image of the surroundings. And finally, **radar sensors** are used to measure the distance to nearby objects. All this information is used to recreate the surroundings by the computer and navigate around it.

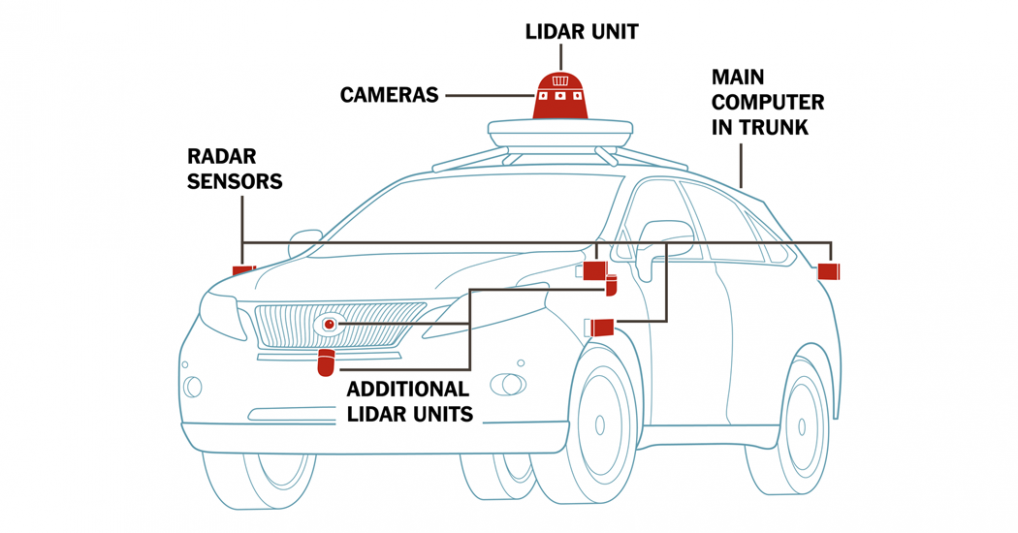


Figure 3 - self-driving car sensors from www.nytimes.com

The onboard computer has multiple modules installed including collision avoidance, drift warning and self-parking. The computer then feeds all the data into the modules to compute the correct action and executes it. This is how a self-driving car works in theory. My system will build upon and improve the collision avoidance system of autonomous cars.

### Current Systems and Problems

Autonomous vehicles are an emerging technology and with all new technology, it has its flaws. **Uber**, a transportation network company has begun to push out approximately 400 self-driving cars onto the roads of a few countries. Incidents have already occurred such as the death of a pedestrian in March 2018 (Bensinger, 2018). This is due to the fault where collision detection system failed to recognize the danger.

**Tesla** is another company that has been utilizing self-driving capabilities in its **Tesla Autopilot** system where the car is not completely autonomous but has modules that serve as emergency systems. A person in the car was killed in May 2016 after its collision detection system was unable to again recognize a truck directly in its path. I think a more intelligent system would have been able to determine the threat and avoid the truck. (Dowling, 2017)

The problem in most of these cases involve the collision detection system. It is the most important system and makes sure the car does not collide with anything. Thus, it is responsible for the lives of pedestrians and drivers.

### Potential Solution

I believe I can improve upon the current systems by designing a **collision avoidance** **system** that can be trained with a **reinforcement learning algorithm**. I think such a system can be designed which can use real-time data from its surroundings to steer or and even completely take over the vehicle’s controls. This system could be used as a failsafe, which only activates when it determines a collision is highly probable and corrects the car’s course to prevent it. It could also be similar to or even built over the Tesla Autopilot system which activates in emergency situations (e.g. the driver falls asleep) and takes complete control of the car. Not only could this system be used as a failsafe system, but it can serve as its own autonomous car system and be used on self-driving cars as a safety feature or modified to totally manage the driving system.

There are numerous applications for this system, but I will tackle the collision avoidance aspect of this problem.

### Applications

This software could be used in several fields to enhance vehicles. This isn’t just applicable for personal cars, but for new technology like **BigDog** (Raibert, 2006). This is a quadruped robot modelled after a dog used by the US military to manoeuvre around rough complicated terrain for delivery or to support their troops. My system can be used in projects like these to enhance the awareness and mobility of the robots. This will allow it to be artificially intelligent and able to operate without a nearby human operator. This can also be applied to any other devices in bio-robotics.

Other uses could be in drones which allow it to essentially sense its surroundings and avoid contacting anything harmful. The algorithm will however have to be modified to consider height as well. A human operator will not be able to react nearly as fast and the added latency from their distance to the drone will mean the drone will probably not be able to avoid the danger. This proposed system can gather data from its sensors and instantly make a decision to avoid any collision.

## Potential Clients

I could release this software as an **open source machine-learning library** that could then be used by anyone primarily using AI for autonomous driving in their projects. I think this basic model can be further developed by major companies like Tesla and used as an additional system for increased safety in their products.

## Objectives

### User Interface

I will keep the user interface simple and focus more on the learning algorithm. The only purpose of the UI is to allow the user to see the data in real-time, allowing them to train the AI as required. I will use the Pygame python library for all the UI rendering as previously explained in analysis.

1. The UI must be **compact**, covering no more than 20% of the width or height of the window.
2. A **graph** for the average population fitness for each generation will be displayed and updated frequently (at least 30Hz).
3. The **generation number** and **time** spent will be displayed.
4. The UI must **update** each frame to display the information at that moment.
5. The Pygame window must be **responsive** and be able to exit the main while loop without freezing.

### Environment

This is the class that will simulate the training instances for the AI.

1. **Collision detection** with obstacles must be **pixel-perfect** to ensure the AI trains correctly. For collision checking, all the mathematical functions must return accurate values.
2. The **decision** from the neural network must be evaluated correctly; turning and forward acceleration should be visible in the environment.
3. Obstacles will be **generated randomly at the start**. They will be initialized with a random velocity at random points with random radiuses. They will also not spawn in the immediate vicinity of the car spawn area to prevent an instantaneous collision.
4. The **sensors** from the cars should also return a correct value (the number of points in the sensor ray that do not collide).

### Neural network

The data structure responsible for storing and manipulating the data.

1. A neural network should be able to **initialize** from the structure and weights provided.
2. The neural network should be able to **evaluate an output** and return the decision to the environment (right or left).
3. A neural network should be able to **update** **all the** **weights** and the structure of its self when provided an array of weights.
4. **Randomly generated neural networks** should be created for each member in the current population

### Genetic Algorithm

1. The genetic algorithm should be able to **initialize** a population, each with their neural network when provided with the number of members, N.
2. The fittest member (best solution) from the previous generation should be explicitly **marked** and shown in the next generation.
3. Crossover and mutation should also occur with the **probability** that is specified.
4. The simulation process should run **indefinitely** until the minimum search criteria has been met.

## Extensions

These are additional extensions that could augment the software. However, I will not implement these as they go beyond the scope of the planned solution.

1. **Real Application**: After the AI has been trained and tested in the virtual environment, I can install it onto any vehicle provided sensors can reliably provide it data. I can then train it further to perfect it for that specific vehicle. This could also be applied to other vehicles like drones. It would use a model similar to mine, with a few modifications.
2. **Advanced physics**: Another extension could be to improve the physics in the virtual environment, so the vehicle can be trained in a more realistic environment. This will produce a more accurately trained model which will perform better in a real environment. Furthermore, it will significantly reduce the time it will take to train it in the real world.
3. **Versatile library software**: Another further extension could be to turn my software into an open source python library that anyone can train and use for their purposes.
4. **Multithreading**: As I am currently using the Python library Pygame to virtualize the environment, I can only use one thread for all the calculations. In order to multithread the learning process, I will need to use more advanced libraries or write a game engine from scratch as Pygame doesn’t currently support multithreading. This will allow the algorithm to utilize all the hardware and train significantly faster.

## Potential Programming Languages

**R**: I considered it as an option due to its strength at manipulating datasets and fast analysis. I decided against using R as it is not as programmer-friendly as Python and I would have to spend more time in learning the language.

**Python**: I have decided to use Python as my language of choice mainly due to the previous experience I already possess. The large number of machine learning libraries that are available also contributed to my decision. The vast number of python resources available on the internet for machine learning will also make it much easier to overcome any hurdles.

**C++**: Low-level languages such as C++ could greatly enhance the training speed by giving much lower level access to hardware. GPU accelerated training could be used. However, this would greatly increase the time spend developing the system and is unnecessary when higher level libraries such as TensorFlow could be used instead.

# Design

## Overall system flow diagram

This is a flow diagram describing the control flow for the entire process. It shows how the data is processed in each module and how the modules interact with each other.

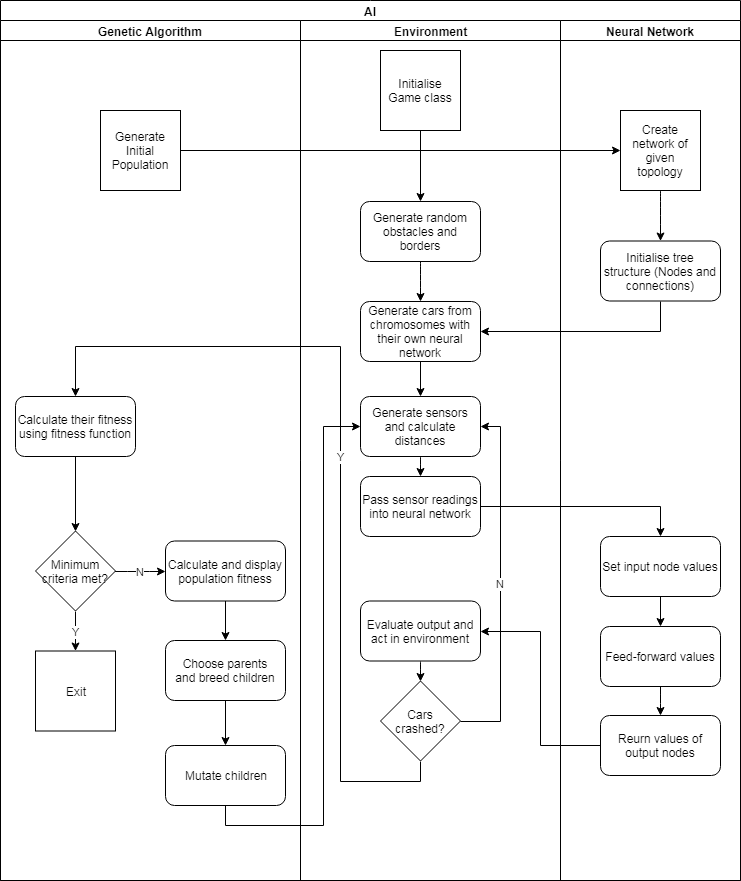
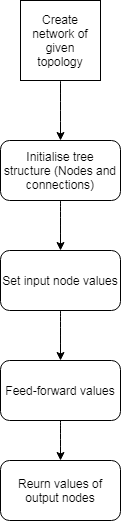


Figure 4 - Overall flow diagram

## Artificial Neural Network

The overall flow of the neural network algorithm is shown in Figure 7. The purpose of the neural network class is to simply decide on a **binary output**, based on the inputs it receives (values from car sensors). A decision is going to be made for each frame to either steer left or right in the environment by the car.

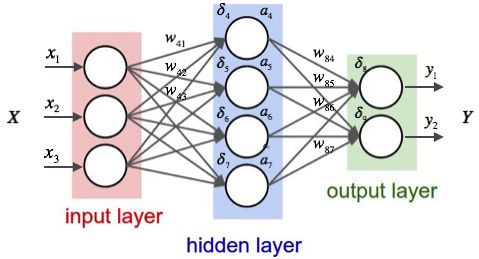


Figure 5 - Neural network structure from Medium.com

This whole structure needs to be produced in code and I will use object-oriented programming to achieve this. I will use objects to represent each of the nodes and connection objects for each of the connections interconnecting the nodes on each of the layers. This entire structure can be classified as a top down **tree structure**. The tree will be generated from the top (input nodes) down to the final output nodes.

Figure 7 - Neural network flow diagram

When a neural network object is initialized, nodes are created for each layer iteratively, while creating connections to each other node and storing the connection objects inside an array for each node. This will allow **traversal** through the neural network tree.

Each node will require a feedforward method which will set the output of a node to the **weighted sum** of all the connections to it from the previous layer.

### FeedForward method:

Below is the pseudocode to calculate the output for one node. The purpose of this method is to sum the value of the connections to adjacent nodes (refer to Figure 5) multiplied by its corresponding weight.

FUNCTION feedForward

sumOutput <- 0

for each connection

sumOutput += connectionValue \* weight

ENDFOR

output <- sigmoid(sumOutput)

ENDFUNCTION

### Sigmoid function

The sum is then passed through the sigmoid function before being returned. The purpose of the **sigmoid function** is merely to flatten the value between zero and one while preserving the data (Figure 7 - Sigmoid function). The formula for the function is given below. (Equation 1)



Equation 1 - Sigmoid formula

Figure 7 - Sigmoid function

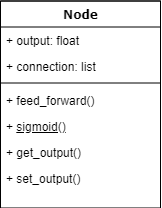
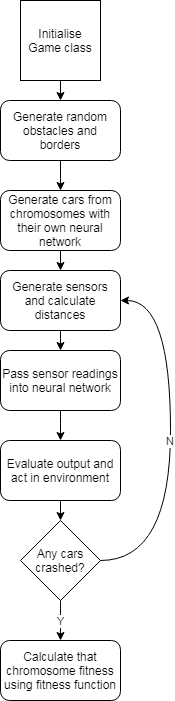
The nodes will be the main object, which will contain all the necessary methods and data structures to perform the calculations. The UML class diagram for this class is shown in Figure 10. This demonstrates **encapsulation** of datain objects. I will also demonstrate **abstraction** by designing the neural network class separately from the other classes. This way the environment can be configured and run without worrying about complications in the neural network class. The only way to interact with a neural network will be through each instance’s set and get methods. This is an example of **defensive programming** used to prevent any errors.

Figure 10 – Node UML Class diagram

## Environment

The sole purpose of this class is to evaluate the viability(**fitness**) of each solution in the current population.

For my project, I have decided to use a virtual car in a **simulated environment** for maximum control and efficiency. The training process will be much faster to run in a simulated environment compared to on a real car as simulations will be able to run simultaneously and much faster than a real car will as I will have full control of the environment clock. The simultaneous instances will be managed by the main controller class using the genetic algorithm.

Designing the car and virtual environment myself will allow me to have **complete control** of what the AI learns and how it learns. It will allow me to design the inputs the network receives and the outputs it produces. This will make it much easier to implement and debug the AI. This, however, means that I will not be able to employ perfect physics, which I would attain if I used a real car for the training process. This can be improved by further training a solution obtained from my algorithm in a real environment. This will minimize time spent on training while also achieving realistic driving. I will be focusing on the virtual aspect of the AI throughout this project.

At first, I considered making the virtual environment 3-dimensional, but I quickly realized that a **2-dimensional** environment is much more appropriate. Not only would it be much easier to design, but also training would be much faster without the added burden of 3D rendering. A 2D environment is perfect for my project as the AI collision detection and navigation only works in one plane. This could easily be extended to 3-dimensions once I have a working proof of concept should I choose to train a different vehicle such as a drone.

I was initially considering programming a full-fledged **physics engine** using the pymunk library for my simulation to achieve maximum realism and effectiveness. However, I decided against it when I considered the possible extension of training in a real environment. Therefore, the AI now only needs to recognize the basic patterns and learn the actions it needs to take. Now without the added burden of complex physics out of the way, I could focus on the learning aspect.

Once an environment object is **initialized** with an array of code objects (each containing a neural network to control each car), a random number of obstacles of random sizes are generated with random movement vectors at random positions. This **randomness** is required so that the AI doesn’t develop any bad habits such as just spinning in circles as this will be against the overall objective: to create an intelligent driver which can react to any random situation. Next, borders of fixed size will be created so the cars attempt to stay inside the screen. And finally, the car objects are initialized from the provided solutions and put into an array for evaluation.

Figure 11 - Environment flow diagram

After initialization is complete, an **iterative process** is started to compute all the information for each frame. In each iteration, the sensors for each car are calculated and displayed. These are five lines of points originating from the body of the car and spread evenly across the front of the car; essentially providing the car with a 180° field of view. This will return an array of distances from each sensor. Now, for each car, its sensors’ values are passed into its neural network and an output is produced. This output is the decision it makes, either drive left or right. This is repeated for each car. Once this has been done, the next frame is generated, by moving all the obstacles according to their vectors, and cars steered according to their own decision. Below, you can see a rough sketch of the way the neural network (brain) is going to interact with the car in the environment. This will be repeated for multiple cars with several obstacles.

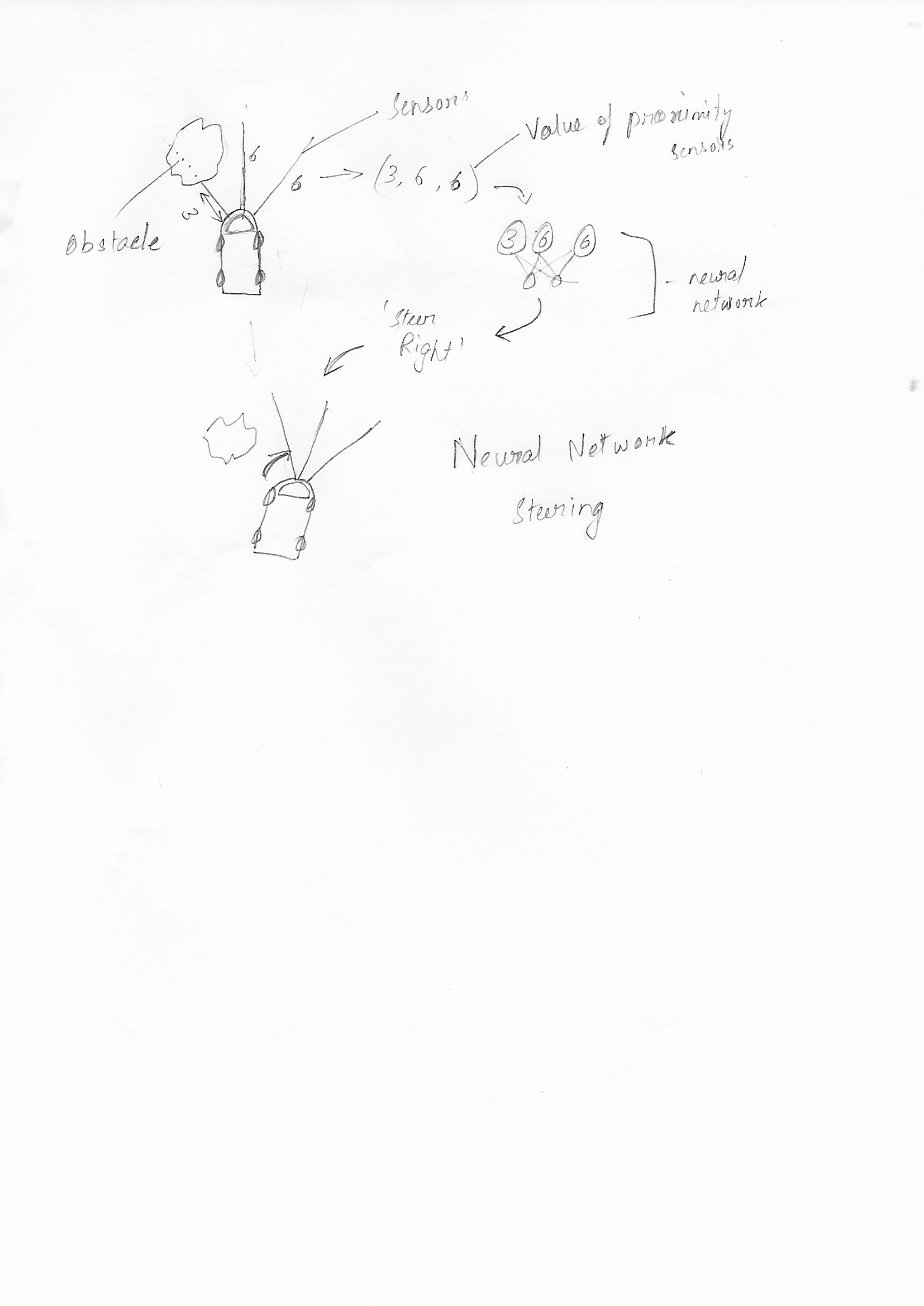


Figure 10 – Learning summary sketch

This process is repeated until every car has crashed or the maximum time limit has been reached. Then fitness is calculated for each car depending on how well they performed (how far they travelled without crashing). The fitness for each solution has now been evaluated and is returned.

### Environment pseudocode

Shown below is the basic logic and functions called in the environment to simulate each car and call their respective training subroutines.

FUNCTION Environment:

Create obstacles

Move obstacles

FOR each car:

get sensor data for car

pass sensor data into neural network

feed forward data in neural network

car output <- decision from neural network

ENDFOR

IF decision is left:

steer left

ELSEIF decision is right:

steer right

ENDIF

Move forward

IF crashed:

Calculate fitness

ENDIF

IF no cars left OR time limit reached:

FOR each car:

calculate fitness

Remove from evaluation queue

ENDFOR

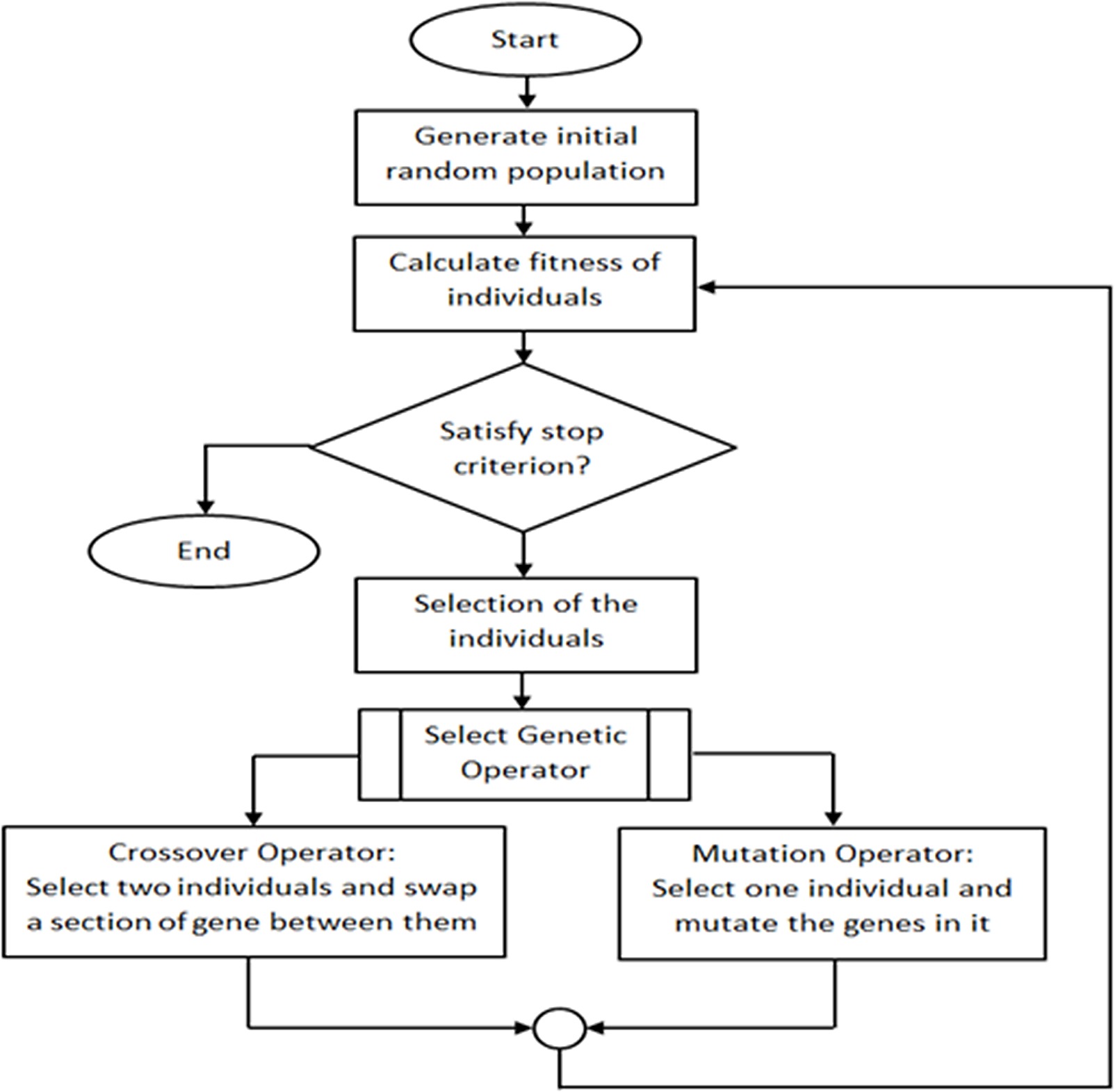
RETURN cars

ENDIF

Drawcars

ENDFUNCTION

## Genetic Algorithm

The only purpose of the genetic algorithm is to **improve** the neural network by finding the best combination of weights to allow the AI to drive. This is how it will learn.

A genetic algorithm inspired by natural selection can be used to solve **optimisation** problems like pathfinding or image processing. I combined this type of algorithm with neural networks which can carry out similar tasks but in a different way to create a **neuroevolutionary** algorithm.

In a genetic algorithm, a population of candidate **solutions** (called members) to an optimisation problem is evolved toward better solutions using an **iterative process**, with the population in each iteration called a **generation**. The idea is that with enough iterations, the population will eventually **converge to a solution**. I plan on terminating the algorithm when a satisfactory solution is found, or a maximum amount of generations have completed.

Figure 13 - Genetic algorithm flow diagram

Each candidate, called a solution, has its own set of properties, which can be altered to make it “fitter”. A fitter candidate is better at solving a problem. Which is in this case, driving. Therefore, my goal is to evolve the candidates until **maximum fitness** is achieved.

First, I will create a main controller class. This will be responsible for managing everything using the environment class, neural network class and all the genetic algorithm operator functions. After everything has been initialized, an iterative process will run until the satisfactory condition is met (time spent without crashing in this case). This iterative process will involve parent selection, crossover and mutation which I will explain below.

### Initialization

I will first generate an initial population with N members. I will discuss the choice of the size of the population later. Each member will have will be represented by a solution, a specific code.

Each solution has two main attributes, fitness and the neural network with random weights. The next step is to evaluate and assign a fitness to each solution depending on how well its neural network performs so that the genetic operators can be applied to “evolve” them.

This initial population is then evaluated using the environment by creating a environment object and passing in all the solutions.

Note: Due to the random nature of the initialization process, there is a chance that the solution we are looking for is present in this initial population, but this is extremely unlikely.

### Fitness Function

A fitness function is an **objective function** that is used to summarize how close a solution is to the real solution and must be maximised or minimized to solve the problem; i.e. how close the AI is to being perfect; navigating perfectly while avoiding obstacles. This means that each member of a population will be assigned a fitness value computed based on how that member performs in the simulation. The better it performs, the higher the value.

To allow my algorithm to **converge** to a solution, I must design a fitness function that is accurately able to represent how proficient each member is at performing the task. If designed incorrectly, it could take a long time and possibly not even converge. Since my aim is to create an AI that can survive and navigate, I need to consider the distance it managed to travel before colliding. This is also known as a **reward function**. I plan to implement additional heuristics like total displacement from starting position and average speed if my algorithm cannot perform well. These should provide additional incentive for the AI to continuously move instead of staying still to avoid collisions.

This function is used to assign a fitness value to each solution by the environment class.

### Selection

After the initial evaluation and on each generation, a portion of the existing population is selected to breed a new generation through a process called crossover. Individuals for breeding are selected randomly with fitter individuals having a **greater chance** of being selected. This ensures that those individuals with characteristics that produce better results have a greater chance of breeding. Since the next generation’s members are created from characteristics of the best of the current generation, the **average fitness increases**. Theoretically, converging to a population of perfect solutions.

For selecting individuals, I will use a **fitness proportionate selection** algorithm, also known as roulette wheel selection where fitter individuals have a greater chance of being selected.

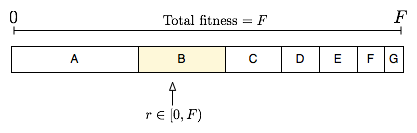


Figure 12 - Fitness proportionate selection en.wikipedia.org

Selection pseudocode:

FUNCTION choose parent:

Total fitness <- 0

for each member:

total fitness += fitness of member

ENDFOR

Pie size <- total fitness \* random float between 0 and 1

fitness <- 0

for each member:

fitness += member fitness

IF fitness >= pie size:

RETURN member

ENDIF

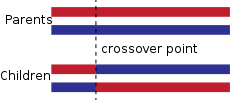
ENDFOR

ENDFUNCTION

This algorithm creates a pie of the total fitness of the population by summing the fitness of each individual and assigns each member a chunk proportional to their fitness. A random point is chosen on the pie and the member it belongs to is chosen, causing fitter members to have a higher change of selection.

### Genetic operators

To generate a new population through new child solutions, two main processes will occur: crossover and mutation.

Once two parents are selected through selection, **crossover** occurs. To create a child who shares characteristics from the parents, the code from the parents is combined to form the child’s code. New parents are selected for each child and this process is repeated until a new population of size N is created. I will use a single crossover point; after which I will take the average of the numerical weight values from each parent to produce a child.

I will then **mutate** the child’s code by randomizing a weight for a few randomly chosen children. This will introduce diversity in their code and avoid converging to local minima. In other words, this randomness introduces code that might not be present in any of the other members which alters the solutions slightly, potentially discovering a better solution.

Figure 15 - Crossover diagram

### Limitations

Although a genetic algorithm is a viable solution for optimising a neural network, it can be very **inefficient** if designed incorrectly.

One of the biggest problems usually is the time spent in **evaluating fitness** for each individual as each individual has to be evaluated inside the simulated environment especially since Pygame has to render all of this in one thread. This could potentially take very long if a solution isn’t converged to quickly. Therefore, I must keep the population size, N relatively low as it will take a long time otherwise.

Another limitation is that the solution is **never perfect**. It only converges to one of the possible solutions, often ignoring other even better solutions. However, this is fine as my problem only requires a good enough solution, one capable of driving intelligently.

I do, however, think that the genetic approach I am taking is perfect for my problem as it will find a relatively good solution in a realistic time.

## Techniques demonstrated

Here I will briefly describe the technical skills and coding practices used. I will explicitly reference some techniques which are not obvious.

E.g. *[environment]->(Car)->update()* references the **update** method of the **car** class in the **environment.py** file

### List operations

Multidimensional lists are used inside multiple classes to store data. One such instance is storing a specific solution or weights for a neural network. List operations such slicing to separate lists and list comprehension to create new lists have been used to manipulate data in the model.

*[neural\_network]->(Network)-> layers*

### Tree generation and traversal

The neural networks are defined as tree structures with nodes and connections. Each edge of the tree is represented by a connection object.

*[neural\_network]->(Network)-> \_\_init\_\_()*

The neural network is traversed using the feedforward algorithm when the output is retrieved from the output nodes.

*[neural\_network]->(Network)-> feed\_forward(),*

*[neural\_network]->(Node)-> feed\_forward(),*

### Complex Scientific model

My project models evolution using complex user-defined algorithms. The proposed collision avoidance is a complex optimisation problem which I attempt to solve using a genetic algorithm.

### File saving

I have implemented a file save function in my project to allow exporting the trained neural networks to a file, so they can be loaded and used later.

*[environment]->(Environment)-> save\_solution()*

### Complex user-defined **O**bject-**O**riented **P**rogramming

I have used object-oriented programming throughout my project. Dynamic generation of objects are used for the network, car and code classes as they are instantiated by the environment as required and directed by the user.

I have also demonstrated the use of some advanced OOP concepts such as overriding, inheritance and static methods in my code.

*[car]->(Car)->update(),*

*[car]->(Car),*

*[environment]->(Environment)-> circle\_rect\_collision()*

### Coding practices

I have used the PEP-8 (Warsaw, 2001) coding convention throughout my code. All functions and variables names use lowercase underscore convention and class names use the CapWords convention.

### Functional Programming

I have demonstrated functional programming in each of the classes. This makes the code easier to debug and modify in the future; which will be necessary as I plan on open sourcing this software.

### Implementation of AI/machine learning

My entire project is based around AI and machine learning. The AI is designed to learn to train a collision avoidance algorithm using machine learning techniques.

### Use of version management

I have used Git as my primary version management and backup software where I have backed up every significant change in the code through regular commits. All the code is on a GitHub hosted private repository as shown below in Figure 14. I will use this repository to host the source code when it is open sourced.

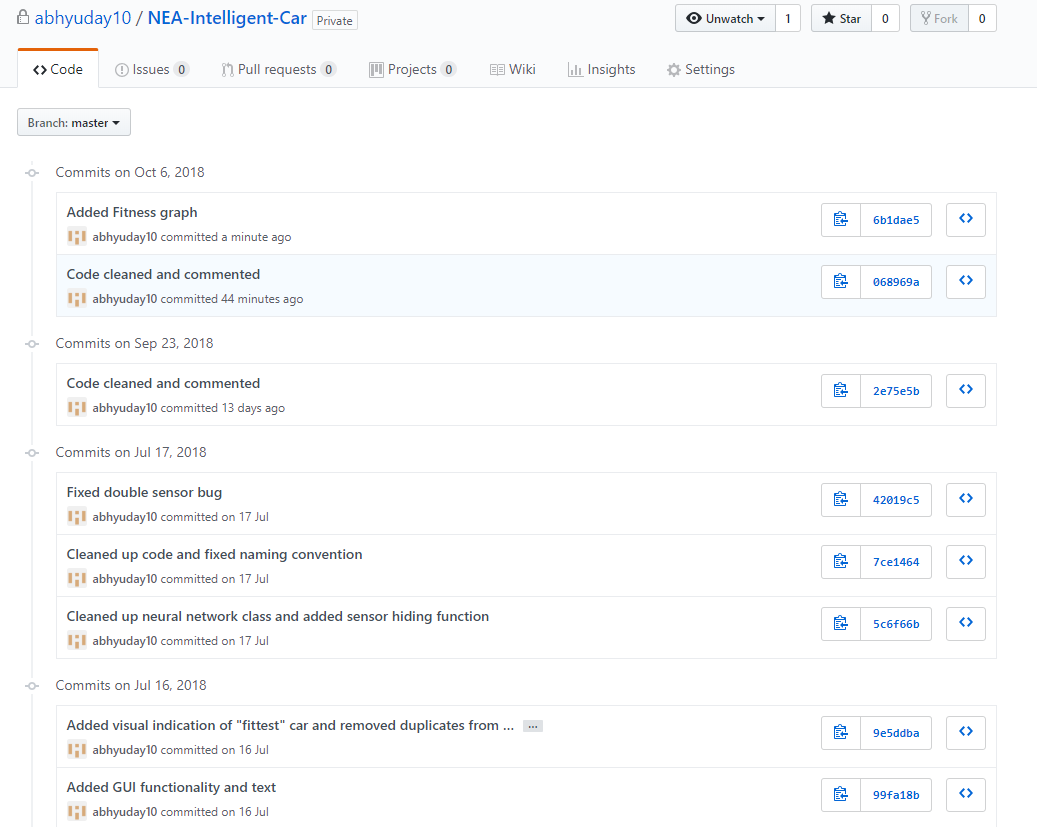


Figure 14 - GitHub commit history

I have also used GitHub pages to host my API documentation online at <https://abhyuday10.github.io/NEA-Intelligent-Car/>

All the classes, methods and variables are described and documented so my code is can be easily used by anyone.

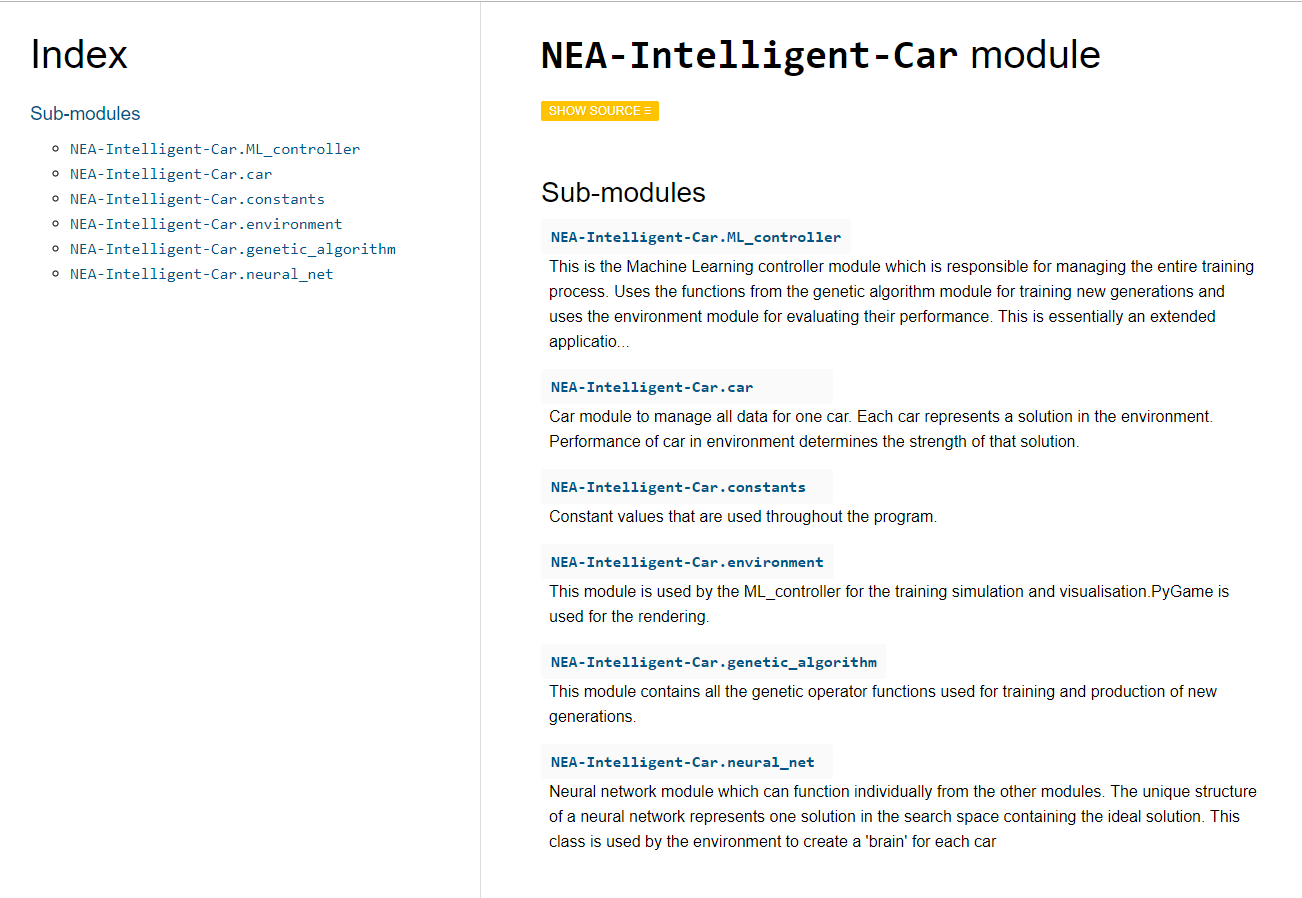


Figure 15 - GitHub Pages API Documentation

## Further Considerations

### Output

The outputs of the neural network are the values computed using the inputs provided from the input layers in the neural network. Each output is a **numerical value** when evaluated, results in an **action** in the environment. Designing what each output node represents is key to having the neural network learn quickly. Through my preliminary testing, I found reinforcement-learning networks using **fewer outputs** learn much faster because it has fewer possible actions to learn from.

Firstly, I need to decide the outputs for the neural network to control the car. One possibility I considered was to reduce the problem down to its basics. The movement would be controlled by adjusting **two variables, left and right**. Each variable is a float and designates a **force** to each side of the car. By applying two forces to the car, complete control is possible in the forward direction. This mechanic would be similar to how tank treads work.

To simplify it further, instead of a variable force I could have each variable represent a **probability** instead. An output of L=0.35 and R=0.65 would mean it would be 65% sure that turning right is the correct action. This would reduce the output to a **binary decision**, left or right. I will go with this model as it means the training process and the physics to represent it will be simpler. Therefore, the output layer will contain two nodes.

### Input

The input is the data passed into through the input layer, the first layer in the network. These inputs need to be relevant to the problem we are trying to solve so a **pattern between the inputs and the solution** can be established. Each set of inputs will only produce one specific output. If the inputs are irrelevant, for example if a person’s favourite colour is used to classify their athletic ability, it would end up establishing a totally non-existing correlation if there even is one.

For my project, I need to decide on the inputs I will provide the network. As mentioned previously in environmental design, because the learning environment is going to be designed by me, it will be much easier for me to design the inputs.

My problem is related to a car’s movement. This means my inputs need to be about the object’s position or more precisely, its **position relative to obstacles**. I could provide the network with the exact coordinates of any environmental obstacle, but since I am designing my system for real use, I would like to design it in a way that can be replicated in real life. Therefore, I will provide the car with virtual sensors to detect anything nearby. The **distance** returned by each sensor will serve as inputs for its network. This means the network will have one node for each sensor on the car.

### Training time

Training time is crucial in machine learning. The training algorithm must be **optimised** so that a good solution can be reached quickly in a realistic time. Several variables can be changed to achieve this like generation pool size, number of threads used and optimising the algorithms. In a reinforced-learning algorithm, most of the time is spent **evaluating the performance** in the environment. Therefore, I will have to optimise many of the collision checking and rendering algorithms.

### Generation pool size

The generation pool size is the number of cars **simultaneously** being improved in the environment. The pool size must be large enough so there is a **variance** in the population otherwise the learning will be limited as they will learn from a small previous generation. It also has to be small enough as all the physics must be evaluated for each member. I believe a size between **20 and 30** should be sufficient. This can be fine-tuned through trial and error.

# Technical Solution

Here, I will break down the code, explaining what each individual class and function does

.

All the code is hosted on a private GitHub repository listed below and will be made open source after the marking and moderation period have ended.

**https://github.com/abhyuday10/NEA-Intelligent-Car**

API documentation is available at <https://abhyuday10.github.io/NEA-Intelligent-Car/>

## constants.py

*"""Constant values to be defined by the user that are used throughout the program."""  
  
# Defining colour values for UI rendering in RGB format*BLACK = (0, 0, 0)  
WHITE = (255, 255, 255)  
BLUE = (0, 0, 255)  
GREEN = (0, 255, 0)  
RED = (255, 0, 0)  
GRAY = (169, 169, 169)  
  
*# Environment parameters*DRAW\_SENSORS = **True** *# Whether to render car sensors in environment or not.*TIME\_LIMIT = 820 *# Maximum time spent evaluating each generation.*SCREEN\_HEIGHT = 800  
SCREEN\_WIDTH = 1200  
  
*# Training parameters*POOL\_SIZE = 20 *# Population size for each generation.*TARGET\_TIME = 5999 *# Minimum time in seconds required for training termination.*TOPOLOGY = [5, 3, 2] *# Neural network structure for the cars.  
  
# Genetic algorithm parameters*CROSSOVER\_RATE = 0.7 *# Probability of child being produced.*MUTATION\_RATE = 0.05 *# Probability of mutation occurring on a weight.*

## environment.py

*"""This module is used by the ML\_controller for the training simulation and visualisation. PyGame is used for the rendering. This is where most of the time is spent."""***import** math   
**import** random  
**import** sys  
  
**import** pygame  
**import** thorpy **as** tp *# tkinter wrapper for the GUI***import** constants  
**import** car  
  
*# Initialize the Pygame library*pygame.init()

### Class CircleObstacle

**class** CircleObstacle:  
 *"""Circular obstacle class to manage an obstacle instance in the environment. Can be instantiated with coordinates, movement velocity and radius."""* **def** \_\_init\_\_(self, colour, x, y, move, radius):  
 self.moveX = move[0] *# x velocity* self.moveY = move[1] *# y velocity* self.colour = colour  
 self.pos = [x, y]  
 self.radius = radius  
  
 **def** draw\_to\_screen(self, screen):  
 *"""Simple method to render this object onto the screen using Pygame: draws a circle with the object parameters at its location."""* pygame.draw.circle(screen, self.colour, self.pos, self.radius)

### Class Environment

**class** Environment:  
 *"""Environment class to evaluate one generation of solutions  
 Takes a list of the solutions as input  
 Returns the evaluated solutions and their fitness"""  
  
 # Initialising environment variables* clock = pygame.time.Clock() *# Pygame clock to sync FPS* screen\_size = width, height = constants.SCREEN\_WIDTH, constants.SCREEN\_HEIGHT  
 screen = pygame.display.set\_mode(screen\_size)  
 pygame.display.set\_caption(**'Intelligent Driver'**)  
  
 **def** \_\_init\_\_(self, solution\_list, gen):  
 self.running = **True** rectx = 200  
 recty = 400  
  
 *# Specifying area where obstacles do not spawn* self.spawnrect = [(self.width / 2) - rectx / 2, (self.height / 2) + recty / 20, rectx, recty]  
  
 *# Specifying area where cars spawn* self.carStartPos = [self.spawnrect[0] + 100, self.spawnrect[1] + 100]  
  
 *# Generate obstacles, borders and cars into environment* self.obstacles = self.generate\_obstacles()  
 self.borders = self.create\_borders()  
 self.cars = self.generate\_cars(solution\_list)  
  
 self.evaluatedCars = [] *# List to put cars in once evaluated* self.generation\_number = gen  
 self.time = 0  
  
 self.pop\_size = len(solution\_list)  
 self.set\_borders()  
  
 *# Begin evaluation process for each frame* self.main\_simulation()  
  
 **def** generate\_cars(self, solution\_list):  
 *"""Generate cars from the list of solutions(neural networks) to be evaluated"""* cars = []  
 **for** solution **in** solution\_list:  
 cars.append(car.Car(solution, self.carStartPos[0],  
 self.carStartPos[1]))  
  
 **return** cars

**def** move\_obstacles(self):  
 *"""Updates position for each obstacle based on their velocity on that frame"""* **for** obstacle **in** self.obstacles:  
 *# Changes position of each obstacle by their velocity* obstacle.pos[0] += obstacle.moveX  
 obstacle.pos[1] += obstacle.moveY  
  
 *# Simple algorithm to create bounce from borders by reversing direction on impact* **if** obstacle.pos[0] + obstacle.radius > self.width:  
 obstacle.moveX = obstacle.moveX \* -1  
 **if** obstacle.pos[0] - obstacle.radius < 0:  
 obstacle.moveX = obstacle.moveX \* -1  
 **if** obstacle.pos[1] + obstacle.radius > self.height:  
 obstacle.moveY = obstacle.moveY \* -1  
 **if** obstacle.pos[1] - obstacle.radius < 0:  
 obstacle.moveY = obstacle.moveY \* -1

**def** save\_solution(self, solution):  
 *"""Save a trained solution to a file to be used later"""* **with** open(**"trained\_solution.csv"**, **"w"**) **as** file:  
 file.write(solution.weights)

**def** generate\_obstacles(self):  
 *"""Create randomly positioned objects with random velocities in Environment"""* obstacles = []  
 colour = constants.BLUE  
 number\_of\_obstacles = random.randint(4, 9)  
  
 *# Creates obstacles with the same position near spawn on each generation to encourage cars to travel further* position = [int(self.spawnrect[0] - 80), int(self.spawnrect[1])]  
 obstacles.append(CircleObstacle(colour, position[0], position[1], [1, 0], 60))  
 position = [int(self.spawnrect[0] + self.spawnrect[2] + 80), int(self.spawnrect[1])]  
 obstacles.append(CircleObstacle(colour, position[0], position[1], [-1, 0], 60))  
  
 **while** len(obstacles) < number\_of\_obstacles:  
 *# Generates a random velocity* move\_x = random.randint(-3, 3)  
 move\_y = random.randint(-3, 3)  
  
 *# Generates a random radius* radius = random.randint(40, 80)  
  
 *# Checks to make sure obstacles do not spawn on top of cars* position = [random.randint(0, self.width - radius), random.randint(0, self.height - radius)]  
 **if not** self.check\_if\_circle\_overlaps(position[0], position[1], radius, obstacles):  
 **if not** self.circle\_rect\_collision(self.spawnrect[0], self.height - self.spawnrect[1], self.spawnrect[2],  
 self.spawnrect[3], position[0], position[1], radius):  
 obstacles.append(CircleObstacle(colour, position[0], position[1], [move\_x, move\_y], radius))  
  
 **return** obstacles  
  
 @staticmethod  
 **def** circle\_rect\_collision(rleft, rtop, width, height, *# rectangle definition* center\_x, center\_y, radius): *# circle definition  
 """ Detect collision between a rectangle and circle. """* dist\_x = abs(center\_x - rleft - width / 2)  
 dist\_y = abs(center\_y - rtop - height / 2)  
  
 **if** dist\_x > (width / 2 + radius):  
 **return False  
 if** dist\_y > (height / 2 + radius):  
 **return False  
  
 if** dist\_x <= (width / 2):  
 **return True  
 if** dist\_y <= (height / 2):  
 **return True** dx = dist\_x - width / 2  
 dy = dist\_y - height / 2  
 **return** dx \* dx + dy \* dy <= (radius \* radius)  
  
 **def** check\_if\_circle\_overlaps(self, x, y, r, other\_obstacles):  
 *"""Detects if any obstacles overlaps"""* **for** obstacle **in** other\_obstacles:  
 **if** self.check\_if\_circles\_overlap(x, y, r, obstacle.pos[0], obstacle.pos[1], obstacle.radius):  
 **return True  
 return False  
  
 def** print\_if\_obstacles\_overlap(self):  
 *"""Returns True if any obstacles collide with another"""* overlapping = **False  
 for** obs **in** self.obstacles:  
 **for** obst **in** self.obstacles:  
 **if** obs == obst:  
 **continue  
 elif** self.check\_if\_circles\_overlap(obs.pos[0], obs.pos[1], obs.radius, obst.pos[0], obst.pos[1],  
 obst.radius):  
 overlapping = **True** print(**"overlapping: "**, str(overlapping))  
  
 @staticmethod  
 **def** check\_if\_circles\_overlap(x, y, r, a, b, t):  
 *"""Static method to detect if any circles overlap"""* **if** math.hypot(x - a, y - b) <= (r + t):  
 **return True  
 return False  
  
 def** create\_borders(self):  
 *"""This method creates the rectangle objects that act as borders for the screen"""* line\_width = 10  
 colour = constants.RED  
 width = self.width  
 height = self.height  
  
 *# top line* top = pygame.draw.rect(self.screen, colour, [0, 0, width, line\_width])  
 *# bottom line* bottom = pygame.draw.rect(self.screen, colour, [0, height - line\_width, width, line\_width])  
 *# left line* left = pygame.draw.rect(self.screen, colour, [0, 0, line\_width, height])  
 *# right line* right = pygame.draw.rect(self.screen, colour, [width - line\_width, 0, line\_width, height + line\_width])  
 **return** [top, bottom, left, right]  
  
 **def** set\_obstacles(self):  
 **for** car **in** self.cars:  
 car.obstacles = self.obstacles  
  
 **def** set\_borders(self):  
 **for** car **in** self.cars:  
 car.borders = self.borders  
  
 *# Rendering of different objects in the environment by calling their draw methods* **def** draw\_obstacles(self):  
 **for** obstacle **in** self.obstacles:  
 obstacle.draw\_to\_screen(self.screen)

**def** draw\_cars(self):  
 *"""Calls the update method to recalculate the positions and draws each car."""* **for** car **in** self.cars:  
 car.update()  
 car.draw()  
 **for** car **in** self.cars:  
 **if** car.solution.fittest:  
 car.draw()  
  
 **def** draw\_gui(self):  
 *"""Using the tkinter wrapper, thorpy to render the HUD."""* gen\_text = tp.OneLineText.make(**"Generation: "** + str(self.generation\_number))  
 time\_text = tp.OneLineText.make(**"Time: "** + str(self.time))  
 live\_text = tp.OneLineText.make(**"Cars Alive: "** + str(len(self.cars)) + **"/"** + str(self.pop\_size))  
  
 *# Specifies font size for HUD.* live\_text.set\_font\_size(18)  
 gen\_text.set\_font\_size(20)  
 time\_text.set\_font\_size(20)  
  
 *# Put elements into box to manage them better.* box = tp.Box.make(elements=[gen\_text, time\_text, live\_text])  
 menu = tp.Menu(box)  
  
 *# Render each element onto screen* **for** element **in** menu.get\_population():  
 element.surface = self.screen  
  
 box.set\_topleft(((self.screen\_size[0] - box.get\_rect()[2]) - 10, 10))  
 box.blit()  
  
 **def** draw(self):  
 *"""Calling all the draw methods"""* self.draw\_obstacles()  
 self.create\_borders()  
 self.draw\_cars()  
 self.draw\_gui()  
  
 **def** main\_simulation(self):  
 *"""Main loop that is run after initialising Environment"""* self.time = 0 *# Stores how long the current generation has been running for.* **while** self.running:  
 *# This is the main loop that renders each frame.  
 # Each iteration involves processing and rendering data for one frame.* self.time += 1  
 **for** event **in** pygame.event.get(): *# Managing the pygame event system.* **if** event.type == pygame.QUIT:  
 sys.exit() *# Simple event check to close window.  
  
 # 'Render' each frame by calculating everything until no cars left* self.screen.fill(constants.WHITE)  
 self.set\_obstacles()  
 self.move\_obstacles()  
  
 *# Calculate data for each car* **for** car **in** self.cars:  
  
 *# Get output decision from each car's neural network* car.inputs = car.get\_sensor\_data()  
 car.set\_inputs(car.inputs)  
 car.feed\_forward()  
 car.output = car.get\_outputs()  
  
 *# Evaluate output and react in Environment* **if** car.output == **"left"**:  
 car.rotate\_left()  
 **elif** car.output == **"right"**:  
 car.rotate\_right()  
 car.move\_forward()  
  
 *# If any cars crashed, record their performance(time) and remove from Environment* **if** car.crashed:  
 car.calculate\_fitness(self.time)  
 car.solution.time = self.time  
 self.evaluatedCars.append(car)  
 self.cars.remove(car)  
  
 *# Terminate Environment if all cars evaluated(they crashed) or time limit reached* **if** len(self.cars) == 0 **or** self.time > constants.TIME\_LIMIT:  
 **for** car **in** self.cars:  
 car.calculate\_fitness(self.time)  
 car.solution.time = self.time  
 self.evaluatedCars.append(car)  
 **return** self.cars  
  
 *# Draw current frame* self.draw()  
 pygame.display.flip()  
 self.clock.tick(30) *# Sets FPS by syncing render cycle to Pygame clock*

## car.py

*"""Car module to manage all data for one car. Each car represents a solution in the environment. Performance of car in environment determines the strength of that solution."""***import** pygame  
**import** math  
**import** environment **as** env  
**import** constants

### Class Car

**class** Car(pygame.sprite.Sprite):  
 *"""Defining the car class which inherits properties from the Pygame sprite class"""* DELTA\_ANGLE = 5.5 *# Specifies the maximum rotation vector on each frame* SPEED = 5 *# Specifies the maximum movement possible on each frame* **def** \_\_init\_\_(self, solution, x, y):  
 *# Initialise the sprite masterclass* pygame.sprite.Sprite.\_\_init\_\_(self)  
  
 *# Initialise images required for rendering* self.boom = pygame.image.load(**"images/explosion.png"**)  
 self.boom = pygame.transform.scale(self.boom, (50, 50))  
  
 **if** solution.fittest:  
 *# Load image of different colour car if this member is the fittest from previous generation* self.image = pygame.image.load(**"images/car\_fit.png"**)  
 **else**:  
 *# Otherwise load default car* self.image = pygame.image.load(**"images/car.png"**)  
  
 *# Scale the image for the environment* self.image = pygame.transform.scale(self.image, (30, 60))  
 self.orig\_image = self.image  
  
 *# Create mask from image. Mask contains coordinates of each pixel of the image in the environment.  
 # Required for pixel perfect collision detection at the cost of evaluation time.* self.mask = pygame.mask.from\_surface(self.image)  
  
 *# Create rectangle to store car coordinates* self.pos = [x, y]  
 self.rect = self.image.get\_rect()  
 self.rect.center = (x, y)  
  
 *# Store current angle car is facing.* self.angle = 0  
 *# Boolean to store whether the car has collided in the environment.* self.crashed = **False** *# Store location of obstacles and screen boundary to check for collisions* self.obstacles = **None** self.borders = **None** *# Store the solution this car represents.* self.solution = solution  
  
 *# Imputs and outputs for the neural network making decisions for this car.* self.inputs = **None** self.output = **None  
  
 def** rotate\_right(self):  
 *"""Rotate car right by maximum angle specified"""* self.angle = (self.angle - self.DELTA\_ANGLE) % -360  
  
 **def** rotate\_left(self):  
 *"""Rotate car left by maximum angle specified"""* self.angle = (self.angle + self.DELTA\_ANGLE) % -360  
  
 **def** move\_forward(self):  
 *"""Trigonometric method to determine new position based on the angle car is facing."""* dx = math.cos(math.radians(self.angle + 90))  
 dy = math.sin(math.radians(self.angle + 90))  
  
 *# Update position of car* self.pos = self.pos[0] + (dx \* self.SPEED), self.pos[1] - (dy \* self.SPEED)  
 self.rect.center = self.pos  
  
 **def** update(self):  
 *"""Overriding the default Pygame update method  
 Update mask and collision state"""* self.mask = pygame.mask.from\_surface(self.image)  
 self.crashed = self.check\_if\_crashed()  
  
 *# Helper methods to get and set neural network values* **def** set\_inputs(self, inputs):  
 self.solution.brain.set\_inputs(inputs)  
  
 **def** feed\_forward(self):  
 self.solution.brain.feed\_forward()

**def** get\_outputs(self):  
 **return** self.solution.brain.get\_decision()  
  
 **def** calculate\_fitness(self, time):  
 *"""Simple formula to determine fitness from time spent in environment  
 Can be adjusted to make fitness increase exponentially with time"""* fitness = 2\*time + math.pow(time, 2)  
 self.solution.fitness = math.pow(fitness, 0.5)  
  
 **def** draw(self):  
 *"""Method to draw car data on this frame to the Pygame screen.  
 Rotate image to the angle the car is currently facing."""* self.image = pygame.transform.rotate(self.orig\_image, self.angle)  
 self.rect = self.image.get\_rect(center=self.rect.center)  
 *# Rendering the image to the current coordinates of the car.* env.Environment.screen.blit(self.image, self.rect)  
  
 **def** check\_if\_crashed(self):  
 *"""Algorithm to check if car has crashed by checking overlaps"""* sample\_points = []  
 outline\_points = self.mask.outline()  
 *# Samples points by checking every tenth point in the car outline to reduce time required.* **for** i **in** range(len(outline\_points)):  
 **if** i % 10 == 0:  
 sample\_points.append(outline\_points[i])  
  
 *# Points need to be offsetted as mask does not store absolute position in environment.* **for** point **in** sample\_points:  
 offsetted\_mask\_point = [0, 0]  
 offsetted\_mask\_point[0] = point[0] + self.rect[0]  
 offsetted\_mask\_point[1] = point[1] + self.rect[1]  
  
 *# Checks for overlaps between sampled points to check if crashed into object* **if** self.check\_if\_point\_in\_any\_obstacle(offsetted\_mask\_point) **or** self.check\_if\_point\_in\_any\_border(  
 offsetted\_mask\_point):  
 adjusted\_rect = [offsetted\_mask\_point[0] - 25, offsetted\_mask\_point[1] - 25]  
  
 env.Environment.screen.blit(self.boom, adjusted\_rect) *# Display collision graphic at collision point if crashed* **return True**

*# Otherwise False: no collision* **return False  
  
 def** get\_arm\_distance(self, arm, x, y, angle, offset):  
 *"""Method to return sensor distance to objects"""* i = 0 *# Used to count the distance.  
 # Look at each point and see if we've hit something.* **for** point **in** arm:  
 i += 1  
 *# Move the point to the right spot.* rotated\_p = self.get\_rotated\_point(  
 x, y, point[0], point[1], angle + offset  
 )  
  
 *# Check if we've hit something. Return the current i (distance) if we did.* **if** rotated\_p[0] <= 0 **or** rotated\_p[1] <= 0 \  
 **or** rotated\_p[0] >= env.Environment.width **or** rotated\_p[1] >= env.Environment.height:  
 **return** i *# Sensor is off the screen.* **elif** self.check\_if\_point\_in\_any\_obstacle(rotated\_p):  
 **return** i  
  
 **elif** constants.DRAW\_SENSORS: *# Only render sensor arms if specified* pygame.draw.circle(env.Environment.screen, constants.BLACK, rotated\_p, 2)  
  
 *# Return the distance for the arm.* **return** i  
  
 **def** check\_if\_point\_in\_any\_border(self, point):  
 *"""Method to check for border intersection with a point"""* **for** border **in** self.borders:  
 **if** self.check\_inside\_rect(point[0], point[1], border):  
 **return True  
 return False  
  
 def** check\_if\_point\_in\_any\_obstacle(self, point):  
 *"""Method to check for obstacle intersection with a point"""* **for** obstacle **in** self.obstacles:  
 **if** self.check\_inside\_circle(point[0], point[1], obstacle.pos[0], obstacle.pos[1], obstacle.radius):  
 **return True  
 return False** *# Static helper methods to check for point intersections with circles and rectangles* @staticmethod  
 **def** check\_inside\_rect(x, y, rect):  
 **return** (rect[0] + rect[2]) > x > rect[0] **and** (rect[1] + rect[3]) > y > rect[1]  
 @staticmethod  
 **def** check\_inside\_circle(x, y, a, b, r):  
 **return** (x - a) \* (x - a) + (y - b) \* (y - b) < r \* r  
  
 **def** get\_sensor\_data(self):  
 *"""Method to get sensors readings for the car"""* **return** self.get\_sensor\_readings(self.rect.center[0], self.rect.center[1], math.radians(abs(self.angle) - 90))  
  
 **def** get\_sensor\_readings(self, x, y, angle):  
 *"""Return the values of each sensor in a list"""* readings = [] *# List to store each sensor value  
  
 # Make our arms* arm\_left = self.make\_sensor\_arm(x, y)  
 arm\_middle = arm\_left  
 arm\_right = arm\_left  
  
 *# Rotate them and get readings.* readings.append(self.get\_arm\_distance(arm\_left, x, y, angle, 0.75))  
 readings.append(self.get\_arm\_distance(arm\_left, x, y, angle, 1.55))  
 readings.append(self.get\_arm\_distance(arm\_middle, x, y, angle, 0))  
 readings.append(self.get\_arm\_distance(arm\_left, x, y, angle, -1.55))  
 readings.append(self.get\_arm\_distance(arm\_right, x, y, angle, -0.75))  
  
 **return** readings  
  
 @staticmethod  
 **def** make\_sensor\_arm(x, y):  
 *"""Method to create array of points representing one arm of sensor."""* spread = 16 *# Default spread between sensor points.* distance = 10 *# Gap before first sensor point.* arm\_points = []  
  
 *# Make an arm.* **for** i **in** range(1, 15):  
 arm\_points.append((distance + x + (spread \* i), y))  
 **return** arm\_points  
  
 @staticmethod  
 **def** get\_rotated\_point(x\_1, y\_1, x\_2, y\_2, radians):  
 *"""Algorithm to rotate a point by an angle around another point.  
 Rotate x\_2, y\_2 around x\_1, y\_1 by angle."""* x\_change = (x\_2 - x\_1) \* math.cos(radians) + \  
 (y\_2 - y\_1) \* math.sin(radians)  
 y\_change = (y\_1 - y\_2) \* math.cos(radians) - \  
 (x\_1 - x\_2) \* math.sin(radians)  
 new\_x = x\_change + x\_1  
 new\_y = y\_change + y\_1  
 **return** int(new\_x), int(new\_y)

## neural\_network.py

*"""Neural network module which can function individually from the other modules. The unique structure of a neural network represents one solution in the search space containing the ideal solution. This class is used by the environment to create a 'brain' for each car."""***import** math  
**import** random

### Class Connection

**class** Connection:  
 *"""Connection class to store connections between nodes in the network"""* **def** \_\_init\_\_(self, connected\_node):  
 self.connected\_node = connected\_node *# Node on the previous layer* self.weight = random.normalvariate(0, 1) *# Generating random weight from gaussian distribution (mu=0, sigma=1)*

### Class Node

**class** Node:  
 *"""Node class for network. These are connected to each other through the connection class."""* **def** \_\_init\_\_(self, prev\_layer):  
 self.connections = [] *# Store all connection objects to previous layer* self.output = 0.0 *# Default output value  
  
 # No connections if this is the first layer* **if** prev\_layer **is None**:  
 **pass  
 else**:  
 **for** node **in** prev\_layer:  
 *# Otherwise connect to each node of previous layer* connection = Connection(node)  
 self.connections.append(connection)  
  
 **def** feed\_forward(self):  
 *"""Propagate values through network to produce output"""* sum\_output = 0  
 **if** len(self.connections) == 0:  
 **return  
 for** connection **in** self.connections:  
 sum\_output += (connection.connected\_node.get\_output() \* connection.weight) *# Values scaled by weight.* self.output = self.sigmoid(sum\_output) *# Flattens output between 0 and 1.* @staticmethod  
 **def** sigmoid(x):  
 *"""Simple static method to flatten input between 0 and 1"""* **return** 1 / (1 + math.exp(-x \* 1.0))  
  
 **def** set\_output(self, output):  
 *"""Sets the output of this node; required to set the values of the input layer"""* self.output = output  
  
 **def** get\_output(self):  
 *"""Returns the output value stored in this node"""* **return** self.output

### Class Network

**class** Network:  
 *""" Network to act as 'brain' for each car. Created from the nodes and connections previously defined.  
 Requires a parameter, topology; a list which specifies the number of nodes on each layer.  
 eg. [2,3,1] creates a network with input layer of 2 nodes, hidden (middle) layer of 3 nodes and 1 node on the output layer."""* **def** \_\_init\_\_(self, topology):  
 *"""Generates the entire tree structure of the defined structure with random weights and biases."""* self.layers = []  
 self.topology = topology  
 layer\_number = 0  
  
 **for** layerNum **in** range(len(topology)):  
 *# Iterate through each layer in network, creating nodes and connections as required* layer\_number += 1  
 layer = []  
 **for** i **in** range(topology[layerNum]):  
 *# Iterate through each node on this layer* **if** len(self.layers) == 0:  
 *# Create a null node (node with no connections to previous layer) if this is the first layer* layer.append(Node(**None**))  
 **else**:  
 *# Otherwise connect this node to each node on previous layer* layer.append(Node(self.layers[-1]))  
  
 **if not** layer\_number >= len(topology):  
 *# Create a bias node on every layer except the last* layer.append(Node(**None**)) *# bias node* layer[-1].set\_output(1) *# setting output of bias node as 1* self.layers.append(layer)  
 *# Add layer to list of all the layers of the network* **def** print\_network\_structure(self):  
 *"""Method to output the number of nodes in each layer in the format [L1, L2, L3 … Ln] including the bias nodes for debugging purposes."""*

structure = []  
 **for** layer **in** self.layers:  
 structure.append(len(layer))  
 print(**"Initiated Neural network with structure: "**, structure)  
  
 **def** print\_network\_weights(self):  
 *"""Method to output the values of each connection in network for debugging purposes."""* **for** layerI **in** range(len(self.layers)):  
 layer = []  
 **for** node **in** self.layers[layerI]:  
 **for** connection **in** node.connections:  
 layer.append(connection.weight)  
 print(**"Layer "**, layerI, **": "**, layer)  
 print(**""**)  
  
 **def** get\_network\_weights(self):  
 *"""Returns the weights of the network in a specific order.  
 Required to retrieve weights when creating child networks from current one."""* layers = []  
 **for** layer **in** self.layers:  
 **for** node **in** layer:  
 **for** connection **in** node.connections:  
 layers.append(connection.weight)  
 **return** layers  
  
 **def** set\_network\_weights(self, weights):  
 *"""Sets the weights of this network to the ones provided in the list.  
 Required for setting weights of child network.  
 Order of weights same as returned by the method get\_network\_weights()"""* i = 0  
 **for** layer **in** self.layers:  
 **for** node **in** layer:  
 **for** connection **in** node.connections:  
 **if** i > len(weights) - 1:  
 **return** connection.weight = weights[i]  
 i += 1

**def** set\_inputs(self, inputs):  
 *"""Sets the values of the nodes on the input layer to the ones in the list provided"""* **for** i **in** range(len(inputs)):  
 self.layers[0][i].set\_output(inputs[i])  
  
 **def** feed\_forward(self):  
 *"""Propagates values through node"""* **for** layer **in** self.layers[1:]:  
 **for** node **in** layer:  
 node.feed\_forward()  
  
 **def** get\_results(self):  
 *"""Returns output value from the output layer"""* output = []  
 **for** node **in** self.layers[-1]:  
 output.append(node.get\_output())  
 **return** output  
  
 **def** get\_decision(self):  
 *"""Generates decision from value returned from the method get\_results()*

*makes choice depending on node with higher probability. If they are equal, goes straight"""* **if** self.layers[-1][0].get\_output() > self.layers[-1][1].get\_output():  
 **return "left"  
 elif** self.layers[-1][0].get\_output() < self.layers[-1][1].get\_output():  
 **return "right"**

## genetic\_algorithm.py

*"""This module contains all the genetic operator functions used for training and production of new generations."""***import** random  
  
**import** neural\_net **as** nn  
**import** constants

### Class Code

**class** Code:  
 *"""Code class to manage all data for one entity.  
 This unique code is what represents a solution."""* **def** \_\_init\_\_(self, topology):  
 self.topology = topology  
  
 *# A brain is the neural network specifically created from this code.* self.brain = nn.Network(topology)  
  
 *# Weights of neural network stored here so it can be easily modified for training* self.weights = self.brain.get\_network\_weights()  
  
 *# Fitness: calculated value representing how well this solution performs* self.fitness = **None** *# Time: Time spent in environment without crashing; used to calculate fitness.* self.time = 0  
  
 *# Bool to store whether this solution is the best of their generation* self.fittest = **False  
  
 def** set\_weights(self, weights):  
 *"""Setter method to set weights of the neural network"""* self.brain.set\_network\_weights(weights)  
 self.weights = weights

**def** generate\_population(pool\_size, topology):  
 *"""Function to generate population of random members"""* pool = []  
 **for** solution **in** range(pool\_size):  
 member = Code(topology)  
 pool.append(member)  
 **return** pool  
  
**def** produce\_child\_solution(first, second):  
 *"""Crossover to create one child base on probability."""* **if** random.uniform(0, 1) <= constants.CROSSOVER\_RATE:  
 *# Only creates child if required probability met.  
  
 # Determines random point in the list of weights to cross the data between parents (refer to design section).* crossover = random.randrange(0, len(first.weights))  
  
 *# Child created has same network structure* topology = first.topology  
  
 *# Function create to child code by crossing weights of parents from the point previously determined* offspring\_weights = first.weights[0: crossover] + second.weights[crossover:]  
  
 *# Code object created from newly created weights* child = Code(topology)  
 child.set\_weights(offspring\_weights)  
  
 **return** [child]  
 **else**:  
 **return** [first, second]  
  
**def** mutate(solution):  
 *"""Mutation of one solution to give random features"""* weights = solution.weights  
  
 **for** i **in** range(0, len(weights) - 1):  
 *# Scales some of the weights based on probability specified* **if** random.uniform(0, 1) <= constants.MUTATION\_RATE:  
 weights[i] = weights[i] \* random.uniform(0.5, 1.5)  
 solution.set\_weights(weights) *# Updates the weights of the solution* **return** solution  
  
**def** choose\_parent(population):  
 *"""Roulette parent selection  
 algorithm to choose an individual from population with members with higher fitness  
 having a greater probability of being selected."""* population\_fitness = 0  
 **for** i **in** population:  
 population\_fitness += i.fitness  
  
 pie\_size = population\_fitness \* random.uniform(0, 1)  
 *# Assigns a slice of radius proportional to members' fitness  
 # Random value determined in pie  
 # Member with landing slice is selected*

fitness = 0  
 **for** i **in** population:  
 fitness += i.fitness  
 **if** fitness >= pie\_size:  
 **return** i

## ML\_controller.py

*"""This is the Machine Learning controller module which is responsible for managing the entire training process.  
Uses the functions from the genetic algorithm module for training new generations and uses the environment module for evaluating their performance.  
This is essentially an extended application of the genetic algorithm"""***import** environment **as** env  
**import** genetic\_algorithm **as** ga  
**import** matplotlib.pyplot **as** plt *# For graphing the data***import** constants  
  
**def** reset\_fittest(population):  
 *"""Set all members' fitness to 0"""* **for** solution **in** population:  
 solution.fittest = **False  
 return** population

### Main training function

**def** train():  
 *"""Function that begins the training process"""  
 # Setup Graphing variables* generation\_fitnesses = [] *# To store average fitness for each generation (y coordinate)* figure = plt.figure(figsize=(6, 3))  
 fitness\_graph = figure.add\_subplot(1, 1, 1)  
 fitness\_graph.set\_xlabel(**"Generation"**) *# X axis label* fitness\_graph.set\_ylabel(**"Fitness"**) *# Y axis label  
  
 # Generate Population and set up simulation* population = ga.generate\_population(constants.POOL\_SIZE, constants.TOPOLOGY)  
 print(**"Initial population generated"**)  
  
 *# Display generated population* population[0].brain.print\_network\_structure()  
 **for** i **in** population:  
 i.brain.print\_network\_weights()  
  
 generation = 0 *# Stores the current generation of the training process* solution\_found = **False** *# Whether the minimum criteria for the solution met* print(**"Solution not found in initial population, Continuing evaluation…"**)  
 **while not** solution\_found:  
 *# Repeat evaluation and training process until suitable solution found  
  
 # Evaluate population fitness* print(**"Evaluating current population..."**)  
 env.Environment(population, generation)  
  
 *# Find best Code and show info* solution = population[0]  
 **for** solution **in** population:  
 **if** solution.fitness > solution.fitness:  
 solution = solution  
  
 *# Mark solution as fittest in population* solution.fittest = **True** print(**"Best solution located and marked at: "**, hex(id(solution)))  
  
 *# Calculate average population fitness* total\_population\_fitness = 0  
 **for** i **in** population:  
 total\_population\_fitness += i.fitness  
 avg\_fitness = total\_population\_fitness / len(population)  
  
 *# Visualization* generation\_fitnesses.append(avg\_fitness)  
 x = [i **for** i **in** range(0, len(generation\_fitnesses))] *# Generation number* y = generation\_fitnesses *# Average fitness achieved on this generation* fitness\_graph.clear()  
 fitness\_graph.set\_xlabel(**"Generation"**)  
 fitness\_graph.set\_ylabel(**"Average Fitness"**)  
 fitness\_graph.plot(x, y)  
 plt.pause(0.1) *# Required for rendering  
  
 # Display current generation info* print(**""**)  
 print(**"Generation: "**, generation)  
 print(**"Average generation fitness: "**, avg\_fitness)  
 print(**'Target Number: '** + str(constants.TARGET\_TIME))  
 print(**"Best Code fitness = "**, solution.time)  
 print(**""**)  
  
 *# Stop simulation if minimum criteria met* **if** solution.time >= constants.TARGET\_TIME:  
 solution\_found = **True** print(**"Minimum search criteria met, terminating simulation..."**)  
 **else**:  
 print(**"Solution criteria not met, producing new population..."**)  
  
 *# Generate new generation* new\_population = [] *# Stores members of the next generation* **while** len(new\_population) < constants.POOL\_SIZE: *# Keeps producing children until required population size met  
  
 # Choose two parents* parent1 = ga.choose\_parent(population)  
 parent2 = ga.choose\_parent(population)  
  
 *# Create and add one child to next population* childs = ga.produce\_child\_solution(parent1, parent2)  
 **for** child **in** childs:  
 **if** child **not in** population: *# Prevent identical children if crossover doesn't occur* child = ga.mutate(child) *# Apply mutation* new\_population.append(child)  
  
 *# Add best solution from last iteration* new\_population.append(solution)  
 population = new\_population  
 generation += 1  
  
  
print(**""**)  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 train()

# Testing

To test the functionality of my system, I will test each component of my system individually and compare them to the objectives initially discussed in the analysis and list the results in a table. I will then comment on the outcome of **certain tests** and provide screenshots as evidence.

The working of components that cannot be sufficiently demonstrated through screenshots will be shown in the video along with the test numbers.

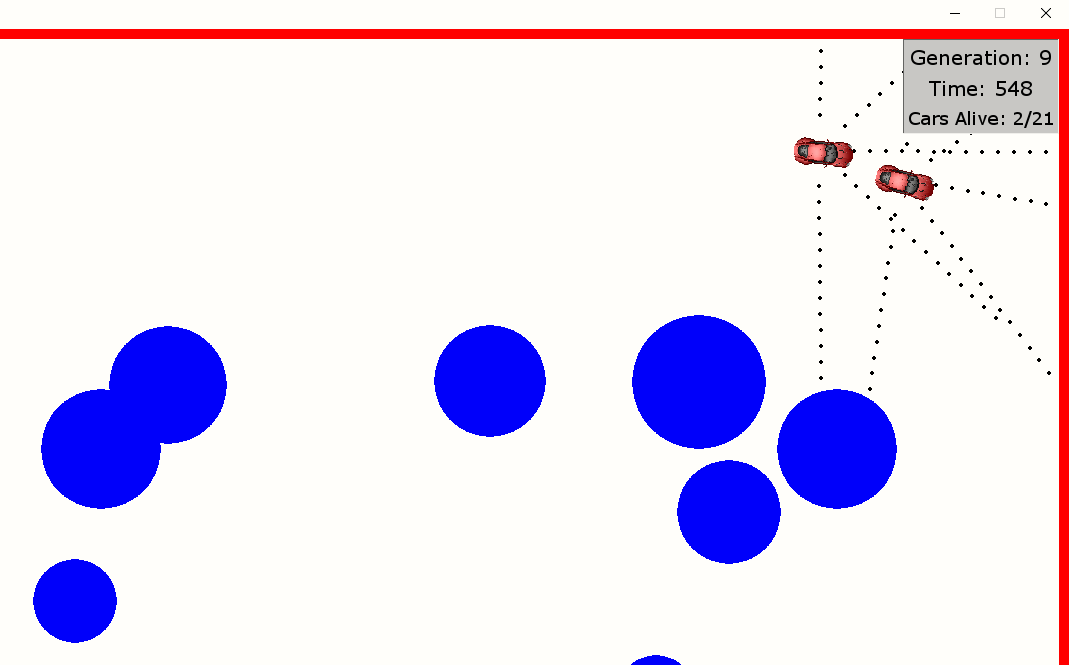
Each section is assigned a letter from A to E and each test within a section is assigned a number. E.g. The second test of user interface is test A2.

## A: User Interface

The user interface includes the **HUD** (**H**ead-**U**p **D**isplay) showing information about the current frame and Pygame window containing the environment.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Outcome |
|  | The HUD must be **compact**, covering no more than 20% of the width or height. | Both conditions are met. The HUD only takes up 1/8th of the screen width and height. |
|  | A **graph** for the average population fitness for each generation must be displayed and updated frequently. | A graph is plotted and updated with real-time data. |
|  | The UI must display useful information such as **current generation, time and number of cars** alive. | The current generation, time and number or cars alive on that frame are displayed in the HUD. |
|  | The UI must **update** each frame to display the information at that moment. A framerate of 30Hz is to be achieved. | The HUD is rendered properly and the information from the current frame displayed. |
|  | The Pygame window must be **responsive** and be able to exit without freezing. | The window is responsive at all times and is able to close with user input as demonstrated in the video. |

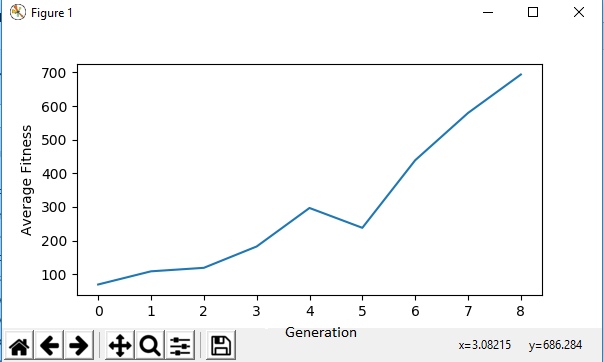
### Test 1,3,4: HUD Rendering



**1, 3:** The HUD is rendering properly in the correct position with the correct data as shown above. It displays the **current generation** of the simulation, the **time** the current generation has been simulated for in seconds and the **number of cars** still alive at that instant. The number of cars is accurately depicted as shown above. The HUD is also taking up minimal space as intended and not blocking any objects on the screen.

**4:** The image shows the correct number of cars and the correct generation number. This means the HUD has **updated** to display the current state of the environment correctly and the test has been successful. However, the testing video shows that the environment is not able to constantly render at **30Hz** (1Hz = 1 frame rendered per second) when there are too many cars to process.

### Test 2: Real-time graph



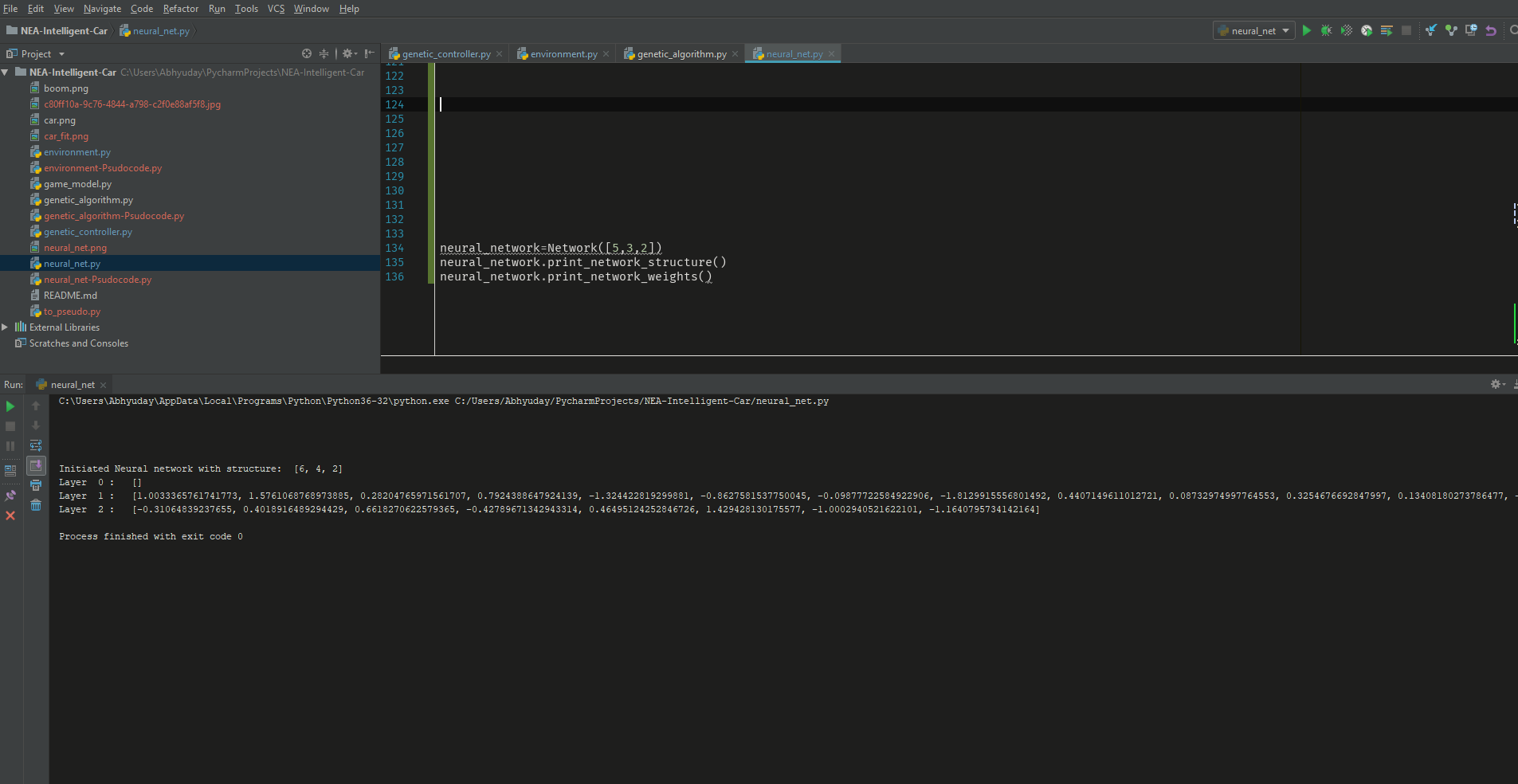
The effectiveness of a machine learning algorithm is more accurately represented through statistics. Therefore, a **graph** showing the result of each successive generation should provide evidence of the algorithm working. For this, a graph plotting the **current generation** against the **average fitness** which is a realistic figure of effectiveness should be displayed. This is working correctly as shown above.

## B: Neural network

The neural networks are the “brains” of this process and will be unique for each member. They will need to be generated, assigned to a member and updated with **improved weights** throughout the training process.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Outcome |
|  | A neural network should be able to **initialize** from the structure and weights provided. | Neural networks of specified structure and weights are generated. |
|  | A neural network should be able to **update** **all the** **weights** and the structure of its self when provided an array of weights. | The neural network is able to update its weights when provided with an array of floats. |
|  | **Randomly generated neural networks** should be created for each member in the current population | A unique neural network is generated for each with random values from a normal distribution as intended. |
|  | The neural network should be able to **evaluate an output** and return the decision to the environment (right or left). | The neural network produces an output of “left” or “right” from the data as intended. |

### Test 1: Initialisation

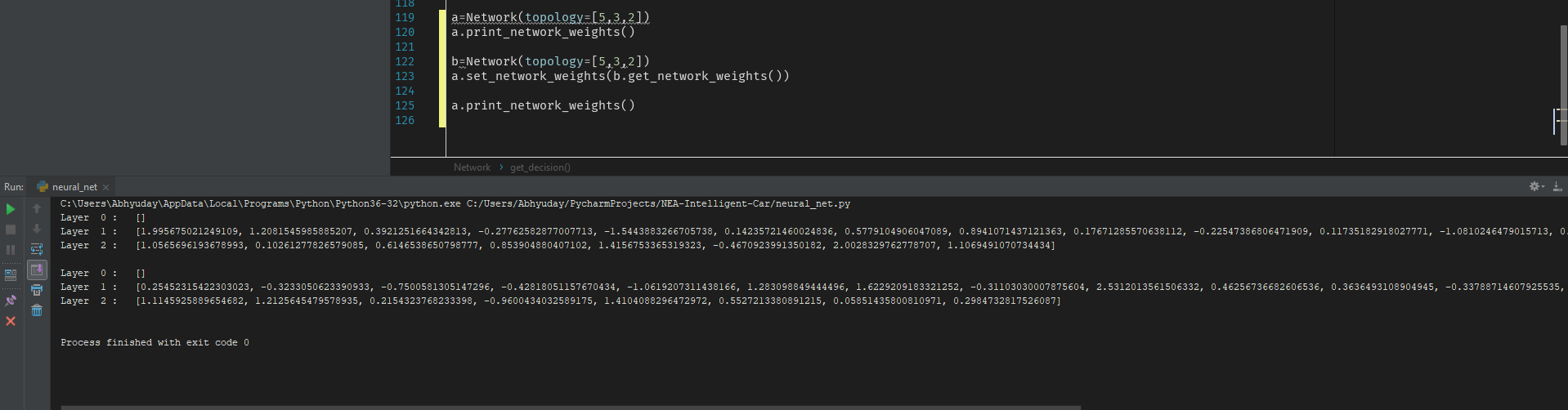


For the entire process to work correctly, the neural network class should be able to produce a neural network with the **correct structure** with the intended **number of nodes** in each layer and initial **configuration of weights** for the nodes. These should be inputs from the user and generated through the machine learning controller.

Note: In a neural network **[L1, L2, L3 … Ln]** the value of L1 represents the number of nodes on the first layer, L2, the number of nodes on the second layer and so on. The value n is the cardinal number representing the number of layers in the structure and could be any integer greater than 0.

As shown above, upon attempting to use the class to create a neural network with structure **[5,3,2]**, one is indeed produced (with an additional bias node on all but final layer producing one of [6,4,2].) The values of the weights are also displayed for each layer. These are all float values from a normal distribution with a mean of 0 as intended. These were **randomly generated** as should be for the first iteration.

### Test 2: Updating the network



For the training process, the genetic algorithm must be able to **produce new neural networks** with a specific configuration of weights. It will do this using the set\_network() method which takes an array of floats of same length as number of weights in the neural network and **set the weights** of a neural network to these. The results of the test are shown above where two neural networks a and b are created with random weights. The weights are displayed before and after updating the weights of a with weights of by using the set\_network() method. Since the initialising process has been tested to be working correctly, there is no need to check the weights of b as the only condition is that they are almost guaranteed to be random and therefore different from a. The test for this condition has been shown below.

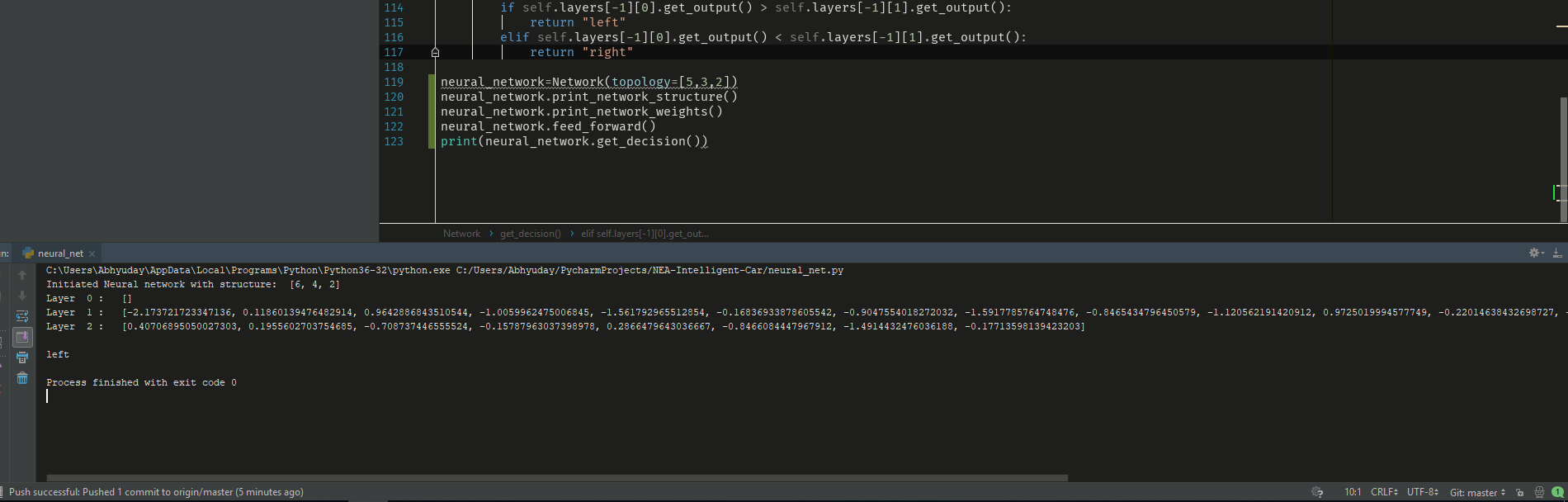
Note that the method print\_network\_weights() is also used but is not necessary in the actual system and has only been included for debugging purposes.

### Test 3: Weight generation:

For the algorithm to function correctly and for purposes of the previous test, the neural network weights have to be **generated randomly** on the first generation. This has been tested as shown above by generating several neural networks as they will be on the first generation. As it can be seen, all the weights are reliable random and **normally distributed** with a mean of 0 and standard deviation 1.

Note: Due to the way neural networks operate, it is convenient to have the weights distributed normally with a **mean of 0** and **standard deviation of 1**. Since values are multiplied by weights and this is essentially scaling the values without infinitely increasing or decreasing the values. This is not tested as the math.normal function is considered reliable and is used for this purpose.

### Test 4: Output evaluation:



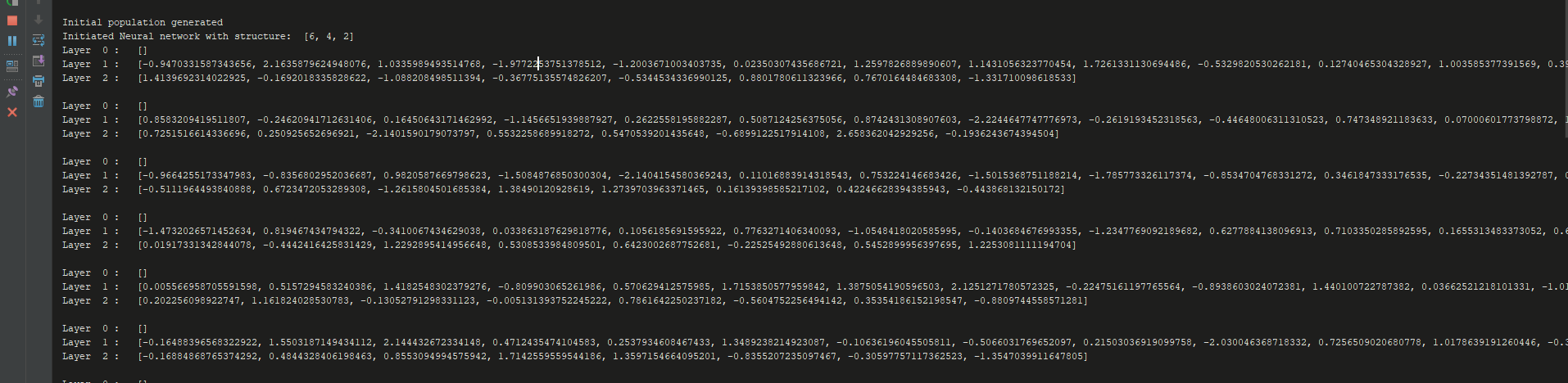
The neural network is responsible for making the decisions to steer the car either **left or right**, therefore, it should be able to produce a relevant output based on the data from the sensors. To test this, I made a randomly generated neural network print the decision which resulted in an **output of “left”** as shown above. This is working as expected and means that this decision can be then be evaluated in the environment.

## C: Genetic algorithm

The genetic controller is responsible for the training process. It should create a population of N members, evaluate them, breed the next generation of members and repeat this process until a suitable solution is found.

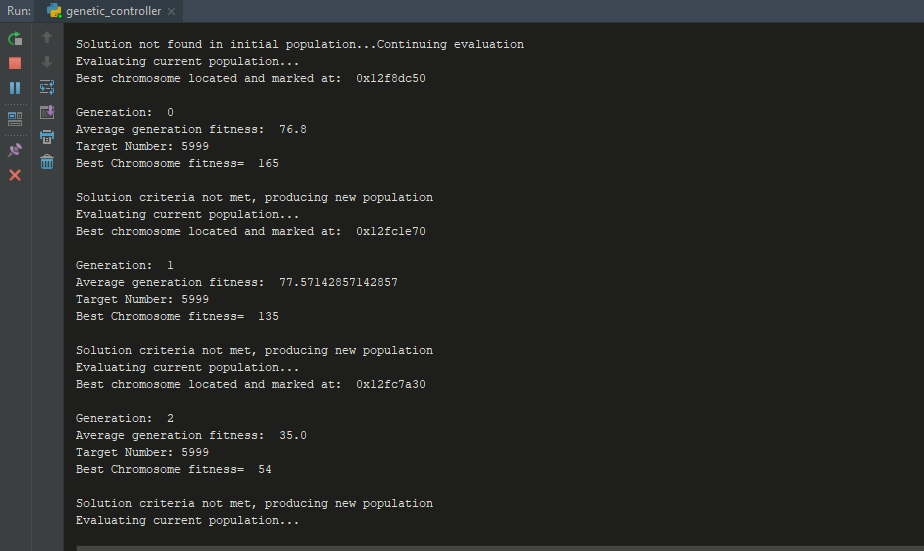
|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Outcome |
|  | The genetic algorithm should be able to **initialize** a population, each with their neural network when provided with the number of members, N. | N number of members with neural networks are initialised. |
|  | The fittest member (best solution) from the previous generation should be explicitly **marked** and shown in the next generation. | The strongest member is marked, and relevant data is displayed. |
|  | Crossover and mutation should also occur with the **probability** that is specified. | Crossover and mutation occur to produce each child from two parents. |
|  | The simulation process should run **indefinitely** until the minimum search criteria has been met. | The simulation runs until the search criteria has been met. This is demonstrated in the video. |

### Test 1: Initialisation



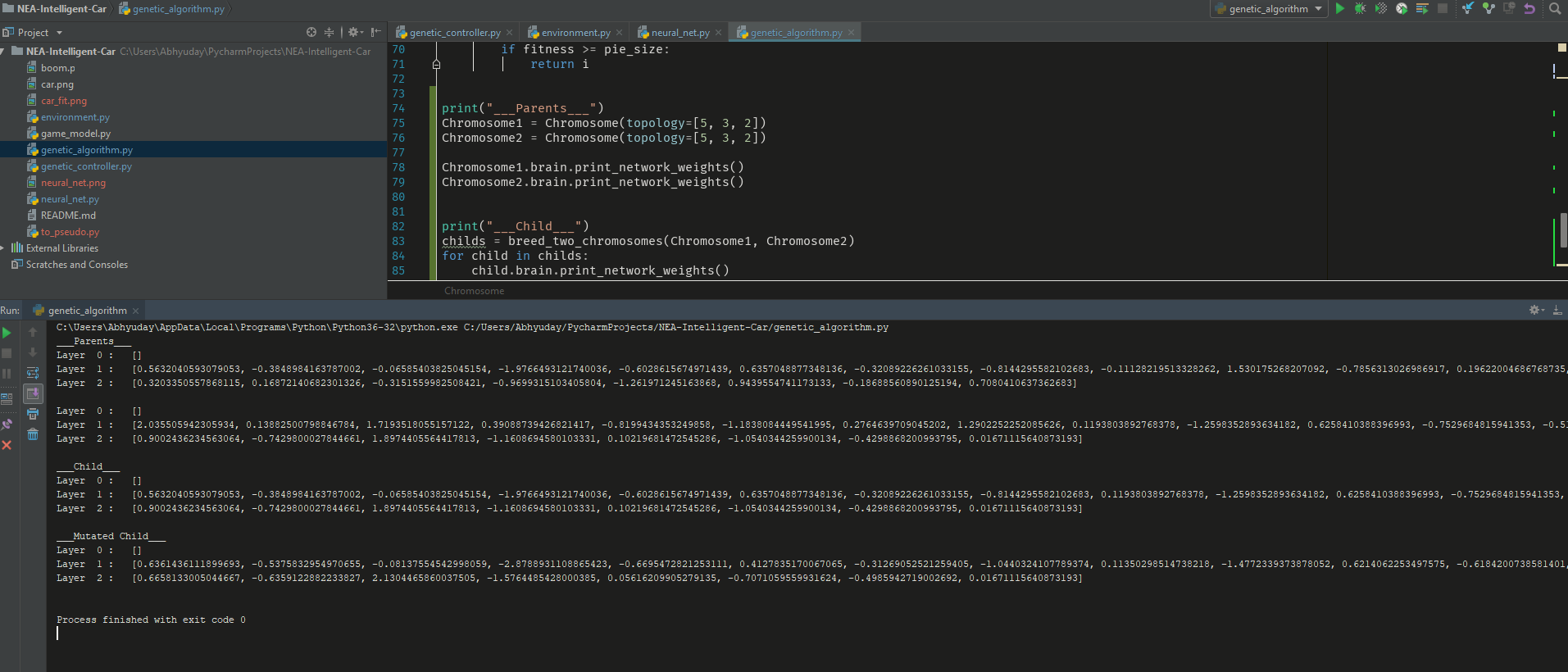
As shown above, it is able to create a random population on the first generation with N members as specified by the user. Each member is given a **uniquely generated neural network** from the **neural\_network.py class**. No further testing is required as this section heavily relies on the initialisation of the neural network which has been tested to be working.

### Test 2: Member marking



The genetic algorithm should be able pick out the best member from each generation since the training process relies on being able to pick out the stronger members. As an extension and as evidence that the selection process is working the algorithm is able to **mark the strongest member** from the **previous generation** and display the car object id as shown above. This is further demonstrated visually in the environment testing section and video.

### Test 3: Crossover and mutation



Crossover and mutation are key processes which aim to create better individuals in future generations. This algorithm should be able to create **one child from two parents** and mutate it depending on a **user-defined probability** of the process occurring. This process should produce a child with similar code to the parents but slightly modified due to the mutation to introduce variance.

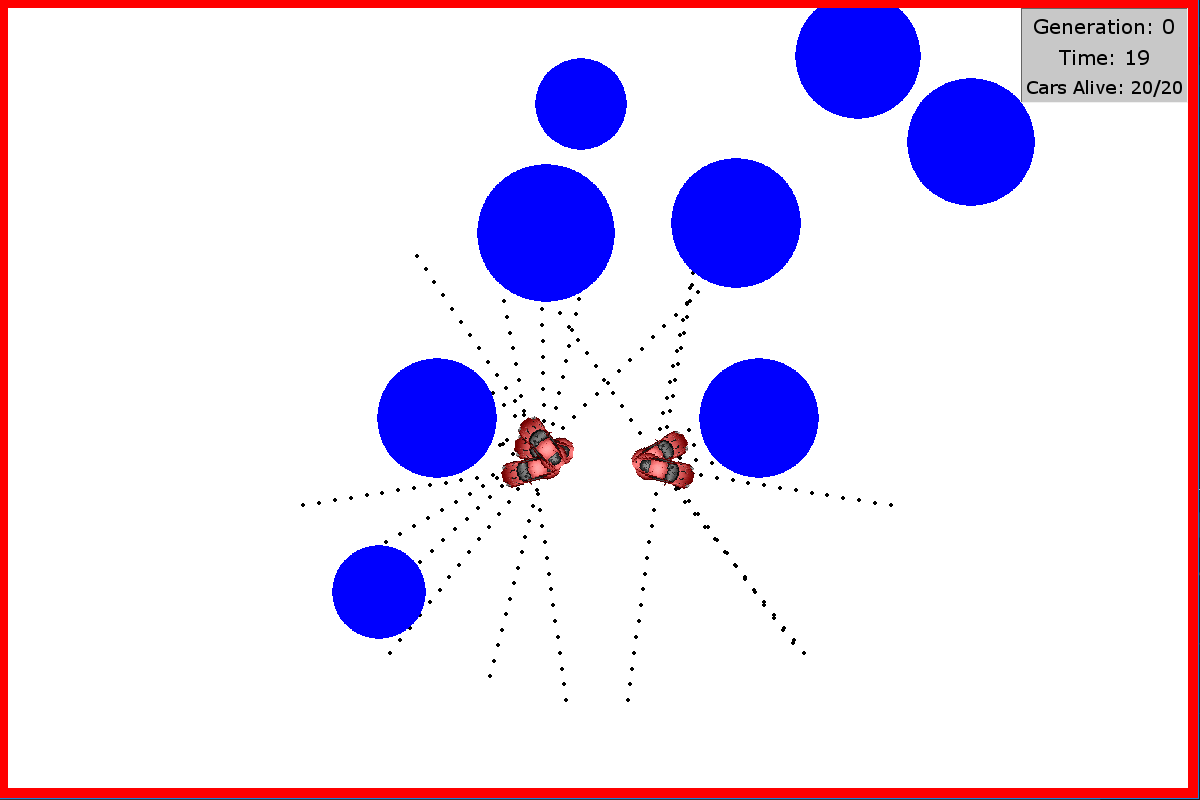
This works as intended as shown above where two randomly generated parents are created and crossover and mutation are applied to their child. As shown above, the child produced has weights that are **slightly different** to that of the parents. The weights are again **different** after it has been mutated. Hence, the breeding algorithm is working correctly.

## D: Environment

The environment is where the training process for each generation occurs and is rendered for the user to see. It should be able to render all the obstacles, vehicles, sensors, HUD and check for collisions. It will then send the evaluated data back to the genetic algorithm to process.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Outcome |
|  | Obstacles will be **generated randomly at the start**. They will be initialized with a random velocity at random points with random radiuses.  They will also not spawn in the immediate vicinity of the car spawn area to prevent an instantaneous collision. | Obstacles are generated randomly in the environment as required. |
|  | **Collision detection** with obstacles must be **pixel-perfect** to ensure the AI trains correctly. For collision checking, all the mathematical functions must return accurate values. | Collision is pixel perfect and explosion graphic is shown on collision. |
|  | The **sensors** from the cars should also return a correct value (the number of points in the sensor ray that do not collide). | Sensors return accurate data from environment to genetic algorithm. |
|  | The **decision** from the neural network must be evaluated correctly; turning and forward acceleration should be visible in the environment. | The cars are able to freely accelerate forwards in the environment. The decision from the neural network is accurately evaluated by the environment. They steer either left or right depending on the decision calculated. This is demonstrated in the video. |

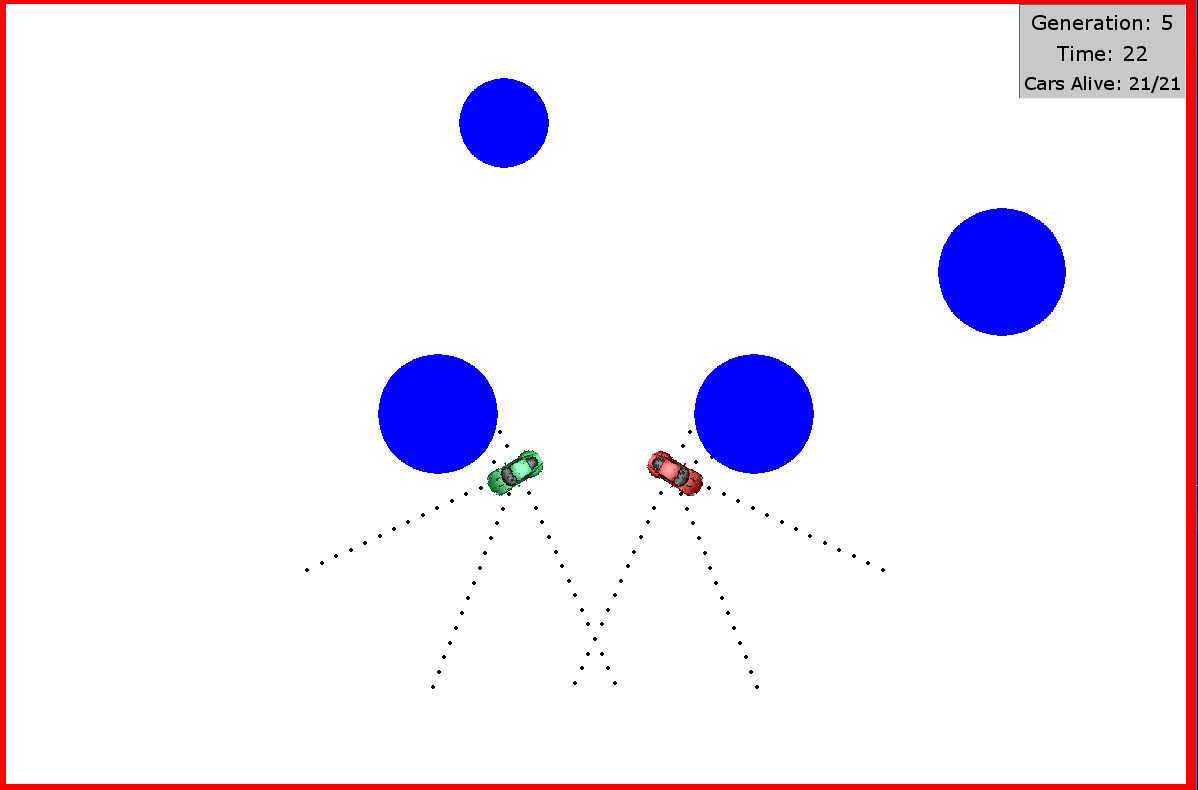
### Test 1: Initialisation:



Shown above is an image taken 19 frames after initialisation of the first generation as shown in the HUD (as one-unit time represents one frame). For the training algorithm to work, the environment has to produce **random elements**, so the AI can learn a variety of patterns for collision avoidance.

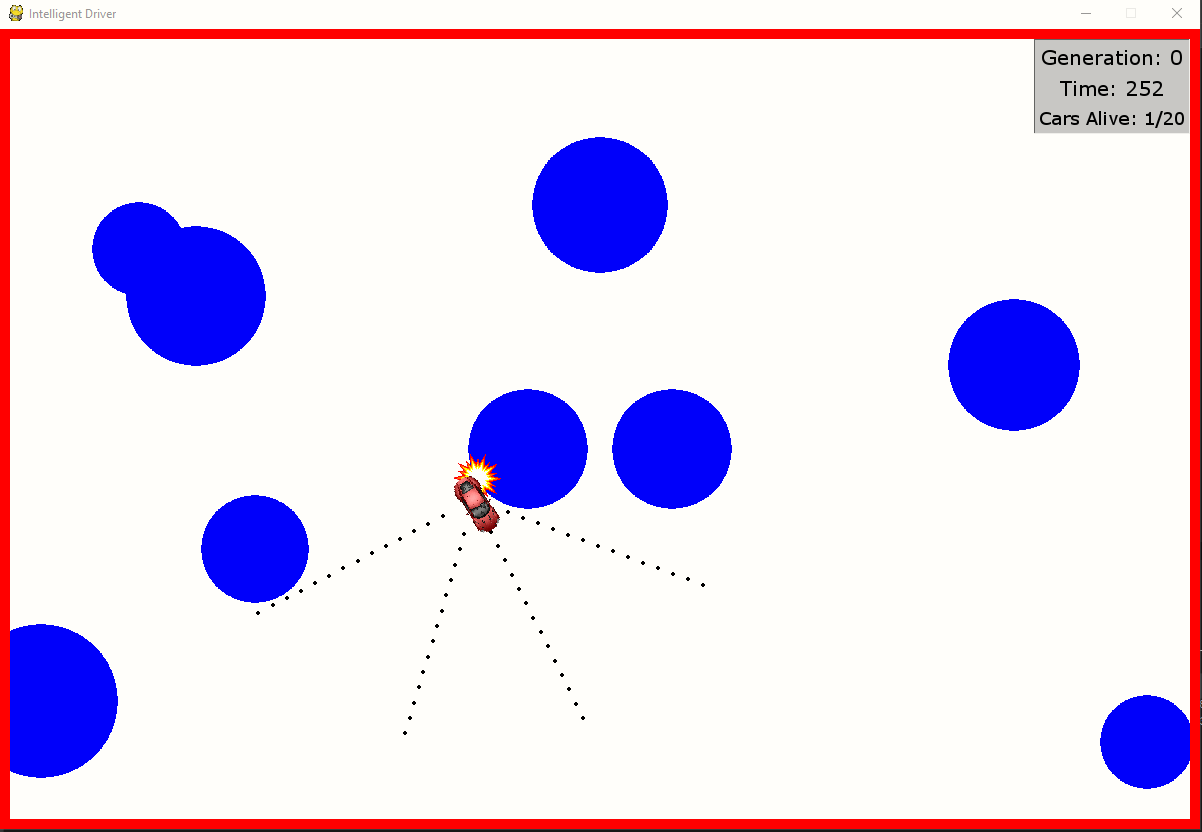
Therefore, a random number of obstacles of varying sizes are generated with random velocities in random positions. The sizes and positions of the obstacles are indeed different. As shown in the image below taken right after initialisation of a separate generation. It can be seen that the **generation is random** since all the factors are different in this instance.

It can also be seen in both images that that obstacles do not spawn in the immediate vicinity of the car spawn area as intended to **prevent instantaneous collisions**.



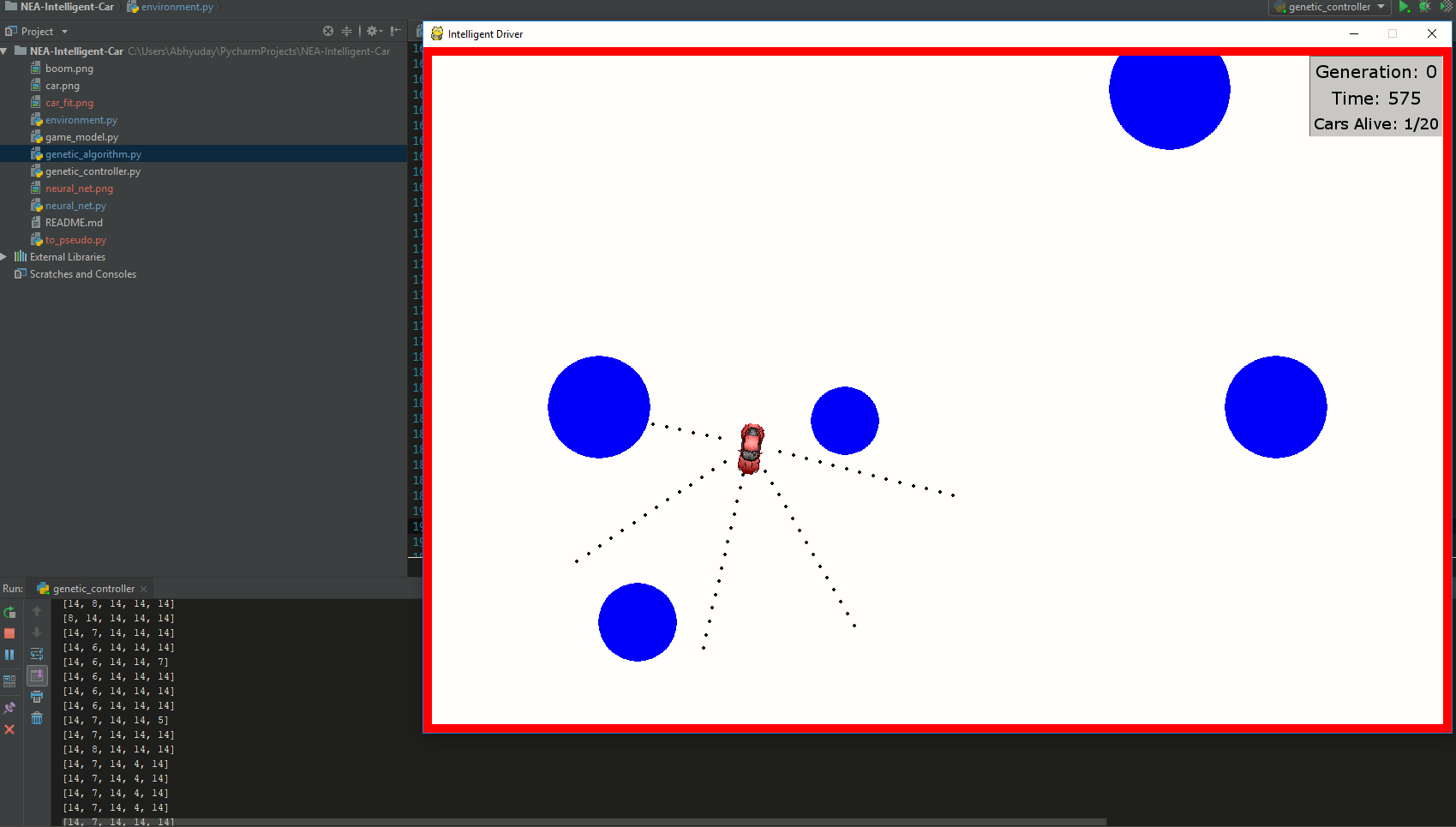
Following on from **member marking** in the testing of the genetic algorithm, the environment is successfully able to render the strongest member from the previous generation in a different colour to differentiate from the others. This further proves that the marking is working, and the genetic algorithm is able to utilise the environment properly.

### Test 2: Collision detection



The collision detection system has to be perfect as this is the main aspect of the software. Therefore, the environment should check if any pixel intersects with the obstacles. This is working perfectly as shown in the image above. The environment detects the collision between the car and the obstacle and car and displays an explosion graphic at the pixel where the collision occurred. It is then able to remove that car from the simulation and return its performance data as intended. The accuracy of **pixel perfect collision detection** will be demonstrated further in the video.

### Test 3: Sensor data



The main way that the environment is connected to the neural networks are through the **sensor data** from cars serving as the **inputs for the neural network**. This data is in the form of an array of length N, which is the number of sensors of a car and the number of inputs for the neural networks.

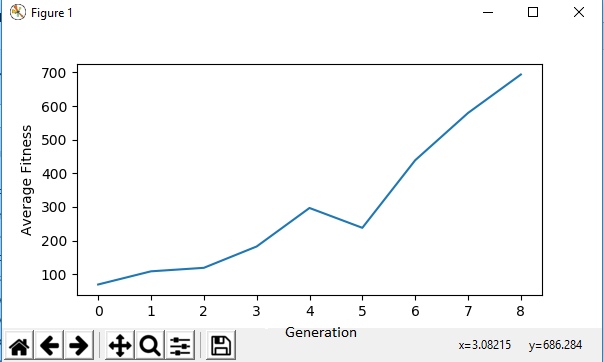
Each integer in the array represents the number of dots not intersecting an obstacle for an arm of the proximity sensor. The environment returns a list of 5 integers as expected due to the **5 sensors** for the car. The last row of the test data represents the current frame and as expected only one of the integers is 7, while the others are 14 as they do not intersect, the max range of the sensor. Therefore, the environment is returning the sensor data accurately.

## E: Evidence of Machine learning/Artificial Intelligence

In this section I will test the success of the overall machine learning algorithm by evaluating the results of successive generations.

|  |  |  |
| --- | --- | --- |
| Test No. | Objective | Outcome |
|  | There must be evidence that the machine learning algorithm is **improving**. | Real-time fitness graph proves effectiveness. |
|  | There must be evidence that the **entire population is learning** from previous generations. | On each successive generation, the number of cars surviving due to learning is increasing as demonstrated in the video. |
|  | The artificial intelligence must be **capable of driving** the cars in the environment on its own. | The artificial intelligence can drive and manoeuvre around obstacles on its own. This driving is improving with each successive generation as shown in the video. |

### Test 1,2: Effectiveness of machine learning



Since it is difficult to measure the effectiveness of machine learning using traditional methods, the best way would be to look at **statistics of its performance with time** of one specific training case. Above is a graph where a population of **50 members** are trained for **8 generations**, each lasting a **maximum of 820 seconds**.

As the graph of average fitness against generation number shows where the fitness represents the number of seconds survived by the cars, the average fitness increases substantially with each generation and reaching 700 by the 8th generation. As each generation takes a maximum of 820 seconds, achieving 700 in 8 generations is remarkable and shows that the algorithm is working well. The **overall increasing gradient** of the graph in this range also proves that the algorithm is working.

Note the decreased fitness on generation 5. This is normal for such algorithms as unexpected scenarios can occur due to the random nature of the simulations. The most important point is that the performance should be increasing in general over a large range.

## Video evidence

Working of components which are hard to test through screenshots have also been tested in a video. Some sections such as extended training have been sped up. Note that not all the sections have been covered.

# Evaluation

With all aspects of my project covered, I will now evaluate the success considering my **initial objectiv**es and the **scope of the problem solved**. I will also describe some **improvements** that could be made. This could be contributed by others once the software is **open sourced**.

## User Interface

The HUD was intended to be **simple**, **responsive** and able to **display** any necessary information about the environment. These objectives have been met as the HUD is displayed in the top right corner of the screen and does not take up much of the screen. The current generation, time and number or cars alive on that frame are displayed in the HUD. The HUD has a 60Hz refresh rate, in sync with the environment rendering engine. This provides real-time information to display the information for that each frame.

The HUD, however, does not display more detailed info such as the **structure** **of the neural networks** used or sensor data for the cars. Another improvement would be the addition of buttons to process user input, such as pressing a button or detect the pressing of a key to take further action such as speeding it up or loading a file. However, the program can easily be modified due to its **modular design** to be display any information necessary for training.

The graph has met my objectives as it displays precisely what is needed to monitor overall success from a higher level by only observing the results from successive generations. This information allows the user to notice the effects of their inputted parameters. I think this can be improved by again, providing more specific information to the user. There could be **additional graphs** to show best fitness from the population, or to visualize individual performance of each car. This could be further improved by providing more precise details about the environment such as sensor data from each car and colour coded information.

The Pygame window is able to respond to user inputs without any delay. This is achieved by using an **event queue** which is processed after rendering each frame, i.e. 60 times a second. Hence, the Pygame window is extremely responsive to user input even during the training process. This, however, is not true for the matplotlib graph visual as it processes user inputs in its own thread. This occasionally results in the graph crashing when commands are sent when it is rendering. To improve my system, I could either create my own graph plotting module or introduce an event queue for the graph to **improve responsiveness**.

## Neural Network

The neural network functions perfectly as it was designed as a **standalone module** to act as a ‘brain’ for each car. It processes the data from the sensor inputs and outputs a decision to steer the car as intended. The decision is specific to this scenario and can be modified to represent a decision for any scenario my changing the **get\_decision()** method.

Currently, the actions the network can perform are **limited to steering**. Even though this is fine for a simple training model, this would not be accurate for a real-driving case. One major improvement could be allowing more precise control by allowing the neural network to **accelerate, brake and stop** in the environment. This would however require an overall improvement of the system as more information would be required by the neural network to compute more precise controls.

The neural networks are initialised with random numbers from a **normal distribution** for each weight in their structure as intended to produce a random initial population. To further improve my algorithm, I could consider a **better initial population** by starting with the most optimum combination of weights for the neural networks. This could be achieved by using statistical models to limit the search space and decrease the time required to find the solution. However, this would be unnecessary for the scope of this project and could be added for training more complex networks.

## Genetic Algorithm

The genetic algorithm is responsible for running the entire process until a suitable solution is found. It is able to create an **initial population** of neural networks, **evaluate** them and use **crossover** to create a better population. The techniques used in a genetic algorithm are hard to evaluate as there is no perfect solution. Therefore, the only way is to evaluate the results produced and tweaking the variables until successful results are produced.

The genetic algorithm **explicitly marks** the best solution from the previous generation by colouring its corresponding car in green. An improvement on this system could be to extend the marking of solutions to **different species**, and colour coding them differently to differentiate between the diversity of techniques the algorithm develops to drive. This would allow the user to choose from different methods of collision avoidance. For example, one model might choose to avoid obstacles while maintaining the **shortest distance** to it as possible whereas another might attempt to get as **far from them** as possible.

Crossover and mutation are responsible for creating a similar but slightly different and better child for the next generation. This is functioning as intended as demonstrated in testing and resulting in improving the solution each generation. The probability values and specifics of the algorithm can be modified for the problem that is being undertaken. There is no perfect crossover or mutation algorithms hence, they will need to be tested for the use case and modified accordingly.

The algorithm runs **indefinitely** until a suitable solution is found. Therefore, depending on the minimum criteria specified by the user, the algorithm could take a different amount of time to produce a solution. The time taken to produce a solution heavily varies with the problem complexity and changes in variables. For a problem the variables such as POOL\_SIZE and TIME\_LIMIT will need to be modified to find a solution in an acceptable amount of time.

## Environment

Obstacles are generated randomly on each generation. The obstacles also spawn outside the car spawn area to prevent collision on initialisation. This reduces anomalies caused obstacles spawning on top of cars. **Pixel perfect collision detection** is achieved in the environment allowing accurate training of the algorithm. The correct translation of the data such as car heading and coordinates into the environment also suggest that all the mathematical functions used to perform the calculations are working correctly. The collision detection can be optimised further to reduce the training time as most of the time is spent evaluating this. Additional techniques such as processing overlapping cars only once can reduce the overall computation time for the algorithm.

The sensors return the correct values as demonstrated in testing. The neural networks can establish a pattern in the inputs from the sensors. This suggests that the sensors in the environment are working correctly. This can be further improved by providing more data such as position and angle to the neural networks. This might allow the algorithm to establish better patterns in the data.

As shown in the video, the cars are controlled completely by the decisions made by the neural networks. The environment is able to represent this decision in the environment accurately. As discussed previously, additional functionality can be added to the system to allow the AI more precise control of the driving such as **acceleration, braking and stopping**.

The environment used to train this model has been created in Pygame. Due to this, the physics in the environment are very simple and will need to be greatly improved for a real use case. For this, more complicated environment engines such as Unity3D can be used. Due to the **modular nature** of my code, the training and neural network modules can be ported to any other environment.

## Conclusion

Overall, I think my objectives were appropriate for this project as they were able to challenge me and keep me interested throughout development. I think these objectives were met to a suitable extent to create a complete system that performs exactly what it is supposed to.

I was able to produce a **modular program** that can create **neural networks**, represent them in an environment with **simple physics**, **evaluate** their performance at a task, **train** them using an **evolutionary algorithm** and export this solution to **any other environment**. This will allow anyone to use my system to achieve any of those tasks. I will make the **GitHub** repository public once this is released along with API documentation, so it is easy to use.

This project has allowed me to learn more about this field and provided me the opportunity to create something that can hopefully benefit many others.

# References

Anon., 2018. *Machine Learning.* [Online]   
Available at: https://en.wikipedia.org/wiki/Machine\_learning

Bensinger, G., 2018. [Online]   
Available at: https://www.wsj.com/articles/video-shows-final-seconds-before-fatal-uber-self-driving-car-crash-1521673182

Błażej Osiński, K. B., 2018. *deepsense.* [Online]   
Available at: https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/

David Rumelhart, G. H. a. R. W., 1986 . [Online]   
Available at: http://www.nature.com/nature/journal/v323/n6088/pdf/323533a0.pdf

Dowling, T., 2017. *The Guardian.* [Online]   
Available at: https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk

Fridman, L., 2018. *Autonomous Car.* [Online]   
Available at: https://arxiv.org/pdf/1711.06976.pdf

Gates, G., 2017. *New York Times.* [Online]   
Available at: https://www.nytimes.com/interactive/2016/12/14/technology/how-self-driving-cars-work.html?\_r=0

Raibert, M., 2006. *BigDog.* [Online]   
Available at: https://www.cs.swarthmore.edu/~meeden/DevelopmentalRobotics/bigdog.pdf

Stanley, K. O., 2002. [Online]   
Available at: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

Topham, G., 2017. [Online]   
Available at: https://www.theguardian.com/business/2017/nov/06/government-urged-to-plug-insurance-black-hole-over-self-driving-cars

Wagenaar, T., 2017. *towardsdatascience.* [Online]   
Available at: https://towardsdatascience.com/neuro-evolution-on-steroids-82bd14ddc2f6

Warsaw, B., 2001. [Online]   
Available at: https://www.python.org/dev/peps/pep-0008/