Intelligent Systems

Academic Report

Table of Contents

[1 Introduction 4](#_Toc71802130)

[1.1 Project Outline 4](#_Toc71802131)

[1.2 Report Outline 4](#_Toc71802132)

[1.3 Development Tools 4](#_Toc71802133)

[2 Theoretical Background 5](#_Toc71802134)

[2.1 Genetic Algorithms 5](#_Toc71802135)

[2.1.1 Fitness Function 5](#_Toc71802136)

[2.1.2 Selection 5](#_Toc71802137)

[2.1.3 Crossover 6](#_Toc71802138)

[2.1.4 Mutation 7](#_Toc71802139)

[2.2 Neural Networks 9](#_Toc71802140)

[2.2.1 Neurons and Layers 9](#_Toc71802141)

[2.2.2 Feed-forward and Backpropagation 10](#_Toc71802142)

[2.2.3 Activation Functions 11](#_Toc71802143)

[2.2.4 Learning Process 14](#_Toc71802144)

[3 Unity Engine 15](#_Toc71802145)

[3.1 The Car 15](#_Toc71802146)

[3.2 The Track 16](#_Toc71802147)

[3.3 The GA Object 17](#_Toc71802148)

[4 Implementation 18](#_Toc71802149)

[4.1 The C# Language 18](#_Toc71802150)

[4.2 The GeneticAlgorithm Class 19](#_Toc71802151)

[4.2.1 Variables 19](#_Toc71802152)

[4.2.2 Methods and Functions 20](#_Toc71802153)

[4.3 The NeuralNetwork Class 23](#_Toc71802154)

[4.3.1 Variables 23](#_Toc71802155)

[4.3.2 Initialising the network 23](#_Toc71802156)

[4.3.3 Randomising Weights 24](#_Toc71802157)

[4.3.4 The Sigmoid Function 25](#_Toc71802158)

[4.3.5 Running the network 25](#_Toc71802159)

[4.4 The Car Controller Class 25](#_Toc71802160)

[4.4.1 Variables 26](#_Toc71802161)

[4.4.2 Functions 27](#_Toc71802162)

[5 Results 29](#_Toc71802163)

[5.1 Testing and Analysis 29](#_Toc71802164)

[5.1.1 Selection and ANN Architecture 29](#_Toc71802165)

[5.1.2 Epoch 30](#_Toc71802166)

[5.1.3 Measuring Performance 30](#_Toc71802167)

[6 Conclusion 31](#_Toc71802168)

[6.1 Overview 31](#_Toc71802169)

[6.2 Improvements 31](#_Toc71802170)

[6.2.1 More Meaningful Results 31](#_Toc71802171)

[6.2.2 Saving the Network Values 32](#_Toc71802172)

[6.2.3 The Environment 32](#_Toc71802173)

[6.3 Potential Uses 32](#_Toc71802174)

[7 Further Discussion 33](#_Toc71802175)

[7.1 Notes on GA & ANN 33](#_Toc71802176)

[7.2 Genetic Algorithms: Advantages and Disadvantages 33](#_Toc71802177)

[7.3 Neural Networks: Advantages and Disadvantages 33](#_Toc71802178)

[8 Bibliography 34](#_Toc71802179)

# Introduction

## Project Outline

For this project, a self-driving car simulator was built. Two key algorithms were used for this project: the genetic algorithm (GA) and artificial neural network (ANN). The goal was to get the car to complete a track by itself. Normally, the ANN alone would suffice, however a different approach was taken for solving the task – GA is not commonly used in conjunction with ANN, but it was nevertheless an interesting experiment.

## Report Outline

The report covers the background theory of these algorithms, it introduces some of the tools the game engine has to offer, the implementation with snippets of code included, how the virtual space was created (for example tracks, how the car interacts with the track, etc), results, and a conclusion, which also covers potential improvements and further developments.

The report describes, in detail, how GA and ANN work and how they can be used together to meet the goal of the project.

## Development Tools

The Unity Engine was used to realise the project, and the scripting was done in C#. All assets and plugins are available for free.

This engine is excellent for creating AI-related projects as Unity Technologies has published several tools (ML Agents e.g.), however the project was built from the ground up, the only ready-to-use assets/components were the car, paint for the grass, paint for the road, and a mathematics plugin.

# Theoretical Background

## Genetic Algorithms

There are some problems that cannot be solved using traditional algorithmic approaches. Examples include the knapsack problem, SEND+MORE=MONEY, game theory equilibrium, ML feature selection, scheduling, and many others. GA is excellent for optimisation problems in general and can be applied along with other algorithms to boost efficiency.

The first step in the process is to create a population. Each chromosome (individual) is evaluated by a fitness function, following the evaluation the selection takes place. A certain number of chromosomes are selected (with the guidance of the fitness function) for reproduction. The reproducing chromosomes are the parents, and these produce two offspring or child chromosomes. The children do not inherit the exact same features or traits of the parents because this way the GA wouldn’t converge to a solution. The first step in diversifying the new generation is the application of crossover. Crossover allows for a child chromosome to retain its parents’ traits without keeping them exactly the same: both children should have traits inherited from both parents. To further evolve the generation, a small chance of mutation is applied on each chromosome. Mutation diversifies the population even more and is a good tool for exploring the solution space.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Generation |  |  |
|  |  |  |  |  |
| Mutation |  |  |  | Evaluation |
|  |  |  |  |  |
|  |  |  |  |  |
|  | Crossover |  | Selection |  |

*Figure 2.1: GA cycle*

### Fitness Function

The fitness function determines how close the solution is and helps the algorithm to converge to an optimal solution. There is no exact definition for the fitness function; its features depend on the problem domain and how the GA is customised. Generally, there are two types of behaviours a fitness function can follow: one where the function is fixed, and one where the function is mutable. The fitness function always seeks to identify the best set of solutions within a current population for the selection process.

### Selection

GA is inspired by biology, more precisely evolution theory – process of genetic mutation, selection, and generation. In nature, natural selection dictates that the fittest candidate is selected for reproduction, meaning the candidate has the most optimal genes that can be passed on to the next generation to promote the species’ survival. In GA, this is the *selection* process. The algorithm picks the best individual with the closest solution – the individual is called the *chromosome*.

### Crossover

Chromosomes that have been selected for reproduction create the offspring. The children of the parent chromosomes will inherit the *genes* (traits) of the parents; however, the inherited sequence must not be identical – the next generation will not offer new solutions this way. There are different ways of performing crossover: it can be single-point, two-point, k-point, or uniform crossover. A point in the sequence is chosen, then the offspring’s traits are swapped. (TutorialsPoint, 2021) Let’s take an 8-bit chromosome as an example to show how each type of crossover works:

* Single-point crossover: parents are selected. After selection, the point of crossover (point A) is chosen randomly. Child A will have a matching sequence of genes as Parent A until the sequence reaches Point A. Same applies to Child B and Parent B. After Point A, Child A’s gene sequence will match with Parent B’s genes, Child B’s genes will match with Parent A.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | Point A |  |  |  |  |
| Parent A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Parent B | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
|  |  |  |  |  |  |  |  |  |
| Child A | 0 | 1 | 2 | 3 | 104 | 105 | 106 | 107 |
| Child B | 100 | 101 | 102 | 103 | 4 | 5 | 6 | 7 |

*Figure 2.1.3.1: single-point crossover*

* Two-point crossover: parents are selected. After the selection process, 2 random points of crossover are chosen. It is a similar process to single-point crossover, except two points are chosen. K-point crossover is a multi-point crossover, where K is a positive integer. It is more suitable for larger gene pools.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Point A |  |  |  | Point B |  |  |
| Parent A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Parent B | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
|  |  |  |  |  |  |  |  |  |
| Child A | 0 | 1 | 102 | 103 | 104 | 105 | 6 | 7 |
| Child B | 100 | 101 | 2 | 3 | 4 | 5 | 106 | 107 |

*Figure 2.1.3.2: two-point crossover*

* Uniform Crossover: each gene has a chance for being chosen for crossover. This will result in children with highly random traits. There can be multiple crossover points, or there can be none, it is left to chance.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Point A |  | Point B | Point C |  |  | Point D |
| Parent A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Parent B | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
|  |  |  |  |  |  |  |  |  |
| Child A | 0 | 1 | 102 | 103 | 104 | 105 | 6 | 7 |
| Child B | 100 | 101 | 2 | 3 | 4 | 5 | 106 | 107 |

*Figure 2.1.3.3: uniform crossover*

### Mutation

Each child has a chance to have its traits modified slightly after the crossover process. This is mutation, where at least one gene in the chromosome is randomly changed with no correlation to the parents’ traits or those of its sibling. This feature is very important in GA because it introduces diversity to the algorithm. Additionally, the algorithm will not be limited to the set of values it produces, instead these random changes allow exploration of the search space. It is important to set the probability correctly; if it is too high the algorithm becomes a random search, or if it’s too low the most optimal solutions might be omitted.

There are different methods for mutation, the simplest example is bit flip mutation, where the genes are binary numbers, and one of the bits changes from 0 to 1 or from 1 to 0. GA is not restricted to binary numbers, decimals can also be used, depending on the problem.

Swap mutation can really make a difference when working with decimals – it takes two random genes in the chromosome and swaps their positions in the sequence.

Scramble mutation allows a whole subset of genes to be shuffled – shuffling implies random outcome – which is done by randomly choosing two indexes then shuffling every gene that falls in the range.

Another mutation technique is the inversion mutation, which is similar to scramble mutation. Two indexes are chosen at random, genes within the specified range are reordered to create an inverse of the initial sequence (first becomes last, last becomes first, etc). (TutorialsPoint, 2021)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Child A | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| Child A(m) | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

*Figure 2.1.4.1: flip mutation*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Child A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Child A(m) | 1 | 2 | 7 | 4 | 5 | 6 | 3 | 8 |

*Figure 2.1.4.2: swap mutation*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Child A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Child A(m) | 1 | 2 | 3 | 5 | 7 | 4 | 6 | 8 |

*Figure 2.1.4.3: scramble mutation*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Child A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Child A(m) | 1 | 5 | 4 | 3 | 2 | 6 | 7 | 8 |

*Figure 2.1.4.4: inverse mutation*

## Neural Networks

In concept, artificial neural networks (ANNs) seek to simulate the human brain to solve problems. It is designed to take in information, analyse it, and process it, then make decisions. A key feature of ANNs is the learning process during which the algorithm interprets the information. Data is fed into the network via the input layer. From the input layer, the data is forwarded to the hidden layer (or the first hidden layer, depending on the architecture) where a mathematical function is applied to the data. Note that the data is just a number without any real meaning at this point. After the application of the mathematical function – referred to as the activation function – the value is then passed down onto the output layer. Each neuron is connected via synapses, each of these connections have a weight assigned to them. The correct value is known, so the neural network knows what the output should be. Based on the error (difference between the actual value and the expected value) the network adjusts the synapses weights. This process is repeated until a certain error rate is reached or a number of iterations has passed. (Kopec, 2019)

ANNs are used in various fields, most commonly they are applied for classification (for example image, speech, audio recognition). Furthermore, ANNs can be used for pattern recognition – fraudulent activity, terror plotting, etc. If enough data is available, ANNs can even make very accurate predictions in fields like medicine, finance and economy, cyber security and defence, and environmental forecasts (geologic, atmospheric, astronomic).

### Neurons and Layers

An artificial neural network consists of neurons that are organised in layers. A neuron is the smallest unit in the network, and it holds a vector of weights, which are floating-point numbers. Data is fed into the network via the input layer, which holds the input neurons. Neurons are connected via synapses, each of these synapses have a random value (weight) assigned to them, between -1 and 1. The inputs are combined with the synapse’s weight using the dot product (vector multiplication), then the activation function is applied on the product to transform the output, which is another floating-point number. The neurons also have a bias value randomly set by the network. (Frankenfield, 2021)

Diagram, shape

Description automatically generatedThere are three distinct layers in the architecture of an ANN: the input layer, a set of hidden layers (up to two), and an output layer. Data travels through the network in this order in the feed-forward process. Another process is backpropagation, which is only used during the training of the network.

*Figure 2.2.1: a basic ANN architecture*

### Feed-forward and Backpropagation

Data travels through the network, starting in the input layer, through the hidden layers, ending in the output layer, which eventually gives a result. When a network in untrained, backpropagation needs to happen.

During the process of backpropagation, the network determines the error rate and identifies neurons responsible for the wrong output. Using these errors, the network adjusts the weight of the synapses. The correct output is known to the network in the training process, so it can be used as guidance. First, the *error* is calculated which is then distributed across all neurons in the output layer. The *derivative* of the output neuron’s activation function is then applied to the neuron’s original output without the application of the original activation function. This result is then multiplied by the neuron’s error to find its *delta*. The delta for each neuron is calculated in the hidden layers, which allows the network to determine how much of an impact a neuron had in getting the wrong output. The following sequence describes the process for calculating the delta in the previous layer in the network:

1. Take the dot product of the next layer’s weights with respect to the current neuron, and deltas already calculated in the next layer.
2. Calculate the derivative of the application function that was applied to a neuron’s last output.
3. Multiply the value of the dot product in step 1 by the value of the derivative in step 2.

The delta calculation starts in the output layer using the following formula:

*delta = f’(outputCache) x e*

Key:

*outputCache*: what was received by the activation function during the feed-forward process.

*e*: the difference between the expected output and the actual output.

*f’()*: derivative of the activation function

The delta is then stored in the output neuron as this process is repeated for each neuron in the layer.

Let’s assume the deltas have been calculated for the output neurons in a 3-output neuron architecture using the above method, and are stored in O1, O2, O3 (On is an output neuron). The below figure visualises the delta calculation for the first neuron in the previous hidden layer.

Diagram

Description automatically generated

*Figure 2.2.2: a basic ANN architecture*

Key:

*Onw1*: these are the weights between each neuron in the output layer and the first neuron in the hidden layer. *w1* represents “weight 1” of a particular output neuron.

*OnDelta*: the delta of a particular output neuron.

*tempVar*: stores the dot product of the vector holding the delta values and the vector holding the weight values. (Kopec, 2019)

### Activation Functions

There is a range of different activation functions that can be used, some are better suited than others for solving a particular problem. Neural networks may feature different activation functions in different layers, for example a network can have a sigmoid function in its hidden layers and a hyperbolic tangent function in its output layer. (Brownlee, J., 2021)

Each function is visualised using Python’s matplotlib library.

#### Step Function

It is the simplest activation function – a threshold value is defined, if the output is greater than or equal to the threshold, the neuron is activated. The output is always either 0 or 1.

Defined as:

f(x) = 1, if x >= 0

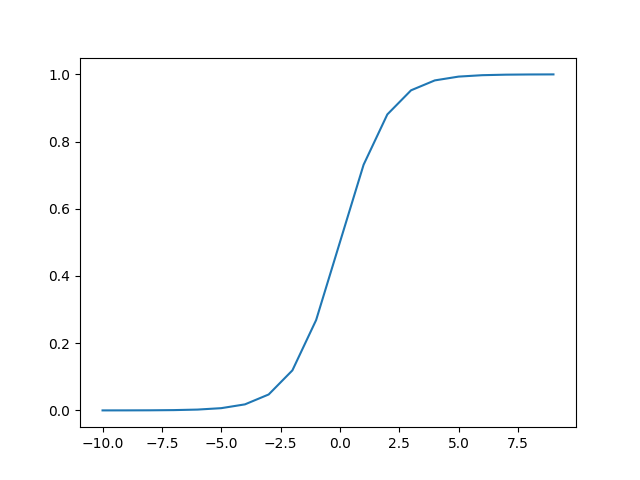
Chart, histogram

Description automatically generated f(x) = 0, if x < 0

*Figure 2.2.3.1: Binary Step function*

#### Sigmoid Function

The sigmoid function is a continuously differentiable (important in backpropagation) and non-linear function. When the function is applied, neurons will output different values, therefore the result will be non-linear, which better for more complex problems. The output ranges between 0 and 1. This function is one of the better options for the hidden layers of a recurrent neural network. The sigmoid function can also be used on the output layer for binary and multilabel classification.

Defined as:

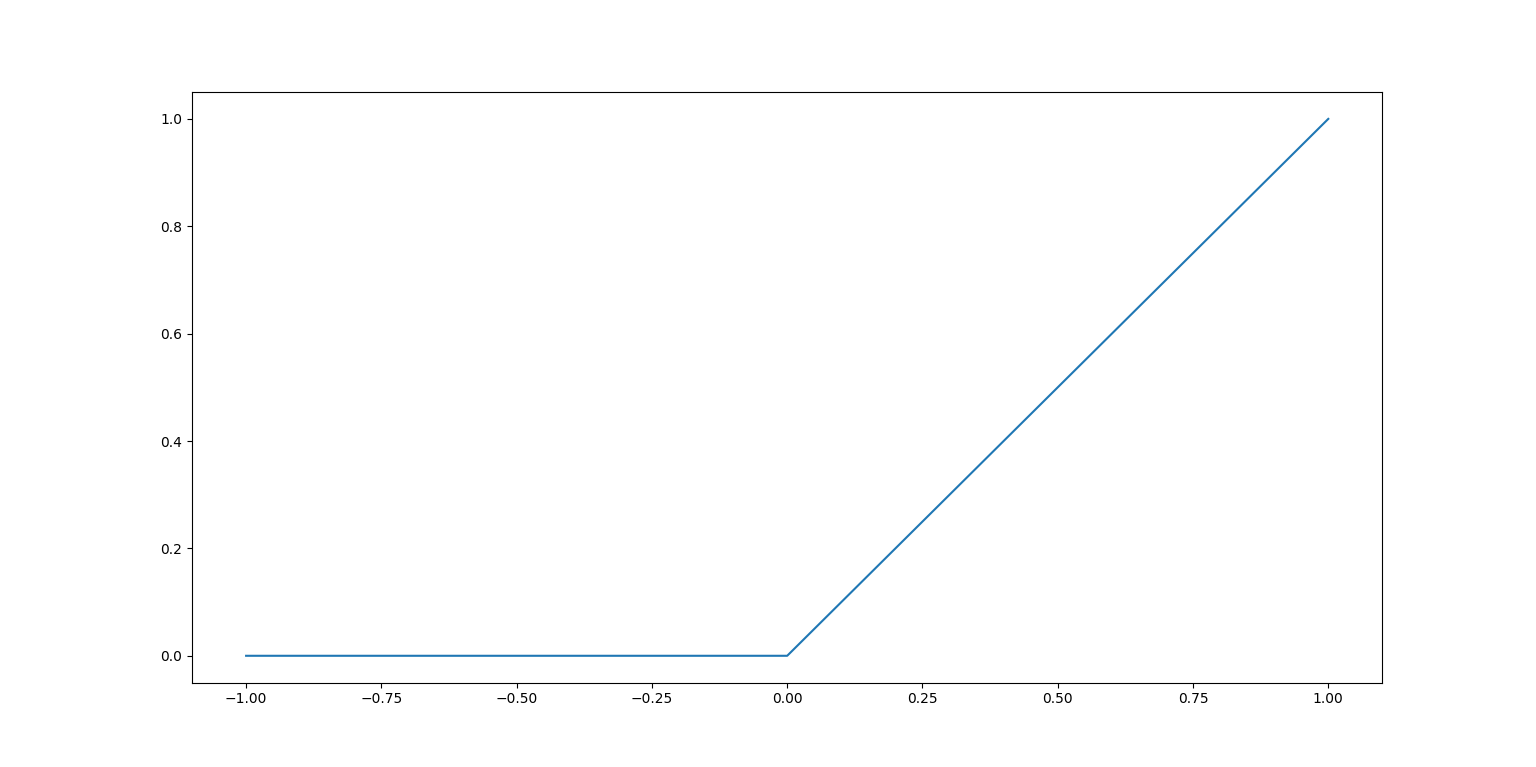
*Figure 2.2.3.2: Sigmoid function*

#### Rectified Linear Unit

ReLU is the most popular activation function; it does not take all neurons into account – only neurons that have an output value greater than 0. Anything less than 0 is set to have a value of 0, and neurons that have the output value of 0 remain inactive. ReLU is a great activation function for the hidden layers of convolutional neural networks and multilayer perceptron.

Defined as:

f(x) = max(0, x)



*Figure 2.2.3.3: ReLU*

#### Tanh function

Chart, line chart

Description automatically generatedThe hyperbolic tangent function is similar to the sigmoid function in terms of its feature. The function outputs a value within the range of -1 to 1. The key difference between tanh and sigmoid is that when it comes to backpropagation, the derivative of the two functions will be different. Tanh avoids bias in gradients. The sigmoid function also has the potential to have a very flat error surface near the origin and because of saturation, the error surface is also flat far from the origin. Tanh is a good activation function to use in the hidden layers of recurrent neural networks. Tanh works well on the output layer when it comes to binary and multilabel classification.

*Figure 2.2.3.4: Hyperbolic tangent function*

### Learning Process

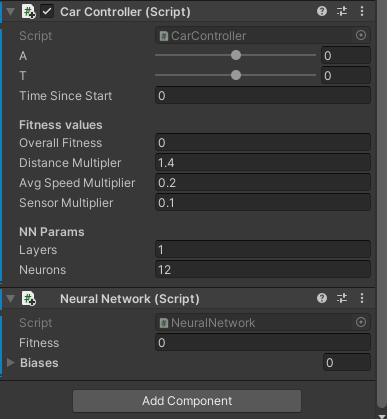
The most difficult part of implementing an ANN is the learning. There is a great degree of complexity to the whole process; once the model is coded it needs to be trained. During the training, the network learns how to solve the problem.

Data traverses back and forth through the network, a single pass or iteration in the training process is called an *epoch*. With each epoch the network gets closer to the optimal solution, however it is important not to overtrain the network because that might lead to the memorisation of the solution, rather than learning it. If the problem is memorised, the network will not be able to solve other similar problems. The loss function, or cost function, is the function that calculates the error made by the network – this links back to backpropagation. The loss function is key in adjusting the weight values of the synapses. This process is repeated over and over until the target number of epochs is reached. (How neural networks are trained, 2021)

# Unity Engine

The Unity Engine is a game engine developed by Unity Technologies. The reason for choosing Unity over any other engines, and plain Python, is because it comes with a wealth of free and open-source assets ready for use. The engine can use C# and JavaScript, for the project the C# language was used. The project is created in a 3D space; it consists of a plain, a track, and a car – since the purpose of the project is to get the car to drive itself on the track nothing else is needed.

## The Car

The car is a free asset from the Unity Store. It runs using a customised script, which has a few variables that are observable in the inspector. This allows continuous experimentation (trial and error) with the values.

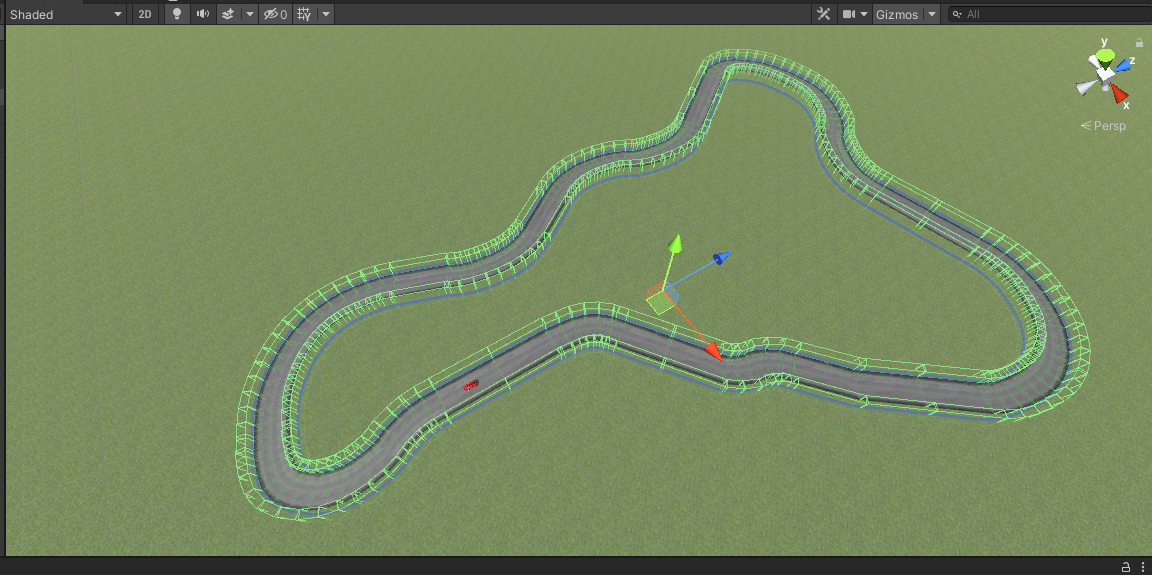
*Figure 3.1: Hyperbolic tangent function*

*Figure 3.1* shows the parameters in the inspector. There is a timer, which is automatically incremented; and a fitness value, which is calculated based on the time and distance since start. The distance and average speed multipliers are used in the fitness calculation – the speed is also randomised when the chromosome is created, and changes randomly. The sensor multiplier is there to help the car navigate.

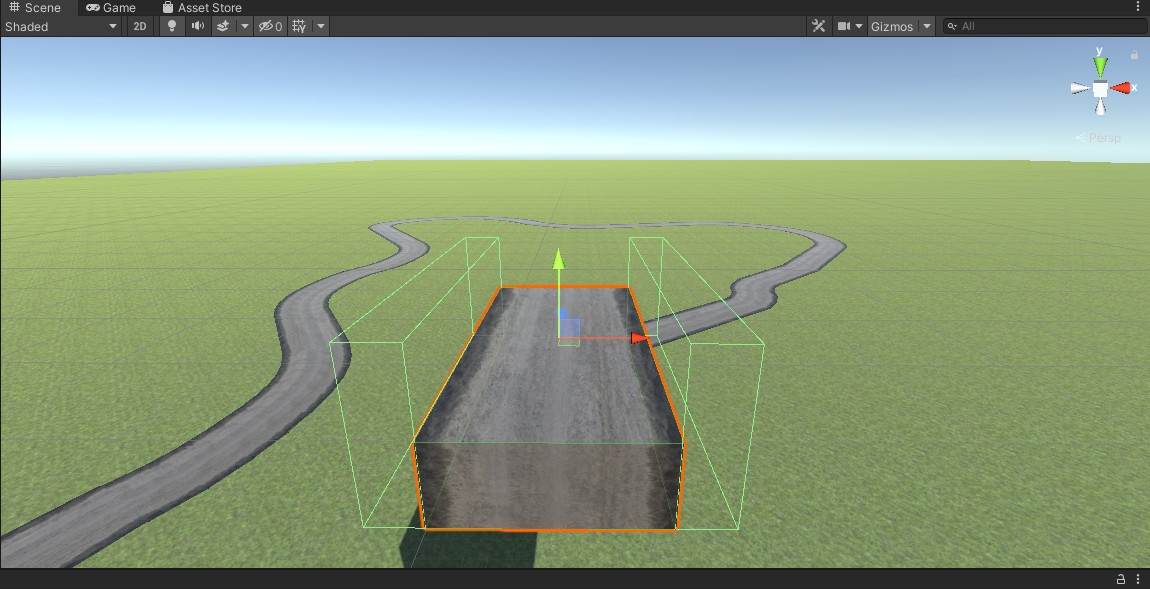
There are additional parameters for the neural network – these are the layer count and the neuron count. Note that an additional layer and several additional neurons would drastically slow down the program due to the values being manipulated. Additionally, the car displays the fitness value (i.e. the fitness value of the current chromosome), and the bias value.

## The Track

The track was hand-built using cubes with textures. On either side of the road another box is added which is used as a collider. The collider is invisible, but it does interact with the car: if the car collides with it, it results in the elimination of that particular car (one car is one chromosome in the population).

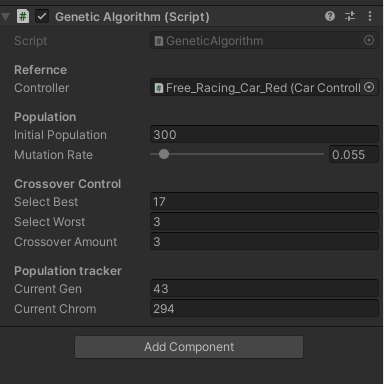


*Figure 3.2.1: the entire track. Green boxes are the colliders.*



*Figure 3.2.2: a single road object with colliders on either side highlighted.*

## The GA Object

In order to get GA working correctly with ANN, a separate game object needs to be created for GA, which is then referenced to the car object. There are several methods and functions called between the two classes which is explained in Chapter 4. The below figure shows a snapshot of the GA object’s behaviour during runtime.

*Figure 3.3: the GA variables as seen in the inspector.*

The population is set to 300, with a mutation chance of 5.5% on each chromosome. For the creation of the next generation, the 17 best chromosomes and worst 3 are selected. The crossover amount indicates how many times these chromosomes will be used to breed offspring.

The population tracker continuously updates while the program is running, giving the current generation count and the chromosome count. Every single chromosome is simulated individually – this is an area where the program could be made more efficient: an entire population of chromosomes could be simulated at the same time. This would require the setup of either 300 duplicates of the track and the car, or 300 cars running on the same track. The latter requires manipulation of the colliders. Note that the values shown in *Figure 3.3* are just experimental values.

# Implementation

Perhaps the most optimal approach to getting the car to drive itself is to implement a feed-forward neural network with backpropagation. The backpropagation during the training process, described in Chapter 2, allows the car to improve its navigation over time. However, a somewhat different approach was taken. Instead of backpropagation, the neural network is evaluated by a genetic algorithm.

The implemented network features the standard structure: one input layer, one hidden layer, one output layer. Each synapse has a randomised weight, and the neurons have random biases. The network utilises the sigmoid and hyperbolic tangent activation functions: the sigmoid activation function is used for the speed; the hyperbolic tangent is used of turning. The speed varies between 0 to 1 – 0 is still, 1 is full speed. The tanh function allows to use -1 to turn left, 0 to keep straight, and 1 to turn right. The values used for speed and turning are both float, which is important for optimising the controller. For example, taking sharper turns or driving near the edge of the track in some cases can lead to an increase in the distance travelled over time.

The training is done using the genetic algorithm, which selects the best performing networks from a population of networks and creates a new generation. To help preventing the network from memorising the solution to the problem, a small portion of the worst solutions are also mixed with the best ones. There would be two ways of discovering the solution space:

* either start with a population consisting of a large pool of chromosomes and eliminate the unfit chromosomes, repeat the process until the population goes extinct, or
* keep a steady population, control what chromosomes get to reproduce, repeat the process until reaching a set number of epochs.

For this project, the second option was implemented. It is a safer approach and less taxing on the system – performance had to be considered – but it is also more time-consuming. The main reason the second option is safer than the first one is because there is a chance the network will not be optimised within a number of iterations equal to the maximum number of possible generations – in other words all chromosomes might be eliminated before an optimal solution is found. Additionally, in case an optimal solution is found using the first option within a reasonable time, if the training continues, there is a risk for overtraining the model.

## The C# Language

The Unity Engine can run scripts written in C# and JavaScript. The project was coded in C#, which is a modern OOPL, widely used by businesses. It runs on Windows and requires the .NET framework. The language is often compared to C++ and Java due to similarities in syntax. It best performs when used for the development of web or desktop applications – since Unity allows the porting of a developed application to mobile platforms, C# has become very popular for mobile application development.

C# is a high-level language; it has been described as a language that is easy to learn and read but is complex to fully utilise. It is a statically, dynamically, strongly typed language, meaning the code is checked for errors before the it is compiled. (Mkhitaryan, 2017)

## The GeneticAlgorithm Class

The GeneticAlgorithm class is used for the training of the network. It evaluates each network and selects the best ones from the population for reproduction after ordering the results using bubble sort.

### Variables

*Figure 4.2.1: global variables in* GeneticAlgorithm

Figure 4.2.1 shows all global variables used in GeneticAlgorithm. To make the progression and controls easier to follow, categories were created and labelled using [Header(“ “)]. For example, the Population header displays the variables initialPopulation and mutationRate in the inspector, and since they are public, they can be modified in the inspector – the hardcoded values are overwritten in that case.

The initialPopulation sets value for the starting population and since the GA maintains the population count, this value is used throughout the whole process of training. For example, the evaluation takes place after 300 chromosomes tried to solve the problem. The variable mutationRate can have a float value between 0 and 1 to represent chance of mutation with 1 being a 100% chance for a mutation to occur. The coded value is kept at 4%.

Crossover is a very important feature of GA. The algorithm allows for the selection of the most optimal and least optimal solutions for crossover. These values can be manipulated in the inspector too, the integer value given to the variable crossoverAmount sets the value for the k-point crossover.

All the chromosomes are stored in a list of integers stored in genePool and there an NeuralNetwork object called population is created.

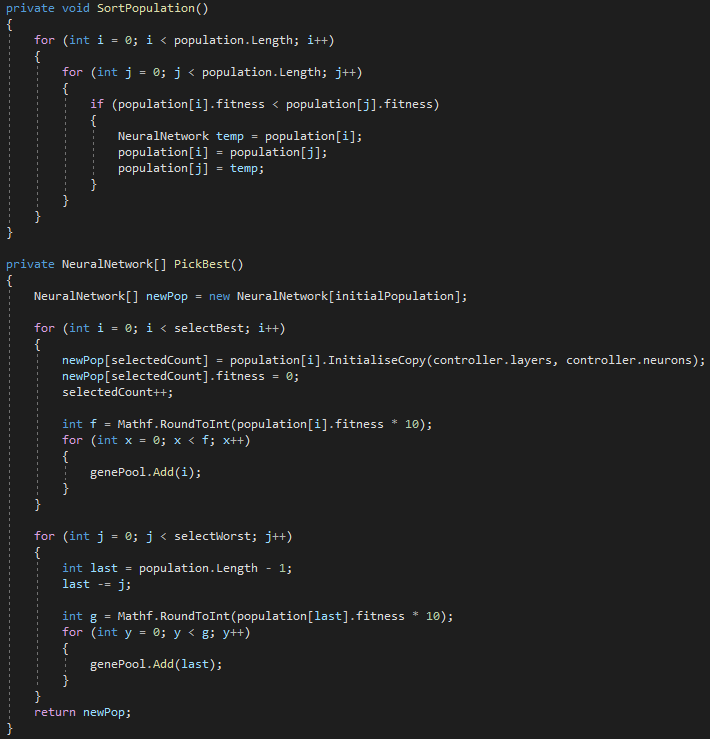
The variables currentGen and currentChrom represent the generation count and the currently performing chromosome, respectively. These provide a live update on the overall performance in the inspector.

### Methods and Functions

When the program begins the InitialisePopulation() function is called which creates a new population of chromosomes. This function also calls a method created in the GeneticAlgorithm class called RandomPopulate() which creates neural network objects and randomises their values. This is how the initial population of chromosomes is created.

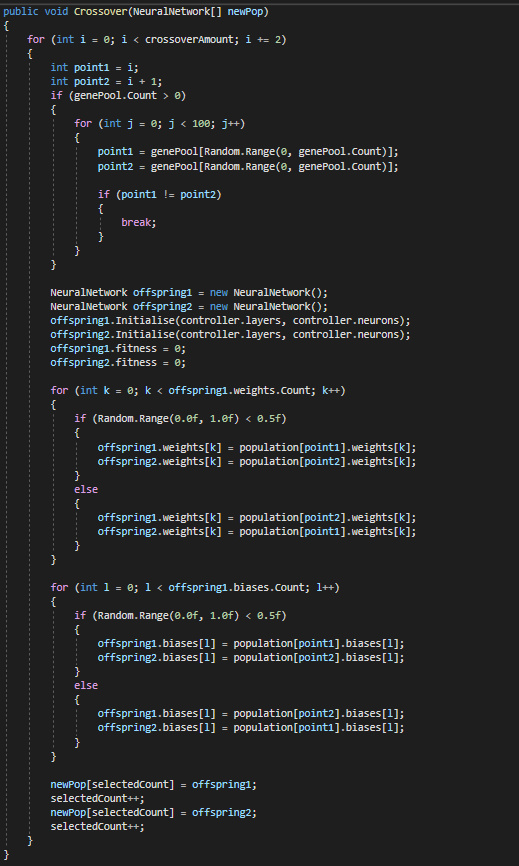
#### Evaluation, Sorting, and Selection

When a chromosome finishes performing, its overall fitness is saved along with the weights and biases, which are the traits of the chromosome – these are done by the NeuralNetwork class. A very simple bubble sort is used to order the chromosomes by fitness (descending). The function PickBest() uses these results, and the selectBest and selectWorst variables to identify candidate chromosomes for reproduction.



*Figure 4.2.2.1: functions used for the selection process.*

#### The Crossover Function

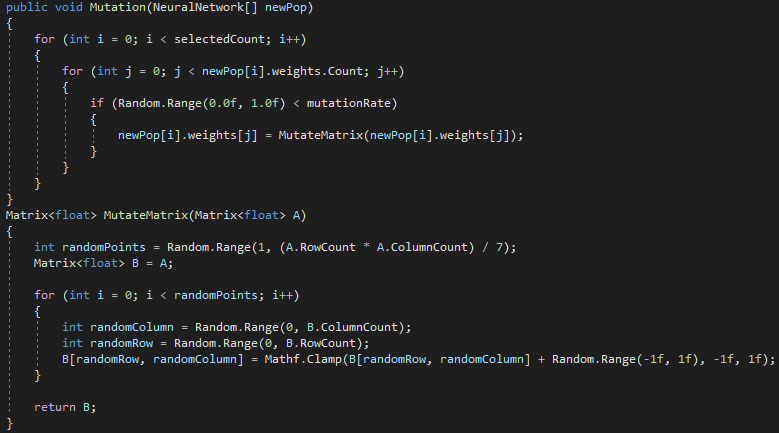
K-point crossover is featured in the algorithm. To create crossover in neural networks, the program takes two offspring objects, takes all the weight values of each of these objects (stored in a list), then identifies the points of crossover. Weight values are then swapped between the neural network objects at given index. Since neurons have biases too, these bias values are also transferred across. The function takes an argument which is a neural network object and updates the population with the result of the crossover.

*Figure 4.2.2.2: crossover*

#### Mutation-Related Functions

When it comes to the mutation of a neural network, only the weight value of the connection (synapse) changes – the bias value of the neuron remains the same. The variable described above, mutationRate, is evaluated by an if statement in which a random float is generated in the range of 0 and 1, and if that value is less than the mutation rate, then the weight at a given index is changed using the MutateMatrix() function. The mutation featured in this GA is uniform mutation – this is not described in Chapter 2; it is similar to uniform crossover: each bit (weight value) has a random chance to mutate.

The function MutateMatrix() takes a matrix as an argument which holds the weight values. Uniform mutation is achieved by selecting random positions in the matrix *n* times where *n* is a random number between 1 and the size of the matrix. This is more efficient than applying a random chance on each weight, and it improves the overall performance of the program.



*Figure 4.2.2.3: functions used for mutation.*

#### Other functions

The function Repopulate() is responsible for creating the new population of chromosomes. It calls functions created within the class and it feeds parameters into them. When this function is called it increments the generation counter, resets the best selected candidates to 0, sorts the population, applies selection, crossover, and mutation on the previous population to create the new population. Finally, it then resets the chromosome count to 0 and clears the genome using ResetToCurrentGenome(), which essentially takes care of the repopulation at the lowest level by resetting the neural network – further calculations are done in the CarController and NeuralNetwork classes. When a chromosome “dies” – for example the car collides or reaches the fitness limit – the Death() function extracts the traits and fitness value of the chromosome and stores them.

## The NeuralNetwork Class

The NeuralNetwork class creates the networks by adding the input layer, hidden layer(s), and output layer, and by assigning random weight and bias values. C# has hyperbolic tangent function build into it; however, the sigmoid function is not defined by default so one is created in the script.

### Variables

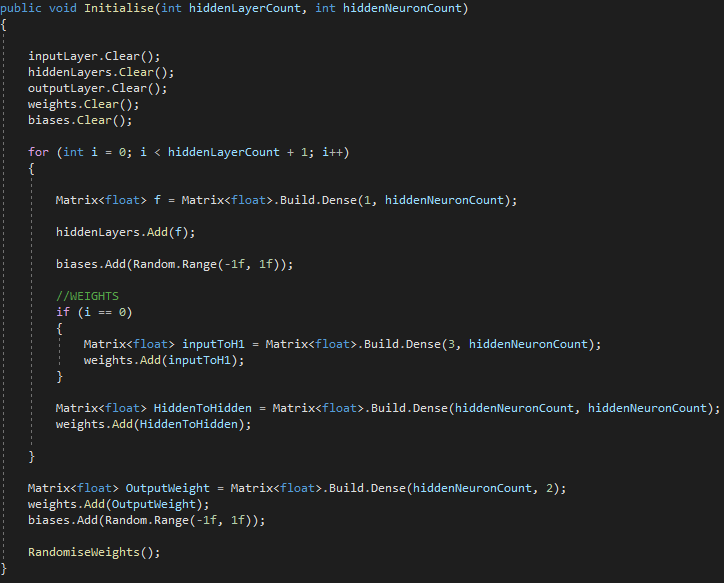
*Figure 4.3.1: variables in the NeuralNetwork class*

When a chromosome (a neural network) dies, its fitness value needs to be stored for further evaluation. The float variable fitness serves this purpose – it is passed to the GA class, where the network’s performance is assessed based on this value.

One matrix is created for each layer. The inputLayer matrix builds a 1x3 matrix using the Build.Dense() method. The hiddenLayer structure is not set at this stage as it must not be hardcoded. Finally, the outputLayer creates a 1x2 matrix. One of those neurons is for setting the speed, the other is for defining how much the car should turn. The bias value of each neuron is stored in a list.

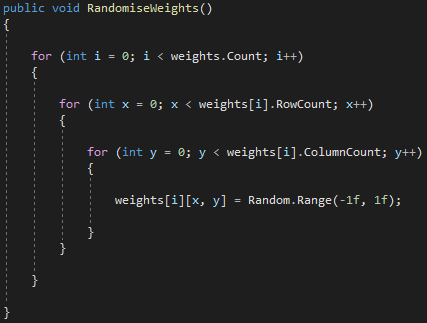
### Initialising the network

All values must be reset when a new network is created. First, the function resets the values of the input layer, the hidden layer, the output layer, the weights, and the biases. Since the input and output layers are already built, the Initialise() function builds the hidden layer(s). The function takes two integers as arguments: one for the hidden layer count, and another for the neuron count in one hidden layer. It then creates the hidden layers by a for loop, in which the bias values are randomised and assigned to each neuron, as well as the weight of each connection. The weight values between the last hidden neurons and the output neurons must also be set.



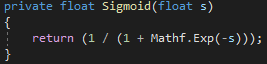
*Figure 4.3.1.2: the initialisation function.*

### Randomising Weights

The function RandomiseWeights() contains a set of nested loops. The outermost loop counts the total number of weights. The first nested loop counts the rows, the innermost loop counts the columns of the matrix. The method inside the innermost loop assigns a random float value between -1 and 1 to each weight in the network.

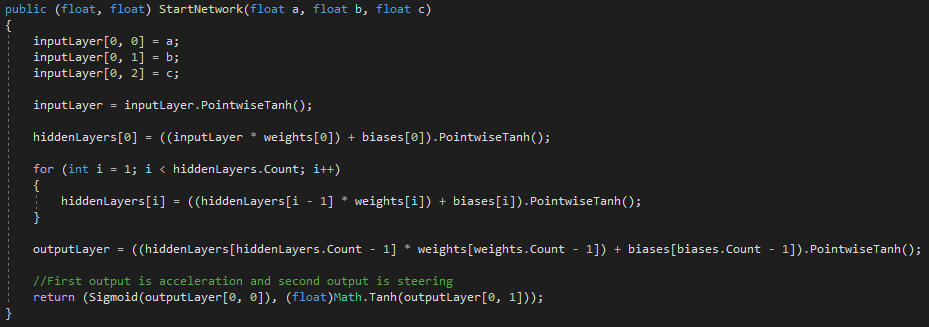
*Figure 4.3.3: randomising the weight values.*

### The Sigmoid Function

The sigmoid function is very easy to create. In the script, the function takes a float argument, which is passed to a Mathf.Exp() method as a parameter.

*Figure 4.3.4: the sigmoid function.*

### Running the network

The StartNetwork() function runs the network, it takes three arguments and has two return values. Each of the three arguments are from the input sensor (more on this in the Car Controller section), and the output values determine how the car should move – speed and rotation. The values from the input sensor are fed into the input layer then the feedforward process takes places. The values in the input layer are transformed using the hyperbolic tangent function. When these transformed values are passed onto the hidden layer, the passed value is multiplied by the weight of the synapse, then the bias value is added to the product. Finally, the hyperbolic tangent function transforms the value to set it within the range of -1 and 1. The values for the output layer are calculated in a similar fashion. The function then returns to float values which are the values of the output layer. The first value represents speed, and it needs to be within the range of 0 and 1, therefore the sigmoid function is used on that value. Steering can be between -1 and 1, so that output is transformed by the hyperbolic tangent function.

*Figure 4.3.5: feedforward script.*

## The Car Controller Class

The CarController defines how the car (the chromosome) behaves. It interacts with the environment; the car object has 3 sensors (left, forward, right) and collision detectors built into it. The input sensors’ values are passed onto the NeuralNetwork class, while the collision detector acts as an alert when the car is going off-track. When the car collides, that particular chromosome is killed and a new one is initiated. The neural network feeds the acceleration and steering values to the car controller. The values that are evaluated by the fitness function are the total distance and the average speed of each chromosome.

### Variables

*Figure 4.4.1: the global variables in CarController*

The car needs to have 3 characteristics in order for it to able to move around, these are the starting position (startPos), starting rotation (startRot), and movement values (moveInput). These are stored in vectors. The float variables a and t are representing acceleration and turning, and they both have a range between -1 and 1. Note that the acceleration never reaches or drops below 0, the range is extended for testing purposes.

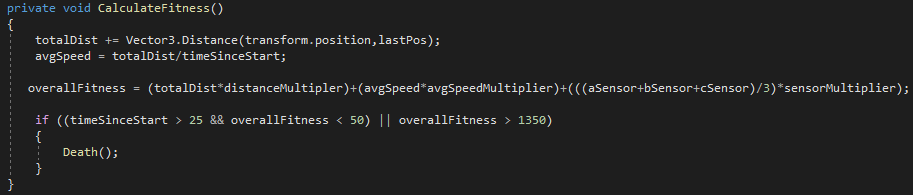
The defining values for calculating the fitness are the distance and the speed. Each of these are of different importance, the most importance is the distance made by the car during its lifetime. Speed is also an important factor as there will be several chromosomes that find a correct solution. This will allow the more optimal solutions to be distinguished. These variables appear in the inspector and can be changed at run time to tweak performance.

Each chromosome has a neural network. The network’s structure can be changed – by default there is one hidden layer with 24 neurons. A time counter is also present – if a chromosome is alive for too long it is eliminated.

### Functions

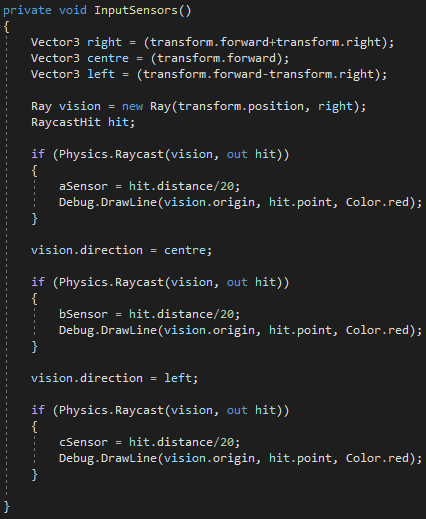
#### Fitness calculation

As mentioned, the fitness score is determined by the average speed and total distance travelled. Each of these variables are multiplied by their corresponding multipliers – the values for these are determined based on importance. The sensor multiplier is also applied – this determines how close to the edge the car is.



*Figure 4.4.2.1: the fitness function.*

#### Knowing Where The Car Is

The input sensors allow the car to be aware of its whereabout – how close it is to the edge of the track and what is ahead of it. This also plays an important role when solving track, and it is vital for making the car intelligent as it needs to be able to make the correct decision based on the sensory inputs. The function uses Unity’s raycast, which allows one object to determine its distance from another object.

*Figure 4.4.2.2: the scripted sensors.*

#### Other Functions

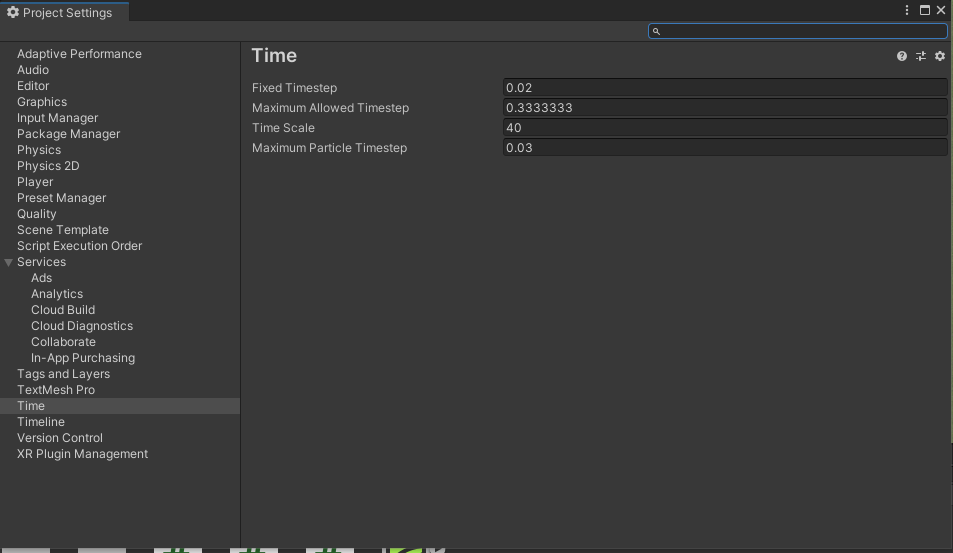
The car is moved using the MoveCar() function, which takes two floats as arguments and it uses those to accelerate and steer.

When the chromosome is eliminated either by being too fit or by collision, the game object is destroyed by calling the Death() function and the neural network is reset by calling the ResetNetwork() method, which takes a NeuralNetwork object as an argument. The Reset() function resets the game object itself to the starting point.

# Results

The GA is able to solve the track within a generation, but this does depend on the network architecture. This is not to say the algorithm solved the problem, it simply means it found a potential solution – the solution space still needs exploring, for example steering angles can be improved.

## Testing and Analysis

Unity allows the simulation to be sped up which results in the training to finish faster. This setting can be found under “Edit”, “Project Settings”, “Time”. A time scale setting can be found there, for the training process it is set to 40, meaning the program will run 40 times faster.

*Figure 5.1: speeding up the emulation.*

### Selection and ANN Architecture

The first hyperparameter that was subject to heavy modification was the selection technique in GA. A small group of selected chromosomes were tested against a larger group, and the ratio of best to worst chromosome selection was also modified.

Selecting only a few of the top agents with a small amount of the worst chromosomes resulted in the algorithm slowly reaching a plateau. The performance got worse as the car was not able to make it past the first turn after getting to nearly 200 generations. On the other hand, the overall performance was better with less generation.

Selecting more of the better performing chromosomes and combining them with only a few of the worst ones resulted in a more consistent learning process, but it also took more time. Not only the population becomes more diverse, but the algorithm also becomes more accurate over time as it can explore more of the solution space.

Increasing the hidden layer count from one to two does help, but performance significantly improves when the neuron count is increased in the hidden layer.

### Epoch

The epoch is defined by the generation count. A hard limit can be placed on the number of generations and if that limit is reached, the training finishes.

### Measuring Performance

One way of measuring the performance is by using the generation and solution count and comparing them after repeating the training process several times. This is especially useful when the hyperparameters are being adjusted.

The results can be analysed after the training is done – the generation and solution count values are saved into a comma-separated values (CSV) file. The values that are being exported are the solution count, the generation count, and the rate at which the solutions are being presented. This helps with finding out when the algorithm performs at its best.

# Conclusion

## Overview

The training of the model is done using GA rather than the commonly used backpropagation technique. The car uses 3 input sensors for each direction: left, right, and centre. These sensors work with Unity’s collider feature do help determine the car’s whereabouts. The values obtained are then passed onto the neural network which uses the hyperbolic tangent activation function, except for one of the input neurons – which uses the sigmoid activation functions. The reason for this is because steering can range from -1 to 1 (indicating left to right respectively, 0 is centre), and the acceleration can be between 0 and 1 (stationary to full speed).

GA uses its key features (selection, crossover, mutation) on neural networks to improve the overall performance. The traits of the network are the biases with the connecting synapse weights – upon crossover these values are swapped in offspring chromosomes. Mutation only affects the weight values, however.

The results of the training are exported into a CSV file for evaluation – this file can be used in Python for graph plotting.

This project can be a good starting point for the development of software designed to solve specific tasks, but it can also be great for experimenting with the combination of GA and ANN.

## Improvements

The end product of the project is vey basic; it allows experimenting with GA and ANN. It can be developed further, the main improvements would be the addition of a GUI, the creation of a .exe file, and saving the results of the training.

The GUI would display exactly what is shown in the inspector. There would be a main menu with 3 buttons:

* One option would allow the training to begin. On click, a warning message would appear if the training file is already present (see 6.2.2), otherwise the training would just take place using the hyperparameters set by the user.
* Another option would allow the user to start the car. A warning message would appear if no training file is present.
* The final option would be the quit button, which would close the application. If the user wishes to terminate early, a home button can be added to the corner of the screen.

### More Meaningful Results

The testing phase yields numbers, but these are raw results. These values could be converted into data that could be represented on a graph. The graph would show how accurate the model gets over time: on the x-axis the epoch would be represented, on the y-axis the rate of good solutions would be shown.

To show a detailed breakdown of the algorithm’s performance over time, the number of good solutions yielded within one epoch could be shown on the y-axis. The reason for the application of this visualisation technique is that it allows us to identify when the algorithm made a mistake and measure the severity of the mistake – for example 3 solutions are found within one population, but no other results are found for the next 2 populations.

Visualising data can further allow assessing the importance of several traits, such as the distance, speed, and sensory values. It is possible to create these graphs in Unity, and it is also possible to extract the data and save it in a file that can be used by Python to create these graphs.

### Saving the Network Values

The training would become truly meaningful if the values of the neural network were extracted into a JSON file. Training would only happen if such file were absent from the directory – an if statement in the script can be used to check if this is the case. If the file is present, the values of the neural network would then be set based on what is stored in the file.

### The Environment

There would be several factors in real life that could affect the car’s physics. Road material, weather, weight, tyres, etc. are all major contributing factors. If the weather is rainy, it could potentially affect the sensors, and the wet road will have an impact on how well the car could turn. This means that the car would have to slow down before taking the turn. If the road is icy, the car needs to slow down well before turn is taken, and this is when the weight of the car and the type of tyres equipped become extremely important. The car needs to learn how to interact with its environment; it needs to learn how to slow down. This however is an extremely complex task to perform.

## **Potential Uses**

A fully functional virtual self-driving car can be applied in many areas. Gaming would be the most common example; the car could be deployed in racing games or in open-world games where the traffic is procedurally generated.

Furthermore, virtual self-driving cars can be applied in infrastructure design, for example predicting flow of traffic, designing roads and streets where traffic lights and roundabouts are being used. This can be useful for the development of government projects.

Section 6.2.3 mentions other variables that could affect the car’s behaviour – if these are fully implemented the result would be a software that can be used for designing land vehicles either for personal use, law enforcement, or military purposes. The design of the car could be put to a thorough test. Efficiency, effectiveness, safety, robustness, and many other traits and features can be tested which could help developing a car that would be used in the real world.

# Further Discussion

## Notes on GA & ANN

Neural networks heavily rely on hyperparameters. These are parameters that control the learning process and behaviour and is set manually – therefore it can be said that these values are set heuristically. These values cannot be learned by gradient based neural networks. The only way for a pure ANN to learn hyperparameters is by implementing nested learning. Common examples of hyperparameters are the learning rate, epochs, number of hidden layers, and the activation function. (Goodfellow, Bengio and Courville, 2016)

GA allows the NN hyperparameters to be optimised. Each network is trained, and they yield a fitness value. These values are evaluated by the GA and the chromosomes (neural networks) go through the selection-crossover-mutation process. (Suryansh, 2018) This is a basic approach of combining GA with ANN and it is used for this project.

## Genetic Algorithms: Advantages and Disadvantages

GAs are known to be computationally cheap as there is no algebra involved. Since the application of GA on the ANN eliminates the need for backpropagation, the whole model becomes less demanding. Pointed out in Section 2, GAs can be used for a variety of purposes. They are adaptable and can be designed for a specific problem. Additionally, GAs can provide insightful outputs which allows us to determine why a specific decision was made.

Since everything in GAs is left to chance, it allows an extensive exploration of the solution space. On the other hand, when awful crossover and mutation is applied not by design but by chance, the algorithm can deviate from the path to the correct solution. There is a risk of reaching a loss threshold, where the algorithm no longer improves. (Using Genetic Algorithms to Train Neural Networks, 2021)

## Neural Networks: Advantages and Disadvantages

Neural networks can handle large amounts of data and it truly shines when more data is available for the problem. Algorithms for neural networks are constantly evolving. The computation power available makes contribution to this – for example with the release of more powerful GPUs we have access to much more processing power. Normally, programs run on the CPU, neural networks tend to be trained on the GPU instead as it is much more powerful than the processor.

Neural networks are described to have a black box nature, which means data is given to the program, something happens, then it outputs the result. It is known what exactly is taking place between the input and the output, however it is very difficult to find out why a particular prediction is made. A significant drawback of ANN is non-interpretability, which might be needed for some problems. Due to the increasing amount of data available – or better quality of data –, there is a constant growth in data volume at a rate higher than the growth in computational power. This means different types of ANN algorithms must be used on different problems to maintain efficiency. (Donges, 2019)

# Bibliography

Brownlee, J., 2021. *How to Choose an Activation Function for Deep Learning*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/> [Accessed 30 April 2021].

Donges, N., 2019. *Pros and Cons of Neural Networks*. [online] Experfy Insights. Available at: <https://www.experfy.com/blog/ai-ml/pros-and-cons-of-neural-networks/> [Accessed 1 May 2021].

Frankenfield, J., 2021. *Artificial Neural Network (ANN)*. [online] Investopedia. Available at: <https://www.investopedia.com/terms/a/artificial-neural-networks-ann.asp> [Accessed 29 April 2021].

Goodfellow, I., Bengio, Y. and Courville, A., 2016. *Deep learning*. The MIT Press.

Kopec, D., 2019. *Classic computer science problems in Python*. Manning Publications Co., pp.127-152.

Medium. 2021. *Using Genetic Algorithms to Train Neural Networks*. [online] Available at: <https://towardsdatascience.com/using-genetic-algorithms-to-train-neural-networks-b5ffe0d51321> [Accessed 2 May 2021].

Mkhitaryan, A., 2017. *Why Is C# Among The Most Popular Programming Languages in The World?*. [online] Medium. Available at: <https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb> [Accessed 1 May 2021].

Ml4a.github.io. 2021. *How neural networks are trained*. [online] Available at: <https://ml4a.github.io/ml4a/how\_neural\_networks\_are\_trained/> [Accessed 1 May 2021].

Suryansh, S., 2018. *Genetic Algorithms + Neural Networks = Best of Both Worlds*. [online] Medium. Available at: <https://towardsdatascience.com/gas-and-nns-6a41f1e8146d> [Accessed 2 May 2021].

TutorialsPoint, 2021. *Genetic Algorithms Tutorial - Tutorialspoint*. [online] Available at: <https://www.tutorialspoint.com/genetic\_algorithms/index.htm> [Accessed 28 April 2021].