
Optimization of an AI Car in a Physics- Based Driving Simulator using Evolutionary Algorithms and Neural Networks

**EVOLUTIONARY COMPUTING
WRCV402**

ASSIGNMENT 05
29 October 2019






CHRIS ABRAHAM	s216257603
GEOFFREY OLLS	s213463857
JASON HOWARTH	s216014727
SIYABULELA MZOMBA	s214335658
WANGA SIDLOYI	s216063531

AUTHENTICATION

We declare that this material submitted for assessment, is entirely our own work and has not been taken from the work of others. The assignment work is based solely on our own study and/or research. We completely understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should we engage in plagiarism, collusion or copying.

We have identified and included all sources of all facts, ideas, opinions, and viewpoints of others in the reference list. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study, or in any other institutions.

We understand that this assignment may undergo electronic detection for plagiarism and an anonymous copy of the assignment may be retained on the database and used to make comparisons with other assignments in future.

Name, Surname	Student Number	Signature
Jason Howarth	216014727	
Geoffrey Olls	213463857	
Chris Abraham	216257603	
Siyabulela Mzomba	214335658	
Wanga Sidloyi	216063531	

1 Table of Contents

2	<i>Abstract</i>	<i>iv</i>
3	<i>Introduction</i>	<i>1</i>
3.1	Topic	1
3.2	Hypothesis.....	1
3.3	Proposed Methodology.....	1
4	<i>Implementation</i>	<i>2</i>
5	<i>Parameters</i>	<i>6</i>
5.1	Recurrent Neural Network	6
5.2	Particle Swarm Optimization	6
6	<i>Experiments Conducted</i>	<i>7</i>
6.1	Fitness vs Generations for various number of hidden neurons.....	7
6.2	Throttle and Velocity vs Distance for various number of recurrent neurons	7
6.3	Fitness vs Time for various number of recurrent neurons	8
6.4	Repeatability	8
7	<i>Observations</i>	<i>9</i>
7.1	General Observations.....	9
7.2	Experiment 6.1	10
7.3	Experiment 6.2	10
7.4	Experiment 6.3	10
7.5	Experiment 6.4	10
8	<i>Conclusions</i>	<i>11</i>
9	<i>Appendices – The code</i>	<i>12</i>
9.1	Scripts that make up the Evolution/Training manager game object (containing PSO): ..	12
9.2	Car-side evolutionary behaviour script:	22
9.3	The Neural Network script:	25
9.4	The distance sensor or “feeler” script:	29
9.5	Other miscellaneous scripts:.....	30

List of Figures

Figure 1: Proposed inputs to the neural network controlling the vehicle	1
Figure 2: Scribante Racetrack imported into Unity	2
Figure 3: Altered Scribante track, implemented as an unseen track	2
Figure 4: Physics and Dynamics of the Vehicle.....	3
Figure 5: Sensory Data Captured from environment	4
Figure 6: Neural Network Topology.....	4
Figure 7: Visual Implementation of PSO training	5
Figure 8: Fitness evaluation over generations for various numbers of hidden neurons	7
Figure 9: Speed and Throttle measured against the distance of the trained track for various numbers of recurrent neurons.	7
Figure 10: Speed and Throttle measured against the distance of an unseen track for various numbers of recurrent neurons.	8
Figure 11: Fitness evaluation vs Time for various number of recurrent neurons	8
Figure 12: Testing repeatability of results	9

2 Abstract

This project uses neural networks and evolutionary algorithms to optimize a vehicle controlled by artificial intelligence (AI). Through Unity the vehicle will initially be trained on the local Celso Scribante track, here the vehicle is optimised to complete the track safely in the quickest possible time. Once the vehicle has been optimised it will be introduced into an unseen environment, where it should be able to navigate through the track safely.

A multilayer neural network will be used to navigate the vehicle through the track. The output of the neural network will be utilized to control the cars speed and turning angle. A recurrent neural network will also be tested to feedback output connections to the input layer and allow the neural network to remember certain characteristics and use them as it wishes. A particle swarm optimization evolutionary algorithm is used to emulate the success of neighbouring vehicles and their own successes in which an optimal path of the track and time can be found.

The results have shown that the car was able to traverse the training track in a very fast time. One could see how the AI gradually improved as the training progressed. The resulting car was able to safely navigate an unseen track although in a relatively longer time.

3 Introduction

3.1 Topic

Optimization of an artificial intelligence driven vehicle in a physics-based driving simulator with the objectives of minimizing time taken per lap on the racetrack used in training, and be able to safely navigate an unseen racetrack of similar complexity after training.

3.2 Hypothesis

Initially all vehicles will be assigned random weights to ensure that the initial population is a uniform representation of the entire search space. The genetic algorithm should achieve an absolute measure of fitness for the path the vehicle should travel.

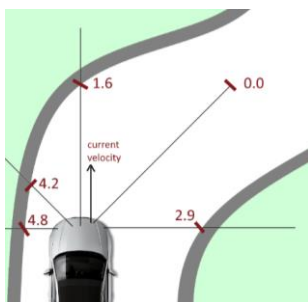
Thereby, we can expect that over the generations, the cars will learn to:

1. Navigate the training track without crashing and be able to (at least partially) carry this auto-navigation ability over to unseen tracks.
2. To find the racing line of the training track that makes the vehicle complete the track in the shortest possible time.

3.3 Proposed Methodology

The project will most likely comprise the following objectives/milestones:

- Set up a virtual driving simulator/environment using Unity.
- Experimentally figure out what spatial and environmental information regarding the vehicle will be useful as inputs into the driving controller's neural network, and how to capture them from the environment.
- Determine the best structure of the neural network for driving the cars (such as activation functions and outputs).
- Set up a genetic algorithm to find the optimal weights for the neural network:
 - Experimentally determine the best fitness function with which to evaluate the "fitness" of different neural networks' performance. Possible factors to consider are: how far the individual made it around the track before crashing (further is better), then the time taken to cover that distance (shorter is better). The evaluation of an individual stops when the individual crashes (or possibly finishes a lap).
- Determine how to evolve the population as efficiently as possible – seeing as it will be an actual race in a 3D environment – it may take a long time per generation – especially once the evolution progresses.



Left: possible type of inputs to be fed into the neural network. Outputs could then translate to which parts of the car controller (such as accelerate, turn left, turn right, or brake) should be activated, deactivated, or remain unchanged from previous state.

Figure 1: Proposed inputs to the neural network controlling the vehicle

turn right, or brake) should be activated, deactivated, or remain unchanged from previous state.

4 Implementation

The simulation was created with 2 main scenes. The 1st scene consisted of a racetrack which was used for training a population of cars, each controlled by a neural network. The 2nd scene was used to test the best individual on an unseen “test” track.

The racetrack modelled into the training scene was recreated from the road profile of the Celso Scribante Racetrack situated in Schoenmakerskop, Port Elizabeth.

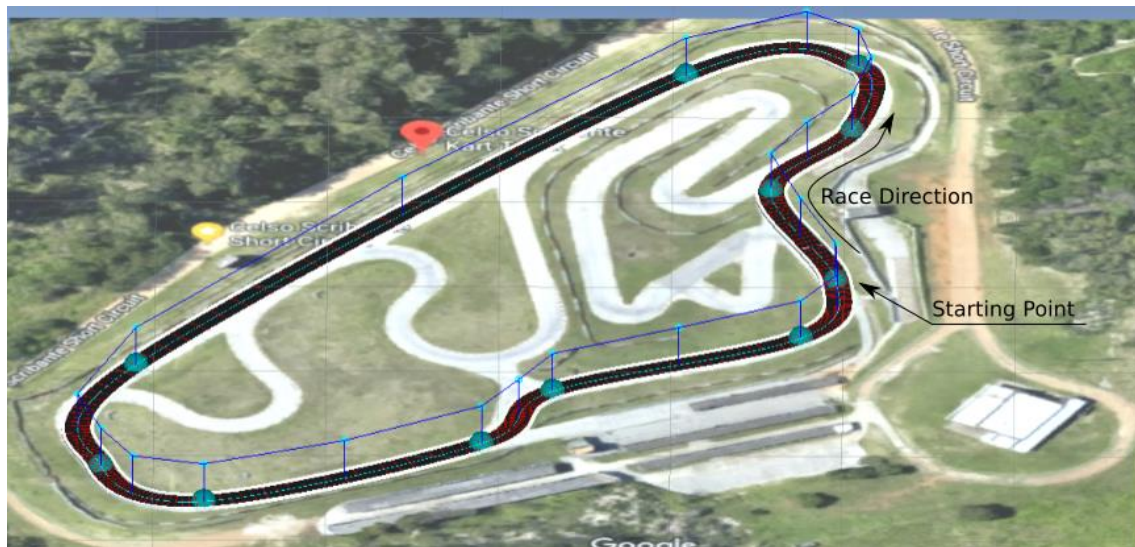


Figure 2: Scribante Racetrack imported into Unity

The “unseen” racetrack was created by modifying the above track, changing the start position, and changing the race direction. This racetrack would show whether the car had overfitted the racetrack. The idea of switching the race direction would be interesting, as the car may want to have a tendency to turn left, as the previous track was counter-clockwise.



Figure 3: Altered Scribante track, implemented as an unseen track

Next, a car was imported from Unity's Standard Assets Package. This car already incorporated realistic physics and dynamics models in a script (CarController.cs) which could be modified as needed.

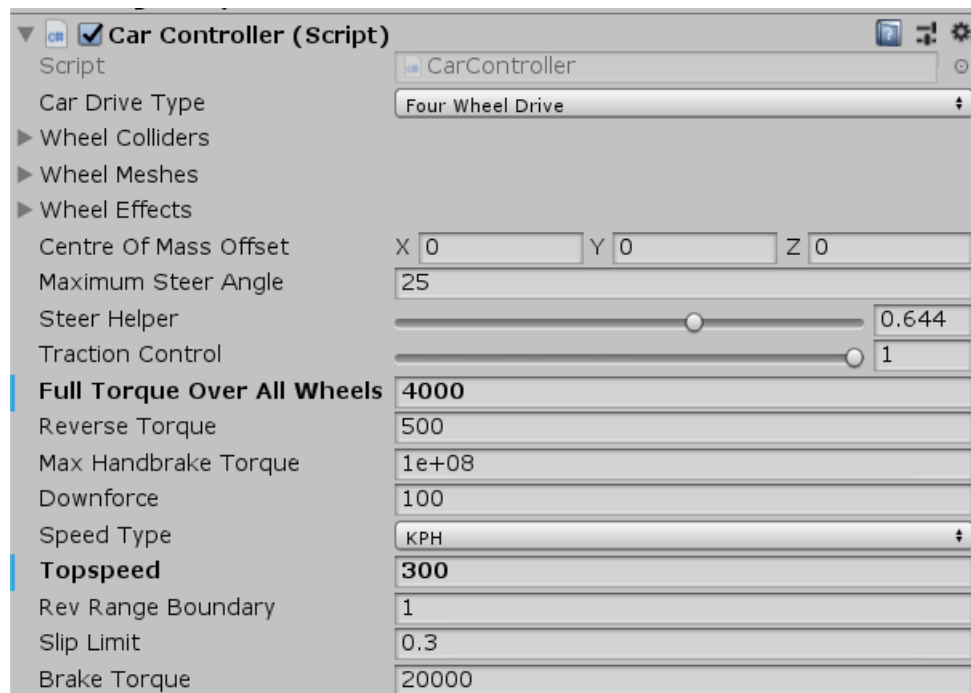


Figure 4: Physics and Dynamics of the Vehicle

The car controller script initially received a horizontal input from -1 to 1 corresponding to the left and right arrow keys (or joystick horizontal axis), and a vertical input also from -1 to 1. The horizontal input would control the steering angle, and the vertical input the throttle of the car. It was therefore proposed that these two values be controlled by the outputs of a neural network.

Before creating a neural network, it was necessary that some sort of sensory data be captured from the environment for input into the NN. This was implemented through "feelers", which would each measure the distance from the end of the feeler to the wall nearest the car.

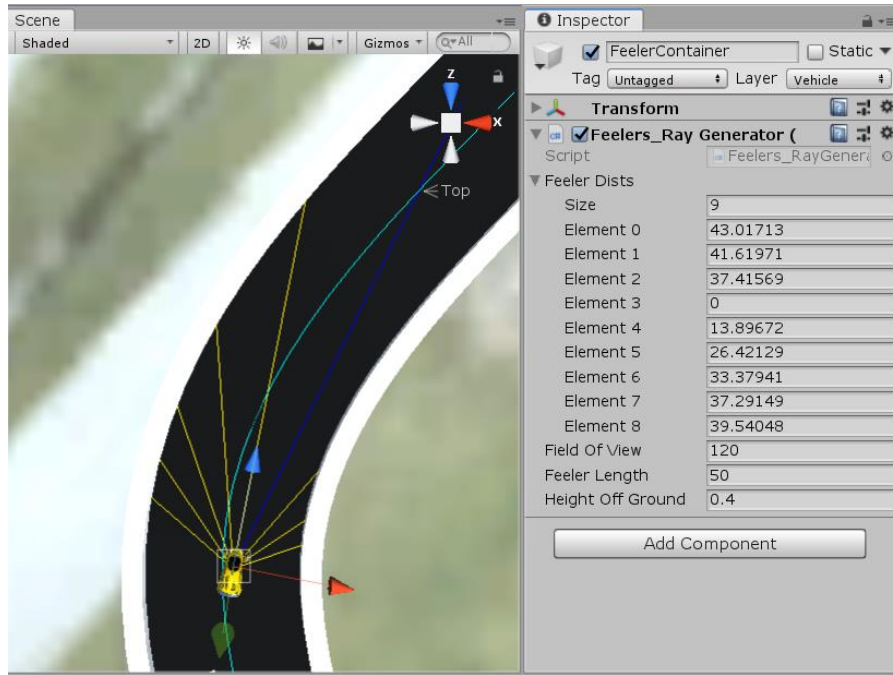


Figure 5: Sensory Data Captured from environment

A neural network script was then developed to output the horizontal and vertical values to the car controller script based on the feeler inputs. The NN would override the user's control of the vehicle. The topology of the neural network investigated is shown in Figure 6. The neurons with a dashed outline are recurrent neurons – i.e. they are outputs that feed back into the NN without actuating the vehicle. It is expected that the NN might learn to use these NN's to store data that it might find useful. For example, they may trigger when the car is at a certain point in a track (e.g. a straight section of the track might trigger one of these outputs). When the trigger is fed back as an input, it may cause the throttle to increase, and accelerate the car.

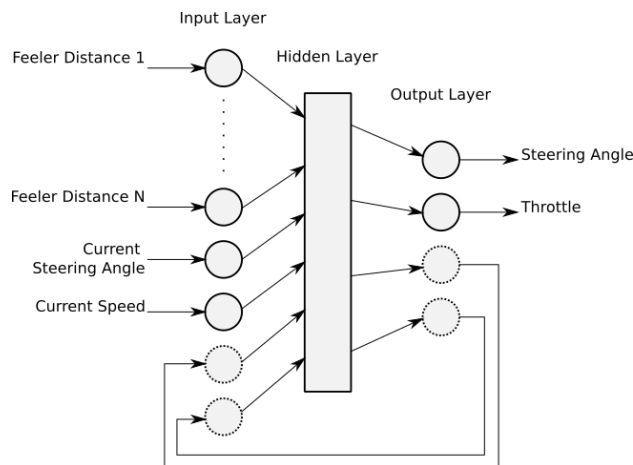


Figure 6: Neural Network Topology

Once the neural network was implemented for the individual vehicles a PSO script (PopulationManager.cs) was developed to control the training/optimization process of the neural networks in the vehicles. Changes to the performance of the vehicles are based on

tendency of the vehicles to emulate the success of other individuals. The population size consisted of 30 individuals.

Initially the algorithm followed a “batch” approach, only updated the weights once all 30 vehicles had terminated – to obtain the global best of the first generation. This was inefficient as when one individual started learning the track, it could take up to two minutes for it to crash, while most of the population would crash within the first few seconds. So about 90% of the individuals would be idle (not exploring any other permutations) until the last few cars eventually crashed.

So to improve the implementation, a continuous evolutionary approach was developed: where the vehicles are assessed, evolved and re-spawned immediately after they crash or are terminated – maintaining a total of 30 vehicles on the track at all times (and enabling more permutations to be explored within a shorter timeframe), constantly updating weights as individuals’ performance increases and the global performance increases. As shown in Figure 7, a red colour was used to indicate the car with weights corresponding to the global best fitness.

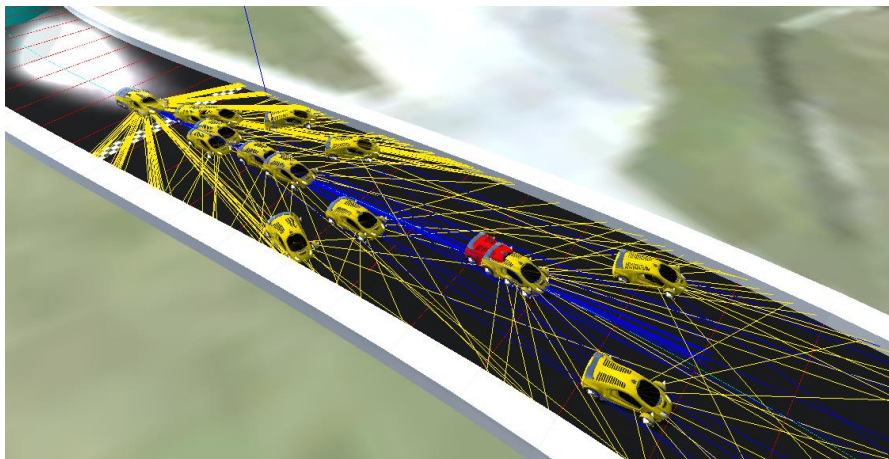


Figure 7: Visual Implementation of PSO training

Through testing, a PSO evolutionary algorithm was decided upon, it was found that the PSO algorithm could be made more efficient through continuous evolution. As soon as a car crashed, it was immediately reinitialized to the start position, and would continue evolving without waiting for the cars in the previous generation to finish. This would not have been possible if another algorithm was used, because in other neural networks, the individuals that would fail sooner, would be able to reproduce more and propagate their bad genes. Due to the nature of PSO, and that there is no reproduction of bad genes, each individual continuously improves over time which is why continuous evolution was more efficient.

Continuous evolution works in PSO because the global and personal bests are stored while the particles (current cars on the road) move on to explore more (possibly worse) solutions. “Crashed” particles can thus be evolved based on the particle’s most recent weights, its personal best and the global best at the time of its crash. There may be a car driving which would achieve a new global best, but the crashed particle doesn’t need to wait for it, as it’ll just base its next evolution on this newest global best next time it crashes.

5 Parameters

Feed Forward Neural Network Parameters:

Inputs = 12 (x9)

- Feeler Distance (x9)
- Current Steering Angle
- Bias

Hidden Layers = 1

Hidden Layer Size = 5

Outputs = 2

- Steering Angle
- Throttle

Initial neurons and weights are randomly initialized.

Activation Function

- *Sigmoid*: $f(net) = \frac{2}{1+e^{-\lambda(fnet)}}$

5.1 Recurrent Neural Network

The recurrent neural network has a similar structure to the FFNN, with an additional three inputs and outputs. These additional outputs are fed back in as inputs, which in theory will help the neural network model temporal characteristics of the track and over time help improve the performance if implemented successfully.

5.2 Particle Swarm Optimization

Population Parameters:

- Population Size = 30

PSO Parameters:

- Cognitive Constant = 0.7
- Social Constant = 1.0
- W = 0.6

Termination Conditions of vehicle fitness:

- If a vehicle crashes or completes the track too slowly

Terminating conditions of training:

- Best solution streak = 5000
 - If 5000 particles cannot beat the global best, training is terminated and the solution is saved:

$$Fitness = FinishingBonus * \frac{Distance\ Travelled^2}{Time}$$

Finishing Bonus Multiplier = 1 (i.e. $Fitness = 1 * Fitness$) if track not completed

Finishing Bonus Multiplier = 1.5 (i.e. $Fitness = 1.5 * Fitness$) if track completed

The fitness function above was decided upon as it optimized much faster than initially using distance and then adding time to the function later. When distance was the only initial fitness vehicles took too long to optimize.

6 Experiments Conducted

6.1 Fitness vs Generations for various number of hidden neurons

In this experiment, a population of cars was trained repeatedly, using various number of hidden neurons. The recurrent neurons were fixed at 3, and the hidden neurons were varied between the values: 0, 5, 10, and 20. The experiment then looked at how long each of the cars took to train and what fitness they obtained.

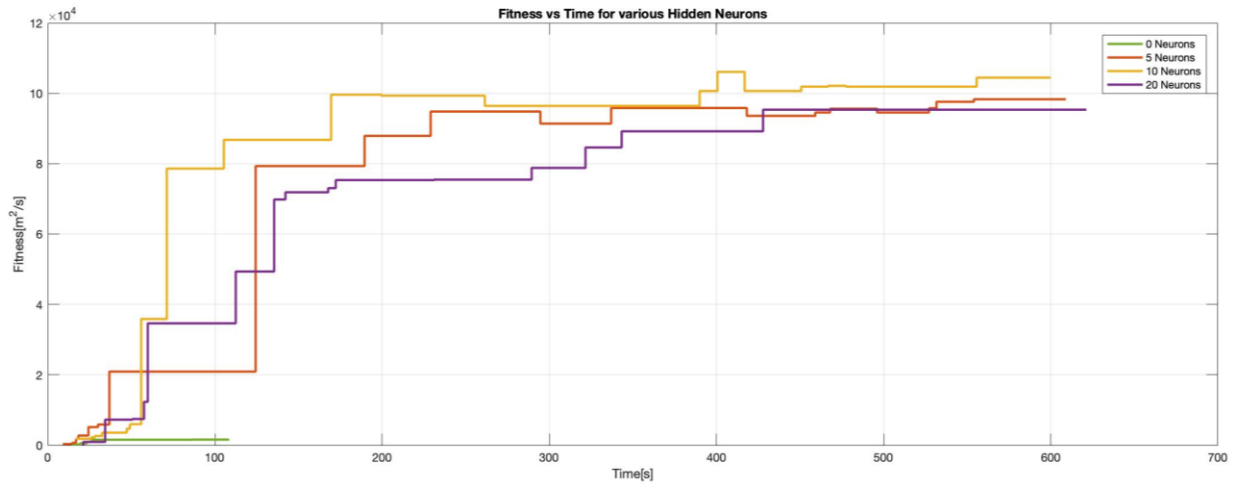


Figure 8: Fitness evaluation over generations for various numbers of hidden neurons

6.2 Throttle and Velocity vs Distance for various number of recurrent neurons

In this experiment, the number of recurrent neurons were varied between 0, 3, and 10. The number of hidden neurons were kept at 5. At first, the 3 cars were trained, then they were each driven around the training track. The speed and throttle profiles were plotted over the distance of the training track in Figure 9, and were plotted over the test track in Figure 10.

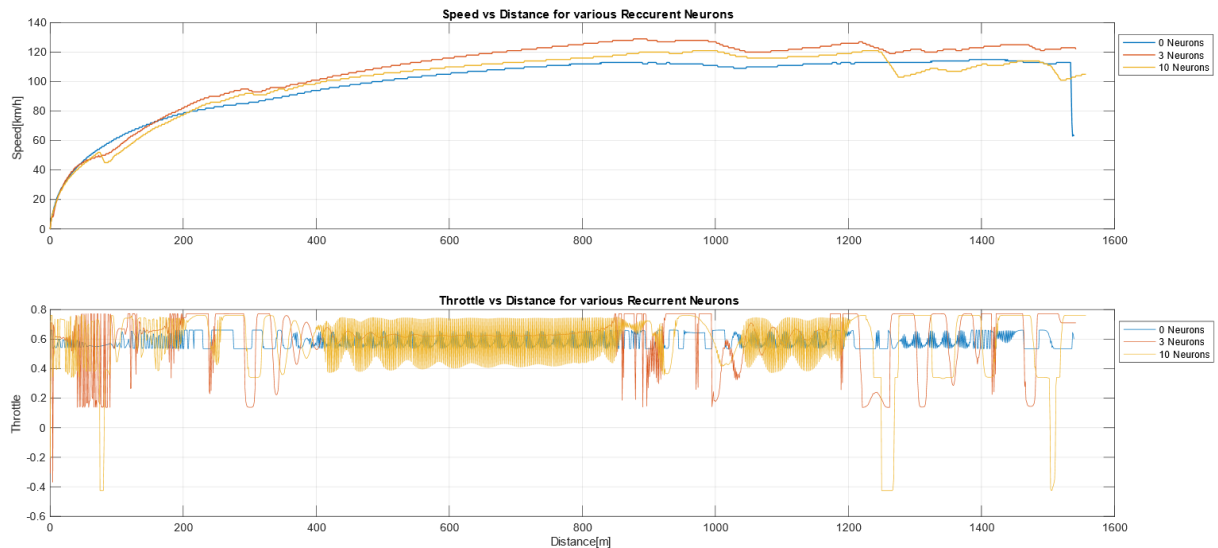


Figure 9: Speed and Throttle measured against the distance of the trained track for various numbers of recurrent neurons.

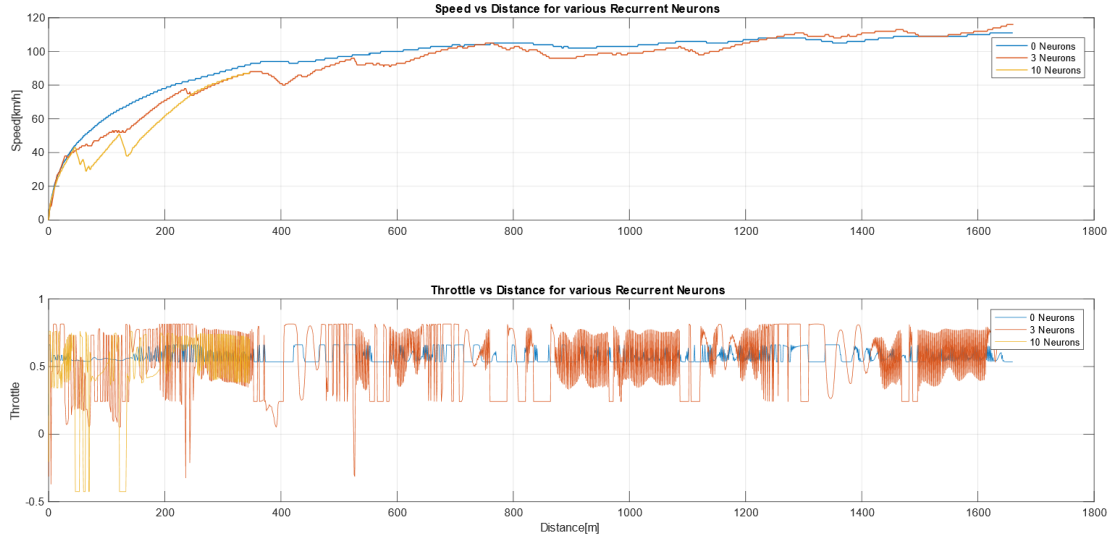


Figure 10: Speed and Throttle measured against the distance of an unseen track for various numbers of recurrent neurons.

6.3 Fitness vs Time for various number of recurrent neurons

In this experiment, the 3 cars were trained. One with 0, another with 3, and the last with 10 recurrent neurons. Each of the cars had 5 hidden neurons. It was investigated to see how quickly the cars trained and to what fitness.

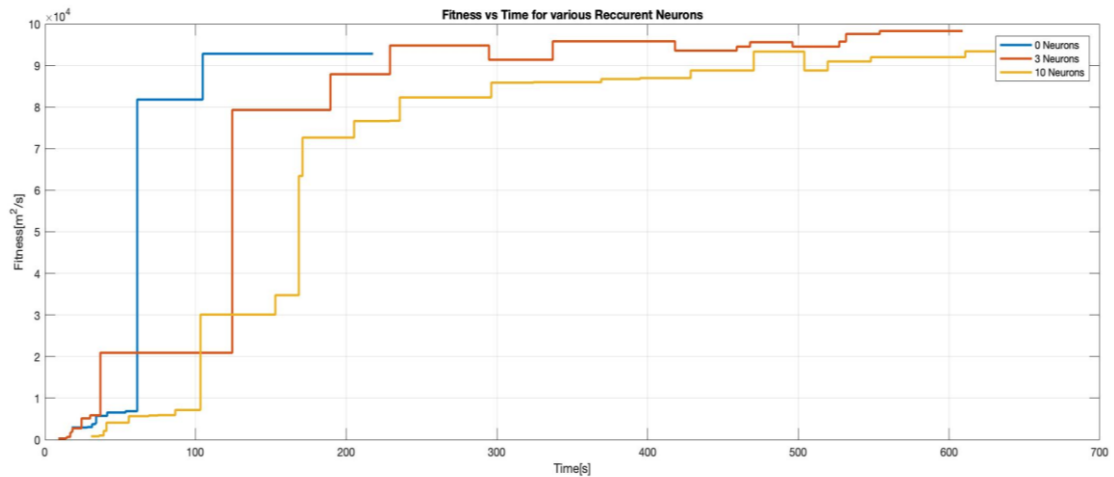


Figure 11: Fitness evaluation vs Time for various number of recurrent neurons

6.4 Repeatability

In this experiment, a trained car was made to navigate around the training track, and then the simulation was restarted with the same car. The speed and throttle vs time from both simulations were plotted to see if the outputs were the same, hence obtaining an idea of the repeatability of the experiments. The parameters of the car were: 3 Recurrent Neurons, 5 Hidden Neurons, and the track was the training track.

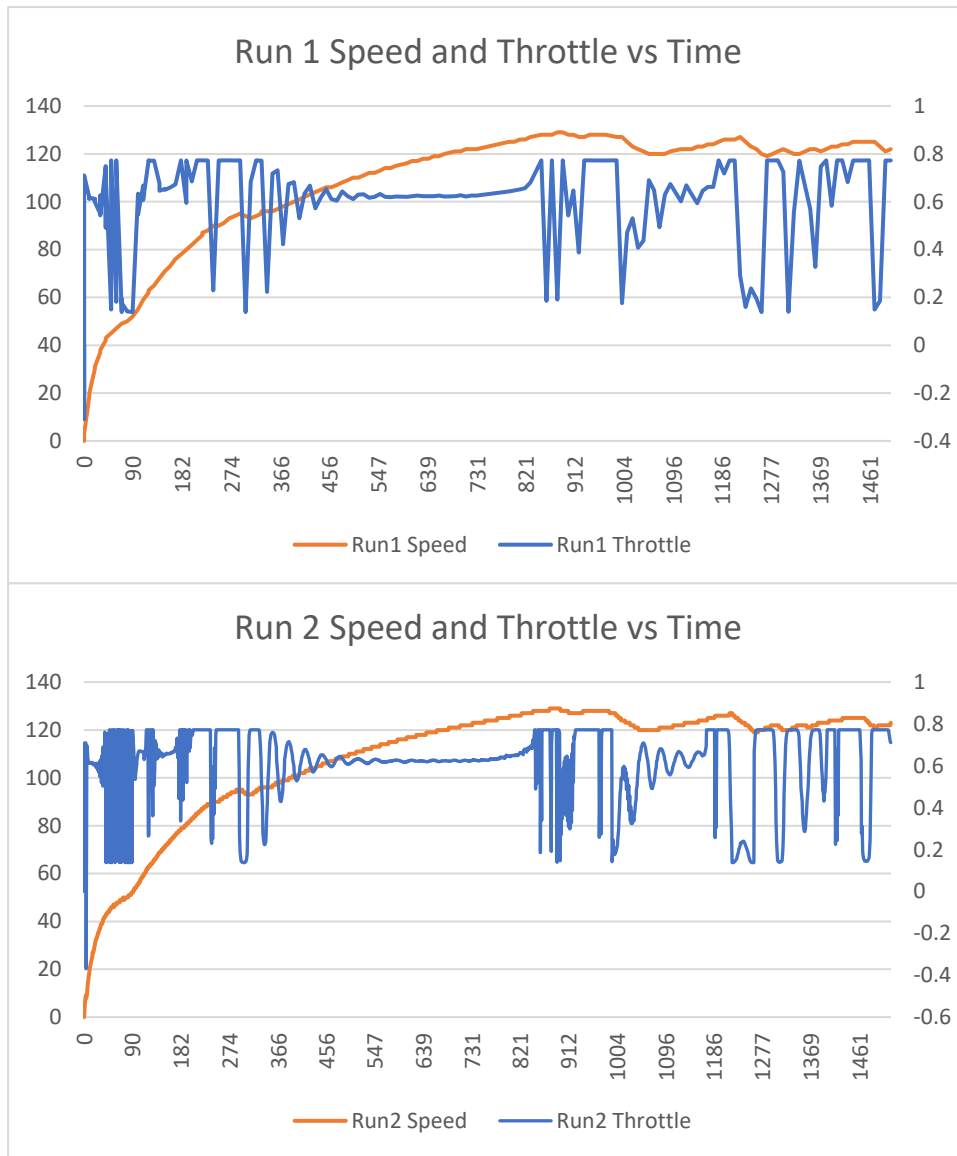


Figure 12: Testing repeatability of results

7 Observations

7.1 General Observations

As shown in experiment 6.1 and 6.3, a distinct jump in the fitness was found when the evolution had reached a point where a car had successfully navigated the whole track. In other evolutionary algorithms, this would mean cars that have completed the track would have quite a significant effect on the gene pool, as they would reproduce the most. However, in PSO it simply means that the car which finishes the track would be global best particle, and cars which crash will be gradually attract to that optimum, while passing through other positions. This encourages exploration.

Also, in experiment 6.1 and 6.3, another observation was that the best fitness would sometimes fluctuate down. This was because the weights of the car did not always produce the same fitness, due to variations in the processing power of the computer. The vehicle did not perform exactly the same each time it ran even if it had the same parameters and weights, as shown in experiment 6.4.

7.2 Experiment 6.1

This experiment showed that extreme sizes of hidden layer neurons can negatively affect the evolution. When the hidden layer size was too small, i.e. 1, the evolution did not do any optimization, and the fitness remained a very low value. On the other hand, when the hidden layer size was too large, i.e. 20, the population took noticeably longer to train, without much gain.

7.3 Experiment 0

One of the main reasons this experiment was conducted was to investigate the effect of the recurrent neurons on the car's ability to "remember" features of the track. This proved to be true. If the reader looks at the car with no recurrent neurons, it can be seen that the speed remains fairly constant from distance 400 to 1200 m. This range corresponds to the straight portion of the racetrack as seen in Figure 2. However, the car with 3 recurrent neurons made a significant improvement to this.

Firstly, the car accelerated more on the back straight, achieving a top speed of 120 km/h (vs 100 km/h in the previous case). The throttle, was also much smoother, compared to the oscillating throttle in the previous case.

Unfortunately, these gains were not carried through to the case of the 10 recurrent neurons. The reason for this could be that the training period was not long enough to sufficiently train the weights of the recurrent neurons to have a meaningful impact. However, the speed did improve on the training track when compared to the original case with no recurrent neurons.

In Figure 10, the same 3 cars were tested on the unseen racetrack. This test had a major implication on the findings. As shown in the speed graph, the car with no recurrent neurons had the smoothest speed profile. The other two cars had a fluctuating speed (i.e. a "jerky" ride), and the car with 10 recurrent neurons crashed quite early in the track.

In the throttle plot, a similar behaviour was seen, as the cars with the recurrent neurons tended to have an oscillating value for the throttle, while the car without recurrent neurons had a smooth throttle. The recurrent neurons caused the cars to "over-fit" the training track, in the sense that the cars were unable to navigate unseen tracks well. The car with the 10 recurrent neurons couldn't even complete the track without crashing.

7.4 Experiment 6.3

In this experiment, the effect of the number of recurrent neurons was investigated. It was clear that more recurrent neurons made training slower. This was shown by the fact that the time it took for a car to complete a track (i.e. to get a fitness above ~70000) took longer as the number of recurrent neurons went up. However, more recurrent neurons led to a higher fitness in the end although it took longer to train.

7.5 Experiment 6.4

In this experiment, it was clear that the results produced were repeatable. The overall trend of the velocity and throttle graphs remained the same when the run was repeated. This gives the reader some confidence that the data is reliable.

8 Conclusions

The main objective of this project was to create an Artificial Intelligence driven vehicle that can run through unseen racetracks without colliding with walls and the time the car takes per lap must be minimized as much as possible.

The first step was to create a car model using Unity Asset packages. Secondly, two racetracks were developed where the training and testing of performance would take place,

To control the car movements, the car -controller script received inputs from -1 to 1 which correspond to horizontal movement(steering) , and another set of inputs from -1 to 1 which correspond to the vertical movement(acceleration) of the car. It was then decided that these values would be controlled by the output of the neural network.

The final results show that the neural networks and genetic algorithms drive the car as expected, the car can manoeuvre around the meanderings of the racetrack without colliding with the walls.

Applications of this topic could include: Developing AI opponents for racing simulators and games.

Finding optimum driving strategies in real-life tracks that can be useful for race drivers.

Helping engineers optimize tuning parameters of vehicles.

For example, the evolution process can be re-run using various parameters for the vehicle. These parameters can affect engine performance (top speed, acceleration, etc.), change the mass of the vehicle, and affect vehicle handling.

9 Appendices – The code

9.1 Scripts that make up the Evolution/Training manager game object (containing PSO):

```
public class PopulationManager : MonoBehaviour
{
    // Parameters
    public int POPULATION_SIZE = 10;
    public int INITIAL_WEIGHTS_UPPER_BOUND = 1;
    public int MAX_GENERATIONS = 1500;
    public int BEST_INDIVIDUAL_STREAK = 5000;
    // The car population script Component
    public CarPopulation cars;
    // Individuals of the population's fitness
    private List<float> CurrentNN_Fitness = new List<float>();
    // Stored velocity of each individual particle
    private List<List<float>[]>> ParticleVelocityVectors = new List<List<float>[]>>();
    // Stored Personal Best of each individual
    private List<List<float>[]>> PersonalBestNN_Weights = new List<List<float>[]>>();
    public List<float> PersonalBestNN_Fitness = new List<float>();
    // Stored y-Hat of each individual (using Ring topology)
    private List<List<float>[]>> Y_HatNN_Weights = new List<List<float>[]>>();
    private List<float> Y_HatNN_Fitness = new List<float>();
    // Stored Global Best Weights and fitness achieved
    private List<float>[]> GlobalBestWeights = new List<float>[]>();
    public float GlobalBestNN_Fitness;
    // Links to used Game Objects
    public GameObject BestCarDemo;
    // PSO parameters
    public float cognitiveConst = 0.7f;
    public float socialConst = 1.0f;
    public float w = 0.6f;
    public float socialRandom;
    public float cognitiveRandom;

    // Other variables
    private float sessionStartTime;
    public int curGeneration;
    public int leadCounter;
    [SerializeField]
    private int numberCarsDriving = 0;
    [SerializeField]

    // Variables For Outputting Data
    public bool logOutputs = false;
    public string folderToSaveTo;
    private StreamWriter outputStream;
```

```

// Create genes(weights) property and instantiate and position cars
void Start()
{
    // Initialize some variables
    curGeneration = 0;
    leadCounter = 0;

    // Open logging files
    if (logOutputs && sessionStartTime == 0)
    {
        sessionStartTime = Time.time;
        Feelers_RayGenerator feelerSettings = cars.carPopulation[0].transform.GetChild(
0).gameObject.GetComponent<Feelers_RayGenerator>();
        NeuralNetwork NN = cars.carPopulation[0].GetComponent<NeuralNetwork>();
        // Initialize naming conventions (open/create all the files) add a time stamp l
ine to each
        if (folderToSaveTo.Length != 0)
            folderToSaveTo = folderToSaveTo + "/";
        string filepath = String.Format("{0}Log(Feelr#{1}len{2};NN-
w={3}socC={4}cognC={5};#recur={6};#hid={7})"
, folderToSaveTo, feelerSettings.feelerDists.Length,
feelerSettings.feelerLength, w, socialConst, cognitiveConst, NN.outputs-
2, NN.hLayer_size);

        outputStream = new StreamWriter("Assets/logs/" + filepath + ".txt", true);
        outputStream.WriteLine("//////////////////// START //////////////////////");
        outputStream.WriteLine("///// " + DateTime.Now);
        outputStream.WriteLine("///// " + filepath);
    }

    //Initialize Velocity Vectors to Zero
    ParticleVelocityVectors = new List<List<float[][]>>();
    for (int x = 0; x < POPULATION_SIZE; x++) {
        ParticleVelocityVectors.Add(InitializeParticleVelocityToZero());
    }
    // Can't evaluate fitness until raced at least once...
    // So can build that in on Update()
}

private void OnDestroy() {
    if (logOutputs)
    {
        outputStream.WriteLine("//////////////////// END //////////////////////");
        outputStream.Close();
    }
}

```

```

void FixedUpdate()
{
    numberCarsDriving = 0;
    foreach (GameObject cur in cars.carPopulation)
    {
        if (cur.GetComponent<CarSideEvolutionaryBehaviour>().isDriving)
        {
            numberCarsDriving++;
        }
    }
    // Still part of setup - run initialised cars once to get initial fitness
    if (curGeneration == 0)
    {
        // If no cars are still driving
        if (numberCarsDriving <= 0)
        {
            // Note: Have fitness calculated by each car as it crashes. Have a method t
            hat the car calls to decrease NumberCarsDriving by 1.
            // Initialize Personal bests (weights and fitness)
            PersonalBestNN_Fitness = new List<float>();
            PersonalBestNN_Weights = new List<List<float[][]>>();
            InitializeParticlePersonalBests(); // to the weights currently in the car a
            nd the resultant fitness

            // Initialize "Global" bests (y-hat) All-
            best and Ring topology (Based on Personal bests initially - then subsequently on Y-Hats)
            GlobalBestNN_Fitness = 0;
            GlobalBestWeights = new List<float[][]>();
            Y_HatNN_Fitness = new List<float>();
            Y_HatNN_Weights = new List<List<float[][]>>();
            UpdateGlobalAndY_HatRingTopologyBestVectors(true);

            // reset test cycle - new generation
            curGeneration++;
            EnableAllTheCrashedCarsNNs();
        }
    }
    // Start of evolutionary cycles
    else if (leadCounter < BEST_INDIVIDUAL_STREAK)
    {

        // have fitness calc in each car as it crashes. have a method that the car call
        s to decrease NumberCarsDriving by 1.
        // Fetch each car's latest NN fitness
        for (int curIndividual = 0; curIndividual < POPULATION_SIZE; curIndividual++)
        {
            if (!cars.carPopulation[curIndividual].gameObject.GetComponent<CarSideEvolu
            tionaryBehaviour>().isDriving)

```

```

        {
            // Update personal best
            if (GetA_CarsNN_Fitness(curIndividual) > PersonalBestNN_Fitness[curIndi
vidual])
            {
                PersonalBestNN_Fitness[curIndividual] = GetA_CarsNN_Fitness(curIndi
vidual);
                PersonalBestNN_Weights[curIndividual] = CloneOfWeights(GetA_CarsNN_
Weights(curIndividual));
            }
        }
    }

    // Recalculate latest ring Y_Hats
    UpdateGlobalAndY_HatRingTopologyBestVectors(false);

    // Calculate new velocity for each individual
    for (int bob = 0; bob < cars.carPopulation.Count; bob++)
    {
        if (!cars.carPopulation[bob].gameObject.GetComponent<CarSideEvolutionaryBeh
aviour>().isDriving)
        {
            List<float[][]> bobIndividual = GetA_CarsNN_Weights(bob);
            List<float[][]> bobVelocity = ParticleVelocityVectors[bob];
            List<float[][]> bobPersonalBest = PersonalBestNN_Weights[bob];
            //List<float[][]> bobYHat = Y_HatNN_Weights[bob];
            // for each dimension of the Vectors
            for (int layer = 0; layer < bobVelocity.Count; layer++)
            {
                for (int to = 0; to < bobVelocity[layer].Length; to++)
                {
                    for (int from = 0; from < bobVelocity[layer][to].Length; from++
)

                    {
                        socialRandom = UnityEngine.Random.Range(0f,1f);
                        cognitiveRandom = UnityEngine.Random.Range(0f,1f);
                        // Gbest
                        bobVelocity[layer][to][from] = w*bobVelocity[layer][to][fro
m] + cognitiveConst*cognitiveRandom*(bobPersonalBest[layer][to][from] - bobIndividual[layer
][to][from]) + socialConst*socialRandom*(GlobalBestWeights[layer][to][from] - bobIndividual
[layer][to][from]);
                    }
                }
            }
            // Apply new velocity to Particle
            for (int layer = 0; layer < bobVelocity.Count; layer++)
            {

```

```

        for (int to = 0; to < bobVelocity[layer].Length; to++)
        {
            for (int from = 0; from < bobVelocity[layer][to].Length; from++)
            {
                bobIndividual[layer][to][from] = bobIndividual[layer][to][from] + bobVelocity[layer][to][from];
            }
        }
        // Place new Gen weights back in the car
        cars.carPopulation[bob].GetComponent<NeuralNetwork>().weights = bobIndividual;
    }
    // reset test cycle - new generation
    //curGeneration++;
    leadCounter++;
    // Set cars to be driving again
    // ...here... Generation 1.. GO!
    EnableAllTheCrashedCarsNNs/*AndResetTimer*/();
}
else
{
    //Solution is reached..
    Debug.Log("PSO SOLUTION: Final fitness: " + GlobalBestNN_Fitness);
    if (logOutputs)
    {
        //Log final best fitness for graph
        outputStream.WriteLine((Time.time - sessionStartTime + ";" + GlobalBestNN_Fitness).Replace(',', '.'));
    }
    // Record the Solution's weights
    WriteBestWeightsToFile();
    Debug.Log("Best Weights saved to file...");
    leadCounter = 0;
    //Save BEST Global Weights to a new text file: or put them in one car and let it race
}
}

public void PositionCarAtStartLine(GameObject car)
{
    car.transform.position = cars.CarStartingPosition;
    car.transform.rotation = cars.CarStartingRotation;
}

```

```

public void ResetDemoCar()
{
    // Update NN with Best Weights
    if (BestCarDemo.GetComponent<CarSideEvolutionaryBehaviour>().isDriving != true)
    {
        if (GlobalBestWeights.Count != 0)
            BestCarDemo.GetComponent<NeuralNetwork>().weights = CloneOfWeights(GlobalBestWeights);
        BestCarDemo.GetComponent<CarSideEvolutionaryBehaviour>().fitness = GlobalBestNN_Fitness;

        // Reset and Start
        BestCarDemo.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
        BestCarDemo.GetComponent<Rigidbody>().velocity = Vector3.zero;
        BestCarDemo.GetComponent<CarSideEvolutionaryBehaviour>().isDriving = true;
        BestCarDemo.GetComponent<NeuralNetwork>().WakeUp();
        BestCarDemo.GetComponent<Timer>().ResetTimer();
    }
}

private List<float[][]> CloneOfWeights(List<float[][]> weightsToClone)
{
    List<float[][]> newCopy = new List<float[][]>();
    foreach (float[][] cur in weightsToClone)
    {
        float[][] newClone = new float[cur.Length][];
        for (int x = 0, length = cur.Length; x < length; x++)
        {
            newClone[x] = new float[cur[x].Length];
            for (int y = 0, innerLength = cur[x].Length; y < innerLength; y++)
            {
                newClone[x][y] = cur[x][y];
            }
        }
        newCopy.Add(newClone);
    }
    return newCopy;
}

public void WriteBestWeightsToFile()
{
    NeuralNetwork NN = BestCarDemo.GetComponent<NeuralNetwork>();
    string path = String.Format("Assets/Best weights Logs/Best_log_NNFormat({0}-HidUnits,{1}-Inputs).txt", NN.hLayer_size, NN.inputs);

    StreamWriter sr = new StreamWriter(path, true);
    sr.WriteLine("////////////////// START ////////////////////");
    sr.WriteLine("Hidden layers: {0}", NN.hiddenLayers);
}

```

```

sr.WriteLine("Hidden layer size: {0}", NN.hLayer_size);
sr.WriteLine("Hidden layers: {0}", NN.hiddenLayers);

sr.WriteLine("Fitness: " + GlobalBestNN_Fitness);
int layer = 0;
foreach (float[][] cur in GlobalBestWeights)
{
    for (int x = 0, length = cur.Length; x < length; x++)
    {
        for (int y = 0, innerLength = cur[x].Length; y < innerLength; y++)
        {
            sr.WriteLine("{0}:{1}:{2}:{3}", layer, x, y, cur[x][y]);
        }
    }
    layer++;
}
sr.WriteLine("////////////////// END //////////////////");
sr.Close();
}

// Initialise Particle velocity to zero
private List<float[][]> InitializeParticleVelocityToZero()
{
    List<float[][]> magnitudes = CloneOfWeights(cars.carPopulation[0].GetComponent<NeuralNetwork>().weights);
    foreach (float[][] cur in magnitudes)
    {
        for (int x = 0; x < cur.Length; x++)
        {
            for (int y = 0; y < cur[x].Length; y++)
            {
                cur[x][y] = 0f;
            }
        }
    }
    return magnitudes;
}

// Initialize Particle personal best weights to the weights currently in the car
private void InitializeParticlePersonalBests()
{
    for (int x = 0; x < cars.carPopulation.Count; x++)
    {
        PersonalBestNN_Weights.Add(GetA_CarsNN_Weights(x));
        PersonalBestNN_Fitness.Add(GetA_CarsNN_Fitness(x));
    }
}

```

```

// Switch the cars' NNs back on
private void EnableAllTheCrashedCarsNNs/*AndResetTimer*/()
{
    for (int x = 0; x < cars.carPopulation.Count; x++)
    {
        CarSideEvolutionaryBehaviour cur = cars.carPopulation[x].GetComponent<CarSideEvolutionaryBehaviour>();
        if (!cur.isDriving)
        {
            cur.isDriving = true;
            cars.carPopulation[x].GetComponent<NeuralNetwork>().WakeUp();
            cars.carPopulation[x].GetComponent<Timer>().ResetTimer();
        }
    }
}

// This will be moved to the CAR object // has been
/**/ Calculate fitness for a given car
private float CalculateFitnessTheWeightsAchieved(GameObject car)
{
    // Get Distance car travelled
    float dist = car.GetComponent<DistanceTracker>().distanceTravelled;
    // Get time elapsed
    float timeElapsed = car.GetComponent<Timer>().timeElapsedInSec/60;
    // some function to combine them
    return dist^2/timeElapsed;
}*/

private void UpdateGlobalAndY_HatRingTopologyBestVectors(bool initializationStep = false
)
{
    List<List<float[][]>> originWeights = PersonalBestNN_Weights;
    List<float> originFitness = PersonalBestNN_Fitness;
    int indexOfGlobalFittest = GetIndexOffittest(originFitness);
    if (GlobalBestNN_Fitness < originFitness[indexOfGlobalFittest])
    {
        // Update the global best weights
        GlobalBestNN_Fitness = originFitness[indexOfGlobalFittest];
        GlobalBestWeights = CloneOfWeights(originWeights[indexOfGlobalFittest]);
        // Reset Demo car
        BestCarDemo.GetComponent<CarSideEvolutionaryBehaviour>().ResetAndLogCarTermination(true);
        //Print Best Fitness
        Debug.Log("Best fitness:" + GlobalBestNN_Fitness + "; Generation: " + curGeneration + "; First weight: " + GlobalBestWeights[0][0][0]);

        if (logOutputs)
        {
            //Log change in best fitness for graph

```



```

        float time = Time.time-sessionStartTime;
        outputStream.WriteLine((time + ";" + GlobalBestNN_Fitness).Replace(',', '.'))
    );
    }
}

private int GetIndexOffittest(List<float> theFitnessValues, int[] subsetIndices = null)
{
    int indexOffittest = -1;
    if (subsetIndices == null || subsetIndices.Length == 0)
    {
        //Debug.LogError("Please implement GetIndexOffittest() for the full population"
    );

        indexOffittest = 0;
        float fitnessOfCurFittest = theFitnessValues[indexOffittest];
        for (int x = 1; x < cars.carPopulation.Count; x++)
        {
            float fitnessOfCur = theFitnessValues[x];
            if (fitnessOfCurFittest < fitnessOfCur)
            {
                indexOffittest = x;
                fitnessOfCurFittest = fitnessOfCur;
            }
        }
    }
    else
    {
        indexOffittest = subsetIndices[0];
        float fitnessOfCurFittest = theFitnessValues[indexOffittest];
        for (int x = 1; x < subsetIndices.Length; x++)
        {
            float fitnessOfCur = theFitnessValues[subsetIndices[x]];
            if (fitnessOfCurFittest < fitnessOfCur)
            {
                indexOffittest = subsetIndices[x];
                fitnessOfCurFittest = fitnessOfCur;
            }
        }
    }
    return indexOffittest;
}

private List<float[][]> GetA_CarsNN_Weights(int carIndex)
{
    if (carIndex < cars.carPopulation.Count)
    {

```

```

        return CloneOfWeights(cars.carPopulation[carIndex].GetComponent<NeuralNetwork>(
).weights);
    }
    else
        return null;
    }
    private float GetA_CarsNN_Fitness(int carIndex)
    {
        if (carIndex < cars.carPopulation.Count)
        {
            return cars.carPopulation[carIndex].GetComponent<CarSideEvolutionaryBehaviour>(
).fitness;
        }
        else
        {
            return 0;
        }
    }
}

```

```

public class CarPopulation : MonoBehaviour
{
    // Car objects - Population
    public List<GameObject> carPopulation = new List<GameObject>();

    // Links to used Game Objects
    public GameObject NeuralNetworkControlledCar;
    public GameObject StartingBlocks;
    public Vector3 CarStartingPosition;
    public Quaternion CarStartingRotation;

    // Start is called before the first frame update
    void Start()
    {
        // Find and store reference to Starting Block where to initialize the cars
        Transform startPosition = StartingBlocks.transform.Find("Start Position Solo");
        CarStartingRotation = startPosition.rotation;
        CarStartingPosition = startPosition.position;
        // Instantiate the car population
        int popSize = this.gameObject.GetComponent<PopulationManager>().POPULATION_SIZE;
        for (int x = 0; x < popSize; x++)
        {
            carPopulation.Add(Instantiate(NeuralNetworkControlledCar, CarStartingPosition,
CarStartingRotation) as GameObject);
        }
    }
}

```

9.2 Car-side evolutionary behaviour script:

```
public class CarSideEvolutionaryBehaviour : MonoBehaviour
{
    public GameObject evolutionManager;
    public float distanceTravelled = 0;
    Vector3 lastPosition;
    Vector3 StationaryPos;
    Vector3 StartPos;
    public float fitness;
    private float startTime;
    public bool isDriving;
    public bool isDemo = false;
    static int NumberCarsFinished = 0;
    private float finishingBonus;
    private bool culled;
    float tempStoreOfDistTravelled;

    // Start is called before the first frame update
    void Start()
    {
        isDriving = true;
        distanceTravelled = 0;
        lastPosition = transform.position;
        StationaryPos = transform.position;
        StartPos = this.gameObject.transform.position;;
        startTime = Time.time;
        finishingBonus = 1;
        culled = false;
    }

    // Update is called once per frame
    void Update()
    {
        distanceTravelled += Vector3.Distance(transform.position, lastPosition);
        lastPosition = transform.position;
        //Stationary?
        float t = Time.time - startTime;
        if (t > 5)
        {
            if (Vector3.Distance(StationaryPos, this.gameObject.transform.position) < 5)
            {
                GameObject car = this.gameObject;
                if (car.GetComponent<NeuralNetwork>().sleep == false && !isDemo)
                {
                    culled = true;
                    ResetAndLogCarTermination();
                }
            }
        }
    }
}
```

```

    }
    StationaryPos = transform.position;
    startTime = Time.time;
}
}

private void OnCollisionEnter(Collision other)
{
    if (isDriving == true) {
        ResetAndLogCarTermination();
    }
}

public void ResetAndLogCarTermination(bool usurped = false)
{
    if (!isDemo)
    {
        GameObject car = this.gameObject;
        PopulationManager popMan = evolutionManager.GetComponent<PopulationManager>();
        car.GetComponent<NeuralNetwork>().Sleep();
        CalculateTheIndividualsFitness();
        popMan.PositionCarAtStartLine(car);
        lastPosition = transform.position;
        isDriving = false;
        culled = false;
    }
    else
    {
        GameObject car = this.gameObject;
        PopulationManager popMan = evolutionManager.GetComponent<PopulationManager>();
        // Stop

        // Meant to be a check to penalize the known best if it crashes - but maybe
not working right
        if (!culled && NumberCarsFinished > 15 && !usurped)
        {
            float distCovered = Vector3.Distance(StartPos, this.gameObject.transform.po
sition);

            if (distanceTravelled != 0)
                tempStoreOfDistTravelled = distanceTravelled;
            float latestFitness = CalculateTheIndividualsFitness();
            if (latestFitness < 0.9f*popMan.GlobalBestNN_Fitness && tempStoreOfDistTrav
elled > 100 && isDriving)
            {
                this.gameObject.transform.position = StartPos;
                // need to get the corresponding Personal-
Best and adjust by same amount
                for (int x = 0; x < popMan.PersonalBestNN_Fitness.Count; x++)

```

```

        {
            if (popMan.PersonalBestNN_Fitness[x] == popMan.GlobalBestNN_Fitness
)
            {
                popMan.PersonalBestNN_Fitness[x] = latestFitness;
                popMan.GlobalBestNN_Fitness = latestFitness;
            }
        }
    }

    popMan.PositionCarAtStartLine(car);
    lastPosition = transform.position;
    distanceTravelled = 0;
    finishingBonus = 1;
    car.GetComponent<NeuralNetwork>().Sleep();
    isDriving = false;
    culled = false;

    popMan.ResetDemoCar();
}

private float CalculateTheIndividualsFitness()
{
    // Get the Time elapsed
    //float time = evolutionManager.GetComponent<Timer>().timeElapsedInSec; // Using a
master timer
    float time = this.gameObject.GetComponent<Timer>().timeElapsedInSec; // used if eac
h car has a timer

    if (distanceTravelled != 0)
        fitness = finishingBonus * Mathf.Pow(distanceTravelled,2)/time;
    // If too close to the starting Line make fitness negative
    float distCovered = Vector3.Distance(StartPos, this.gameObject.transform.position);
    if(finishingBonus == 1 && distCovered < 10 && distanceTravelled != 0)
    {
        fitness = - distCovered - 10;
    }
    distanceTravelled = 0;
    finishingBonus = 1;

    return fitness; // fitness is global -- so doesn't necessarily have to return
}

private void OnTriggerEnter(Collider other) {
    if (isDriving == true)
    {

```

```

        NumberCarsFinished++;
        float time = this.gameObject.GetComponent<Timer>().timeElapsedInSec;
        finishingBonus = 1.5f;//1000 * Mathf.Pow(distanceTravelled/time,1);
        ResetAndLogCarTermination();
    }
}
}

```

9.3 The Neural Network script:

```

public class NeuralNetwork : MonoBehaviour
{
    //[RequireComponent(typeof(CarController))]

    private CarController m_Car; // the car controller we want to use

    // Neural Network parameters
    public int hiddenLayers = 1;
    public int hLayer_size = 5;
    public int outputs = 4;
    public int inputs = 0;
    public float maxValue = 1f;
    public bool sleep = false;
    public float[] outInput;

    public float throttle;

    public float steering;

    // List of neuron outputs and weights
    public List<List<float>> neurons;
    public List<float[][]> weights { get; set; }

    private int layers;
    int size = 0;
    void Awake()
    {
        m_Car = GetComponent<CarController>();
        Feelers_RayGenerator feelerNum = this.GetComponentInChildren<Feelers_RayGenerator>(
    );
        size = feelerNum.feelerDists.GetLength(0);
        inputs = size +3 +(outputs-
2); //bias, carspeed and angle included in input layer, and the inputs for the recurrentNN
from output
        layers = hiddenLayers + 2; // total layers including input and output layers
        weights = new List<float[][]>(); //weight initialisation
        neurons = new List<List<float>>();
    }
}

```

```

outInput = new float[outputs-2];

// Assign Values to neurons
for (int i = 0; i < layers; i++)
{
    float[][] layerWeights;
    List<float> layer = new List<float>();
    int layerSize = getSizeLayer(i);
    if(i != hiddenLayers + 1) // checking that not the last layer (it is + 1, rather than + 2 because i is zero-based)
    {
        layerWeights = new float[layerSize][];
        int nextSize = getSizeLayer(i + 1); // size of the next layer
        for (int j = 0; j < layerSize; j++)
        {
            layerWeights[j] = new float[nextSize];
            for (int k = 0; k < nextSize; k++)
            {
                layerWeights[j][k] = getRandom();
            }
        }
        weights.Add(layerWeights);
    }
    //What is this for? Geoff = Think it stores the Neuron's value (Input/lastestResultantFNet)
    for (int j = 0; j < layerSize; j++)
    {
        layer.Add(0);
    }
    neurons.Add(layer);
}

private void FixedUpdate()
{
    if (!sleep)
    {
        Feelers_RayGenerator feelerNum = this.GetComponentInChildren<Feelers_RayGenerator>();

        float[] inputs = new float[size + 3 + (outputs - 2)]; // initialised size of inputs as the num of feelers + 2 vars(speed and angle),inputs from Output and bias
        int j = 0;
        for (int i = 0; i < feelerNum.feelerDists.Length; i++)
        {
            inputs[i] = feelerNum.feelerDists[i];
            j++;
        }
    }
}

```

```

        // j is to keep track of the counter used to add to the inputs list
        int io = 0; // to iterate through the outputs, with the first 2 items passed as
input to the car
        for (int i = j; i < j + outInput.Length; i++)
        {
            inputs[i] = outInput[io];
            ++io;
        }
        inputs[inputs.GetLength(0) - 3] = m_Car.CurrentSpeed;
        inputs[inputs.GetLength(0) - 2] = m_Car.CurrentSteerAngle;
        inputs[inputs.GetLength(0) - 1] = -1;//bias value

        Feedforward(inputs);
        //Get outputs to be used for recurrent NN
        for (int i = 2; i < outputs; i++)
        {
            outInput[i-2] = getOutputs()[i];
        }
        // pass the input to the car!
        steering = getOutputs()[0];
        throttle = getOutputs()[1];
        float h = getOutputs()[0];
        float v = getOutputs()[1];
        m_Car.Move(h, v, v, 0f);
    }
    else
    {
        m_Car.Move(0, 0, 0, 0/*h, v, v, 0f*/);
        m_Car.gameObject.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
        m_Car.gameObject.GetComponent<Rigidbody>().velocity = Vector3.zero;
        m_Car.Move(0, 0, 0, 0/*h, v, v, 0f*/);
    }
}

public void WakeUp()
{
    sleep = false;
}

public void Sleep()
{
    sleep = true;
}

public void Feedforward(float [] inputs)
{
    // Create reference to input layer.
    List<float> inputLayer = neurons[0];

```



```

// Replace values in input layer to input argument.
for (int i = 0; i < inputs.Length; i++)
{
    inputLayer[i] = inputs[i];
}

// Update neuron values in layers 1 to output layer
for (int j = 0; j < neurons.Count-1; j++)
{
    // Create reference to weights of layer j
    float[][] weightsLayer = weights[j];
    // value to keep track of the next layer
    int nLayer = j + 1;
    // Create reference to neuron values of layer j
    List<float> neuronLayer = neurons[j];
    // Create reference to neuron values of layer j + 1
    List<float> neuronNLayer = neurons[nLayer];
    // Updating values of all neurons of layer j+1
    // Looping through neurons of layer j+1
    for (int k = 0; k < neuronNLayer.Count; k++)
    {
        float fnet = 0;
        // Multiplying the value of each neuron in layer j by the each weight in la
        // Looping through the number of weights in a neuron from layer j
        for (int l = 0; l < neuronLayer.Count; l++)
        {
            //weight from neuron l of the jth layer to neuron k of the j+1th layer
            fnet += weightsLayer[l][k] * neuronLayer[l];
        }
        neuronNLayer[k] = Fnet(fnet);
    }
}

public List<float> getOutputs()
{
    return neurons[neurons.Count - 1 ];
}

public float Fnet(float x)
{
    float sigmoid = 2 / (float)(1 + Mathf.Exp(-x)) - 1;
    return sigmoid;
}

public float getRandom()
{
    return Random.Range(-maxValue, maxValue);
}

```

```

public int getSizeLayer( int i)
{
    int size = 0;
    if (i == 0)
        size = inputs;
    else if (i == hiddenLayers + 1)
    {
        size = outputs;
    }
    else
        size = hLayer_size;

    return size;
}
}

```

9.4 The distance sensor or “feeler” script:

```

public class Feelers_RayGenerator : MonoBehaviour
{
    public float[] feelerDists = new float[9];
    public float fieldOfView = 120;
    public float feelerLength = 70;
    public float heightOffGround = 0.4f;

    // Update is called once per frame
    void Update()
    {
        for (int i = 0; i < feelerDists.Length; i++)
        {
            feelerDists[i] = feelerLength - getDistInDir(-
fieldOfView / 2 + i * fieldOfView / (feelerDists.Length - 1));
        }
    }

    float getDistInDir(float angleDeg)
    {
        float angle = -(float)(angleDeg * Math.PI / 180);
        float dist;
        RaycastHit hit;
        //Getting direction vector, first in local space, then world space
        Vector3 directionVector = transform.localPosition + new Vector3((float)Math.Sin(ang
le), 0, (float)(Math.Cos(angle)));
        directionVector = transform.TransformVector(directionVector);
        LayerMask mask = 9;
    }
}

```

```

        if (Physics.Raycast(transform.position + new Vector3(0,heightOffGround,0), directionVector, out hit, feelerLength,mask))
        {
            Debug.DrawRay(transform.position + new Vector3(0,heightOffGround,0), directionVector * hit.distance, Color.yellow);
            dist = hit.distance;
        }
        else
        {
            Debug.DrawRay(transform.position + new Vector3(0,heightOffGround,0), directionVector * feelerLength, Color.blue);
            dist = feelerLength;
        }
        return dist;
    }
}

```

9.5 Other miscellaneous scripts:

```

public class Timer : MonoBehaviour
{
    public Text TimerText;

    public float timeElapsedInSec;
    private float startTime;
    // Start is called before the first frame update
    void Start()
    {
        startTime = Time.time;
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        timeElapsedInSec = Time.time - startTime;
        string minutes = ((int)timeElapsedInSec / 60).ToString();
        string seconds = (timeElapsedInSec % 60).ToString("f2");

        if (TimerText != null)
            TimerText.text = minutes + ":" + seconds;
    }

    public void ResetTimer()
    {
        startTime = Time.time;
    }
}

```

```

public class Speed : MonoBehaviour
{
    public Text SpeedText;
    public float speed2;
    public GameObject car;

    // Start is called before the first frame update
    void Start()
    {
        //lastposition = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        speed2 = car.GetComponent<UnityStandardAssets.Vehicles.Car.CarController>().CurrentSpeed;
        string SpeedToShow = Mathf.Abs(((int)speed2)).ToString();

        if (SpeedText != null) SpeedText.text = SpeedToShow + "km/h";
    }
}

```

```

public class LoadWeightsFromFile : MonoBehaviour
{
    public TextAsset File;
    // Start is called before the first frame update
    void Start()
    {
        SetCarsNN_WeightsToThoseReadInFromFile();
        this.gameObject.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
        this.gameObject.GetComponent<Rigidbody>().velocity = Vector3.zero;
        //this.gameObject.GetComponent<CarSideEvolutionaryBehaviour>().isDriving = true;
        this.gameObject.GetComponent<NeuralNetwork>().WakeUp();
        this.gameObject.GetComponent<Timer>().ResetTimer();
    }

    // Update is called once per frame
    void Update()
    {
        //this.gameObject.GetComponent<CarSideEvolutionaryBehaviour>().isDriving = true;
        this.gameObject.GetComponent<NeuralNetwork>().WakeUp();
    }
}

```

```

    }
    private void SetCarsNN_WeightsToThoseReadInFromFile()
    {
        NeuralNetwork NN = this.gameObject.GetComponent<NeuralNetwork>();
        string path = "Assets/Best weights Logs/";
        path = path + File.name + ".txt";

        StreamReader sr = new StreamReader(path);
        sr.ReadLine();//WriteLine("//////////////////// START //////////////////////
//");
        sr.ReadLine();//WriteLine("Hidden layers: ", NN.hiddenLayers);
        sr.ReadLine();//WriteLine("Hidden layer size: ", NN.hLayer_size);
        sr.ReadLine();//WriteLine("Hidden layers: ", NN.hiddenLayers);

        sr.ReadLine();//WriteLine("Fitness: " + GlobalBestNN_Fitness);
        foreach (float[][] cur in NN.weights)
        {
            for (int x = 0, length = cur.Length; x < length; x++)
            {
                for (int y = 0, innerLength = cur[x].Length; y < innerLength;
y++)
                {
                    float[] vals = Array.ConvertAll(sr.ReadLine().Split(':'),
float.Parse);
                    NN.weights[(int)vals[0]][(int)vals[1]][(int)vals[2]] = val
s[3];//sr.WriteLine("{0}{1}{2}:{3}",layer,x,y,cur[x][y]);
                }
            }
            //sr.WriteLine("//////////////////// END //////////////////////");
            sr.Close();
        }
    }
}

public class SpeedThrottleVsDistancePlotter : MonoBehaviour
{
    // Variables For Outputting Data
    public bool logOutputs = false;
    public string folderToSaveTo = "Throt&Velo&Dist";

    public string customLabel;
    private StreamWriter outputStream;

    private float startTime;
    private int lastPrintedDist;

    // Start is called before the first frame update

```

```

void Start()
{
    // Open logging files
    if (logOutputs)
    {
        lastPrintedDist = -1;
        startTime = Time.time;
        Feelers_RayGenerator feelerSettings = this.gameObject.transform.Ge
tChild(0).gameObject.GetComponent<Feelers_RayGenerator>();
        NeuralNetwork NN = this.gameObject.GetComponent<NeuralNetwork>();
        // Initialize naming conventions (open/create all the files) add a
time stamp line to each
        if (folderToSaveTo.Length != 0)
            folderToSaveTo = folderToSaveTo + "/";
        string filepath = String.Format("{0}{5}-
Log(Feelr#={1}len{2};#recur={3};#hid={4})"
, folderToSaveTo, feelerSettings.feeler
Dists.Length, feelerSettings.feelerLength, NN.outputs-
2, NN.hLayer_size, customLabel);

        outputStream = new StreamWriter("Assets/logs/" + filepath + ".txt"
, true);
        outputStream.WriteLine("////////// START //////////
////////");
        outputStream.WriteLine("///// " + DateTime.Now);
        outputStream.WriteLine("///// " + filepath);
    }
}

// Update is called once per frame
void Update()
{
    if (logOutputs)
    {
        float distTravelled = this.gameObject.GetComponent<CarSideEvolutio
naryBehaviour>().distanceTravelled;

        //float t = Time.time - startTime;
        //print every second if (t > 1)
        int curDist = (int)distTravelled;
        if (curDist != lastPrintedDist)
        {
            float time = this.gameObject.GetComponent<Timer>().timeElapsed
InSec;

            float throttleSetting = this.gameObject.GetComponent<NeuralNet
work>().throttle;

```

```

        float speed = this.gameObject.GetComponent<UnityStandardAssets
.Vehicles.Car.CarController>().CurrentSpeed;;
        string SpeedToShow = Mathf.Abs(((int)speed)).ToString();
        string lineToPrint = String.Format("{0};{1};{2};{3}",time,(int
)distTravelled,throttleSetting,SpeedToShow);
        outputStream.WriteLine(lineToPrint.Replace(',','.'));
        Debug.Log(lineToPrint);
        startTime = Time.time;
        lastPrintedDist = curDist;
    }
}

private void OnTriggerEnter(Collider other) {
    if (logOutputs)
    {
        float time = this.gameObject.GetComponent<Timer>().timeElapsedInSe
c;
        if (time > 10)
        {
            outputStream.WriteLine("//////// Lap Time: {0}",time);
            outputStream.WriteLine("////////// END //////////
////////");
            outputStream.Close();
            logOutputs = false;
        }
    }
}

```