|  |
| --- |
| Lady Manners School |
| NEA self-driving car |
| Archie McMullan |

|  |
| --- |
| LMS  [Date] |

Contents

[Abstract 1](#_Toc23763731)

[Analysis 1](#_Toc23763732)

[Background 1](#_Toc23763733)

[Supervisor 1](#_Toc23763734)

[End user 1](#_Toc23763735)

[The problem 1](#_Toc23763736)

[Algorithms 1](#_Toc23763737)

[Car physics 1](#_Toc23763738)

[Track generation 1](#_Toc23763739)

[Training the neural-network 1](#_Toc23763740)

[Objectives 1](#_Toc23763741)

[Design 1](#_Toc23763742)

[Car simulation 1](#_Toc23763743)

[Track generation 1](#_Toc23763744)

[Neural network 1](#_Toc23763745)

[Complex structures **Error! Bookmark not defined.**](#_Toc23763746)

[Uml class diagrams **Error! Bookmark not defined.**](#_Toc23763747)

[Technical Solution 1](#_Toc23763748)

[Testing 1](#_Toc23763749)

[Evaluation 1](#_Toc23763750)

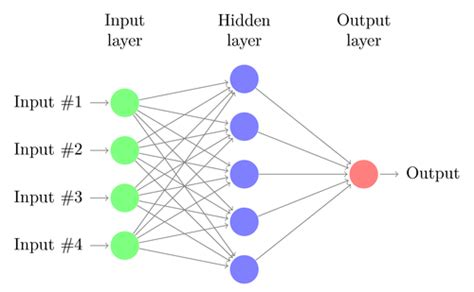
# Abstract

The purpose of this project is to teach a neural-network to drive a 2D car around a track. The use of neural-networks is likely only to increase in the future and so the purpose of this project is to educate people.

# Analysis

## Background

A neural network is a mathematical function that can be ‘trained’ to behave in certain ways. It is made up of; neurons, which hold a value; the connections between them, each a fixed value and biases, again a fixed value. These can all be thought to be arranged in layers as in the diagram.

https://tse2.mm.bing.net/th?id=OIP.BQ0SxdqC9Pl\_3ZQtd3e45AHaEk&pid=Api

To go forward through the network the value of each neuron is equal to the sum of the previous layers neurons times by their corresponding weights and the bias added. This continues until the output layer. The values of the neurons in this layer are the output of the network. To train the network to produce a given output for a given input the values of the weights and biases are changed in a process called training. in this way the network can ‘learn’ patterns of what to output for what inputs and so represent an invaluable tool in today’s world.

<https://mlfromscratch.com/neural-networks-explained/>

Despite their first being designed back in the back in the 1950s, neural networks have only recently started to become widely used. Despite this they have already had a great effect in many areas from stocks markets to video recommendations and will only increase in utility in the future. One of the areas that will certainly benefit from this form of computer control is self-driving cars. Whilst they have not yet reached the level where they can used by all, it is possible that within the next ten years they will have completely replaced human drivers.

My project will centre on educating people on the workings of neural networks and how these are being applied to the problem of self-driving cars. My intention is that teachers will be able to use my project to aid explanations of machine learning but also to develop people's interest in computer science.

## Supervisor

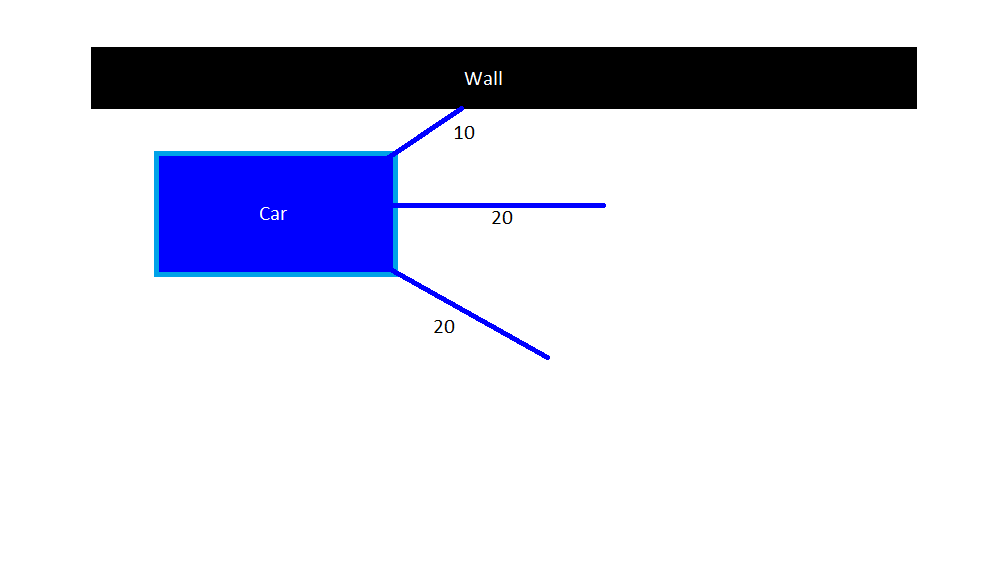
# TODO

End user

The end user for my project will likely be teachers and students who wish to gain insight into the way’s computers learn through a game like interface that will make the subject more approachable than trying to learn from a book. With this in mind, my software should be easy to use but more importantly should show visually the steps of the car learning in order to enhance its teaching ability.

## The problem

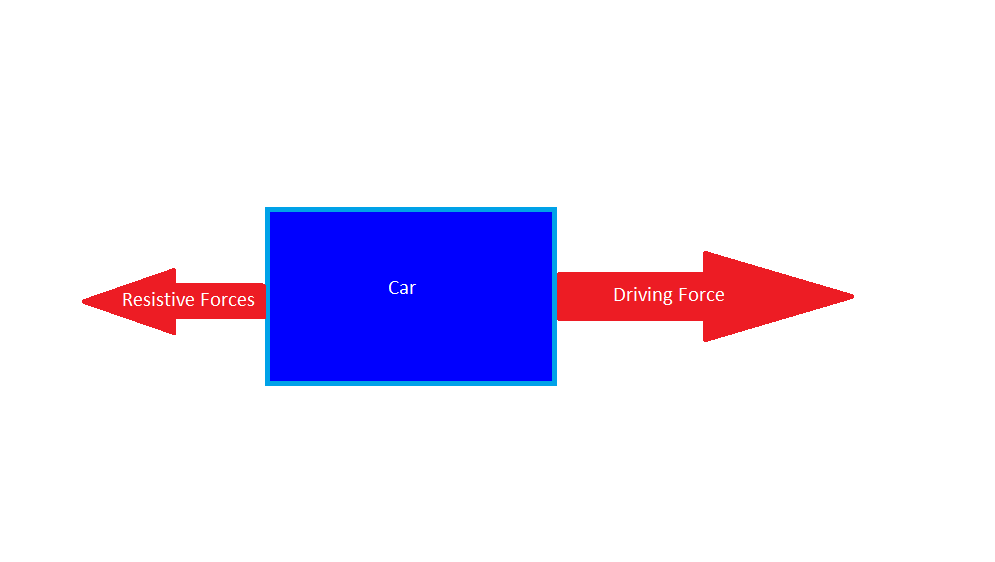
I am aiming to create a 2d top down driving simulation, that will generate race tracks for a car to drive around. A neural network will be fed the distance to the nearest wall along lines drawn at angles around the car (see image). If the car hits the side of the track then this should be detected and have the car try again. To aid the programs teaching ability, the user will be able to change features of the neural network for example the sizes and amount of the hidden layers, they will then be able to see how this affects its learning ability. To stop people having to train the network each time, the user will be able to save and load previously trained networks and have them drive around the track.



## Algorithms

### Car physics

To simulate the movement of the car I have decided to use the driving force and drag forces acting on the car to calculate the resultant force on the car. As F=ma I will be able to use this resultant force to calculate the acceleration on the car. Using this acceleration and the time between frames I will be able to calculate the change in velocity between the frames. I can then add this value to the car's velocity. I can then use the velocity and the time between frames to calculate the distance the car has moved between frames and so update the position. I believe using this method with the forces and numerically integrating up to the change in position will give me the most realistic movement of the car as I can vary the drag forces based on the velocity of the car, as seen in real life.

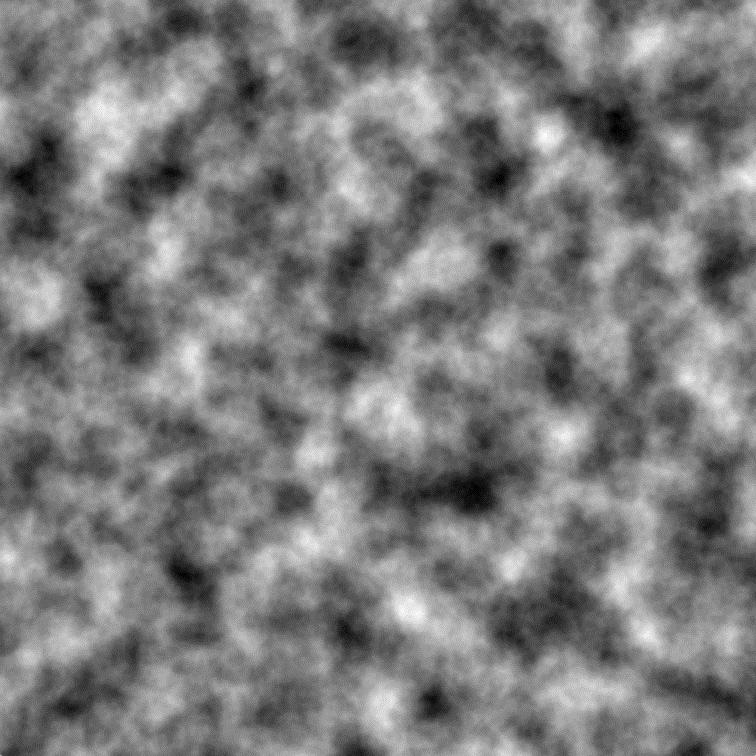


For the turning of the car I will vary the angle of the ‘wheels’ of the car and then use this to calculate the angular velocity and similarly ‘integrate’ this to find the angle the car has turned through.

### Track generation

In order to generate the race track I could use several methods. One of the methods would be to use a cellular automaton similar to Conway’s game of life etc. with set rules for each cell on a grid that would be the computer window using the cells around each to decide whether it is wall or track. This method would likely work well but would produce tracks out of large squares which is not the look I want for my software.

Another way would be to use Perlin noise to generate the tracks. Perlin noise is a type of gradient noise that takes coordinates and return a random value. The values close together will have similar magnitudes (as seen in the image by the smooth changes in the values) making this useful for the track generation as normal random values would lead to sharp spikes instead of smooth curves for the track. I could use these values as the distance from the centre of the track and so generate points around the track that I would then join to create the track. The use of gradient noise is common among procedurally generated video games for instance Minecraft uses it to generate its landscapes.

[https://cdn-images-1.medium.com/max/1200/1\*vs239SecVBaB4HvLsZ8O5Q.png](https://cdn-images-1.medium.com/max/1200/1*vs239SecVBaB4HvLsZ8O5Q.png)

### Training the neural-network

There are lots of different ways to train neural networks one way is back propagation. This involves tweaking the values in the neural network in a way that will lead to the desired output for a given input this is an extremely powerful method that can lead to impressive results however it requires lots of training data to learn that I would have generate by driving the car around the track and recording the values. This would make the software less fun to use as students would have to spend lots of time generating this data themselves and the cars ability would be limited to the ability of the students as the neural network would be seeking to recreate the way they drive.

Another way to train neural networks is to copy the way nature evolves animals. You start with random values in the neural network and then test how well they function. With all the neural networks evaluated you ‘breed’ the best creating new networks with similar properties to the best of the former generation and some with mutations to create desirable changes in the networks. This method will learn on its own without the need to generate training data and so will be more desirable for my project.

## Objectives

1. Simulation will be 2d and have a graphical user interface
2. OOP will be used to hold the separate objects relevant to the task
3. When the program starts a start screen will appear
   1. There will be a graphical input box for each layer to change the size of the layers
   2. There will be an input box to change the size of the population
   3. There will be an input box to change the mutation rate
   4. On the screen there will be a ‘Start New’ button that will start training a new network from scratch with the user specified dimensions, population size and mutation rate
   5. There will be an input box to select a saved neural network
   6. There will be ‘Load’ button that will load the selected neural network and start training it with the user specified population size and mutation rate
4. When a track is generated
   1. A 2d track with a white border will appear on screen
   2. Green will fill the rest of the screen
   3. A 2d car will be placed in the middle of the track nearest the top of the screen
5. For simulating the car
   1. The cars motion will be simulated using the forces on the car
   2. It will be able to travel forwards and backwards
   3. The networks will be able to control the steering angle of the car and so cause it to turn
6. When a network starts training
   1. a track will be generated
   2. Each member of the population will attempt to drive the car around
   3. If a network crashes into a wall this will be detected and the next network will have its attempt resetting the position of the car each time
   4. For each network attempt a fitness will be assigned based on how far it got around the track and, if it completed a circuit, the time it took
   5. The network will then be sorted based on this fitness and the best networks ‘bred’ and the new population created
   6. For each new generation a new track will be generated

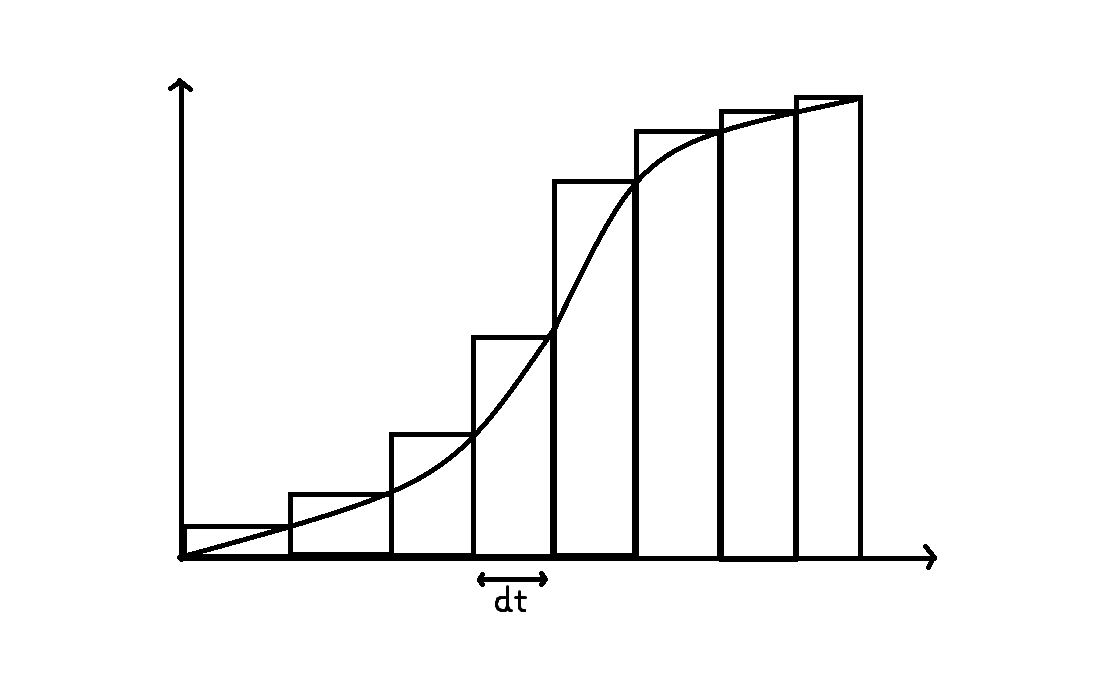
# Design

## Car simulation

In order to simulate the motion of the car I will be using the forces acting on it at any given time to find its acceleration and then using this to find its velocity. Finally, from the velocity I will find how far the car moves between each frame and update its position.

For the purposes of my simulation there will only be two forces on the car the driving force and the resistive forces. The driving force of the car, when applied, will have a fixed magnitude this is a simplification as in reality force from the engine would vary at different rpms etc. The direction of the force will either be forwards with respect to the car or backwards this will allow the agent controlling the car to drive it forward or backwards. The resistive forces of the car will be modelled as a single force acting in the opposite direction to the motion of the car with a magnitude proportional to the square of the car's velocity. By adding these two forces with the directions of the forces maintained the resultant force on the car can be calculated. According to Newton’s second law of motion the acceleration of the car can be calculated by dividing the resultant force by the mass of the car.

As acceleration is the rate of change of velocity, if you integrate the acceleration with respect to time you find the change in velocity. Similarly, to get the change in displacement I will integrate the velocity with respect to time.



To integrate these values, I will take their value between each frame and multiply it by the time between the frames. This is better shown in the graph above where it is clear that an approximate for the area under the graph can be acquired by finding the area of all the rectangles on the graph.

An example of how this might work in pseudocode is below:

IF forward pressed THEN

DrivingForce <- carForce # note the carForce is just a constant that will be tuned

IF backward pressed THEN

DrivingForce <- -carForce

ResistiveForce <- -dragCoefficient\*(CarVelocity)^2 # the dragCoefficient is another constant

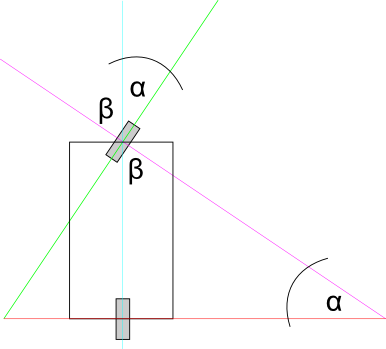
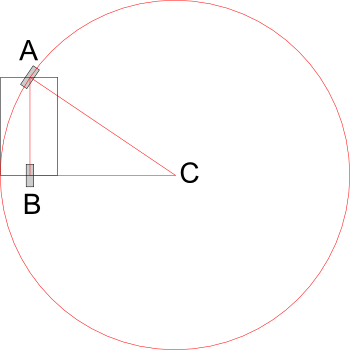
ResultantForce <- DrivingForce + ResistiveForce

Acceleration <- ResultantForce/carMass # uses F=ma and carMass is a constant

Velocity <- Velocity + Acceleration\*timeBetweenFrames

Displacement <- Displacement +Velocity\*timeBetweenFrames

In order to handle the steering of the car I will model the car as having one wheel at a certain steering angle. Using this and a bit of geometry we can find the turning radius of the car and from this and the velocity of the car we can find the angular velocity (how much of a circle it turns in a second).

<http://rmgi.blog/pygame-2d-car-tutorial.html>

If alpha is the steering angle of the car that can be a defined value based on input then the turning radius is equal to the car’s length divided by the sine of the steering angle. As we have the cars velocity and the radius of its circle, we can find its angular velocity as:

angular velocity =velocity/turning radius # note that this gives the value in radians

From this we can use the same trick with the integrating to find the change in the angle between the frames and add this to the angle the car already had to find the cars actual angle.

An example of how this might work in pseudocode is shown:

IF steeringLeft THEN

SteeringAngle <- maxAngle # maxAngle is a constant that will be tuned

IF steeringRight THEN

SteeringAngle <- -maxAngle

TurningRadius <- carLength/sin(SteeringAngle) #

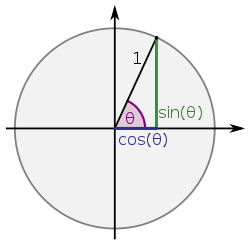
AngularVelocity <- CarSpeed/TurningRadius

ChangeInAngle <- AngularVelocity\*timeBetweenFrames

CarAngle <- CarAngle+ChangeInAngle

Now we can find the cars angle in any frame and the distance it moves in the frame we need to combine the two to find the cars new position.

To do this we will again use geometry:

<https://upload.wikimedia.org/wikipedia/commons/thumb/7/72/Sinus_und_Kosinus_am_Einheitskreis_1.svg/250px-Sinus_und_Kosinus_am_Einheitskreis_1.svg.png>

If θ in the diagram is the cars angle at a given time and its distance moved is known, its change in x and y coordinates can be found by multiplying sin(θ) for y and cos(θ)for x by the distance moved.

In order to store all the values that pertain to the movement of the car I have decided to create a ‘Car’ class. It will store: the coordinates of the car, its angle, its speed, its acceleration etc. It will have methods that take inputs (this will be the forward, backward etc) and processes the cars movement as described earlier and return the coordinates of the car on screen and its, angle.

A UML class diagram might look something like this:

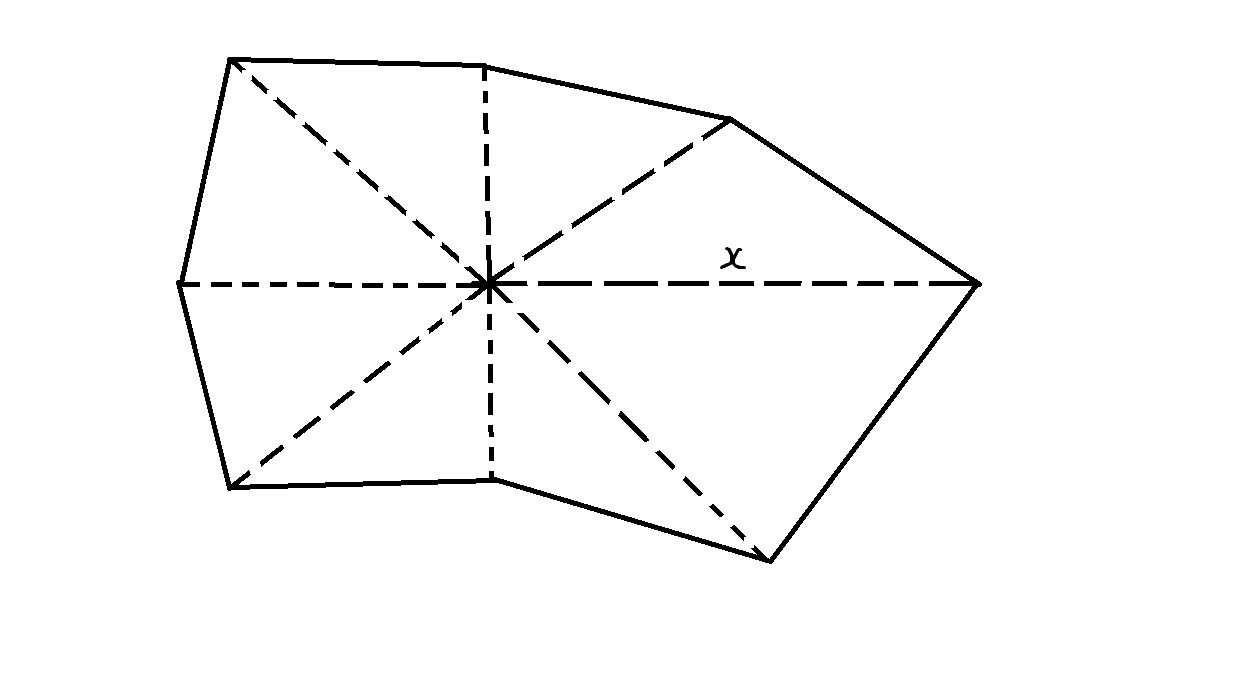
|  |
| --- |
| Car |
| -Xcoords: int  -Ycoords: int  -angle: float  -speed: float  -acceleration: float  -steeringAngle: float |
| +updatePosition(inputArray) |

Note that inputArray will likely have 5 items each either a 0 or a 1 to pass the controls to the car class e.g if inputArry[0] = 1 then drive forward.

## Track generation

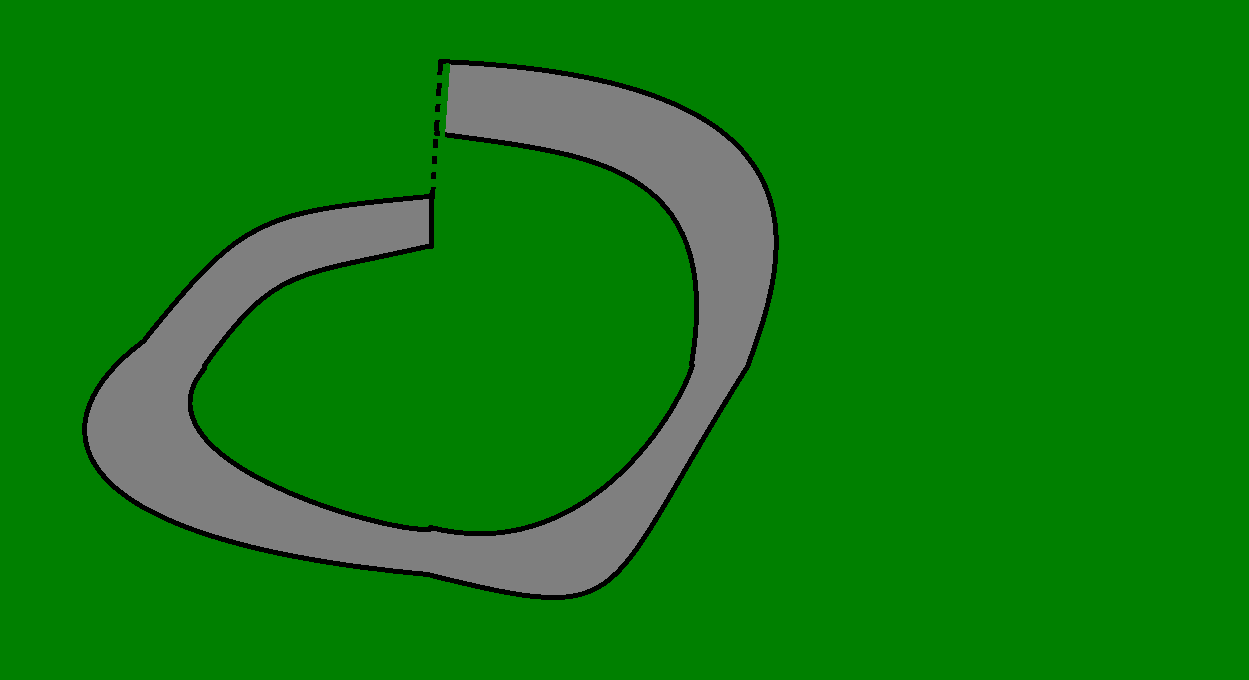
In order to generate the track, I will use a form of gradient noise to get smoothly random values that I will then use to place points random distances away from the centre of the screen. By joining these points together around the circle, I will generate my track.

My plan is to use find points a random length from the centre of the track, each line will be a set number of degrees around the circle and so will come back around to the start. This may be better understood if you look at the diagram below where each line is a random value X long and each line is spaced by 45 degrees, though I will use a smaller change in the angle to give the track a higher ‘resolution’.

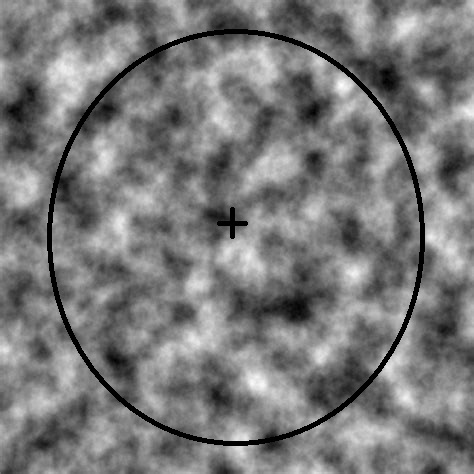


To generate the random lengths, I will use gradient noise. When using gradient noise, you input coordinates into a function and it returns a random value. If two coordinates are close together then they will have similar magnitudes, this makes it very useful for my track generation as unlike regular pseudorandom numbers it will result in a smooth track with the random points close to each other forming a smooth.

As this stands this still leads to a problem where by the start and the end of the tracks circle may not line up as the random values at the start and at the end are not from similar coordinates on the perlin noise plane. This problem is shown in the diagram.



A way to solve this is to pick the random values from the Perlin plane around a circle, this retains the property of having the coordinates for consecutive values close together but also means that when track loops back, it has similar values once again and so smoothly joins back up with its self. In order randomise the values for each track I will change the centre of the circle about which the values are taken.

<https://tse4.mm.bing.net/th?id=OIP.bCD-dUSkWFVrfHDrCFZEdwHaHa&pid=Api>

(This image is an example of how the values will be taken from around the circle)

In order to find the coordinates from around the centre of the circle I will use the parametric equations for a circle, these being:

Xcoord = Cos(θ)\*radius + Xcoord of centre

Ycoord = Sin(θ)\*radius + Ycoord of centre

In these equations theta is the angle around the circle you have gone, for the simplicity of the algorithm I will use this angle for both the angle around the circle in the noise and the angle around the centre of the track.

In order to find to coordinate of the point around the track I will again use the parametric equations this time replacing the radius of the circle with the random length acquired from the Perlin noise.

The pseudocode for this algorithm might look something like this:

CentreX <- random number

CentreY <- random number

FOR angle between 0 and 360

NoiseXcoord <- CentreX + cos(angle)

NoiseYcoord <- CentreY + sin(angle)

DistanceFromCentre <- perlinNoise(NoiseXcoord, NoiseYcoord)

PointX <- DistanceFromCentre\*cos(angle)

PointY <- DistanceFromCentre\*Sin(angle)

APPEND to innerList (PointX, PointY)

DistanceFromCentre <- DistanceFromCentre + 50 #add constant to give width

PointX <- DistanceFromCentre\*cos(angle)

PointY <- DistanceFromCentre\*Sin(angle)

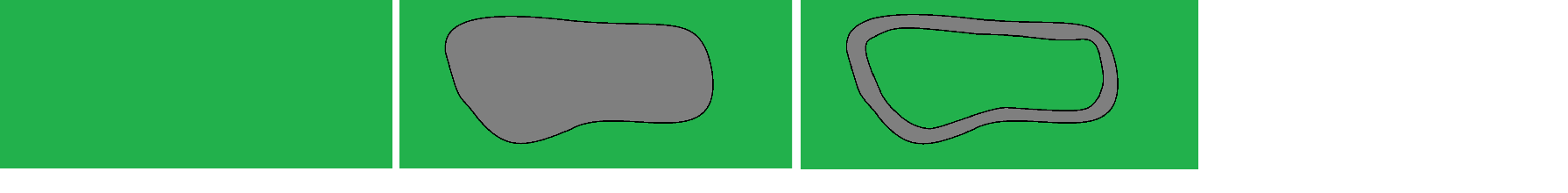
APPEND to outerList (PointX, PointY)

To make my program more modular and therefore easy to manage, I will make a class for the track.

A UML class diagram might look something like this:

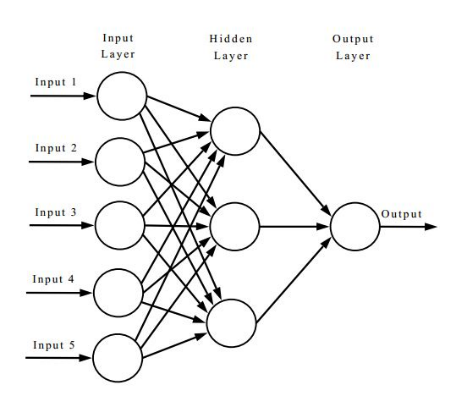
|  |
| --- |
| Track |
| -innerList: array  -outerList: array |
| +generatePoints |

In order to show the track on the screen I will be using the library pygame. This allows you to fill in a shape with colour given its vertices. Therefore, to display my track I will fill in the screen with green then the outerList shape with the grey of the track, and the innerList with green again. These shapes will overlap and so show the track.



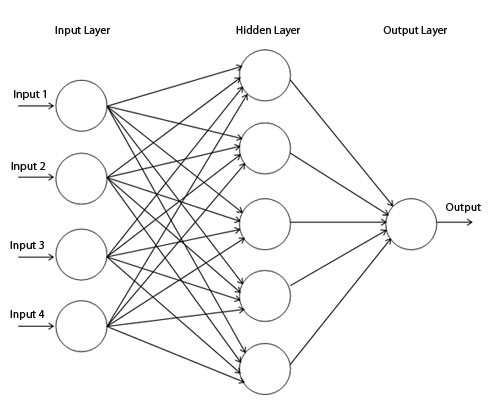
## Neural network

In order to store, generate and train I will create a neural network class. For the purposes of my program I want the dimensions of my network to be changeable and so when instantiating a network, it will be passed an array with the number of neurons in each layer contained.

For instance:

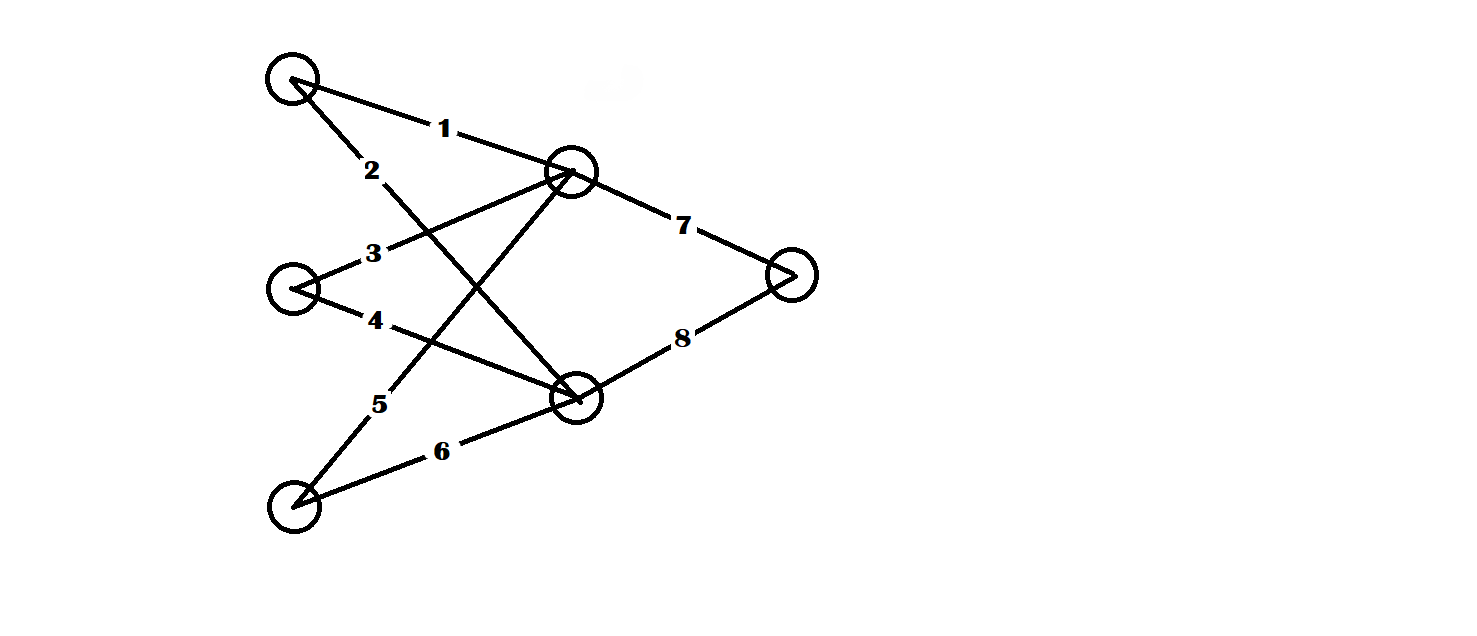
[5, 3, 1]

https://i.stack.imgur.com/9jzpy.jpg

[4, 5, 1]

http://www.mitchellspryn.com/content/Hardware-Neural-Network-Generation-Tool/network.jpg

To store the values of the network I will have three arrays: one for the weights, one for the biases and one for the values that feed through the network. There will need to be arrays within these arrays each representing a layer within the network. For the weights array there need to be values representing the connection between each neuron in a layer and a single neuron in the next layer. It will therefore be an array containing an array for each layer after the first, with an array in these representing each neuron in that layer and each value in these arrays representing a connection to a neuron in the previous layer.

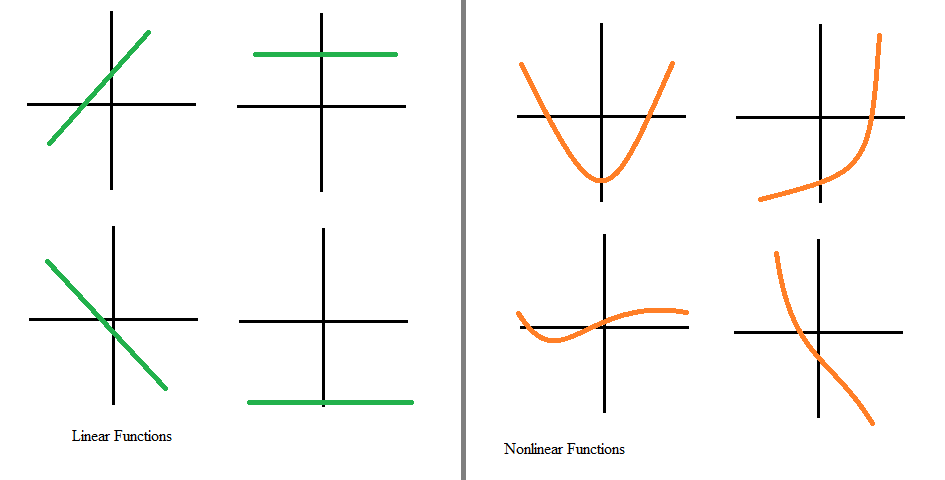


If this were the network with the numbers on the connections being the weight then the weights array would be this:

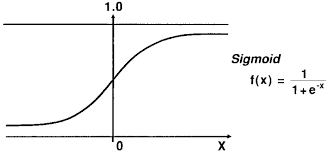
Weights = [[[1, 3, 5], [2, 4, 6]], [[7, 8]]]

Before training the networks, they require random values for its weights and biases to provide a start point for the training to begin. To do this I will use random values from a normal distribution so that whilst the majority of the values centre around the mean of zero there will be a few outliers which take more extreme values, so may provide more extreme behaviours. This results in faster training.

In order to ‘run’ the network the value of a neuron is calculated by multiplying each value of the previous layer by its connecting weight, the bias of the neuron is then added. The resulting value is passed through what is known as the activation function which allows the function to be non-linear (not a straight line) and so replicate more complex functions (and so more useful).

https://saugatbhattarai.com.np/wp-content/uploads/2018/06/linear\_function\_vs\_nonlinear\_function.png

For my project I will use the commonly applied sigmoid function.

https://www.researchgate.net/profile/Knut\_Kvaal/publication/239269767/figure/fig2/AS:643520205430784@1530438581076/An-illustration-of-the-signal-processing-in-a-sigmoid-function.png

There will be a function within the neural network class that feeds forward within the network. It will be passed values in an array; the first layer values will then be set equal to this array. It will use a for loop that goes through each layer and uses matrix multiplication and addition to times the first array of values by their weights and then adds the biases of the next layer.

Some pseudocode for this would be:

FOR each layer after first

layerValues <- sigmoid((weights\*previousLayer)+biases)

RETURN finalLayer

I will be using an evolutionary algorithm to train my network; this will mimic the way animal evolve in nature. Each network will have a calculated fitness value, they will then be sorted by this and then the ‘fittest’ network will ‘breed’ and then randomly mutate to create further generations. The fitness value will be calculated based on how far around the track it got and if it gets around the track how long it took. In this way the networks that can get around track will evolve to complete the track faster.

My breed function will take two networks and will iterate through the weights and the biases of the networks and will randomly choose one from either of the networks to take the place in the resulting network.

My mutate function will iterate through a network and for each weight and bias will generate a random number up to 100. This number will determine whether the value: is added to, multiplied, set to a new random number, or left unchanged. For instance, if it is between 0 and 5 a random number may be added to it. In this way the percentage mutation rate can be easily changed by changing the size of the bounds.

The pseudocode for this may look like this:

AddRate <- 5

TimesRate <- 2

NewRate <- 3

FOR each value of a weight or bias

randoNum <- randomNumber(0, 100)

IF randoNum < AddRate THEN

Value <- Value+randomNumber(-5, 5)

ELSE IF randoNum < AddRate+TimesRate THEN

Value <- Value\*randomNumber(-5, 5)

ELSE IF randomNum < AddRate+TimesRate+NewRate THEN

Value <- ranomNumber(-5, 5)

The UML class for the neural network might look like this:

|  |
| --- |
| Neural Network |
| -layerSizes: array  -values: array  -weights: array  -biases: array  -fitness: float |
| +generateValues  +feedforward(inputArray)  +breed(neuralNetwork)  +mutate |

You need to show how Objects relate to one another in a diagram.

I would also include a hierarchy chart in the work to show how modules relate to each other.

Show finished UI in design section and label any output boxes / buttons / etc.

# Technical Solution

# Testing

# Evaluation