Self-learning ai

AUTEUR
Sten de Boer 500773940

DATE
1 April 2020

Table of contents

Introduction 3

Week 1 4

- Think 4
  - The design 4
  - Base set-up neural networks 4
  - Car integrated with neural network 4
  - Manager 4
- Make 5
  - The base project 5
  - Creating layers & neurons & weights 5
  - Feed forward 6
  - Mutations 6
  - Copying the network 6
  - Implementing the network 7
- Check 7
  - Iterations 7
  - Conclusions for the week 8

Week 2 9

- Think 9
  - New driving method 9
  - How NEAT works 9
- Make 10
  - Support classes 10
  - Gene classes 10
  - Adding the NEAT & genome class 11
  - Creating distance function 11
  - Creating crossover function 12
  - Creating mutations 12
  - Creating connection and node classes 13
  - Creating the calculator 14
  - Creating client 14
  - Creating species 15
  - Creating the evolve function 15
  - Updating the cars 17
  - Updating the manager 18

Check 18
Small iterations 18
Conclusions for the week 18
Week 3 20
Think 20
Change checkpoints 20
Adding live points finish 20
New way resetting 20
Support system 20
New map 20
More input 20
Bugs & more species 20
Make 21
Change checkpoints 21
Adding live points finish 21
Support system 22
New map 22
More input 22
Bugs & more species 22
Check 23
Small iterations 23
Conclusions for the week 23
Week 4 24
Think 24
Drawing the genomes 24
A* pathfinding 24
Calculating the best time 24
Make 25
Drawing the genomes 25
A* pathfinding 26
Calculating the best time 28
Check 29
Small iterations 29
Conclusions for the week 29
Conclusion 31

Introduction

In this time where good ai becomes more important in games, I want to create a powerful self-learning ai that I can use in my games. So, for this project I want to set-up a project that will have a simple car controller. This car has four inputs. These are to give gas, steer right, steer left and to brake. With the use of a neural network it will decide which of these functions to use. Then it needs to find the most optimal route through the track, while driving as fast as possible. For this research I posed this question: How can I create an evolutionary ai that completes a created circuit in the most optimal time?

This question has three main goals it has to adhere to:

- The ai works with a neural network that goes through an iterative process to learn how to drive a track.
- The ai has no support deciding when it needs to steer.
- The ai gets better over time at driving the track.
- The ai reaches an optimal time to complete the track at.

For this project I have two goals. These are that I want to learn how neural networks work. This is because neural networks are strong tools in a programmer's arsenal to create powerful ai that can solve a lot of problems in games. The second goal is to learn how to create an evolutionary ai integrated with a neural network. I want to learn this because evolutionary ai can tackle a lot of unique problems that might be hard to solve otherwise. With the combination of these two some problem can be taken care of that might otherwise be hard to deal with.

Week 1

Think

The design

The project will be created in Unity. First, I will create a car controller. This will have the function to drive around manually and with help of the neural network. Driving around manually is to check if the car controller works correctly. Because there could be for example be a small bug in the speeding up of the car. The car will also speed up less and steer less good when driving at higher speeds. Then I will set-up a circuit to drive around. I will create a starting position that will also count as the finish. To make sure the program can't complete rounds early a checkpoint system will be set-up. This system will double down as a live value system. Every time a car drives through a checkpoint the car will get extra live value. When the car would then be touching the finish, the finish would check whether all the checkpoints are passed by. If this is done, then the car can go to the next round or finish. The car would also get live points based on how much faster it went then the last frame to make the program focus on doing the circuit as fast as possible.

Base set-up neural networks

After this I researched on how the basics of a neural network are implemented. The idea is that there are multiple layers of neurons. The first layer is the input layer. This is where the input is given for the neural network. After this input layer, the hidden layer comes. In this hidden layer the decision gets made on what to do with the information given in the hidden layer from the input layers. After this comes the output layer. This output layer can be used for the program to do actions within the program. The action that gets used can be the biggest number that the neural network gives back. All these layers have neurons in them that help make these decisions. These neurons can be represented by multi-dimensional arrays. The layers can be represented as an array. These layers are connected by weights. These can be represented by a weight matrix. These weights all have numbers that the program uses to do calculations.

This network can then be mixed with an evolutionary ai by mutating the numbers on the weights. This will then be done with a random chance. This chance will be small, because the changes can be bad and the good working ai is useful to preserve. With correct changes on these weights the ai will become better at the program.

For breeding the program will remove the worst performing amount of the cars. The program will then use the better performing cars to breed in new cars. These cars will then be mutated to try and get a better solution at the problem.

Car integrated with neural network

The car will have a function that works without input of a player. This function will make use a neural network. In this function the program will create some input to give to the neural network. Then in the neural network an output will calculated to an array. Then the biggest number of this array can be used to decide what to do. The car will do everything by itself. So, no help with driving forward. If the car then stands still for too long, then the car needs to be removed. When removing a car, a function needs to be called that checks whether a new generation needs to start.

Manager

In this object all the neural networks need to be created. Furthermore, this object will create all the cars. This class will check whether all the cars removed. When that is true it will start a generation. In here it will remove the worst performing cars and breed better cars. Then it will mutate these cars. This process will be repeated until I decide that it is enough.

Make

The base project

I created a unity project with a simple car controller. This car can drive forward, steer and brake. The car can be controlled manually or can be controlled randomly. When driving randomly, the car has a fifty percent chance to drive forward and one sixth chance to steer or brake. When the car goes faster, the pulling up goes slower. This is the same with the steering. So, when going faster, the steering becomes slower. These effects can be amplified to make it harder or easier to drive. When the car is just driving forward it gets live point based on how much faster it goes then the last frame. This is zero when going slower. After that I set up a checkpoint system. If a car passed a checkpoint it would get a hundred extra live points. This would also set a Boolean to true, so the start would now that a car had passed the checkpoint. Then I created a start that checks if all the checkpoints are passed by. If that would be the case the finish lets the car go the next round or if it is the last round the car would end the run and reset afterwards.

```
public void GiveGas()
{
    driving = true;
    if (carSpeed < 0)
    {
        carSpeed = 0;
    }

    carSpeed += (speedUpdate / ((carSpeed + maxSpeed) / maxSpeed)) / effectOfSpeedSpeed;
    if (carSpeed >= maxSpeed)
    {
        carSpeed = maxSpeed;
    }
}

public void Brake()
{
    driving = true;
    carSpeed -= brakeForce;
    if (carSpeed < 0)
    {
        carSpeed = 0;
    }
}

public void SteerLeft()
{
    if (canSteer)
    {
        steerPower = (rotationForce / ((carSpeed + maxSpeed) / maxSpeed)) / effectOfSpeedSteering;
        transform.Rotate(0, -steerPower, 0);
    }
}

public void SteerRight()
{
    if (canSteer)
    {
        steerPower = (rotationForce / ((carSpeed + maxSpeed) / maxSpeed)) / effectOfSpeedSteering;
        transform.Rotate(0, steerPower, 0);
    }
}
```

Creating layers & neurons & weights

```
private void InitWeights()
{
    List<float[][]> weightsList = new List<float[][]>();

    for (int i = 1; i < layers.Length; i++)
    {
        List<float[]> layerWeightsList = new List<float[]>();

        int neuronsInPreviousLayer = layers[i - 1];

        for (int j = 0; j < neurons[i].Length; j++)
        {
            float[] neuronWeights = new float[neuronsInPreviousLayer];

            for (int k = 0; k < neuronsInPreviousLayer; k++)
            {
                neuronWeights[k] = UnityEngine.Random.Range(-0.5f, 0.5f);
            }

            layerWeightsList.Add(neuronWeights);
        }

        weightsList.Add(layerWeightsList.ToArray());
    }

    weights = weightsList.ToArray();
}
```

The neural network I created starts with initializing the layers. After that it creates the data for the neurons. This is done by creating a list of float arrays. The neurons get created by looking into the layers array and creating float arrays with those sizes based on the numbers in the layers list. Then these created lists get converted to arrays. There is also a list created for initializing the weights. This list contains multidimensional float arrays. All the layers need their own weight matrix. These weights start at the first hidden layer. First a new list is created that represents all the weight in the previous layer. Then in the next for loop the program iterates throught all the neurons in the current layer and looks for the connections in the previous layer per neurons. For every connection that it finds it gives a random weight to it. After that it adds this list to the list of all the neurons. And then it adds this to the weight matrix of the neural network.

Feed forward

```
public float[] FeedForward(float[] inputs)
{
    for (int i = 0; i < inputs.Length; i++)
    {
        neurons[0][i] = inputs[i];
    }

    for (int i = 1; i < layers.Length; i++)
    {
        for (int j = 0; j < neurons[i].Length; j++)
        {
            float value = 0f;

            for (int k = 0; k < neurons[i - 1].Length; k++)
            {
                value += weights[i - 1][j][k] * neurons[i - 1][k];
            }

            neurons[i][j] = (float)Math.Tanh(value);
        }
    }

    return neurons[neurons.Length - 1];
}
```

Now the neural network is set-up the function for deciding what it will do with this information needs to be set-up. This is the feed forward function. The first action that is done by this function is that the inputs that are given are put into the first layer of the neural network. Then the program needs to iterate over every layer except the input layer. It needs to iterate over ever neuron in that layer and the previous layer. With that a number is created. This value is created for every neuron in the current layer and will be based on all the weight values in the previous layer. At the end of the function it will then return the output layer.

Mutations

The neural network also needs to be able to mutate. This is done by iterating over ever single weight within the neural network. Then every weight has a chance to be changed in four different ways. These are flipping the sign of the weight, picking a random weight, randomly increasing the weight and randomly decreasing the weight.

```
public void Mutate()
{
    for (int i = 0; i < weights.Length; i++)
    {
        for (int j = 0; j < weights[i].Length; j++)
        {
            for (int k = 0; k < weights[i][j].Length; k++)
            {
                float weight = weights[i][j][k];

                float randomNumber = UnityEngine.Random.Range(0, 100f);

                if (randomNumber <= 2f)
                {
                    weight *= -1f;
                }
                else if (randomNumber < 4f)
                {
                    weight = UnityEngine.Random.Range(-0.5f, 0.5f);
                }
                else if (randomNumber < 6f)
                {
                    float factor = UnityEngine.Random.Range(0f, 1f) + 1;
                    weight *= factor;
                }
                else if (randomNumber < 8f)
                {
                    float factor = UnityEngine.Random.Range(0f, 1f);
                    weight *= factor;
                }

                weights[i][j][k] = weight;
            }
        }
    }
}
```

Copying the network

```
public NeuralNetwork(NeuralNetwork copyNetwork)
{
    this.layers = new int[copyNetwork.layers.Length];
    for (int i = 0; i < copyNetwork.layers.Length; i++)
    {
        this.layers[i] = copyNetwork.layers[i];
    }

    InitNeurons();
    InitWeights();
    CopyWeights(copyNetwork.weights);
}
```

For the last part of the neural network it needs to be able to have a deep copy created of the neural network. This will allow the system to copy the best neural networks. These will then be used to replace the worst ones. Then the system will mutate the copys of the best neural networks. The neural network class will also be an iComparable. This will help the program sort all the neural networks from best to worst.

Implementing the network

```
private void CreateCars()
{
    if (carList != null)
    {
        for (int i = 0; i < carList.Count; i++)
        {
            Destroy(carList[i].gameObject);
        }

    }

    carList = new List<Driving>();

    for (int i = 0; i < populationSize; i++)
    {
        Driving car = Instantiate(carPrefab, startPoint.transform.position, carPrefab.transform.rotation).GetComponent<Driving>();
        car.Init(neat.GetClient(i), this); //This is already new code
        car.name = "Client " + (i + 1).ToString(); //This is already new code
        carList.Add(car);
    }

}
```

I implemented this network and created a manager for the game. This manager spawns in the cars and creates the neural network for all the cars. Then the program creates cars. If it isn't the first generation then it destroys all the previous cars from the program. It creates a new list of cars for every generation. With the init function from the car it gives a manager and a neural network to the car. Then it adds the car to the list of cars. Every time a car gives an input, this input is given to the neural network. Then the network can decide what to do with it. It also updates the live value at the end of every update of every car. This is done with the speed difference compared to the last frame and it gets extra live value when driving through a checkpoint. Whenever a car drives into a wall it calls the call next generation function from the manager. In this function the program resets the value of the number of disabled cars. Then it adds one to it for every car in the car list that is disabled. If this value is then equal to the population size, then the program is going to start the next generation. When all the cars have driven into walls the generation stops. After that it will sort these neural networks based on their live values. Then it will create a new neural network based on the best of the old neural networks. Here the groups were split in half, where the latter half of the were the better neural networks. It will also rest all the live values of the neural networks, so that worse neural networks can't become better then ones that are good. Then for the next generation it will call the create cars function again.

```
public void CallNextGeneration()
{
    amountOfCarsDisabled = 0;
    for (int i = 0; i < carList.Count; i++)
    {
        if (!carList[i].gameObject.activeSelf)
        {
            amountOfCarsDisabled++;
        }
    }

    if (amountOfCarsDisabled == carList.Count)
    {
        StartNextGeneration();
    }
}
```

```
nets.Sort();
for (int i = 0; i < populationSize / 2; i++)
{
    nets[i] = new NeuralNetwork(nets[i+(populationSize / 2)]);
    nets[i].Mutate();

    nets[i + (populationSize / 2)] = new NeuralNetwork(nets[i + (populationSize / 2)]);
}

for (int i = 0; i < populationSize; i++)
{
    nets[i].SetFitness(0f);
}
```

Check

Iterations

I did some tests with this code. Here I mainly focused on if the program was improving at controlling the car. I checked whether they could reliably get past the first corner. What I noticed was that the program couldn't get past the first corner reliably. No matter how many generations I waited the cars were still making the same mistakes as before. So, I thought that maybe I made some mistakes in how I added live values and thus I removed adding live values based on speed difference. However, this made no difference. Furthermore, I made the effect of getting a checkpoint stronger. With the live value added getting stronger per checkpoint. This also made no difference. I also tried changing the code of how many of the better neural networks would survive, but this didn't change anything either.

Conclusions for the week

I came with two conclusions. The first was that I wanted to implement a different neural network. The second was that I needed to change the way the cars were driving. I did some research for evolutionary ai based on neural networks and I came across the neural network named NEAT. This stands for Neuro Evolution of Augmenting Topologies. In this technique of evolutionary ai the topology of the node's changes. The start of the system would only start with input and output nodes and then through the process of combining the best performing cars and mutating the system would learn how to drive. The mutations in NEAT aren't only changes weights, but also adding nodes and links. The last mutation that can happen is that nodes would be able be turned on or off. The new way off driving I wanted to do was with raycasts. This would be more consistent in the numbers it returned and thus the network would be more easily able to make use of the information it receives.

Week 2

Think

New driving method

For the driving with raycast, the car would shoot three raycast that will ignore the checkpoints and other cars. When the ray then hits a wall, it will calculate the distance between it and the wall. The car then shoots three rays. To the front, left and right. The neural network would then also have four inputs from the car. The raycast that casts to the front will be the input for driving, raycast left for steering left and raycast right for steering right. The braking would be decided by an interpolation of the closest wall from the left or right and the forward raycast.

How NEAT works

Base idea

For NEAT I needed to do some research and thus I searched on the internet for information. As mentioned above NEAT stands for neuro evolution of augmenting topologies. This term can be spliced in two parts: the neuro evolution and augmenting topologies. Neuro evolution means that it is an evolutionary ai combined with a neural network. It still goes through the same steps of an evolutionary ai: first selection, then crossover and mutations. Augmenting topologies means that instead of the regular neural networks where the system is built up with a set number of layers, that it instead changes this layout. So, in this neural network there aren't any specific layers. There only is an input, hidden and output layer, where the hidden layer can be any number of nodes. Through the mutation method of this system it changes how this hidden layer looks and how many neurons and links it has. NEAT start with only input and output nodes, because if the program would start too large, it would take too long to find the best solution. And if the small network isn't enough it grows bigger and then checks whether it is good at that point.

Breeding

NEAT doesn't have a weight array because of the changing topologie of the system. Instead of that it has genes. All these genes are put into one genome. These genes are split up into node genes and connection genes. The node genes store their identification number. The connection genes store the node where it from and the node where it is going to. It also has an innovation number. These numbers help the system identify when a connection has been created. The innovation numbers are the same for all the genomes. When a genome is trying to create a new connection that already exist in another genome then that connection gets the same innovation number else it gets a new one. This is important for the crossover of the system. This is because when looking at two genomes to breed, they might not have the same amount of connections, but because every connection always has the same innovation number, the program doesn't get wrong nodes for connections. This ensures that the next generation gets better. When two parents have a differing amount of connections, the system matches these parents against each other and looks which one is fitter. There are two types of extra connections. These are disjoint connections and excess connections. Disjoint connections are connections that one parent has, but the other one doesn't have, but there are still innovation numbers that are higher. Excess connections are connections that one parent has that are higher than the highest innovation number of the other parent. If the parent with the excess genes is better than the child gets these excess genes, else it doesn't get them.

Mutations

NEAT mutates in five different ways. These mutations are a mutation where the system creates a new link between two nodes with a random weight, a mutation where the system creates a new node between any random connection. The first connection weight is going to be one and the second

connection weight is going to be the weight that the connection used to be. This connection will get an innovation number, because of the random placing of the node, it technically shouldn't exist in another genome. Next a mutation where the system randomizes a weight, a mutation where the system adds to a weight and a mutation where a link gets turned off or on.

Selection

Selection in NEAT is works with species. These species get clients with genomes that look alike. When a client differs too much from another species, the program creates another species. When all the clients are sorted into species, the program kills of for example fifty percent off all the clients in a species. Then the program will remove the species have less than a certain amount of species in them. However, in these species, clients with a bad score might survive. This is positive, because they might have a good topology, but they just needed some improvements in other fronts. All the clients that live will then breed together and then all the clients will mutate.

For creating this system, I followed a tutorial on how to create a NEAT system in Java. So, I needed to do some translating for my implementation in unity. This tutorial did seem like a tutorial that explains everything very clear.

Make

Support classes

The first thing that needs to be done for creating NEAT is creating a few helping classes that will store data. First the class random hash set is created. The class contains two variables. A hash set called set and a list called data. In this class there can be checked if a set contains certain element, it can grab random elements, remove elements, add element in a sorted way if needed and it can check the amount of data. The next class that is created is a random selector that chooses objects based on how high their score is. This is more often if the score is higher. It does this by first setting a total score. It then chooses a random number between zero and the totalscore. Because the this number has the highest chance to be around the middle of the total score the program will grab better performing clients or species. Then it creates a second variable. The program will then iterate through all the variable and add the all the vairiable scores to the second variable. This is done till the second variable passes the random number. Then it will return the variable it is iterating on.

```
public T RandomT()
{
    double v = Random.Range(0f, (float)totalScore);

    double c = 0;

    for (int i = 0; i < objects.Count; i++)
    {
        c += scores[i];
        if (c > v)
        {
            return objects[i];
        }
    }

    return default;
}
```

Gene classes

Next the gene class is created. This class contains an innovation number with a getter and setter for this. The connection and node gene will inherit from this class. Next the node gene class is created. These need an x and y value. These need getters and setters. Furthermore, the innovation numbers need to be compared with an equals function. Then the connection gene is created. This contains the node it's from, the node it goes to, whether it's turned on or off and the weight of the connection. These need getters and setters. These also need to be compared and this needs to be done with the from and to node genes, because this will show whether this is the same connection as other connections. The weights and if it is on or off doesn't matter for this.

```
public Gene(int innovationNumber)
{
    this.innovationNumber = innovationNumber;
}
```

```
public bool EqualsObject(Object o)
{
    if (o.GetType() != typeof(NodeGene))
    {
        return false;
    }

    return innovationNumber == ((NodeGene)o).GetInnovationNumber();
}
```

```
public ConnectionGene(NodeGene from, NodeGene to)
{
    this.from = from;
    this.to = to;
}
```

```
ConnectionGene c = (ConnectionGene)o;
return from.EqualsObject(c.from) && to.EqualsObject(c.to);
```

Adding the NEAT & genome class

```
public Genome(Neat neat)
{
    this.neat = neat;
}
```

Next the genome and NEAT classes are created. The genome contains two random hash sets for the connection genes and for the node genes and a NEAT object. In the constructor asks for a NEAT object will be set and all the variables have getters.

```
public void Reset(int inputSize, int outputSize, int maxClients)
{
    this.inputSize = inputSize;
    this.outputSize = outputSize;
    this.maxClients = maxClients;

    allConnections.Clear();
    allNodes.Clear();
    clients.Clear();

    for (int i = 0; i < inputSize; i++)
    {
        NodeGene n = GetNode();
        n.SetX(0.1f);
        n.SetY((i + 1) / (double)(inputSize + 1));
    }

    for (int i = 0; i < outputSize; i++)
    {
        NodeGene n = GetNode();
        n.SetX(0.9f);
        n.SetY((i + 1) / (double)(outputSize + 1));
    }
public Genome EmptyGenome()
{
    Genome g = new Genome(this);

    for(int i = 0; i < inputSize + outputSize; i++)
    {
        g.GetNodes().Add(GetNode(i + 1));
    }

    return g;
}
```

In the NEAT class a constructor is created with an input size, output size and the number of clients. Next a function will be created for adding new nodes. The innovation number the node will get is the size of the hash set they are in plus one. Nodes can also be grabbed from this class with their innovation number. The NEAT class gets a reset function. The reset function will clear everything and then add the input and output nodes again, because these nodes will always exist in the genome. The input nodes will be on the left and the output nodes are on the right. This reset function is used in the constructor.

The NEAT class needs a function for creating a new genome. This NEAT object of the genome will be set to this and all the input and output nodes will be added.

```
public ConnectionGene GetConnection(NodeGene node1, NodeGene node2)
{
    ConnectionGene connectionGene = new ConnectionGene(node1, node2);

    if (allConnections.ContainsKey(connectionGene))
    {
        connectionGene.SetInnovationNumber(allConnections[connectionGene].GetInnovationNumber());
    } else
    {
        connectionGene.SetInnovationNumber(allConnections.Count + 1);
        allConnections[connectionGene] = connectionGene;
    }

    return connectionGene;
}
```

Next is getting a connection gene. In here the program will check if the connection already exists. If it does it gets the innovation number of the connection gene else, it creates an innovation number for the connection gene. Then a function for getting connection genes, but this one is for the cross over function in the genome.

Creating distance function

Now the genome class is going to get further developed. In this class a distance and crossover function will be created. The distance is for checking whether two genomes belong to the same species. This function will ask for a second genome. Then it initializes the current genome as the first genome. It is important that the first genome is bigger than the second genome. Bigger is based on which genome has the highest innovation number. This is because then the system will check reliably for excess genes. First the system will set the first genome as the bigger genome by checking which of the two genomes is bigger. Then variables are created for the excess, disjoint and similar genes. There also is a variable for the weight difference. There are also two indexes created. While both these indexes are smaller than the size of the amount of connections of the genomes, similar and disjoint genes are counted. If both contain the connection then similar gets a one added else if the index of the first genome is

```
while (indexG1 < g1.GetConnections().Size() && indexG2 < g2.GetConnections().Size())
{
    ConnectionGene gene1 = g1.GetConnections().Get(indexG1);
    ConnectionGene gene2 = g2.GetConnections().Get(indexG2);

    int in1 = gene1.GetInnovationNumber();
    int in2 = gene2.GetInnovationNumber();

    if (in1 == in2)
    {
        similar++;
        weightDiff += Mathf.Abs((float)gene1.GetWeight() - (float)gene2.GetWeight());
        indexG1++;
        indexG2++;
    }
    else if (in1 > in2)
    {
        disjoint++;
        indexG2++;
    }
    else
    {
        disjoint++;
        indexG1++;
    }
}

weightDiff /= Mathf.Max(1, similar);
excess = g1.GetConnections().Size() - indexG1;
```

bigger than the index of the second then there is a disjoint in the first genome, which means that the index of the second needs to be added. Else there is a disjoint in the second genome. When a connection is similar the difference between the weights gets added to the weight difference variable. After the while loop the weight difference is set to an average weight difference. The amount excess genes are the amount genes left over in the bigger genome. Next a variable N is created that is equal to which genome has more nodes. Then this calculation is done with these numbers:

```
return neat.GetC1() * disjoint / N + neat.GetC2() * excess / N + neat.GetC3() * weightDiff;
```

This is based of the more disjoint genes, excess genes and weight difference a genome has, the more difference they have between each other. More difference means they are less compatible with each other. The C variables allow the program to shift importance of these variables.

Creating crossover function

```
while (indexG1 < g1.GetConnections().Size() && indexG2 < g2.GetConnections().Size())
{
    ConnectionGene gene1 = g1.GetConnections().Get(indexG1);
    ConnectionGene gene2 = g2.GetConnections().Get(indexG2);

    int in1 = gene1.GetInnovationNumber();
    int in2 = gene2.GetInnovationNumber();

    if (in1 == in2)
    {
        float r = Random.Range(0f, 1f);
        if (r > 0.5f)
        {
            genome.GetConnections().Add(neat.GetConnection(gene1));
        } else
        {
            genome.GetConnections().Add(neat.GetConnection(gene2));
        }

        indexG1++;
        indexG2++;
    }
    else if (in1 > in2)
    {
        indexG2++;
    }
    else
    {
        genome.GetConnections().Add(neat.GetConnection(gene1));
        indexG1++;
    }
}
```

After the distance function the crossover function is created. This function is for breeding two genomes. The function asks for two genomes. Before executing this function, it is made sure that the first genome has the higher live value. In the function a NEAT object is gotten. This is done so all the connections can be gotten. There is also a new genome created with the empty genome function. Then two indexes are created again. While these values are lower connections genes are added. If the indexes are the same, then the new genome gets one connection gene randomly of one of the two parents. If the second genome has a disjoint gene, then no connection gene is added. If the first genome has a disjoint gene, then this one is added to new genome. If the first genome has excess genes, then these are added. Then all the nodes are added in the new genome based on the connection nodes in there. Then the new genome is returned.

Creating mutations

```
public void MutateWeightShift()
{
    ConnectionGene con = connections.RandomElement();
    if (con != null)
    {
        float r = Random.Range(0f, 1f);
        con.SetWeight(con.GetWeight() + r * neat.GetWeightShiftStrength());
    }
}

public void MutateWeightRandom()
{
    ConnectionGene con = connections.RandomElement();
    if (con != null)
    {
        float r = Random.Range(-1f, 1f);
        con.SetWeight(r * neat.GetWeightRandomStrength());
    }
}

public void MutateLinkToggle()
{
    ConnectionGene con = connections.RandomElement();
    if (con != null)
    {
        con.SetEnabled(!con.IsEnabled());
    }
}
```

Now the mutations are going to get added in the genome class. First the mutate weight shift is added. In this function a random connection is grabbed. Then it gets a random value added between zero and one times the strength of the weight shift that is defined in the NEAT class. Then the mutate weight random is added. In this function a random connection is grabbed. Then it gives the connection a random weight that is times a random weight strength in the NEAT class. After this the toggle link function is added. In this function a random connection is grabbed, and it reverses the active state of the link.

```
public void MutateLink()
{
    for (int i = 0; i < 100; i++)
    {
        NodeGene a = nodes.RandomElement();
        NodeGene b = nodes.RandomElement();

        if (a.GetX() == b.GetX())
        {
            continue;
        }

        ConnectionGene con;

        if (a.GetX() < b.GetX())
        {
            con = new ConnectionGene(a, b);
        } else
        {
            con = new ConnectionGene(b, a);
        }

        if (connections.Contains(con))
        {
            continue;
        }

        float r = Random.Range(-1f, 1f);

        con = neat.GetConnection(con.GetFrom(), con.GetTo());
        con.SetWeight(r * neat.GetWeightRandomStrength());
        connections.AddSorted(con);
        return;
    }
}
```

Then the mutate link is added. In this function a for loop is created that loops a hundred times. In this for loop two random nodes are grabbed. First it is checked whether these nodes have the same x value. If that is the case, then this connection cannot exist, and the function skips the rest and tries again. If the x values differ then a new connection gene gets created with from the node with the lowest x to the node with the highest x. Then the program checks if the connection already exist in the current genome. If it does the function skips the rest and tries again else the connection is created and is given a random weight. Then the connection is added using the add sorted method in the random hash set class.

As last the mutate node function is added. First a random connection is grabbed. Then the two nodes from the connection gene are grabbed. These are the from and to nodes. In between these two nodes a new node is generated. The x position is the average of the two other nodes. The y value is the average of the other two nodes plus a random number. Then two connections are added between from to the new node and between the new node to the to node. The weight of the first connection is set to one and the weight of the second connection is set to the weight of the original connection. Then active state of the second connection is set to the original active state. Then the original connection is removed from the genome and the two new ones are added and the new node is added. Then the mutate function is added. Each of the mutations get probabilities of happening and then a random chance decides whether the mutation happens or not.

```
public void MutateNode()
{
    ConnectionGene con = connections.RandomElement();
    if (con == null)
    {
        return;
    }

    NodeGene from = con.GetFrom();
    NodeGene to = con.GetTo();
    NodeGene middle;

    float r = Random.Range(0f, 1f);

    middle = neat.GetNode();
    middle.SetX((from.GetX() + to.GetX()) / 2);
    middle.SetY((from.GetY() + to.GetY()) / 2 + r * 0.1f - 0.05f);

    ConnectionGene con1 = neat.GetConnection(from, middle);
    ConnectionGene con2 = neat.GetConnection(middle, to);

    con1.SetWeight(1);
    con2.SetWeight(con.GetWeight());
    con2.SetEnabled(con.IsEnabled());

    connections.Remove(con);
    connections.Add(con1);
    connections.Add(con2);

    nodes.Add(middle);
}
```

Creating connection and node classes

Next the calculations classes are created for the neural network. The first two classes that are created are the connection and node class. The connection class is a copy of the connection gene where the node genes are replaced with nodes and without the equals function. This class is created because this makes the code more structured. Then the node is created. This class is an iComparable. This node class only gets an x value and no y value. This is because the y value isn't important for sorting the nodes in order. The constructor of this class only asks a x value. So, with the comparable all the nodes can be sorted from left to right. Next a calculation function is created. In this function a variable s for the sum is created. The program loops the connections of the current node and adds a value to the sum based on the weight and output of the connections. Then the sum goes through the sigmoid function. This value is set equal to the output.

```
public void Calculate()
{
    double s = 0;
    foreach(Connection c in connections)
    {
        if (c.IsEnabled())
        {
            s += c.GetWeight() * c.GetFrom().GetOutput();
        }
    }

    output = ActivationFunction(s);
}

private double ActivationFunction(double x)
{
    return 1d / (1 + Mathf.Exp((float)-x));
}
```

Creating the calculator

```
public Calculator(Genome g)
{
    RandomHashSet<NodeGene> nodes = g.GetNodes();
    RandomHashSet<ConnectionGene> cons = g.GetConnections();

    Dictionary<int, Node> nodeDictionary = new Dictionary<int, Node>();

    foreach (NodeGene n in nodes.GetData())
    {
        Node node = new Node(n.GetX());
        nodeDictionary[n.GetInnovationNumber()] = node;

        if (n.GetX() <= 0.1f)
        {
            inputNodes.Add(node);
        }
        else if (n.GetX() >= 0.9f)
        {
            outputNodes.Add(node);
        }
        else
        {
            hiddenNodes.Add(node);
        }
    }

    hiddenNodes.Sort();

    foreach(ConnectionGene c in cons.GetData())
    {
        NodeGene from = c.GetFrom();
        NodeGene to = c.GetTo();

        Node nodeFrom = nodeDictionary[from.GetInnovationNumber()];
        Node nodeTo = nodeDictionary[to.GetInnovationNumber()];

        Connection con = new Connection(nodeFrom, nodeTo);
        con.SetWeight(c.GetWeight());
        con.SetEnabled(c.IsEnabled());

        nodeTo.GetConnections().Add(con);
    }
}
```

Next a calculator class is created. This class has three nodes lists. These are for the hidden, input and output nodes. The constructor of the calculator will ask for a genome. In the constructor all the node genes and connection genes will be set separate random hash sets. Also, a dictionary is created from integers to node genes. Then for all the nodes in the genome will be placed into the lists for hidden, input and output nodes. If the x value is smaller or equal to zero point then it is an input node, if the x value is bigger or equal to zero point nine then it is an output node, else it is a hidden node. All the nodes are also put in the dictionary. Then the hidden nodes get sorted. Next all the connections are added. The program will loop through every connection gene in the genome. It will grab the from and to node genes from the connection gene. Whit help from the dictionary it will convert these node genes to nodes using the innovation numbers from the node genes. Then a new connection is created, and it copies the stats of the connection gene. This connection then gets added to the connections list of the to node genes.

```
public double[] Calculate(double[] input)
{
    if(input.Length != inputNodes.Count)
    {
        Debug.LogWarning("Data doenst fit");
    }

    for(int i = 0; i < inputNodes.Count; i++)
    {
        inputNodes[i].SetOutput(input[i]);
    }

    foreach(Node n in hiddenNodes)
    {
        n.Calculate();
    }

    double[] output = new double[outputNodes.Count];
    for (int i = 0; i < outputNodes.Count; i++)
    {
        outputNodes[i].Calculate();
        output[i] = outputNodes[i].GetOutput();
    }

    return output;
}
```

Next the calculate function is created. This function asks an input. This input is set as the output for all the input nodes. Then the calculate function of the node is done for every node in the hidden nodes. An array for the output nodes is created. Then the calculate function is done for the output nodes. The numbers from this calculate function are put in newly created array. Then the program returns this array.

Creating client

```
public void GenerateCalculator()
{
    calculator = new Calculator(genome);
}

public double[] Calculate(double[] input)
{
    if (calculator == null)
    {
        GenerateCalculator();
    }

    return calculator.Calculate(input);
}

public double Distance(Client other)
{
    return genome.Distance(other.GetGenome());
}

public void Mutate()
{
    genome.Mutate();
}
```

Next the client class is created. The client gets a function to generate a calculator. It also gets a calculate function. In this function it checks whether the calculator is null. If it is it generates a calculator. Then it calculates the inputs given to it. The client can also calculate the distance with the genome to another client to check whether they are in the same species. The client also has a mutate function that uses the genome mutate function. This class also has a score with a getter and setter for this. The client also is an iComparable. This is used for sorting the clients based on the live value they have. After this the species class is created.

Creating species

```
public Species(Client representative)
{
    this.representative = representative;
    this.representative.SetSpecies(this);
    clients.Add(representative);

    for (int i = 0; i < 3; i++)
    {
        int r = Random.Range(0, chars.Length);
        char add = chars[r];
        name += add;
    }
}
```

The species class has a list of clients with a representative client. It also has a score and a name. The constructor asks for a representative client. In there the representative for the class is set and is added to the list of clients. This client is set to be part of this species in here. This is because it will be the first of the species. There also is a random name generated for the species for debug purposes.

```
public bool Put(Client client)
{
    if (client.Distance(representative) < representative.GetGenome().GetNeat().GetCP())
    {
        client.SetSpecies(this);
        clients.Add(client);
        return true;
    }
    return false;
}

public void ForcePut(Client client)
{
    client.SetSpecies(this);
    clients.Add(client);
}
```

Next the put function is created. This function asks for a client and must return a bool. In this function the distance function of the client is used. If the distance between the representative and the given client is small enough then the client gets added to the species and the function returns true else the function returns false. There also is a force put function created to put a client for sure in a species. This is for offspring of the species.

```
public void EvaluateScore()
{
    double v = 0;
    foreach(Client c in clients.GetData())
    {
        v += c.GetScore();
    }

    score = v / clients.Size();
}

public void ResetSpecies()
{
    representative = clients.RandomElement();
    foreach(Client c in clients.GetData())
    {
        c.SetSpecies(null);
    }
    clients.Clear();

    clients.Add(representative);
    representative.SetSpecies(this);
    score = 0;
}

public void Kill(double percentage)
{
    clients.GetData().Sort();

    double amount = percentage * clients.Size();

    for(int i = 0; i < amount; i++)
    {
        clients.Get(0).SetSpecies(null);
        clients.Remove(0);
    }
}
```

After this a go extinct function is created that removes the species from all the clients it has. This is for when there is only one client left in the species. Then the evaluate score method is created that averages the score of all the clients in the species and sets the score of the species to that number. This is used for selecting the higher performing species more than the worse performing species when reproducing new clients. Next is the reset species function. Here a random client gets grabbed first. Then everything gets removed from the species and then the random client is set as the representative and is added to the client list. This is for when a new generation starts, and all the clients need to be reordered to all the species. Next the kill function is created where all the clients get sorted first based on their score. Then a percentage of these clients get removed from the species.

```
public Genome Breed()
{
    Client c1 = clients.RandomElement();
    Client c2 = clients.RandomElement();

    if (c1.GetScore() > c2.GetScore()) return Genome.CrossOver(c1.GetGenome(), c2.GetGenome());
    return Genome.CrossOver(c2.GetGenome(), c1.GetGenome()); ;
}
```

The last function that is created is the breed function that grabs two random clients from a species. Then it uses the cross over function from the genome to breed them where the first genome that is given to the is the client with a higher score.

Creating the evolve function

```
for (int i = 0; i < maxClients; i++)
{
    Client c = new Client();
    c.SetGenome(EmptyGenome());
    c.GenerateCalculator();
    clients.Add(c);
}
```

The last addition for the NEAT code is in the NEAT class. First the clients are going to be added to the NEAT class. This is done in the reset function of the NEAT class. This is done in at the end of the method in a for loop. In the for loop clients are created with a calculator and empty genome. Then a function is made for getting clients with a certain index. Then the evolving of the program is added. This consist of five parts. These are the generating of the species, killing a percentage of the species, removing the species with one client, breeding new clients and then mutating the species.

```
private void GenSpecies()
{
    foreach (Species s in species.GetData())
    {
        s.ResetSpecies();
    }

    foreach (Client c in clients.GetData())
    {
        if (c.GetSpecies() != null) continue;

        bool found = false;

        foreach (Species s in species.GetData())
        {
            if (s.Put(c))
            {
                found = true;
                break;
            }
        }

        if (!found)
        {
            species.Add(new Species(c));
        }
    }

    foreach(Species s in species.GetData())
    {
        s.EvaluateScore();
    }
}
```

First is generating the species. In this function all the species call their reset species function, so they only have their representative left. Then the program loops through all the clients and for every client it loops thought all the species using the put function of the species to look if the client fits into a species. If all the put functions return false, then program creates a new species with the current client as their representative. After that it evaluates the score of every species.

After this the killing function is called. In this function a percentage of every species is killed off. After that all the extinct species need to be removed. In this function the program loops through all the current species and looks if they have more than one current client. If they don't then the go extinct function of the species is called and the species is removed from the list of species.

```
private void Kill()
{
    foreach (Species s in species.GetData())
    {
        s.Kill(1 - survivors);
    }
}

private void RemoveExtinctSpecies()
{
    for (int i = species.Size() - 1; i >= 0; i--)
    {
        if (species.Get(i).Size() <= 1)
        {
            species.Get(i).GoExtinct();
            species.Remove(i);
        }
    }
}
```

```
private void Reproduce()
{
    RandomSelector<Species> selector = new RandomSelector<Species>();
    foreach(Species s in species.GetData())
    {
        selector.Add(s, s.GetScore());
    }

    foreach(Client c in clients.GetData())
    {
        if (c.GetSpecies() == null)
        {
            Species s = selector.RandomT();
            c.SetGenome(s.Breed());
            s.ForcePut(c);
        }
    }
}

private void Mutate()
{
    foreach(Client c in clients.GetData())
    {
        c.Mutate();
        c.SetScore(0);
    }
}
```

Next the reproduce function is called. In this function a new random selector is created. This selector gets all the species added to its list. Then the program loops thought all the client and looks if it hasn't gotten a species yet. If it doesn't then the random selector chooses a random species based on their score. Then the client gets a genome with the breed function of the species. After this the force put function is called from the species class to add the client to the species where its bred from. Lastly the mutate function is called. This function calls the mutate function in every client. At the end of the evolve function a calculator is created for all the clients.

Updating the cars

```
public void Init(Client client, Manager manager)
{
    this.client = client;
    this.manager = manager;
}
```

After this NEAT system was implemented, I updated the car to incorporate this system. I removed all the original contents for my driving without inputs function. First, I changed the init function to incorporate a client instead of a neural network. I added two functions for deciding what to do. These are the decide output and the direction from output function. The first one I added was the decide output function. In this function the raycast are shot. There is one shot from the front, left and right.

```
private double[] DecideOutput()
{
    double[] outputs = new double[manager.inputs];

    if (Physics.Raycast(transform.position + new Vector3(0, 0, rayCastAddZ), transform.forward, out RaycastHit hit1, Mathf.Infinity, mask))
    {
        if (hit1.transform.tag == "Wall")
        {
            outputs[0] = hit1.distance;
        }
    }

    if (Physics.Raycast(transform.position + new Vector3(0, 0, rayCastAddX), transform.right, out RaycastHit hit2, Mathf.Infinity, mask))
    {
        if (hit2.transform.tag == "Wall")
        {
            outputs[1] = hit2.distance;
        }
    }

    if (Physics.Raycast(transform.position - new Vector3(0, 0, rayCastAddX), -transform.right, out RaycastHit hit3, Mathf.Infinity, mask))
    {
        if (hit3.transform.tag == "Wall")
        {
            outputs[2] = hit3.distance;
        }
    }

    outputs[5] = (outputs[0] + outputs[2]) / 2;

    outputs[6] = (outputs[0] + outputs[1]) / 2;

    return outputs;
}
```

These raycast are from an offset from the original position depended on where they need to go. So, the raycast that goes forward has an offset that makes the raycast start at the front of the car. Same for left and right. All the raycast have no limit in how far they can shoot. They also ignore certain layers, so they don't shoot other cars and checkpoints. Only if the object is a wall the distance to the object gets calculated and is given as an output for the input. The last two inputs are interpolations of the raycast going forward and the walls to the left and right.

Next the direction from output function is created. In this function the program searches the output array for the biggest number and uses this index for deciding what function to do.

```
private int directionFromOutput(double[] output)
{
    int index = 0;
    for (int i = 1; i < output.Length; i++){
        if (output[i] > output[index]){
            index = i;
        }
    }
    return index;
}
```

```
private void DriveWithoutInputs()
{
    double[] outputs = client.Calculate(DecideOutput());

    int input = directionFromOutput(outputs);

    if (input == 0)
    {
        GiveGas();
    }
    else if (input == 1)
    {
        SteerRight();
    }
    else if (input == 2)
    {
        SteerLeft();
    }
    else
    {
        Brake();
    }
```

Then the drive without inputs function is updated to incorporate these changes. First it creates an array of doubles that is the size of the inputs of the manager. Then it uses its client to calculate all the inputs. Then it gets the biggest output number from the direction from output. Then it uses this number to decide what to do.

Updating the manager

```
void InitCarNeuralNetworks()
{
    neat = new Neat(inputs, outputs, populationSize);
}
```

```
if (generationNumber == 0)
{
    InitCarNeuralNetworks();
}
else
{
    neat.Evolve();
    neat.PrintSpecies();
}
```

In the manager of the game most of the code remained similar. For the first generation the init car neural networks function would be called. This function would only create a new NEAT system. The create cars system remained the same except that the system would now give clients to the cars instead of neural networks. The code for starting a new generation is changed to only call the evolve method of the NEAT class. Furthermore, it debugs the species.

Check

Small iterations

For the tests I used the same criteria as in the first week. This is that the system would be able to reliably complete the first corner. I started the testing first with only twenty cars. But this proved to make the waiting till things happened too take really long so I changed this number to be a hundred. This made for that the cars could complete the first corner reliably in about fifteen generation, but it usually got stuck at the end of the lap, because the cars didn't drive anymore. So, to get more genomes I upped the number of clients in the NEAT system but didn't increase the number of cars. This would mean there were genomes to evolve and thus get more chances to get the correct genome. However, what ended up happening is that the cars didn't learn anything at all anymore. I initially also only had four inputs for the car instead of the five shown in the code snippet. The original code only gave the interpolation of the closest wall to the left or right, while now it gives both the values. This is done, because with more information the car will be able to drive better. I also had the amount of outputs first accidentally set to three, so that the program never used the brake function of the car.

Next, I upped the criteria of driving to completing the map to completing three quarters of the map reliably. Then I made driving harder by making the speed update of the car really high and doubling the max speed. After that I made steering harder by halving the steering power. Then I ran the tests again. The cars would take a really long time to reliably get three quarters of the round correct, but they did make it eventually. But because this took so much longer and the route for completing the track wasn't really interesting, I decided that I wanted to create a different way of making it harder to drive. So, to make it more interesting I should make the map harder to drive around.

Conclusions for the week

The program this time learned how to drive most of the map, but it usually got stuck on the end of the round, because it never got the output for giving gas anymore. Furthermore, the cars started driving circles on the road and didn't die off because they weren't standing still. I also realized that the cars that made the finish line didn't get extra points based on how fast they completed the track. The last thing I noticed was that after a relatively quick time I wouldn't get any new species anymore.

For these problems I came up with a few solutions. For the problem of not finishing the round I would add a support system for the cars that would force the cars to give gas for a small period of time when they are standing still for too long. Hence cars wouldn't die of for standing still now, they would now get killed off if they didn't reach the next checkpoint fast enough. This time would be generous enough so that they could make use of the support system without timing out. With this system in place the cars would also not be able to drive circles on the map anymore. For getting points on the finish line I would need to add a new function to the system. The amount of point

needed to be more based on how fast the car did the lap. The car would get awarded more points for when the round was done faster and less if the round was done slower. For creating more species there could be bugs in the code for when creating new nodes or links. Furthermore, I could make the effects for calculating the distance stronger whilst making the distance needed to be in another species smaller.

Week 3

Think

Change checkpoints

For the problems I had last week I would need to make an overhaul to my checkpoint and finish line code. The problem now is that a new live value always gets added when driving through the checkpoint, no matter if the checkpoint had been driven through before. So, I wanted to change the checkpoints so that the all the information for how many checkpoints a car had done was all in the car script and wasn't for the finish to decide whether all the checkpoints had been done. So, when a car gets initialized it grabs the children of the checkpoints game object and put them in a list. When going through a checkpoint it would remove the checkpoint it went through. It also will add live points. At the end of the round the start would check if the car went through all the checkpoints. If it did then the start would award points based on the time, add all the checkpoints and add one to the lap counter of the car. If it was the last lap it would only award points based on time.

Adding live points finish

These points would get added the same way I calculate on how much speed the program needs to add to a car when its giving gas. Only now it would use the time values. In the speed calculations the value of the speed added gets lower the faster the goes and the program needs to add less points the bigger the time is. Thus, the same calculations would work.

New way resetting

For resetting car based on when they go through checkpoints, I would create a timer. This timer would reset when going through a checkpoint. If the timer became larger than a certain number, then the reset function would be called.

Support system

The support system of the car that would make it drive forward after standing still for a few seconds would work the same as the old system for resetting the car for standing still for too long, but instead of killing the car it would set a bool to true. When this bool is true the car would always give gas. This bool will turn off if a certain time gets passed again. Also, every time this system gets used the live value that would get added when driving through gets halved, so that the system would encourage driving manually.

New map

I also wanted to see if the system could complete a harder map so I would need to add more obstacles to the map. They could be corners or walls in the middle of the road. If a car then passed an obstacle a checkpoint would need to be placed after that so the system would know that it would need to breed that car.

More input

Next, I wanted to add more input in the car so it would have more information on where to drive. So, I added to oblique raycast to the car. They would shoot out of the front of the car. This would help the car detect walls if the raycast going straight would just miss the wall.

Bugs & more species

Furthermore, I noticed a bug where bad cars would have really high scores, and these would impact the mutations so that the entire population would become bad.

The last thing I wanted to do is that I wanted to check out why the program had only one species left after thirty generations. To create more species, I made the difference needed between species smaller. Furthermore, I made the effects of disjoint genes, excess genes and weight difference stronger. This had not that much effect. So, there must be some bugs in the program. This could be in the distance function. This could also be when creating a new node or link or when changing a weight.

Make

Change checkpoints

```
public void AddCheckpoints()
{
    for (int i = 0; i < checkPointsList.transform.childCount; i++)
    {
        Transform checkpoint = checkPointsList.transform.GetChild(i);
        checkpoints.Add(checkpoint.GetComponent<Checkpoint>());
    }
}
```

On the start and when passing through this code is called to add the checkpoints to the car.

```
public void AddLiveValue(Checkpoint checkpoint)
{
    deleteCP = 0;

    checkpoints.Remove(checkpoint);

    if (!canDriveManually)
    {
        liveValue += liveValueAdded;
        client.SetScore(liveValue);
        liveValueAdded *= 2;
    }
}
```

When driving through a checkpoint this function gets called for the car. The checkpoint gives itself with this function. First it resets the timer of the delete checkpoint value, so that the car gets extra time for reaching the next checkpoint. Next it adds the live value of the car. It does this by adding a certain the live value added and then the sets the score of the clients to be equal to the live value. It then increases the live value added.

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag == "Player")
    {
        if (other.GetComponent<Driving>().ReachedFinish() == 0)
        {
            if (other.GetComponent<Driving>().lap < numberOfRounds)
            {
                other.GetComponent<Driving>().AddRound();
                ResetFinish();
                other.GetComponent<Driving>().AddLiveValueFinish(timeFactor);
            }
            else if (other.GetComponent<Driving>().lap >= numberOfRounds)
            {
                StartCoroutine(Finished(other.GetComponent<Driving>()));
            }
        }
    }
}
```

When going through the finish this code is called from the finish. It first will check if the collider is a car. Then it will check if the car has no more checkpoints in its checkpoints list. If it does then it checks whether the car has still laps to do. If it does then it add a lap and the checkpoints to the car with the add round function. It will also add live value based on how fast the car drove the round. If the car had done all the rounds, then it would add points based on time and check if the next generation can be started.

Adding live points finish

For adding live value when finished this calculation is done:

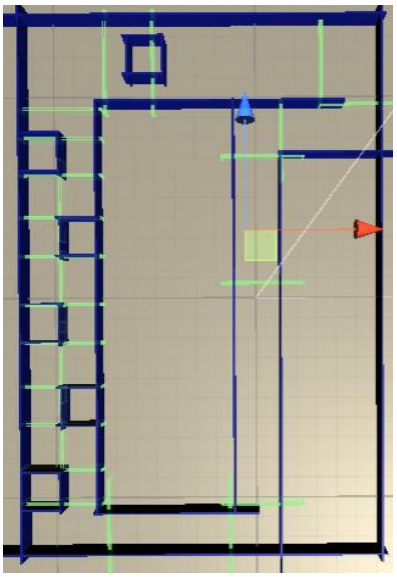
```
liveValue += (time / ((time + timeFactor) / timeFactor)) * 50;
```

The time factor is a value decided in the finish. This is value does not get calculated and is chosen by the user of the program. The time is the current time live time of the car. It is done times fifty to make the effect stronger. By having a static time factor that divides itself by itself and the added time there will come a number that is larger the bigger the added time is. And dividing something by a bigger number will make the following number smaller.

Support system

```
if (carSpeed <= 1 && !giveGas)
{
    timeForGiveGas += Time.deltaTime;
    if (timeForGiveGas > timeForSpeeding)
    {
        liveValueAdded /= 2;
        giveGas = true;
    }
}
else if (giveGas)
{
    timeForGiveGas += Time.deltaTime;
    GiveGas();
    if (timeForGiveGas > timeForSpeeding * 1.5f)
    {
        giveGas = false;
    }
}
else
{
    timeForGiveGas = 0;
}
```

For the support system of the car the system checks first whether the speed of the car is lower than one. If it is then it starts counting. If it is lower than one for too long it will speed the car up and divide the live value added by two. If the speed goes above one than it will reset the timer.



New map

I added some obstacles and corners to my previous map. All the green lines are checkpoints. After completing a corner there always needs to be a checkpoint so that the system knows that a car made progress.

More input

```
if (Physics.Raycast(transform.position + new Vector3(0, 0, -rayCastAddZ), -transform.right + transform.forward, out RaycastHit hit5, Mathf.Infinity, mask))
{
    if (hit5.transform.tag == "Wall")
    {
        outputs[4] = hit5.distance;
    }
}
```

The oblique raycast would start at the front of the car. This is the raycast add z. Than it would mix a transform forward and a left or right with it to shoot oblique. This would then get filled in for one of the inputs.

Bugs & more species

```
private void Mutate()
{
    foreach(Client c in clients.GetData())
    {
        c.Mutate();
        c.SetScore(0);
    }
}
```

The problem that cuased the program to have bad populations would get fixed by resetting the score when all clients where getting mutated. The problem before was that this score wasn't reset when the car didn't reach any checkpoints.

For the last part of I needed to search why I had so little species. For this I changed a few things for the distance calculations done in the genome class. First, I changed the strength of certain variables that decided when to create a new species:

```
private double c1 = 1.5f, c2 = 1.5f, c3 = 1.5f;
private double CP = 2.7f;
```

Calculation for getting the difference between genomes:

```
return neat.GetC1() * disjoint / N + neat.GetC2() * excess / N + neat.GetC3() * weightDiff;
```

C1 is for the amount of disjoint genes there are in the two genomes, C2 is for the amount of excess genes present in the best performing genome, C3 is for the weight difference between the genomes and CP is for the difference needed between the genomes.

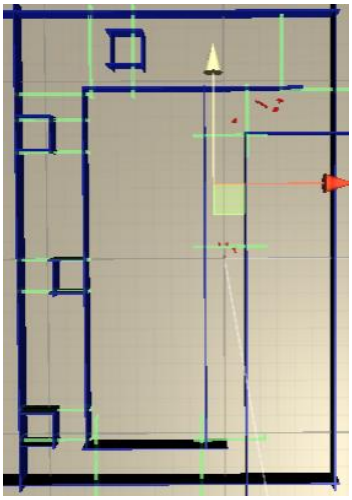
One problem I noticed in the difference function was that the program didn't add anything to the disjoint variable when there was a disjoint making disjoint always zero. Furthermore I noticed a

problem when I created new weights. Normally these get a random value between minus one and one. In my code these got a random value between zero and one. This was also the case for randomizing a weight and thus the program had only positive weights.

```
float r = Random.Range(-1f, 1f);
con.SetWeight(r * neat.GetWeightRandomStrength());
```

Check

Small iterations



For the new tests I would run I would check whether the program actually got better at driving the map. So, I would check whether the first completed time was worse than the later times. The problem I created was that the new map I created was way to hard for the system to complete in a reasonable time. They usually got stuck at the zigzagging part. So I removed some of the blocks it had to pass, but let the rest remain the same. With this map the cars where able to get the circuit completed relativly quick, around generation hundred. The next problem I created was that inflated the amount off species and evolving part of the program so much that the program didn't get a lot of cars that were driving correctly, because there were a lot of species that were bad at driving, but they didn't get removed, because of the inflated numbers. These made it that they still got enough clients. Furthermore, with the high chances for evolving the better performing cars had a chance to dissapear. They did complete the map and got better at it, but it wasn't optimal. They went from 288 seconds to 155 seconds. So I changed the numbers for creating new species. With some testing I came with the numbers shown in the make part of this week. With these numbers the program was able to complete the circuit in 96 seconds.

Conclusions for the week

With these changes to the program the cars could drive three rounds around this harder map. The program also created more species. However, the program is missing a key component for my research and that is that I don't know what the most optimal time is for completing a round. I also want to know what a genome looks like of a successful client. For calculating the most optimal route I can use pathfinding. I can then place a few blocks it has to pass by, because a circle to drive around. Then I can calculate the total distance by doing this times three. Then I can divide the distance by the average speed of the car. This would then get the optimal time to do the track in. For the drawing of the genome I can use images. A line image for drawing connections and circles for nodes. These lines can then be red, or blue based on whether the connection is on or off. In the middle of the line the weight can be shown by text.

Week 4

Think

Drawing the genomes

For drawing the genomes, I would need to create a new button to lead to a UI menu. While on this menu the camera script must be turned off for all the clicking that will be done on this screen else the view in the test scene will be different when turning off the genome screen. The testing must also be put on pause, so that the genomes don't change while looking at them. Also, when turning on the screen when the screen had shown a genome before it should show that same genome again. The genome should also be turned off when closing this part off the program. For showing the genome a dropdown menu could be used. The dropdown button has a function that is called on value changed. When clicking on a value this could be set equal to the number of a client.

Then all the nodes could be drawn using the x and y values within the nodes. For drawing connections, the from and to nodes could be used. The length could be calculated by subtracting the x values from the from and to node. The rotation could be calculated using the positions from the from and to nodes. The enabled state could be shown be giving the connection a colour, red for inactive and blue for active. Then text could be shown in the middle of the connection to display the weight of the connection.

For removing the text, I could make lists for adding the images and the text. When changing the value, I could then remove all the contents from the list and add the new contents to the list.

A* pathfinding

For calculating the most optimal route I could use a-star pathfinding. For this I would first need to create a grid system. There would be multiple targets the system had to pass by to calculate the distance for the round since the start and end of the round are the same. To incorporate this, I could change the start and ending point of the system whenever it found a solution till it is at the back at the start again. For every time it found a solution it could then add the value to the total distance value.

Calculating the best time

When having found the total distance, the program could calculate the best time. This could be done by calculating an average speed. First, I would need to add some distance to the distance variable since with the driving rules the program follows currently it is improbable that the car could follow the best path. Then a calculation could be made for the distance that the car isn't travelling at max speed. This is done by getting the speed update of the car and the effect of speed. The speed update is how much faster the car gets at every frame when giving gas with no interruption. The effect of speed makes it how hard it is for the car the get faster at higher speeds. With these numbers I could calculate the distance that the car isn't at max speed. I could then subtract that distance from the total distance. Then I could calculate average speeds for both distances. Then I could add them together and do them times the amount of rounds the cars would have to drive to get the best time.

Make

Drawing the genomes

For drawing the genomes, I first needed to create a “screen” that would show up. This screen would be a big black image. This could be activated by clicking on a button. When clicking on this button all functions off the UI screen would turn on and it would fill the dropdown with all the clients if it was the first time turning on this UI. The options would have the their name set to: Client and the number plus one the iteration was on. I also gave all the cars the same name in the hierachy so that it was possible to figure out what the best performing car was. It would then also add the function for the on value changed that would draw the genomes. It will also set the values for canvas width and height for calculating the positions of the nodes and connections. Furthermore the activate UI button would also set the time scale to zero when turned on and set it to its original state when turned off. When turning the UI off the system would also disable all the UI again. When hovering over the activate UI button the rotation of the camera couldn’t be changed anymore, because a bool would turn on. This bool was also on when the program was currently showing the UI.

```
public void FillDropDown()
{
    clientsDropdown = GetComponent<Dropdown>();

    clientsDropdown.ClearOptions();
    for (int i = 0; i < manager.GetPopulationSize(); i++)
    {
        int actualNumber = i + 1;
        string clientName = "Client " + actualNumber.ToString();
        clients.Add(clientName);
    }

    clientsDropdown.AddOptions(clients);
    clientsDropdown.onValueChanged.AddListener(ShowClient);

    canvasHeight = canvas.GetComponent<RectTransform>().rect.height;
    canvasWidth = canvas.GetComponent<RectTransform>().rect.width;
}
```

```
private void DrawNode(NodeGene n)
{
    Image node = Instantiate(oval);

    Vector3 pos = new Vector3((float)n.GetX() * canvasWidth, (float)n.GetY() * canvasHeight, 0);

    node.transform.position = pos;

    node.color = Color.gray;

    node.transform.SetParent(canvas.transform);
    images.Add(node);
}
```

In the draw node function the system would ask for a node gene. For this node gene it would instantiate a new circle image. The position of this circle would be set by getting the x and y value of this node gene and doing this times the canvas width and height. This can be done because the x and y values are always below the zero but higher then one. Than the colour is set to gray and the parent of the node is set to be the canvas so it can be drawn. Then it is added to the list of images.

For drawing connections the system would ask for a connection gene. To change the size of the connection the rect tranform needs to be grabbed. Then the length can be calculated using the bigger x position from the to node subtracted by the smaller x position from the from node and doing this times the canvas width. Than the size can be set. Next the position can be calculated in a similar way as the node gene. By grabbing the from node x position and adding the line width divided by two the x position can be decided. The y position is decided by an interpolation of the y position of the from and to node. The rotation can be decided by subtracting the positions from where the line need to look at. Then this can be converted to degrees and then set to be the rotation. The color can be decided based on the enabled

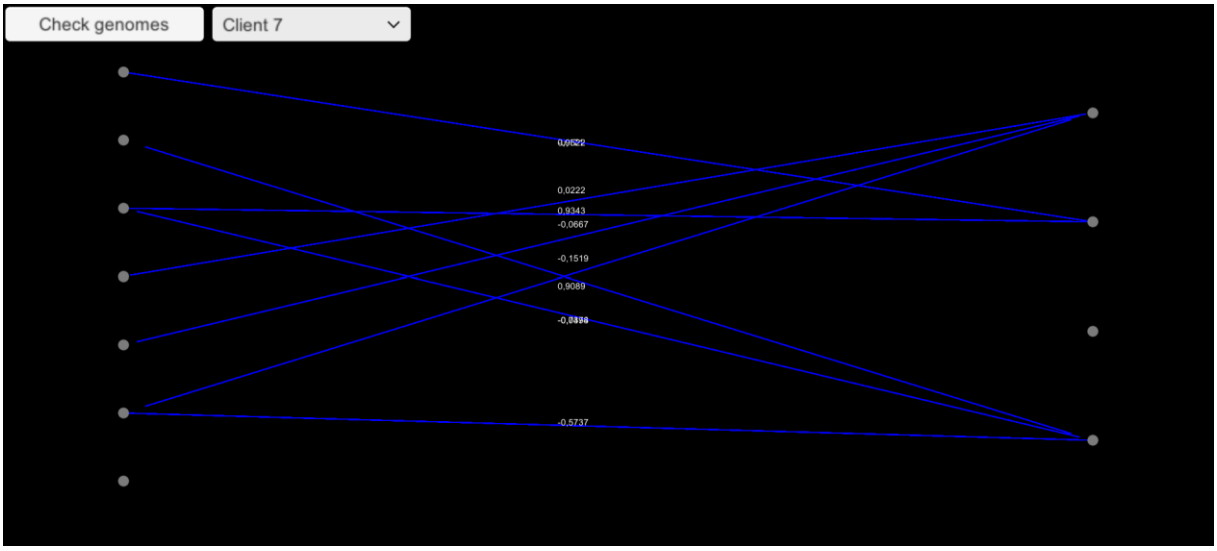
```
private void DrawConnection(ConnectionGene c)
{
    Image connection = Instantiate(line);
    RectTransform connectionTransform = connection.GetComponent<RectTransform>();

    double lineWidth = ((c.GetTo().GetX() - c.GetFrom().GetX()) * canvasWidth);

    connectionTransform.sizeDelta = new Vector2((float)lineWidth, lineHeight);

    Vector3 pos = new Vector3(((float)c.GetFrom().GetX() * canvasWidth) + ((float)lineWidth / 2),
        (float)(c.GetFrom().GetY() + c.GetTo().GetY()) / 2 * canvasHeight, 0);

    connection.transform.position = pos;

    Vector3 lookAt = new Vector3((float)c.GetTo().GetX() * canvasWidth, (float)c.GetTo().GetY() * canvasHeight, 0);
    float angleRad = Mathf.Atan2(lookAt.y - connection.transform.position.y, lookAt.x - connection.transform.position.x);
    float angleDeg = (180 / Mathf.PI) * angleRad;

    connection.transform.rotation = Quaternion.Euler(0, 0, angleDeg);

    if (c.IsEnabled())
    {
        connection.color = Color.blue;
    }
    else
    {
        connection.color = Color.red;
    }

    connection.transform.SetParent(canvas.transform);
    images.Add(connection);

    Text weight = Instantiate(text);

    weight.transform.position = pos;
    float digits = Mathf.Pow(10.0f, amountOfDigits);
    float weightNumber = Mathf.Round((float)c.GetWeight() * digits) / digits;
    weight.text = weightNumber.ToString();
    weight.color = Color.white;

    weight.transform.SetParent(canvas.transform);
    weights.Add(weight);
}
```

state of the connection. Then the parent of the connection is set to be the canvas and it is added to the image list. Next the text is created on the same position as the connection. Then the text is set equal to the weight of the connection. This is then rounded to four digits. And the text color is changed to be white so that it is visible. The parent is then set to be the canvas and this text is added to a text list.

```
private void ShowClient(int setting)
{
    for (int i = 0; i < images.Count; i++)
    {
        Destroy(images[i].gameObject);
    }
    images.Clear();

    for (int i = 0; i < weights.Count; i++)
    {
        Destroy(weights[i].gameObject);
    }
    weights.Clear();

    Genome genome = manager.GetNeat().GetClient(setting).GetGenome();

    foreach(ConnectionGene c in genome.GetConnections().GetData())
    {
        DrawConnection(c);
    }

    foreach (NodeGene n in genome.GetNodes().GetData())
    {
        DrawNode(n);
    }
}
```

When changing a value in the dropdown menu the show client function would activate. This function would first destroy the currently showing genome and clear the lists that these images and text where in. Then it would get the client based on the index that is selected. Then it would call the draw functions for all the connections and nodes in this genome.

This is what it looks like:



A* pathfinding

For creating pathfinding, I would first need to create some targets. These can be moved around in the scene using input of the mouse, the x button and a number on the keyboard. This number is decided based on what number it is in the list. These targets can be turned off using the c button. Whenever a target is moved the program recalculates the total distance.

```
public PathfindingNode(bool isWall, Vector3 pos, int  gridX, int gridY)
{
    this.isWall = isWall;
    this.pos = pos;
    this.gridX = gridX;
    this.gridY = gridY;
}
```

The second piece of code created for the pathfinding are the nodes. These contain a few variables such as their x and y position and if they are a wall. Furthermore, they have three costs. These are the gcost, hcost and fcost. The gcost is the distance from the start. The hcost is the Manhattan distance from the end point. This is a distance calculated purely based on a up, down, left right path. The fcost is the gcost plus the hcost.

Next the grid is created. First there is a start function that sets up all the variables and creates a grid with a create grid function.

```
private void CreateGrid()
{
    grid = new PathfindingNode[gridSizeX, gridSizeY];
    Vector3 bottomLeft = transform.position - Vector3.right * gridWorldSize.x / 2 - Vector3.forward * gridWorldSize.y / 2;
    for (int y = 0; y < gridSizeY; y++)
    {
        for (int x = 0; x < gridSizeX; x++)
        {
            Vector3 worldPoint = bottomLeft + Vector3.right * (x * nodeDiameter + nodeRadius) + Vector3.forward * (y * nodeDiameter + nodeRadius);
            bool wall = true;

            if (Physics.CheckSphere(worldPoint, nodeRadius, wallMask))
            {
                wall = false;
            }

            grid[x, y] = new PathfindingNode(wall, worldPoint, x, y);
        }
    }
}
```

In this function a multidimensional array is created. Next a start position is createed on the bottomleft. Then with two for loops all the nodes are created from the bottomleft. With a check sphere function and a layermask that checks for walls the program looks if the position a node has contains a wall. If it doesn't then the wall variabe will be set to false. Then a node is created with it's needed parameters.

```
public PathfindingNode NodeFromWorldPos(Vector3 worldPos)
{
    float xPoint = (worldPos.x + gridWorldSize.x / 2) / gridWorldSize.x;
    float yPoint = (worldPos.z + gridWorldSize.y / 2) / gridWorldSize.y;

    xPoint = Mathf.Clamp01(xPoint);
    yPoint = Mathf.Clamp01(yPoint);

    int x = Mathf.RoundToInt((gridSizeX - 1) * xPoint);
    int y = Mathf.RoundToInt((gridSizeY - 1) * yPoint);

    return grid[x, y];
}
```

Next a function in the grid is created called node from world position. First the position from the world is set to a position in the grid. Then these coordinates are clamped between zero and one so that if the target leaves the grid the program won't give an out of range error. Next the postion is calculated with an round to int calculation to get the correct node.

Next a function is created to check the neighboring nodes from the grid. This is done by checking the x and y positions from the grid by plus one and minus one. Then it is checked if this position exist on the grid. If it does then the node is added to the list of neighbor nodes. Then this list is returned at the end of the function.

```
public List<PathfindingNode> GetNeighborNodes(PathfindingNode node)
{
    List<PathfindingNode> neighboringNodes = new List<PathfindingNode>();
    int xCheck, yCheck;

    xCheck = node.GetGridX() + 1;
    yCheck = node.GetGridY();

    if (xCheck >= 0 && xCheck < gridSizeX)
    {
        if (yCheck >= 0 && yCheck < gridSizeY)
        {
            neighboringNodes.Add(grid[xCheck, yCheck]);
        }
    }
```

```
private void GetFinalPath(PathfindingNode startNode, PathfindingNode endNode)
{
    List<PathfindingNode> finalPath = new List<PathfindingNode>();
    PathfindingNode currentNode = endNode;

    while (currentNode != startNode)
    {
        finalPath.Add(currentNode);
        currentNode = currentNode.GetParent();
    }

    completeDist += finalPath.Count;

    if (targetIndex < grid.GetTargetList().Count - 1)
    {
        targetIndex += 1;
        startPos = targetPos;
        SetTarget(targetIndex);
    }
    else
    {
        found = true;
        manager.CalculateTime(completeDist);
        manager.EnablePathfinding(false);
    }
}
```

For getting the final path a new list and node are created. The node is set equal to the end node. Then the program adds the nodes to the list and grabs the parents of the nodes till the node is equal to the start position. When it found the final node it calculates the distance of all the nodes and adds it to the complete distance. If there are mare targets to go then it wil grab the next target and set that as the tartget. The current target will be set as the start position. When all targets are found then the found will be set to true and the patfinding wills stop. The best time can then be calculated.

```
openList.Add(startNode);
while (openList.Count > 0)
{
    PathfindingNode currentNode = openList[0];

    for (int i = 1; i < openList.Count; i++)
    {
        if (openList[i].GetFCost() <= currentNode.GetFCost() && openList[i].GetHCost() < currentNode.GetHCost())
        {
            currentNode = openList[i];
        }
    }

    openList.Remove(currentNode);
    closedList.Add(currentNode);

    if (currentNode == targetNode)
    {
        GetFinalPath(startNode, targetNode);
        break;
    }

    foreach (PathfindingNode node in grid.GetNeighborNodes(currentNode))
    {
        if (!node.IsWall() || closedList.Contains(node))
        {
            continue;
        }

        int moveCost = currentNode.GetGCost() + GetManhattenDist(currentNode, node);

        if (moveCost < node.GetFCost() || !openList.Contains(node))
        {
            node.SetGCost(moveCost);
            node.SetHcost(GetManhattenDist(node, targetNode));
            node.SetParent(currentNode);
            openList.Add(node);
        }
    }
}
```

Next the pathfinding class is created. This class contains the findpath function. This class asks for a start position and a target position. With node from world position function from the grid the grid position from these are calculated. Next two lists are created. These are the open and closed lists. Then the start node is added to the start list to start with. While the open list contains objects the program keeps searching for a solution. Then a node is create that is set equal to the first node of the open list. The the program loops throught all the nodes in the open list to see what the fastest route is and sets the current node to whichever node is the best. Then it removes this node from the open list and adds it to the solution list. If the current node then is the target node the program retrieves the final path and stops the function. Then the program will look throught all the neighbor nodes of the best node if the program didn't find the solution yet. In this loop the program will first check if the neighboring node isn't already in the solution or if it is a wall. If it is one of these things then the functio is cut off. Else the function continues and the move cost of the neighboring is calculated. This is the gcost of the node plus the Manhatten distance.

```
private int GetManhattenDist(PathfindingNode currentNode, PathfindingNode neighborNode)
{
    int x = Mathf.Abs(currentNode.GetGridX() - neighborNode.GetGridX());
    int y = Mathf.Abs(currentNode.GetGridY() - neighborNode.GetGridY());

    return x + y;
}
```

If this move cost is then less then the fcost of the node and the node isn't in the openlist yet then the gcost is set to the move cost, the hcost is set to the manhatten distance between the node and the target node and the parent is set to be the current node. Then this node is added to the open list.

Calculating the best time

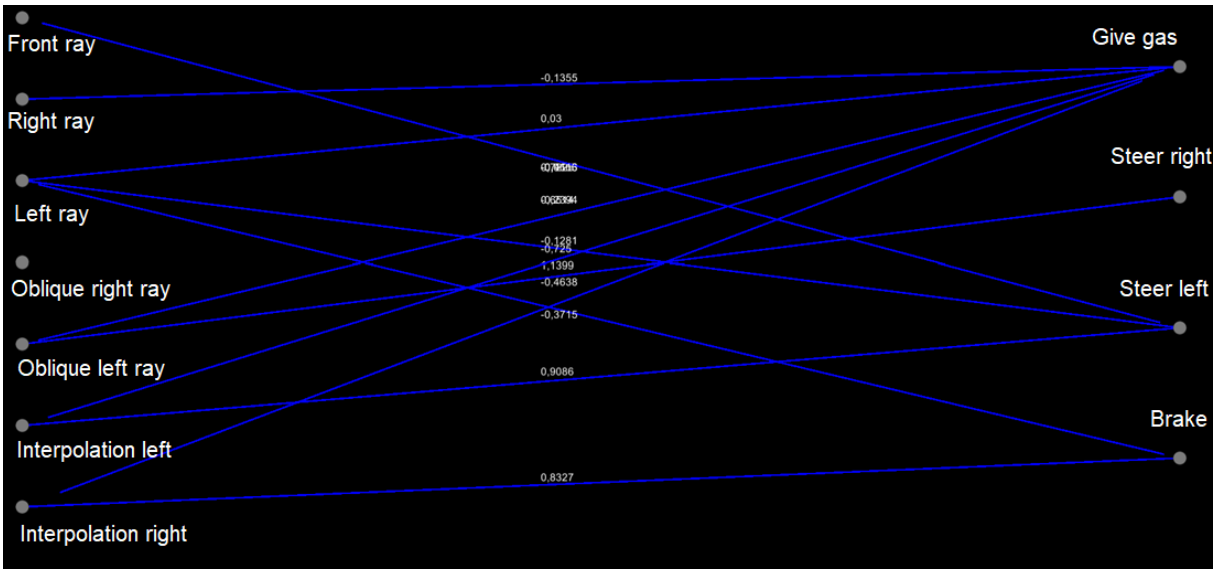
```
public void CalculateTime(float distance)
{
    distance *= 1.25f;
    float distNotMax = distance / (carList[0].GetSpeedUpdate() * 2) * (carList[0].GetEffectOfSpeed() / 100 + 1);
    distance -= distNotMax;

    float addedTime = distNotMax / (carList[0].GetMaxSpeed() / 2);
    float bestTime = distance / carList[0].GetMaxSpeed();

    bestTime += addedTime;

    bestTime *= finish.GetLapAmount();

    bestTimeText.text = "Best time possible: " + bestTime.ToString();
    finish.SetTimeFactor(bestTime);
}
```

For calculating the best time, the complete distance is asked. Since the car can't follow the perfect path correctly. Then the distance that the car does not spent at max speed is calculated. This is done by dividing the distance the speed update times two and doing it times percentage that the effect of speed adds. Then the new distance is calculated. Then average times of these speeds are calculated. Then these times are added to each other and done times the number of laps to get the best time. Then the time factor used to calculate points is set to this best time.

Check

Small iterations

For the tests I would run I would check whether the program actually got better at driving the map. So, I would check whether the first completed time was worse than the later times. I also checked whether the program came close to the optimal time. The first thing I tried was changing the output choices. I changed to first output choice to not do anything anymore, but have the car always give gas. However, by doing this the map got incredibly hard to complete for the program and after close to a thousand generations the program couldn't complete the first sharp corner. So, I reverted this chance and tried something else. To make the program more focused on getting the targets with the best time I would divide the amount of points gotten per checkpoint by the time they completed it in. With the live value being lower per checkpoint if the time was higher, cars with lower times would perform better in the breeding process. I also made it so that the cars only had to complete one round instead of three, so that the best performing cars wouldn't randomly crash in a later round. With these changes the time became a little bit better. The time of a single round was 27 seconds. The average of the previous best time for a single round was 32 seconds, but that was in around 400 hundred generations, while this time was reached in 120 generations. So, this technique where there is more focus on the time does get a better time quicker. The best average time calculated for this lap is fifteen seconds, so the program is still far away from the best time. The genome of a succesful client looks like this:



It is visible that most of the focus is here on the giving gas from the outputs, which makes sense since the car gets more live points if it does the circuit in a faster time. Having the least amount of links for steering right also makes sense, because there is only one turn for right, the rest of the turns are left. Because of the connection to the brake the program reaches a worse time then is possible. The car that did this time still relies on the support system to drive it forward at some points.

Conclusions for the week

For calculating the best time, it would be better to create an ai agent that would follow this path correctly using the rules of the car instead of trying to calculate an average speed and dividing the distance by this. So, for the next step of this project I want to add a driving agent that can follow the path I created using the rules of the current driving class. With this I will be able to have the most accurate showing of the best time that is able to be done over the track. Furthermore, there seems to be a bug with the drawing of the genomes where the connections image is to short, but the position is still correctly set to be in the middle of the nodes, so the correct width is calculated. So,

there is a problem with setting the width of an image. Furthermore, all the cars in the scene need names in the game screen to more easily check the genomes. Because now the program needs to be run in the editor if you want to see which car is doing the best. This is will then be done by clicking on a car and then a UI element will change to the name of the car and then genome menu can be opened to check what the genome looks like. This can be on the right side of the canvas. Lastly, I want to add a function that shows what species are present and how well they are doing. This can also be shown on the canvas with text. This can then show up on the left side of the canvas. This can have a set size and when there are too many species then the text becomes smaller or a scrollbar appears to enable showing all the species.

Conclusion

The question is asked at the start of this report was: how can I create an evolutionary ai that completes a created circuit in the most optimal time? The goals that were linked to this question were:

- The ai works with a neural network that goes through an iterative process to learn how to drive a track.
- The ai has no support deciding when it needs to steer.
- The ai gets better over time at driving the track.
- The ai reaches an optimal time to complete the track at.

The answer to this is that NEAT is does qualify for most of these requirements. NEAT is a neural network that learns through an iterative process. In that process it creates links and nodes to find a solution to the problem at hand. The NEAT system didn't get any help when it comes to steering, though it did get help going forward, although it doesn't need it. It makes the process take a little less time by having a support system. The ai did get better at driving the since it on its first try only goes forward and only after a little while starts steering. The only problem is that the NEAT system might not find the optimal time to complete a track at, but with enough waiting time or luck the system might still find the optimal route. Thus, I might not have found the right answer for answering this question.