Abstract

The main purpose of this project was to investigate into the applications of both neural networks and genetic algorithms, and use the findings to create realistic artificial intelligence for a race car to be able to drive around a race track. The race car is able to drive around a track based solely on the surrounding environment, after being trained to predict and response in a realistic manner. The training phase involves the use of crossover and mutation, and is modelled using Darwinian evolutionary principles. The combination allows the race car to adapt and inherit the best features from a sample of different race cars. This results in the creation of a race car that can drive around tracks efficiently, taking advantage of a basic form of intelligence possessed by living creatures. The software is developed within XNA using C#, and the framework was developed using an adapted version of the MVC design principle. XNA was selected as the language due to the practical applications of AI within gaming. The finished product was designed in such a way that it can be directly applied to applications which require some form of realistic behaviour in the form of a human or other living creature.

Introduction

The concept of neural networks within computing, and especially gaming, has become a particular area of interest within the last 100 or so years. As such, the creation of better and more efficient methods of accomplishing computation problems has increased also, and the concept of a more realistic artificial intelligence is now starting to be realised within gaming and simulations alike. This project aims to combine two areas that have been of particular interest to me in the recent year; artificial neural networks and genetic algorithms.

The applications of both of these areas of computing have been realised in many different ways in the recent years. Face recognition software and fingerprint scanning which not long ago would have seemed to be luxury computer peripherals are now developed with the mass public, and this in turn is due to the growing understanding and research into both neural networks and genetic algorithms. For this project, the aim is to use these methods to create a form of AI which will allow a car the ability to perceive itself within a race track, with one goal in mind; driving around the track in the most efficient manner possible.

In this project, research will be conducted into the principles behind both neural networks, and genetic algorithms. This will include looking into the different ways that these principles exist, such as the different types and variants of each, and will also involve research into applications which already exist using these principles. The chosen development platform is the XNA framework within C#. The XNA framework will allow provide facilities for dealing with 2D graphics, and will allow more focus to be placed on the design model itself. This project aims to hopefully inform the reader of these principles, and also provide the reader with enough knowledge to understand these principles, and gain something in return. This project also hopes to explore new and different ways of looking at what currently exists to-date, and

as a result create something that is informative and unique to people who are beginners to the area of computing, or who seek research for work of their own.

This project is arranged in a series of chapters. The first chapter is the literature review. This chapter collates research that has been gathered from different sources (books, online, etc) into an informative series of sections covering the concepts of both neural networks and genetic algorithms. It is recommended that people who are unfamiliar with either of the previously mentioned topics read through this section and understand as much as possible before reading further. The topic uses an array of cross-referenced sources for ascertaining the information required for the chapter, so if the explanations aren't as helpful as you'd desired then researching into the references may provide a better understanding.

The next chapter is on analysis and specification. This chapter aims to summarise the information gained from the literature review, and produce a set of key considerations for each relevant part of the research. The specification that follows is based upon the considerations made in the analysis, and is designed to aid in confirming whether the project is successful at the end of the report. As this project emphasises upon neural networks and genetic algorithms, considerations for basic design principles and such are not mentioned here. This is due to the time restraints imposed for this particular project, and emphasising on the particulars had a much higher priority. To note, my personal preference of design patterns usually falls between MVC (Model – View - Controller) and/or the Command pattern. This project will almost certainly use the MVC pattern, and there are plenty of resources available on the internet if further research is desired.

The design chapter will then follow, which will detail all the intricate design details for how each of the key components of the system will operate. This section aims to clarify the direction of the project based upon the specification, and is designed to detail each of the main specification points.

The next chapter briefly covers the implementation. A class diagram is displayed here showing the final structure of the software. Considerations and changes made during the development would usually be here, however they've been placed within the next chapter for critical analysis purposes.

The final chapter covers the critical evaluation and conclusion. The final result is analysed here, including the analysis of which parts went well, which parts didn't, and which parts needed changing and why. A reference list containing all research sources is also provided at the back of this report.

Literature Review

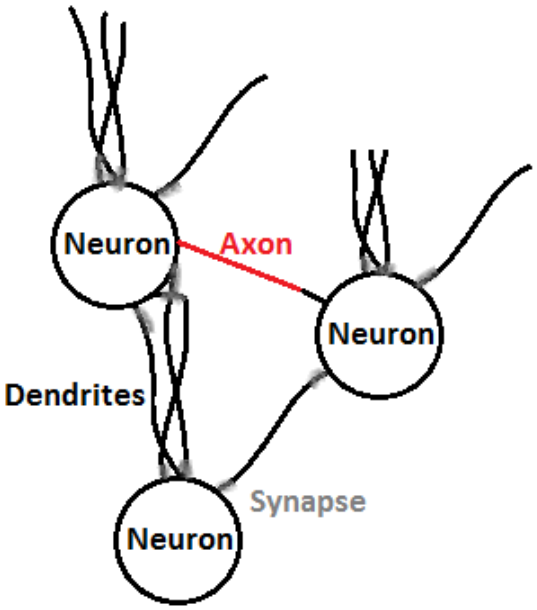
Neural Networks

The project will require a thorough understanding on neural networks. The neural network is the fundamental component of the project, and hence requires a large amount of research to allow for a successful final product. This section covers the areas regarding the key concepts of a neural network. Also considered are the different types of neural networks available, and research into previous examples of where neural networks have been used successfully. These factors will give a clear indication in regards to the direction in which development needs to take place.

What Are Neural Networks?

A neural network is a series of interconnected cells called neurons, which are able to send signals to each other. The signals received by each individual neuron, when combined together, are able to create various outputs. The combinatorial effects of such signal transmissions mean that each neuron individually can influence the way a network behaves, allowing complex computations to be performed when different individual input are used within the network.

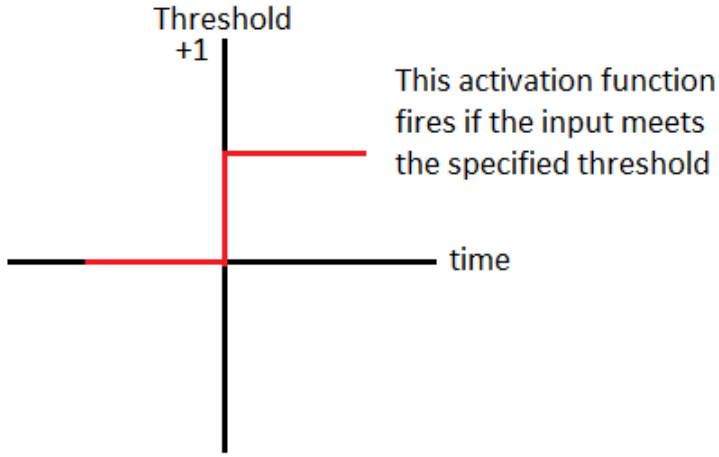
Before being able to understand how a neural network can be emulated in code, it's fundamental to understand the different components which make up a neural network. Neural networks consist of a network of neurons. Neurons contain branches called dendrites, and at the end of each dendrite is a synapse. The synapses within each neuron receive signals from each other neuron they're connected to, resulting in a large network of interconnecting neurons. A neuron generates an electrical signal by calculating the inputs of each of the connecting neurons, and if a certain threshold is met, the neuron fires. When a neuron fires it sends a signal through the axon, which branches to one or many other neurons, and enters through the neurons dendrites, thus repeating the process:



Analysing the model, it can be determined that neurons can be modelled in very basic terms. The neuron receives inputs, and uses these inputs to generate outputs. These outputs then become the inputs for other neurons. In the biological model, incoming signals only fire if the input from other neurons exceeds a defined voltage. This is referred to as the threshold, and this threshold determines the amount of voltage that is transmitted as output if the neuron fires. The biological model is too complex to model, mainly due to the chemical reactions used when generating the specific voltages needed for each neuron firing. Instead, when modelling a neuron, the inputs are summed, and fed into what is known as the activation function. There are many different types of activation function, each one producing an output that is defined within certain constraints. The most common two are as follows:

The Step Function

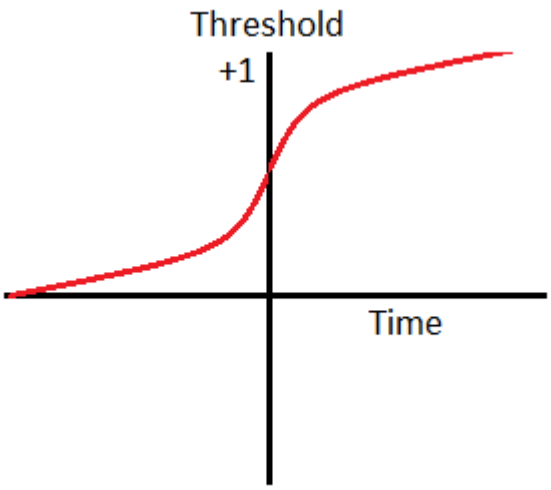
This function simply takes the inputs from each neuron, and after summing the inputs determines whether the result is above the threshold. If the result is above the threshold, than the output produced is 1. If the result doesn't exceed the threshold, then the output is a 0:



This form of function works well for binary computation, where different inputs of 0 and 1 can be interpreted into different output patterns. During research, it was found that these activation functions are useful when emulating the type of computational logic used in traditional logic gate circuitry.

The Sigmoid Function

This function takes inputs from each neuron, and after summing the input feeds the result through an equation, which gives the output in the form of a hyperbolic tangent:



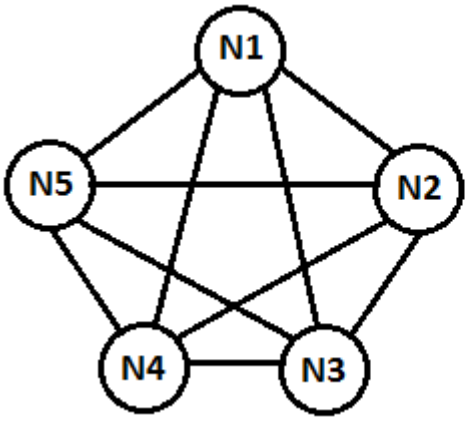
The hyperbolic tangent is best at emulating the process which is performed when a biological neuron fires, and so has much more practical applications when creating realistic responses between neurons. The output created by this activation function is a decimal value between the range of 0 and 1. The function can be altered to enable the outputs to range between -1 and 1. This is known as the bipolar sigmoid activation function.

Types of Neural Networks

There are various different neural network variants that exist; each designed to tailor a specific set of problems. The network types discussed here will focus upon artificial intelligence and the process of learning only. These networks include the Hopfield Net, single-layer perceptron, and multi-layer perceptron.

Hopfield Net

The Hopfield network is a network which provides associative memory by using a series of interconnecting neurons. Neurons act as both inputs and outputs, and except binary threshold values:



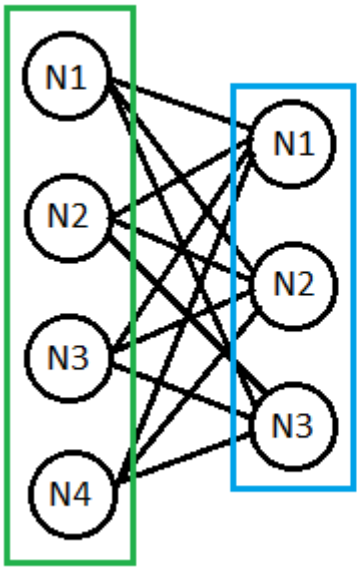
Each neuron is connected to every other neuron, excluding itself.

Weights exist between each connection. These weights are used to calculate whether the desired threshold is exceeded

This network can be used to store binary patterns, which if incomplete will merge at the local minima. In simplistic terms, if a pattern is incorrect by a certain margin of error, the inputted pattern will revert the closest desired. This is the reason why it's considered associative; because if you input a nearly correct pattern, it will associate it with the nearest pattern that it has stored at that time.

Single Layer Neural Network (Perceptron)

A single layer neural network uses neurons in a linear manner, as apposed to the Hopfield network shown previous. Instead of having independent neurons interacting with each other, they instead interact through a layer, which separates input neurons from output neurons:

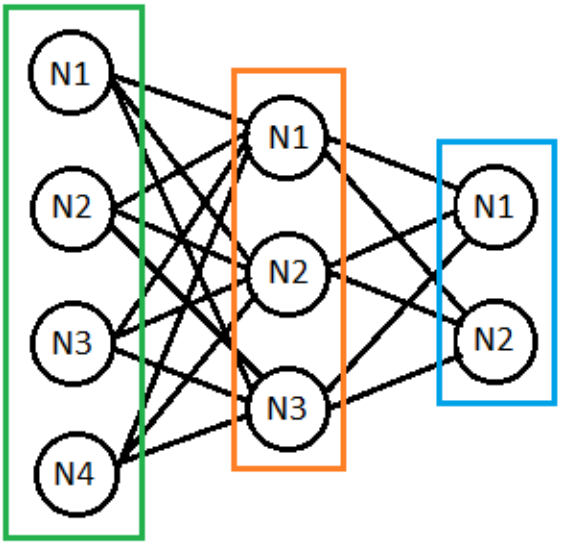


Each neuron within the green area is a part of the input layer, and each neuron within the blue area is a part of the output layer. Each input neuron is connected to every output neuron, with each connection having a specified weight value.

These networks allow input values to be entered, and depending on the weight values for each interconnecting value, as well as the activation function chosen, specific output can be created. These outputs can be used within the learning process (genetic algorithms in this case), to modify the weight values until the neural network learns to accept the inputs that it's given.

Multi-layer Neural Network (Perceptron)

The multi-layer neural network is similar in principle to the single layer network, except that is uses more then one layer. The addition of a 'hidden' layer is used, and provides greater complexity:



This network is similar to the single layer, except there is an additional layer, called the hidden layer (shown as orange).

Examples of Neural Networks

Neural networks are used for a variety of different tasks, usually which involve some form of artificial intelligence. There are other uses however, which cover a large range of difference subject areas, including telephone noise cancelling, associative memory storage, and artificial life, and many many more. A good comprehensive list detailing the different applications which neural networks can be used for can be found here:

http://tralvex.com/pub/nap/ (Travlex Y, 2006)

In the following example Jeff Hannon, a developer at the games studio Codemasters, discusses when he created a neural network design for the game Colin Mcrae Rally 2.

http://www.generation5.org/content/2001/hannan.asp (Hannon J, 2001)

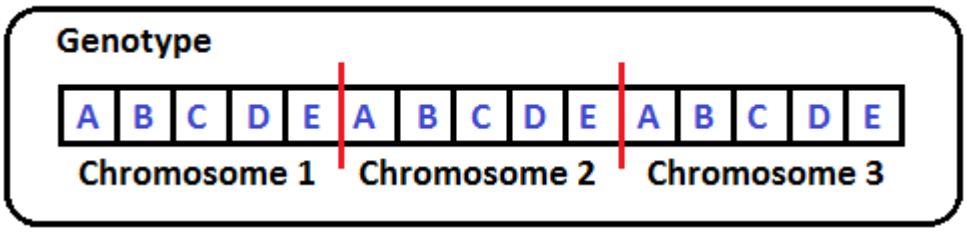
Genetic Algorithms

Genetic algorithms will allow the racing AI to adapt and evolve into a working system. While the neural network may explicitly control the behaviour of a racing car, the genetic algorithms have a direct impact in regards to how the racing car perceives the world around it. This section contains basic information regarding how genetic algorithms function, and looks into the different types of genetic algorithms available. Current examples of genetic algorithms used in similar applications are also covered, providing information regarding the potential pros and cons of already existing solutions.

What are Genetic Algorithms?

Genetic algorithms are essentially search algorithms which use evolutionary methods and techniques for finding a desired result. The use of evolutionary methods such as selection, crossover and mutation are generally used within genetic algorithms, however there are cases when not all of the previously mentioned are used (simulated annealing for example.). This section will briefly cover the basic concepts behind genetic algorithms, as well as some more specific points which relate to the project.

Genetic algorithms use many of the same concepts shown in biological genetics. There will likely be a population of entities, and these entities will likely be of the same species. As such, this species will share common characteristics and traits. These traits make up the blueprint for the species, and are referred to as genes. Each gene is encoded to contain information for a different part of that species make-up, for example eye colour or hair colour, and the possible options that exist for this gene are known as alleles. These long strings of genes are then connected further into what are known as chromosomes. And finally, the species genotype is the combinations of different chromosomes with different variations. The diagram below details the relations between these terms:



This genotype consists of three chromosomes, each containing five genes. The genes in this example, shown in blue, accept the letters A - E as values. These possible values are known as alleles.

The position of a gene within a chromosome is referred to as the locus. Within genetic algorithms, it's often the case that algorithms need to manipulate individual genes within chromosome. This is because unless the algorithm is being developed to accommodate multiple different traits at once, only a single chromosome may need encoding as part of a genotype. This is

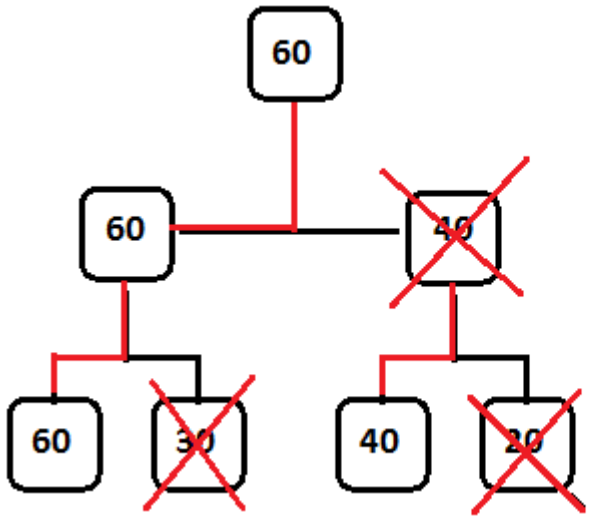
particularly the case when genetic algorithms are used for searching and retrieving information based on set inputs.

Selection

Selection is the process which involves selecting a proportional sample from the population, based upon the samples fitness. The fitness for each member of the population indicates whether or not that member is particularly good or bad at the task it needs to complete. There are various methods for selecting which members of the population are the most suitable.

Roulette wheel is generally the most popular, as it promotes a strong parallelism with how selection occurs in real world scenarios. For roulette wheel selection, each member of the population is given a certain percentage chance for selection based upon their fitness score. This method ensures that even those members of the population with low fitness scores have a chance at being chosen for selection.

The next selection method looked into is the tournament selection method. This method is likely as one would expect, in that the population is placed into a tournament-style competition, and the candidates that reach a certain level in the tournament are selected. Each candidate is randomly picked to face another randomly picked candidate, and the winner is the candidate with the highest fitness. The winners of these rounds progress to compete against other winners, and the process repeats until there is a winner:

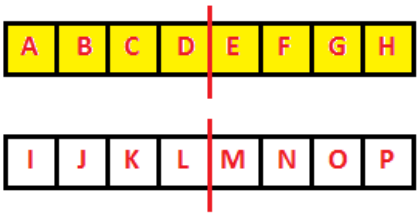


The population here consists of four entities. in this instance, if the desired selection size is two, then the entities with both 60 and 40 for their fitness would be selected.

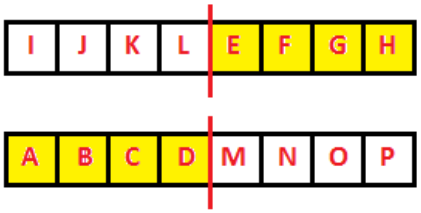
The final selection type which isn't used very often is truncation selection. This method involves ordering the population by fitness, and then keeping the best candidates of a certain fraction, for example keeping the best half. This method isn't used much because of the way it filters out members of the population that may potentially contain desired genes, and therefore rapidly decreases population diversity. There may be a way to counteract this problem with high mutation rates, similar to simulated annealing. It should be noted that this is a theoretical observation made by the author, and not a tested and confirmed method.

Crossover

This is the process which involves switching the genes of two selected sample members so that the resulting result is a combination of both the original sample members. Crossover generally follows the same basic concept; in that genes which have been selected for crossover follow a set pattern. The locus within each chromosome is generally used to split the array of genes up, and then these genes are swapped over at the locus:



After swapping the two sets of genes above at the locus (denoted by a red line), the result is a combination of the original members of the population



Choosing to swap at multiple different loci is also possible, however caution needs to be taken that constraints regarding the chromosome are not broken in the process. Alternating between crossover methods allows for a more realistic evolutionary process to occur, as biological recombination involves the occurrence of dominant and recessive gene traits, as well as particular chances regarding which genes will be crossed over and which will not.

Mutation

Mutation is the process of randomly mutations the value of a gene, based upon a specified level of chance. Within biological evolution, the chances of mutations are very small. For computational purposes however, a higher rate is generally accepted so as to benefit the population in the shortest possible time. Mutation allows for populations where convergence occurs quickly to continue to evolve, allowing the populations level of diversity to be maintained at a steady rate.

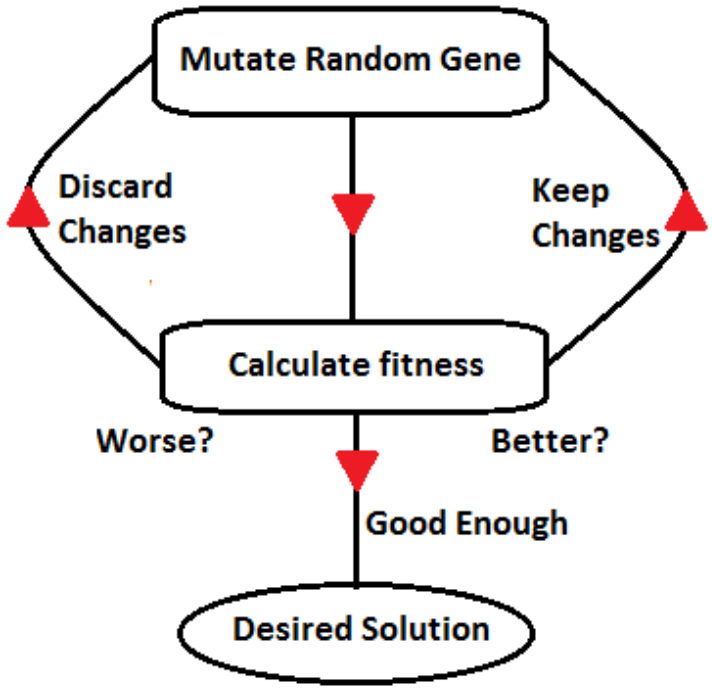
Mutation is performed differently, depending upon the encoding used for genotype. For example, if eye colour was chosen to be mutated, there would be a certain number of alleles that exist as potential eye colours. If these were represented using a numerical value, then the mutation operator would need to decide how best to mutate the value without destroying the constraints of that gene. The rate and size of mutations need to be considered based upon the situation, as small and infrequent mutations may have no effect, while large and consistent mutations may prevent crossover from being effective.

Types of Genetic Algorithms

Genetic algorithms don't seem to necessarily have particular types. Depending on the situation, choosing only certain parts of genetic algorithms may only be required. There is one particular type, which is also considered a meta-heuristic algorithm; simulated annealing.

Simulated Annealing

This process operates in a similar fashion to how mutation occurs within standard genetic algorithms. The big difference however is that only mutation is used throughout the entire process. Crossover can be implemented as an optimisation; however the fundamental concept of simulated annealing is that a mutation occurs based on decreasing chance after each iteration of the algorithm. Crossover also naturally doesn't occur within simulated annealing algorithms because the algorithm generally uses a population of only one. If the fitness increases as a result of the mutation, then this one member of the population keeps the changes made, else the member doesn't change. This cycle is then repeated until a satisfactory result has been produced.



The algorithm first mutates a random gene. If the fitness is better, then it keeps the changes, else it discards the changes. The cycle is then repeated, and once the fitness is good enough, the solution is found.

Examples of Genetic Algorithms

Genetic algorithms are generally used in many specific areas of computing, and one of the most common usages of them is in finding the most optimum solution. The travelling salesman problem is a classic example, which shows the effectiveness of genetic algorithms against iterative algorithms when working with large numbers of cities:

http://www.lalena.com/AI/Tsp/ (Lalena M, unknown)

Another use for genetic algorithms is the ability to evolve the behaviour of artificial entities. In the following example, 'organisms' are evolved so that they can become better eaters of the food growing around them:

http://math.hws.edu/xJava/GA/ (Eck D, 2001)

Analysis and Specification

In order to determine the aims and specifications, it's important that the different ways in which the project can be accomplished are researched into. This section will cover the analysis of the research contained within the literature review, and looks at the different components which will be present in the final system specification. Here, the aims for the project will be established, and a clear set of the major tasks will be identified ready for the design process. A successful analysis will also mean that future potential problems can be isolated easier, and alternative options discussed in this section will allow for different approaches to be taken if necessary.

The major components for analysis, identified after completing the literature review, are as follows:

- Neural Network Design
- Genetic Algorithm Design
- Track Design
- Sensor Design

Neural Network Design

After conducting researching into a few different types, it became sufficiently clear how to design the neural network for the racing car AI. Analysis of the Colin McRae Rally game, which uses a neural network for its racing AI, allowed me to understand on a basic level that it's possible to successfully implement a multi-layer perceptron for the task. After researching, it appears that both single and multi-layer perceptrons are used in a large range of robotic, movement based scenarios, and that the inputs and outputs can be easily acquired without too much hassle. For example, the structure for both the single and multi-layer perceptrons can be created as a stand-alone implementation, which accepts input (information about cars surrounding) and produces outputs (steering direction, speed). The usage of a multilayer network should provide output data that is more precise, and that is capable of handling more complex bends and manoeuvres on the track. The downsides to using a multilayer perceptron are that a more complex crossover system would be required. If however the neural network is designed so that any number of layers can be implemented with ease, then the network topology can be manipulated to facilitate in the creation of a more successful model.

The values used for weights are another key consideration which needs to be taken into account. The options consist of either integer values, or decimal values. Integer values would require the creation of constraints, considering that the range of whole numbers, as well as the requirement for only a fraction of that range, would result in unpredictable results if the constraints are broken. Decimal values, based upon the activation functions available, would appear to be the best choice by far. Decimal values within C# can be created easily using the Random class type, and it just so happens that the sigmoid

activation function utilises the usage of decimal values between 0 and 1. This means that creating values for the weights during the initialisation stage of the network would be considerably easier, and more efficient to develop and maintain in code.

The inputs for the network then would need to represent the track details as effectively as possible. Examples which have been looked at generally make use of straight line ‘sensors’, which in most cases act as rays. The information returned from these sensors is then put into the network as inputs, and the output produced causes the entity to perform some form of action. Steering is usually one of the outputs, so this will almost likely be the case for the neural network design for this project. The usage of acceleration seems to be an interesting idea as well, as many of the examples which were used didn’t feature any form of speed moderation as an output.

Genetic Algorithm Design

The genetic algorithm can easily be encoded by using the weights from the neural network as the genes which make up a chromosome. This data can then be subjected to the standard processes involved in genetic algorithms, such as selection, crossover and mutation. The selection process would likely use roulette selection under most circumstances, however the usage of truncation selection may prove to create a more interesting outcome. In order to make this project more unique, the truncation selection method can also provide more information as to its suitability within this type of project. Also, because the algorithm can easily be interchanged with another selection algorithm, the creation of a roulette selection algorithm may be possible depending on the success and available time during the projects development cycle.

Crossover could possible be performed using a type of ‘staggered’ pattern, such as swapping genes at each odd or even locus. Once again, modifications can be made if this method proves to be ineffective, assuming there is available time to do so.

Mutation should likely occur at a consistently high rate. This would allow the truncation selection to select more diverse samples from the population; by modifying select genes until the best genetically modified car is bred.

Track Design

A track design of some form will need to be created, which enables the car to intact with it, and determine where it is on the track. The use of different colours for the track and non-track space would appear to be a potential option, as the colour of each pixel can be determined by mapping the 2D texture and acquiring the colour that the car is currently on, or is near. Another option would be to use collision boxes, however determining the extent of collision using such a method may be more complex then desired. Based on the research into how the neural network will accept data, the usage of pixel colour checking would better suit the facilitation of some form of line based sensor.

Sensor Design

Based upon the potential idea of using pixel checking for determining the cars surrounding, a line, or vector based, sensor could be used to check whether certain points in a straight line distance away from the car are road or not roads. Another option could involve using a ray to calculate the distance; however there are several difficulties with that idea. Firstly, the implementation of such a thing would require much more extensive research. The second point is that if a ray was used, then the output of the ray would need to somehow be normalised to meet the input criteria for the chosen neural network design. Simulating the usage of a ray by defining the length and checking points along the length would prove much more successful, especially considering that they'd be more resource friendly, and hence wouldn't cause slow down issues when a large population is being iterated through.

Specification

Using the above analysis, and by filling in the details in-between, a list of aims and objectives can be created, detailing each of the tasks which need completing in order to successfully complete the project. The specification objectives are as follows:

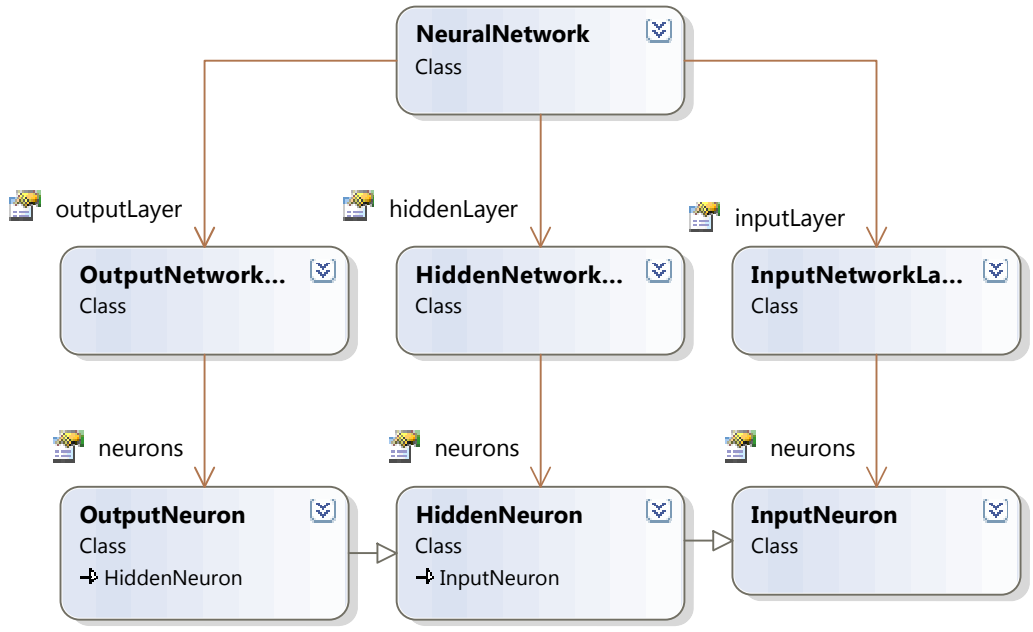
- Complete neural network implementation, which includes the completion of a fully functional activation function, and generation of desired outputs.
- Complete genetic algorithm implementation, which includes the ability to reproduce more successful cars.
- Complete track implementation, including the implementation of track segments and desired direction vectors.
- Complete the sensor implementation for each car, and ensure that the sensors accurately collect data about the layout of the track in range.
- Successfully combine all of the above, and then create a training system which allows the user to train the network under supervision.
- Allow the best car from the population the ability to race on an entirely different track, to test whether the training of the car was successful.

Design

In this section, each of the subjects which are covered in the analysis is worked upon. Designs which incorporate the research and technical information gathered in both the literature review and analysis are used to create implementation-ready designs.

Neural Network Design

The neural network takes advantage of the object-orientated nature of C#, and allows the creation of a completely separate and modular design model, which can be incorporated into the implementation for the car. Each neuron within the network requires connection to every neuron in the layer above. The most effective method available involves first splitting the layers of the network up into three distinct layers; the input layer, the hidden layer, and the output layer. These layers each facilitate in storing three different types of neuron signified by their parent class, these being input neurons, hidden neurons, and output neurons. Each neuron within each of the layers stores a certain amount of information which allows them to function correctly. It just so happens that the amount of information stored by each type of neuron increases, using the same information as neurons that proceeds it. Therefore the following model can be created:



In the model above, the neural network class contains a single instance of each of the required layers that it needs to function. Within these layers, lists of neurons relating to the relevant neuron type are stored.

As shown in the model, the input neuron is used as the base class. This neuron stores only inputs, and because all neurons contain inputs, then this works as a suitable base class. The hidden neuron also contains inputs, and therefore can inherit this from the input class. It also contains additional

information about the weights for the input neurons that it shares connections with. These weights are stored within an array, which contains all the connections relevant to each input neuron in the proceeding layer. Finally, the output neuron uses the same information from both the input and hidden neuron class, and thus only needs to inherit from the hidden class, as it contains the input class data through inheritance already. The output neuron class has additional data regarding the output created from after the activation function is used. The addition of the output data provides clarity for the overall structure of the network. It should be noted that it is possible to link hidden neuron layers together, and thus avoid the need for the output layer. However for the sake of coherence, the output layer exists within this particular instance of model.
The layers of each neuron type alone are not enough, as these layers are only designed to store data. Additional methods are required that allow these layers to interact with other, and therefore create a functioning neural network.

Firstly, there needs to be some form of initialisation for each relevant neuron in the network. This initialisation will provide each of the weights that exist within each relevant neuron with an initial random value. This random value will be within the specified constraints, and will provide a completely a neural network that is created with completely unique neurons as part of its setup. The easiest way to perform this initialisation will be to iterate through the weights both the hidden and output layers' neurons, and to allocate a random number within the ranges permitted (decimal values between -1 and 1). The pseudo code would look something like the following:

```
Foreach (HiddenNeuron neuron in hiddenNeuronLayer)
{
        Foreach (double weight in weights)
        {
                Weight = random value between -1 and 1;
        }
}

Foreach (OutputNeuron neuron in outputNeuronLayer)
{
        Foreach (double weight in weights)
        {
                Weight = random value between -1 and 1;
        }
}
```

The next method which needs implementing is the bipolar sigmoid activation function. As mentioned within the analysis, the bipolar sigmoid activation function will provide outputs within the desired range, and will also accommodate the inputs within the same range. This function is also much easier to implement, and based upon the required results will suit the designed neural network much better then the others mentioned in the analysis. Both the mathematical and pseudo code are shown below:

$$g(x) = \frac{1 - e^{-x}}{1 + e^{-x}}$$

In C# : *((1.0 - Math.Exp(-(input))) / (1.0 + Math.Exp(-(input))));*

Input is the sum of all inputs, and Math.Exp() is used to return the exponent of the passed parameter.

With the activation function determined, all that remains is to create the method that will use the activation function and inputs and weights from each neuron to generate an output. The output for each neuron will then be used as the inputs for the next neuron layer. This will be done by iterating through each neuron, and multiplying the input from the previous neurons with the weights attributed to those neurons. The results will then be summed together, and the result will be used as the input for the bipolar sigmoid activation function. The code pseudo code is as follows:

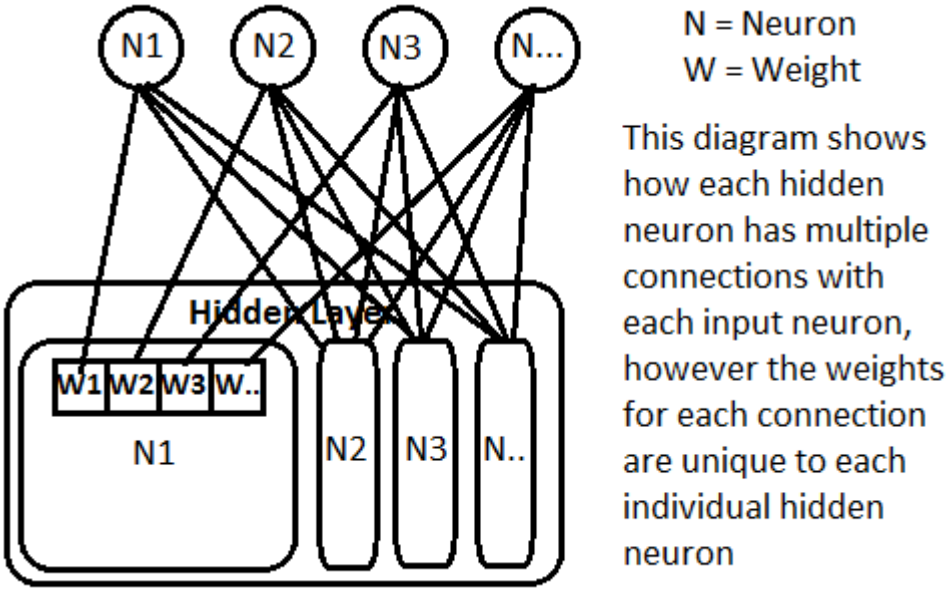
```
Foreach (HiddenNeuron hiddenNeuron in hiddenNeuronLayer)
{
        numberOfConnections = hiddenNeuron.weights.length;
        for (int x = 0; x < numberOfConnections; x++)
        {
                sumOfInput += hiddenNeuron.weights[x] *
        inputLayer.neurons[x].inputValue;
        }

        hiddenNeuron.inputValue =
bipolarSigmoidActivationFunction(sumOfInput);
}

Foreach (OutputNeuron outputNeuron in outputNeuronLayer)
{
        numberOfConnections = outputNeuron.weights.length;
        for (int x = 0; x < numberOfConnections; x++)
        {
                sumOfInput += outputNeuron.weights[x] *
                hiddenLayer.neurons[x].inputValue;
        }

        outputNeuron.output = bipolarSigmoidActivationFunction(sumOfInput);
}
```

The first half of the above code will iterate through the hidden layer, and use the number of connections contained within each hidden neuron to determine how many input neurons exist to connect with. Once the number of connections have been established, the weights at stored at each index in the hidden neuron are multiplied by the input value within the input neuron that matches it. The example below should show this much more clearly:



The second half of the code is identicial to the first, except the results from the activation function in the first half are used to calculate the output within each outputNeuron. This design should enable the creation of the network setup decided within the analysis, using five input neurons, three hidden neurons, and two output neurons. This design also means that if furthur neurons need to be added at any point, the relationships between connections can be done by simply manipulating the number of connections for each layer in relation to the number of neurons in the previous layer.

Genetic Algorithm Design

The genetic algorithm needs to be able to access the data contained within the neural network. In specifics, the data required is the weight data stored within each of the hidden and output neuron. Manipulating this data genetically will allow the neural network for each car to evolve based, upon whether the neural networks being used by each particular car are effective at guiding the car around the track.

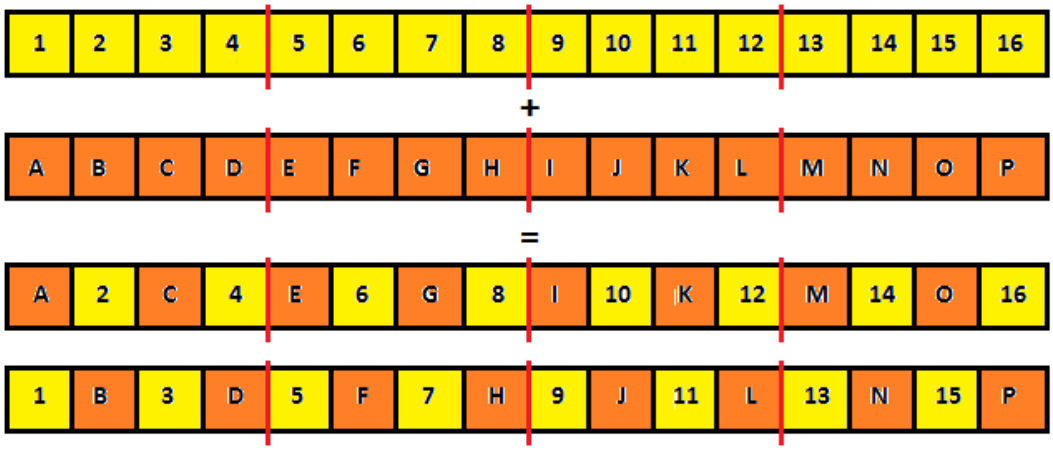
The first important task is establishing how the genetics for each car can be established using the weights for each network. The weight data for each neuron is stored within an array, and this was intentionally done so to provide easier access to the data. So, the data can be theoretically considered to be encoded as shown below:

The red lines shown above outline the locus that separates each encoded neuron within the chromosome. As the neural network already stores these encoded values within an array, the index of the array acts as the locus at each point of the chromosome. This allows easy access to relevant genes within the chromosome, and facilitates the crossover and mutation operators needed to evolve the population

The diagram above details a hidden neuron layer containing four hidden neurons, which each in contain four weights. In this instance, the first four weights would theoretically represent the first four genes in the chromosome. Then the next four would represent the next four, and so on. The advantage to this is that all four elements at the specified locus can be accessed using the index for the array, and therefore the formatting for each individual trait within the chromosome is always kept intact. This means that crossover and mutation will be considerably easier to perform using standard search and swap algorithms, and constraints in regards to the structure and value ranges will be conformed to.

Crossover

The desired algorithm will take the ‘even’ loci of the first chromosome, and swap them with the ‘even’ loci of the second chromosome:



The previous diagram does not represent the values used within the chromosomes. The usage of numbers and letters are simply for clarity. Those values would instead be decimal values between -1 and 1. An algorithm for the above crossover would involve iterating through a hidden neuron using an index, using that reference to access the weight value, and then using the same index, iterate through the second hidden neuron selected for the crossover method. The index will be incremented by 2, as apposed to 1, so that every-other weight is selected, and then these two weights will be crossed over. The pseudo code is as follows:

```
For (int weightCount = 0; weightCount < hiddenNeuron1.weights.length;
weightcount +=2)
{
        tempWeightValue = hiddenNeuron1.weights[weightCount];
        hiddenNeuron1.weights[weightCount] =
hiddenNeuron2.weights[weightCount];
        hiddenNeuron2.weights[weightCount] = tempWeightValue;
}
```

The above code will then be used for each neuron in the neuron layer, as shown in the above diagram as the point after each red line (locus).

Mutation

Mutation will occur based on a set rate, and will occur before crossover of the two selected neurons. The mutation will involve increasing or decreasing one of the weight values stored within each neuron by a random value, which will be generated using a random number generator. The random number generator will also select whether the previously generated number is added or subtracted from the weight. In order to calculate which weight should be selected from the neuron for mutation, a random number within the range of 0 and (neuron.weights.length-1) will be selected.

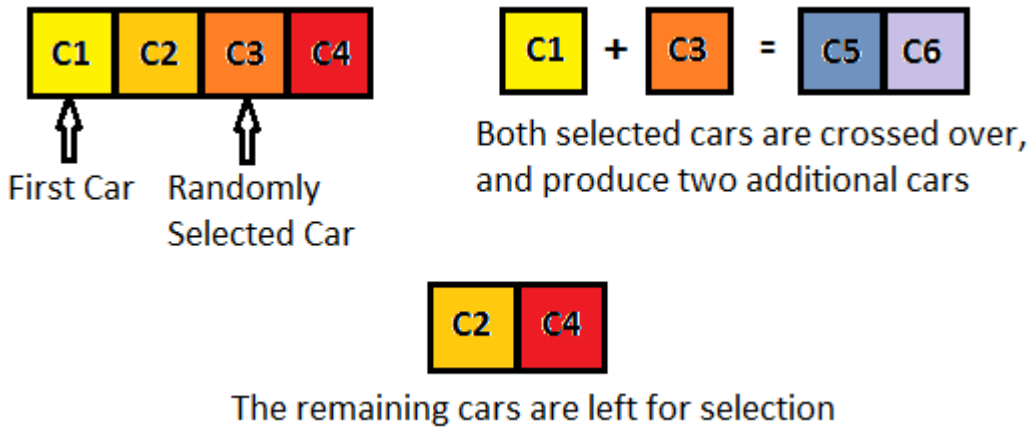
The code for this will simply utilise one random number generator that will first produce a value which defines the size of the change, and a second value, that will be checked against a constant value. This value will almost certainly be 0.5, so that a randomly generated decimal number between 0 and 1 will fall either above or below the value. The original random number is modified to ensure that the change isn't too significant, and then if the value falls above 0.5, then this value is added to weight. If it's less than 0.5, then the value is subtracted from the weight. The pseudo code for this should be as follows:

```
mutateIndex = randomNumber.Next(neuron.weights.Length);
mutateSignCheck = randomNumber.NextDouble();
mutateValue = randomNumber.NextDouble();

if (mutateSignCheck > 0.5)
{
        neuron.weights[mutateIndex] += mutateValue/10;
}
else
{
        neuron.weights[mutateIndex] -= mutateValue/10;
}
```

The Selection Process

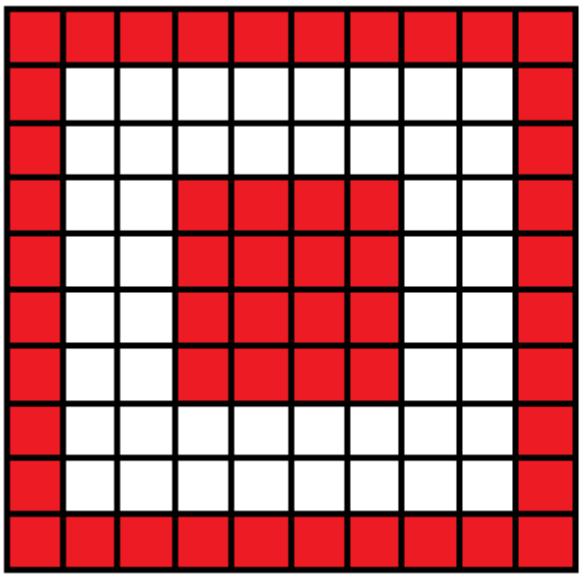
The selection method will be based upon the truncation selection method. A sample size will be defined, and a population of cars will be created. The sample will represent a a number of cars, each with an initally created neural network. These networks will all have a very high chance of being completely unique, and will therefore provide a highly varied sample. Inially, the sample which is created randomly will be used in the first simulation as the first generation, and will each receive calculated fitness levels based upon their success at manouvering around the test track. Once the fitness levels have been established, then a selection process needs to completed which ensures that the best cars are kept for breeding, and the others are discarded from the sample. Based upon the literature review and mentioned in the analysis, it would appear that strictly keeping the best cars using a tournament style selection process is the best method for selection. Keeping the best half of the sample, and then randomly crossing over pairs of cars will be the selected method of selection. So now that the selection type is determined, a method of randomly selecting these cars needs to be defined. This can be done quite simply by selecting the first car in the list of sample cars, and then randomly selecting another car from the remaining list, using the number of cars minus one as the range for the random number selection:



The method above ensures that the number of cars in the newly created sample is identical to that of the original sample. To also ensure that there is always a pair at the last iteration of the selection, the sample size will be set to an even value. Once the the cars have been selected,and crossover and mutation operator have been used upon them, the new sample is then run through the simulation again, and this process is repeated until a satisfactory car has been bred.

Track Design

Based on the analysis, the track will consist of an array of pixels, using two colours to differentiate between whether that pixel is a part of the road or not:

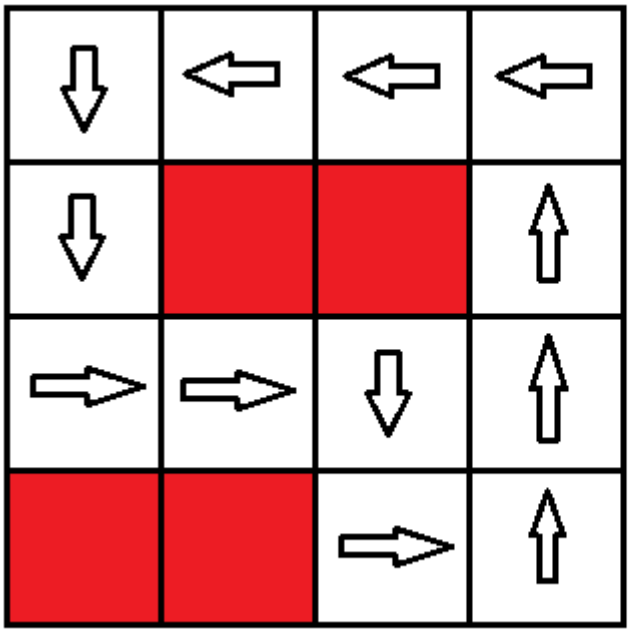


This grid shown left is just a small representation of a larger pixel array, containing red and white coloured pixels.

If the car sensors detect that a pixel is red, then this indicates that there is a wall at the location of that sensor.

Each car will have access to the track data at all times, and to save on memory usage all cars will have access to one instance of the track data.

In order to determine whether a car is travelling in the desired direction on the track, the track will be split into segments, which will each contain a 2D vector. The vector will become available to any car that is within the segment it belongs to, and will provide the car will the data it needs to calculate its fitness value:

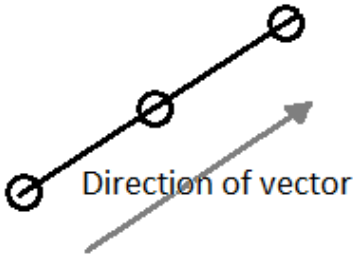


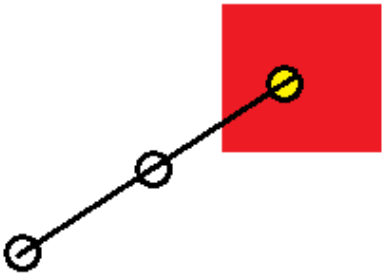
The directional arrows represent the desired direction (vector) that the car should travel in and that track segment. The red segments highlight where road is not present, and any vectors allocated to these segments are not used.

Sensor Design

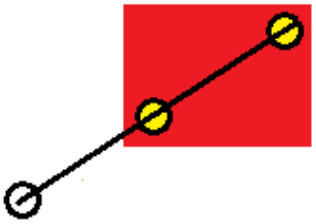
As one of the key components to the final product, the sensor class will need to be capable of successfully relaying data about the track into each cars neural network. First the method of detection needs to be established. Based on the analysis, the easiest and most time effective way of accomplishing this is by using multiple vectors. Then by using the cars current position, modifying the vectors current magnitude, and taking the cars current heading into consideration, multiple points along a line can be used to check for collision with red pixels from the track:

The yellow circles represent the different points along the vector where collision is checked for. The diagram below shows what values would be returned, based on the collision that occurs:



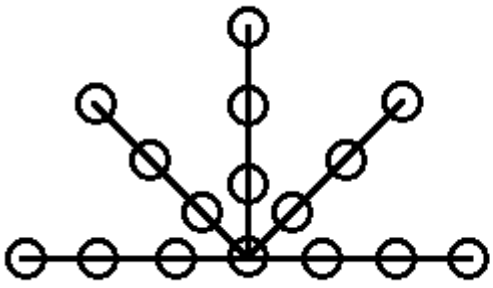


This would return the value 1.



This would return the value 0.5.

As shown above, the return value for the furthest point for the sensor is 1. The range for the sensor will be between 0 and 1, and all points inbetween will have values relative to where they are between 0 and 1. These values will be determined by dividing 1 by the maximum number of detection points. The final sensor sensor which the car will use will consist of 5 of the above sensors, placed as follows:

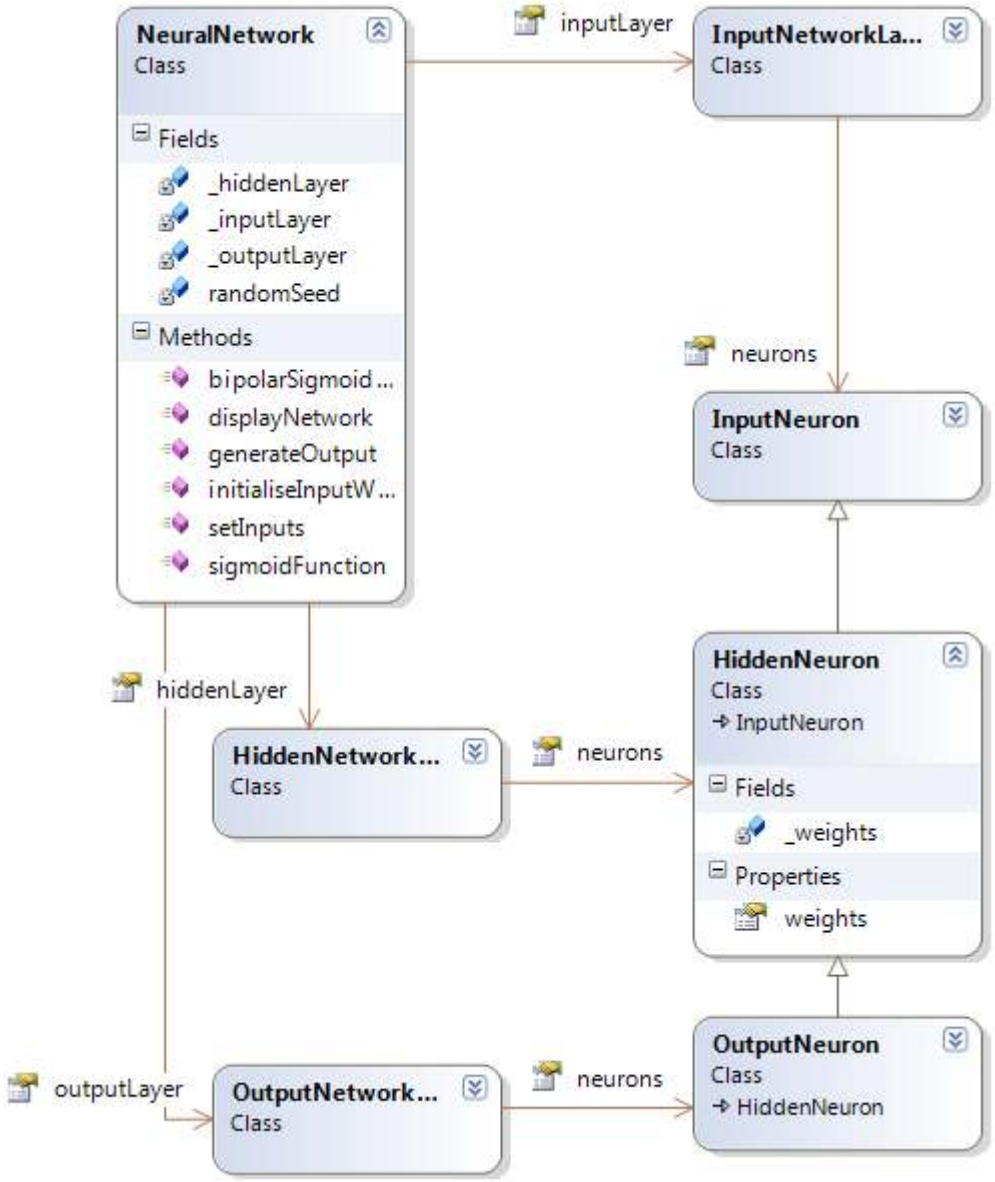


This final sensor uses 5 of the sensors mentioned previous, and combined them into a single unit. The each sensor has 4 detection points, and will span roughly greater then the width of the track.

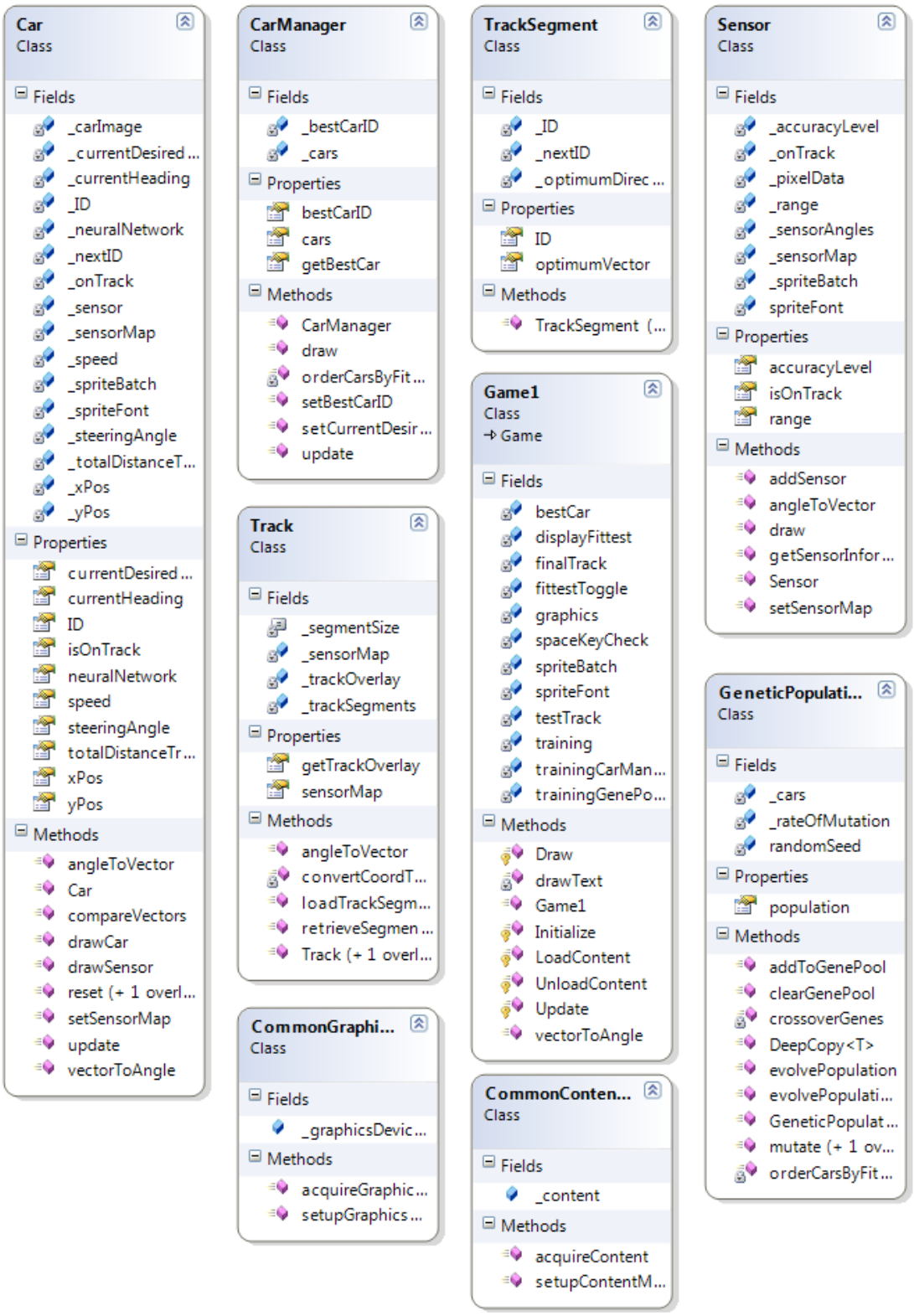
Implementation

This section contains the class diagrams for the final product.

Neural Network



Framework Structure (including Genetic Algorithm Modifications)



Critical Evaluation and Conclusion

The final product has met all the requirements at the very least moderately well. Each specification point will be evaluated separately.

Complete neural network implementation, which includes the completion of a fully functional activation function, and generation of desired outputs.

The neural network itself works exactly as desired, and due to the structural design is also usable as a separate class on its own. There were however issues with the type of neural network which was originally being used. The multi-layer neural network proved to be an unworkable solution when attempting to crossover the genes for each neuron. This was due to epistasis. This occurs when the modifying of one gene has impacts upon other genes which rely on it. This meant that when crossover occurred between genes, the changing of the genes within the hidden layer caused unexpected results at the output from the output neuron layer. Thankfully, the design of the neural network meant that the hidden layer could easily be bypassed, and as a result the multi-layer network became a single layer network. This changed successfully allowed the cars to breed, and so the decision to design the neural network as was done seemed to be a good choice.

Complete genetic algorithm implementation, which includes the ability to reproduce more successful cars.

The genetic algorithm and its operators work pretty much as desired. The genetic algorithms work perfectly as they are, however that's not to say that the choice of selection method was efficient. It seems that using truncation as the selection method worked fine; however the crossover diversity appeared to suffer much more then originally expected. It seems that the amount of potential for better members of the population isn't as high as was expected, evident by the fact that only generally between 1 and 5 out of 200 in the sample set would improve their general fitness values after each iteration. If more time was available, the implementation of the roulette wheel selection would have taken place, as mentioned in the analysis. The choice to use a staggered odd/even locus for gene selection during crossover also seemed like a good initial thought, however with further consideration it actually seems that doing so caused the crossover function to produce new members of the public that were much too different in comparison to their parents. For example, if one of the two members used in the crossover functions only needed one gene from the other member in order to produce better offspring, it wouldn't work correctly as the gene required and every other gene that was odd/even would also be crossed over within the same iteration.

Complete track implementation, including the implementation of track segments and desired direction vectors.

This particular specification task works perfectly without any problems known so far. This is likely due to the lower level of speciality involved in the process, as this objective simply involved the implementation of a storage system

which can be accessed using co-ordinate data. The task of creating a series of classes which store a global array of pixels and track segment objects was fairly straight forward.

Complete the sensor implementation for each car, and ensure that the sensors accurately collect data about the layout of the track in range.

The sensors were surprisingly easier to implement then first expected, and work perfectly. They were also created in such a way that the number of sensors, and the range to which they extended, could be modified and customised with remarkable ease. The original idea was to essentially 'hard-code', and make specific to the layout provided in the design. However, the potential to refactor the design and replace constant values with user configurable values meant that the class would be truly modular, and tweaking could be performed upon the model in order to find the best length for each individual sensor. The only possible missing aspect for the sensor class was that instead of using images to signify each point of the sensor, the letter 'O' was used. This was done as a temporary measure, however once it became clear that time was becoming an increasingly problematic factor, it was decided that it wasn't a major concern and so was left in the final product. The only issue that using the letter 'O' really caused was the issue regarding the origin to which it theoretically should have rotated around. Although the character can, in theory, be rotated and still remain the same, there were some slight discrepancies regarding the sensor display not matching the tracks pixel map. The different is only by roughly 2-3 pixels, and the affects are only superficial. Apart from that it works as desired.

Successfully combine all of the above, and then create a training system which allows the user to train the network under supervision

The training system works as expected. There is a notable limit to the size of the population sample before the frame rate starts to decrease, however this only effects the speed of the model, and not it's efficiency in the task. The limit appears to be roughly 200+, which is acceptable considering the amount of computation performed for each car per frame. The drop in frame rate also only occurs when all cars are displayed on screen. If the option to display the best car only is enabled, then the frame rate increases dramatically. This is likely due to the way in which the method was coded for displaying the sensors. In order to display the sensors, each sensor is essentially checked twice, firstly to acquire track data, and secondly to determine where to draw the sensor points. Both methods were intended to be merged, and a boolean value used to toggle drawing preference, however once again time was a limiting factor. The functionality itself however works exactly as desired so, apart from the point made previous, it seems that this specification point was completed successfully.

Allow the best car from the population the ability to race on an entirely different track, to test whether the training of the car was successful.

The final race track was a simple extension of the training track implementation, and so works just as well. As passing the final track tests whether the car is adequately trained, the conclusion can be made that it's functioning perfectly, as the car is capable of traversing the track after a successful training session. The advantage to the final race track is that there are no desired vector directions required, and so as long as the road width is roughly similar to the road width used in the training track, the track can be as complex as desired without needing to fit within the size constraints of the track segments. The successful completion of this specification point therefore shows that the overall project was a success.

In terms of project management, if this project was to be repeated again, then time management would be a point emphasised on the most. During the time period allocated, the availability of free time, and also that of other module deadlines, really needed to be considered when planning for the design and implementation of the project. Neither was critically affected during this project; however correct time allocation and job distribution for the different tasks may have provided the extra time needed to implement the changes desired for the roulette wheel selection function, the sensor image display implementation, and the combination of sensor operations and drawing. Another improvement that could have been made was to increase the amount of documented research in the literature review, to reflect the amount of research that was actually carried out. Unfortunately however, there was simply not enough time to include everything without jeopardising the completion of the entire report. The limit regarding the guideline number of words was removed after discussing the problems regarding fitting the data gathered within the documentation. Thankfully however, the amount of typed content was able to be reduced to a minimum by providing much more illustrations and diagrams through the documentation. It was essential to try and maintain clarity while not exceeding the limits that were set, and so it seemed that using specifically created diagrams would help in accomplishing that feat.

In the end, with the specification points completed and a working deliverable completed, it can be stated that this project was an overall success. In the end, it seems it is indeed possible to model a form of artificial intelligence that is capable of manoeuvring a car around a track.

References

Unknown. (Unknown). *Neural Networks in Plain English.* Available: http://www.ai-junkie.com/ann/evolved/nnt1.html. Last accessed 15 February 2010.

Valluru B. Rao (1993). *C++ Neural Networks and Fuzzy Logic*. New York: MIS-Press. p43-64

Rojas, R (1996). *Neural Networks : A Systematic Introduction*. Germany: Springer. p03-147.

Mitchell, M (1996). *An Introduction To Genetic Algorithms*. London: The MIT Press. p01-81.

Rabin, S (2004). *AI Game Programming Wisdom 2*. Massachusetts: Jenifer Niles. p467-489.

Jeff Hannan. (2001). *Interview With Jeff Hannan.* Available: http://www.ai-junkie.com/misc/hannan/hannan.html. Last accessed 15 February 2010.

Tralvex, Y. (2006). *Neural Network Applications.* Available: http://tralvex.com/pub/nap/. Last accessed 22 April 2010.

Malasri, S. (2001). *Hopfield Network Applet.* Available: http://www.cbu.edu/~pong/ai/hopfield/hopfieldapplet.html. Last accessed 24 April 2010.

Eck, D. (2001). *A Demonstration of the Genetic Algorithm.* Available: http://math.hws.edu/xJava/GA/. Last accessed 16 April 2010.