A NEAT Driving Simulator

Anton Sterner[1], Rasmus Ståhl[2]

Abstract
Self-driving cars is a hot topic these days. Controlling simulated cars is a slighty simpler task, which this paper aims to do using the *Neuroevolution of Augmenting Topologies* (NEAT) algorithm. The result is a Unity program which can generate neural networks to control cars driving around a track. Within a few generations, the algorithm is able to produce networks stable enough to navigate the whole track without crashing. This shows the NEAT algorithm quickly can generate an effective neural network of minimal structure to solve a given problem.

Source code: https://github.com/antonsterner/TNM095-Artificial-Intelligence-for-Interactive-Media

Authors
[1] *Media Technology Student at Linköping University, antst719@student.liu.se*
[2] *Media Technology Student at Linköping University, rasst403@student.liu.se*

Keywords: NEAT — Driving simulator — Genetic Algorithm — Neural Network

Contents

1. Introduction

In today's society, there is a rapid emergence of computer science solutions being created through A.I. and machine learning. The benefits of letting computers learn how to solve complex problems is proving to be many and large. One of the biggest areas of A.I. research is in self driving cars. There are no perfect self driving cars as of yet, and there are many problems left to be solved before they will occupy our city streets. However, one can imagine the potentially vast benefits of a future more or less perfect self driving car, not least when it comes to safety.

This report aims to evaluate a certain machine learning method in teaching simulated cars how to drive along a track.

2. Theory

This section will describe the idea and theory behind the NEAT algorithm, which is the basis of the project. It will also include a background to the general concepts of neural networks, genetic algorithms and neuroevoloution (NE), all of which the NEAT algorithm builds upon.

2.1 Artificial Neural Networks

Artificial neural networks are inspired by the human brain. Our brains contain a vast network of neurons, or nerve cells. Each neuron takes in electrical signals from multiple others, and interprets these into an output signal which is in turn sent to other neurons. Figure 1 demonstrates the basics of an artificial neural network.

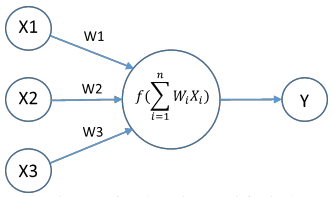

Figure 1. Illustration of a basic artificial neural network.

The network consists of an input-, hidden- and output layer. The input neurons $X1 - X3$ are shown passing data to the hidden neurons affected by weights $w1 - w3$. In each hidden neuron the input is summed up and passed into an activation function f before being output to the output neuron Y. Different activation functions are used for different purposes and they are sometimes used both in the hidden and output layer. A common way of then improving the neural network using supervised learning is to update the weights continuously through a process known as *back-propagation*. However, in NE, the network weights are instead adjusted through genetic algorithms.

2.2 Genetic Algorithms

The basic concept of genetic algorithms is based on the natural process of evolution through natural selection. First, one starts with a population of "individuals", or genomes, possibly represented by neural networks. The population will perform an iteration of their specified task, and based on a fitness criteria, the population will evolve before starting the next iteration. Evolution is carried out through random mutation of individual traits and genetic crossover between different individuals, creating new behaviours.

2.3 Neuroevolution

Neuroevolution is a genetic algorithm where the population genomes are represented by neural networks. The algorithm lets the networks evolve through generations, so that the system converges to the most fit network to handle the desired task. This is the basis of NE algorithms.

2.4 NEAT

The NEAT algorithm was presented by Kenneth O. Stanley and Risto Miikkulainen, described in detail in their 2002 paper *Evolving Neural Networks through Augmenting Topologies*[1]. NEAT is an algorithm that is in the area of Neuroevolution (NE), which is the concept of evolving artificial neural networks using a genetic algorithm.

What sets the NEAT algorithm apart from most NE algorithms is that it allows not only for the weights of the networks to be changed, but the topology of the networks themselves, as the name suggests. This means that hidden nodes and connections between nodes can be added and/or removed. The result will not necessarily have to be a fully connected neural network, like the one shown in figure 1. As with any neuroevolution algorithm, one starts out with a number of individuals, in this case very simple neural networks. They are all the same in the beginning, only containing input- and output nodes, and initiated with random weights. After running the networks one time, the algorithm evaluates each genome according to a custom fitness function. For each time the algorithm is run (each generation) there is a chance that the genome will mutate. Mutation will be explained in detail in section 2.4.2

2.4.1 Network Structure

A NEAT network contains two kinds of nodes, or genes, which together make up the structure of the network. These are *node genes* and *connection genes*[1]. Node genes are all the input-, output- and hidden nodes of the network. They all save a cumulative sum which is used during computation of the network outputs. A connection gene connects two node genes, and also carries a weight, similarly to the weighs of a normal neural network. Because the network topology is dynamic, the computation of the output is not made using matrix to matrix multiplication. The network is traversed in iterations until all the cumulative sums of the node genes have been updated, and an output value is produced. This degrades performance compared to normal NN computations, but allows the possibility for unbounded growth of the networks (within computational limits). Due to starting out with only a basic network with no hidden nodes, the evolution will converge to a solution with the minimal amount of nodes to solve the problem at hand, thus minimizing computational load.

2.4.2 Mutation

There are two kinds of mutations; either a new connection between two existing nodes is added, or a new hidden node is added, see Figure 2.

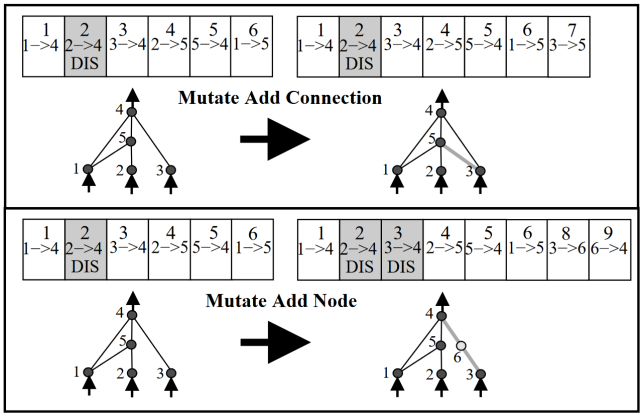

Figure 2. Illustration of the two different possible genome mutations. The top section shows how a connection between nodes can be added, while the bottom one shows how a node can be added between two other ones.

Source: [1]

If a new node is added, the old connection between the two nodes connecting to the new node is disabled, represented by the grey squares in Figure 2. For each connection, there is an innovation number associated, which is the top number of each square representing connections. The innovation number is used to save the history of genome mutations. Each number corresponds to during which generation the genome was attributed this specific node- or connection mutation. This number is important to the crossover step, which is explained in section 2.4.3.

2.4.3 Crossover

Crossover is a procedure which combines the attributes of two successful ”parent” genomes and passes them on to a ”child” genome. In NEAT this is done by first lining up the matching genes (those with the same innovation numbers) of two parent genomes. The genes the parents do not have in common are either *disjoint* or *excess* genes. A gene is disjoint if its innovation number is below the top innovation number of the parents, otherwise its an excess gene. The ”offspring” will inherit these two types of genes from the most fit parent, while the matching genes are inherited randomly from either parent.

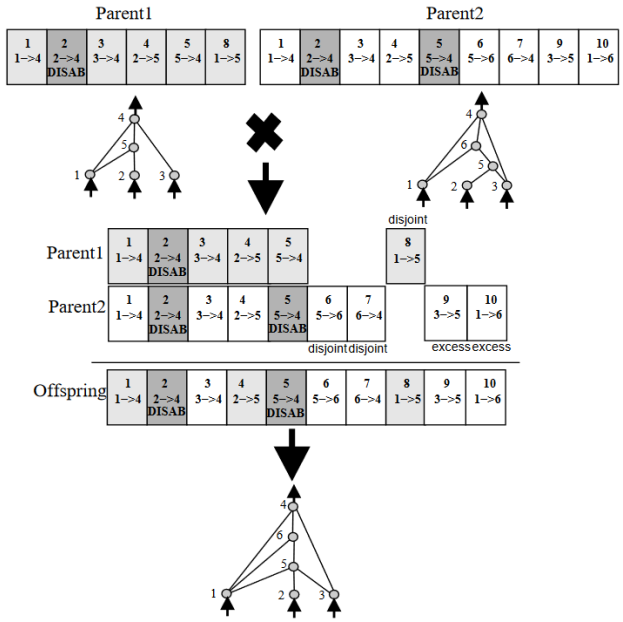

Figure 3. Illustration of the the crossover procedure between two genomes (Parent1 Parent2). The crossover results in a new genome; the *Offspring*.

Source: [1]

2.4.4 Speciation

In order to preserve topological properties to some extent and to allow the longevity of a number of genetic attributes that may become more successful in the future, *speciation* is utilized in NEAT. This means that genomes that are more similar will be categorized as belonging to a distinct ”species”. In order to decide whether a genome belongs to a certain species or whether it should even be categorized into a new species, the *compatibility distance* is measured between the genome and one representative of each current species. Equation 1 shows how the compatibility distance is measured.

$$\sigma = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{1}$$

Here, σ is the compatibility distance, E is the number of excess genes, D the number of disjoint genes and $\overline{W}$ the average weight differences of matching genes. N is the total number on genes in the largest genome and finally c_1, c_2 and c_3 are coefficients used to allow customization of the importance of each of the different factors. The compatibility distance measure in the phase of speciation makes it so that any one species does not become too big. This is important because the NEAT algorithm promotes parallel innovation of different species so that different ways of building a successful genome is tested simultaneously.

3. Method

This chapter will describe in some detail how the NEAT algorithm was implemented in conjunction with the car driving simulation.

The implementation of the NEAT algorithm was written in C# using the cross-platform game engine environment Unity. For the graphics, an open source Unity 3D car driving simulator[2] was imported, and altered to suit the desired simulation. An image of the The basic structure for the implementation of NEAT was inspired by a YouTube tutorial[1], written in Java.

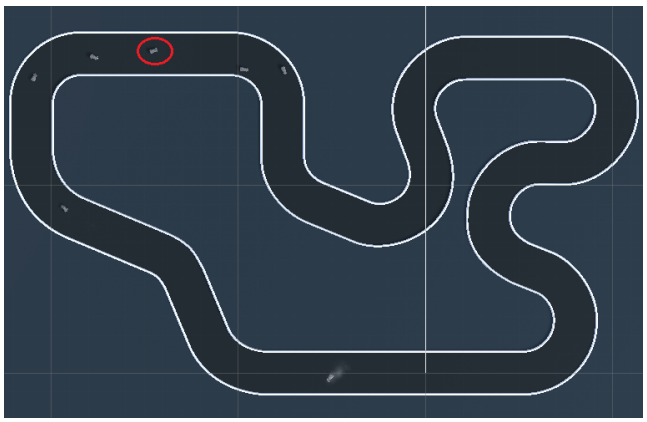

Figure 4. Cars driving on track, one car highlighted in red.

3.1 Program structure

The code provided in the car simulation [2] included all the physics of the car, the throttle and steering process etc. The code was however basically suited for one car controlled by user input. In order to be able to spawn many cars at the same time controlled by the NEAT algorithm, new classes were constructed to more easily handle this task. A class representing each car was implemented, with member attributes such as velocity and steering angle. It also contained the distance from the car and the track verges, which was calculated and updated each frame by casting rays from the car, measuring the distance to the verges. Another class was created to store and keep track of all the cars in the current generation, using

[1] https://www.youtube.com/watch?v=1I1eG-WLLrYt=2s

a C# *dictionary* data structure (similar to hash-maps of other languages).

The functionality of genomes is split into three classes, see Figure 5, where Genome aggregates from ConnectionGene and NodeGene. The Genome class contains all methods necessary to perform mutations and crossovers upon any Genome object.

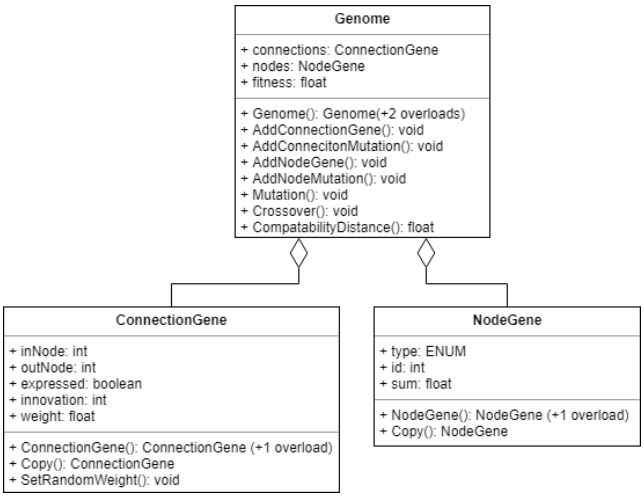

Figure 5. NEAT components class structure.

3.2 Algorithm Walkthrough

The control flow of the main program is described as pseudo code in Algorithm 1 and 2. Here follows a loose description of the process:

At program start a population of cars is initialized. Each car is coupled to a network consisting of input- and output nodes. All networks are identical in both topology and weights at the start (an alternate version was also tested, which used different random weights for all initial networks, a comparison is made in Section 4). The inputs are distances from the car to the track boundaries, measured by ray-casts originating from the front of the car. The outputs are steering angle and throttle input, using a hyperbolic tangent and a logistic activation function, respectively. Another tests was also made, where both outputs used the hyperbolic tangent function.

ALGORITHM 1
NEAT

```
0: procedure NEAT
0:     Initiate N genomes
1: if No cars are active then
1:         Initiate N cars
2: end if
2:     Start CO-ROUTINE to run each 0.1s
```

Every 0.1s, the program reads the distances from each car to the track boundaries and inputs it to the networks. The input is propagated through the network, i.e. multiplied with the connection weights, and summed together to form the outputs. When stepping through each layer of the network (layer numbering is not really applicable here, as the layers are dynamic, but is useful for describing the process), the sum in each node gene is normalized using the *tanh* activation function (produces output in the range $[-1,1]$), to not let the values grow uncontrollably. The output values of the network is then used as input to each car controller.

If a car touches the track boundaries or starts too slowly, it is removed from the simulation and its fitness value is updated. A time limit of 60 seconds for each round was used to speed up the training process. When all cars have been removed, all genomes are evaluated. In the evaluation only the highest fitness genome from each species is preserved unchanged to the next generation. The rest of the population is created from crossovers between previous generation genomes, where high performing genomes have a higher relative chance of being chosen.

ALGORITHM 2
NEAT CO-ROUTINE

```
0: procedure CO-ROUTINE(Cars, Genomes)
1: for all Cars do
1:         Fetch input ray-cast values
2:     for all Inputs do
2:             Multiply input with connection weights
2:             Normalize node values with tanh
3:     end for
4:     for all Hidden nodes do
4:             Multiply node values with connection weights
           until all outputs are calculated
4:             Normalize node values with tanh
5:     end for
5:         Update steering angle and throttle
6:     if car crashed or is too slow or 60 seconds has passed
       then
6:             Destroy car
6:             Update its fitness value
7:     end if
8:     if No cars left then
8:             Evaluate each genome (and prepare the next gen-
           eration of genomes)
8:             Start next generation
9:     end if
10: end for=0
```

3.2.1 Fitness function

The fitness function was chosen to promote cars that can travel as far as possible on the track in the shortest amount of time. Equation 2 shows the fitness function used in the program.

$$f = \frac{d^2}{t} \tag{2}$$

Here, f is the fitness value, d is the total distance traveled by the car, and t is the total time passed since the car spawned.

In order to assure that some cars would not get stuck near the start line, some conditionals were used to destroy the cars if they had not traveled far enough in a certain amount of time. Without this, the algorithm would take much longer to converge to a good result, or simply get stuck; as in some fringe cases the cars barely move at all.

3.3 Activation functions

The choice of final activation function has a large effect on the results, and how quickly a satisfactory solution can be found.

Using a logistic activation function[2] ($tanh$) for the throttle output, the outputs are constricted to the range $[0,1]$. This results in the cars being unable to reverse or use the brakes, as this is done by issuing negative throttle input. But this restriction has benefits if the goal is to find a solution quickly, as the search space for a possible solution is reduced.

Using a hyperbolic tangent ($tanh$) activation function[3] for the throttle output, the outputs are in the range $[-1,1]$. This means the car can break and reverse in addition to throttling forward. This effectively makes the search space much larger, and it takes many more generations to produce good results. But since the cars can use both throttle and brake, this activation function offers the possibility for better solutions.

4. Result

The final results of the simulation was produced by sequentially changing the different parameters of the algorithm, such as the constants c_1, c_2 and c_3 described in section 2.4.4 and testing the simulation. Also by trying different activation functions, and conditionals to improve the algorithm.

4.1 Logistic activation function

Our NEAT implementation was able to produce a network that could lead a car to clear the entire track, even many times over, already in the first or second generation, when spawning 50 cars each run. This car does not drive completely straight or that quick, but subsequent generations would, having a good chance of inheriting attributes from this successful network, finish the track faster and more stable. The limitation to possible throttle input values does speed up the training process, though it also limits the possible performance of the cars, as a car which can use its brakes could possibly complete a lap faster.

4.2 Hyperbolic tangent activation function

Using the hyperbolic tangent activation function offers the possibility for the car to use its breaks (and reverse when standing still). This does increase the number of generations required to produce a good solution, but also makes for better solutions in the end.

[2] https://en.wikipedia.org/wiki/Logistic_function
[3] https://en.wikipedia.org/wiki/Hyperbolic_function

4.3 Genome clones or all random weights

When initializing the genomes, the original implementation only copied one initial genome, which made all genome weights identical in the first generation. This is in hindsight, not good, as it limits the initial starting positions for the search. Using this method will still, eventually, converge towards a working solution. Later tests using random initialization of all genome weights proved to be a much better method, often producing a good solution in $4-5$ generations compared to $20-30$S generations for the genome copy version.

4.4 General observations

The cars generally have a tendency to oscillate left and right over the track. This behaviour may be the result of the update frequency being too low, or the in-build physics of the cars themselves. The cars may require more fine-grained inputs to control in a stable manner.

5. Discussion

The constants used for the final result was as mentioned in chapter 4 was chosen by hand after trying out different values. Not only the constants used in the measuring of compatibility distance but also the mutation rate, the rate of adding connections and nodes. By using this approach, it is difficult to know what other combinations, if any, would be more efficient in producing cars that finish the track in a short amount of time. A program could perhaps be constructed that would try out different values on these constants and compare the results after a certain amount of runs to see which values work best. However, since there are so many different parameters and possible combinations of them, it would take a very long time to thoroughly evaluate these combinations. Also, having changed the values only a few times, it still produced a network satisfying the end result reasonably well, so this would perhaps not be needed depending on the speed and level of perfection needed in the result.

The differences among the species that emerges during the training have not been evaluated due to time constraints. It would be interesting to somehow visualize the different species during the program runs, perhaps give the cars a different color depending on which species they belong to. Also giving a visual representation of the neural networks as they evolve could be a good tool for performance evaluation, to provide some guidance in the adjustment of algorithm parameters.

The choice of a different fitness function could possibly affect the results in another direction. But often the simplest solution is the best solution[4], so other more intricate designs have not been explored.

[4] https://en.wikipedia.org/wiki/Occam%27s_razor

6. Conclusion

It can clearly be said that the NEAT algorithm works well in quickly producing satisfying networks. The possibility of being able to change the the structure of the neural networks seems to have a large impact on producing stable networks quickly, not being bound to static topologies. It can also be concluded that there is a lot more unexplored potential of the NEAT algorithm for this simulation.

References

[1] Kenneth O. Stanley, Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA. 2002.

[2] https://github.com/udacity/self-driving-car-sim. 2017.