# SIMULATION OF CAR CONTROL USING MACHINE LEARNING

Author: Erik Greblo

## Introductions

RaceVsAI is an interactive video game in which the main goal is to explore machine learning and see firsthand how it can be used for numerous implementations.

When playing the game, the player needs to control a car around a racetrack. The main goal is to make one lap around the racetrack in as little time as possible. When the round starts, a timer starts counting down, displaying the remaining time in which the player can try to improve his score. When the time runs up, the best score is saved, and the next stage of the game begins.

Now artificial intelligence has control of the car with the main objective to learn how to control it and how to get a better score than the player. Artificial intelligence uses artificial neural networks to calculate the next best move. The car has three sensors detecting the racetrack borders around and in front of the car. These values are used as input values for the neural network. As output, the neural network gives two values that dictate how the car should be controlled in the next frame. These values are acceleration and steering angle.

Artificial intelligence uses genetic algorithms to learn how to drive the car around the racetrack. It starts with an initial population of cars with randomly generated neural network values and builds a stronger generation each iteration with genetic operators as selection, crossover and mutation.

## 1. Start screen

The first thing that players see when opening the game is the start screen. It consists of four buttons for various actions in the game. Play button starts the game loop and needs to set the highest score possible. The Info button contains general information about the game, and the Options button is for configurating volume levels and full screen mode.

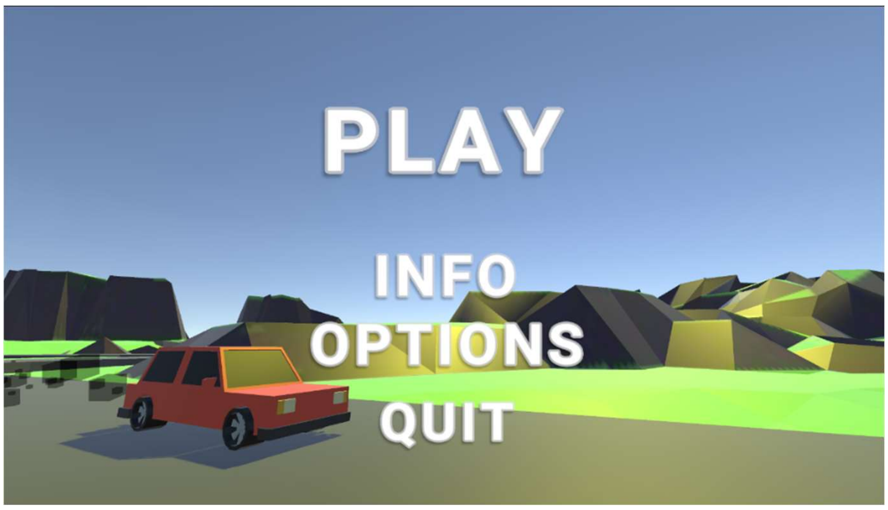
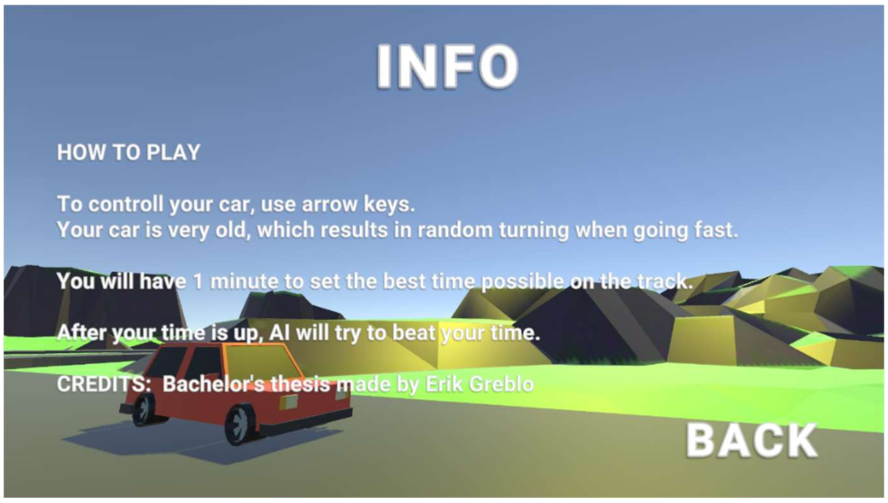


Figure 1 Menu scene

### 1.1 Information's

Info buttons show description of the game, how the game is meant to be played and some information about the creator of the game and project for which is designed.



HOW TO PLAY

To controll your car, use arrow keys.
Your car is very old, which results in random turning when going fast.

You will have 1 minute to set the best time possible on the track.

After your time is up, AI will try to beat your time.

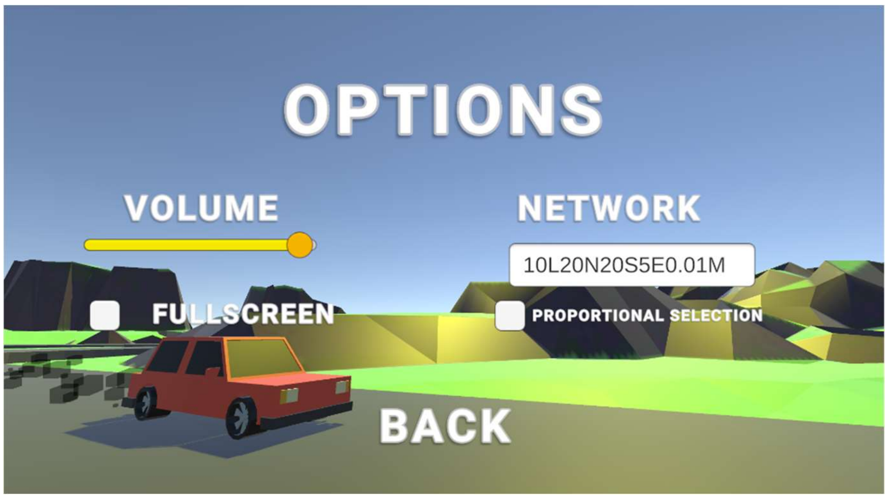
CREDITS: Bachelor's thesis made by Erik Greblo

Figure 2 Info screen

## 1.2 Options

The Options button shows an Option screen that has a slider for volume and a checkbox for full screen mode. The sounds in the game consist of a theme song that plays constantly in the background, an engine sound of the car that is played during the main game loop with cars moving, and a crash sound that is played when a car drives of the road, hitting its hitboxes. The volume slider changes the volume of all said sounds.

On the right side, there is a field for configurating the parameters of neural network and hyperparameters for the genetic algorithm. Changing the parameters will effect how artificial intelligence will learn to control the car.



NETWORK: 10L20N20S5E0.01M

Figure 3 Options screen

## 1.3 Quit

Quit button exits the game.

## 2 Game loop

Pressing the Play button in the main menu starts the game loop. The game loop consists of two stages main stages, and two menu stages.

In the first stage, the player is positioned at the start of the racetrack, the countdown starts and the race for setting the fastest time on the track begins. Using arrow keys on the keyboard, the player needs to drive as far and as fast as possible. Every time the car drives offroad, its score is remembered, the car is positioned at the start, and the player has another chance of setting a better score. This repeats until the countdown reaches zero. That is where the second stage of the game begins.

In the second stage, the player watches how artificial intelligence tries to learn to drive the car from zero, with a goal of beating the player's best score. Artificial intelligence controls the car using neural networks. Its inputs come from three sensors in front of the car, which supply artificial intelligence with information about its environment. Output of its neural network are two decimal values: one for acceleration and one for turning. With this structure, artificial intelligence can "see" the racetrack using its sensors, and according to that information, it can speed up, slow down, turn left or right, depending on the values of the weights in its neural network. In the beginning of this phase, a group of independent cars spawn at the start line, each with a different, random neural network controlling its movement. They start racing at the same time, and when all cars crash, the next batch of cars are spawned, but this time with modified neural networks, generated by a genetic algorithm. Genetic algorithm sorts the current batch of crashed cars by their distance traveled and time taken, and then does multiple operations, with a goal of creating a better performing batch of cars for the next round. The process repeats until the genetic algorithm reaches a certain generation number (25).

Now that both the player and the artificial intelligence have set their best score, a score menu appears, showing the results and declaring a winner. Player can now exit the game or try to beat his personal best score.

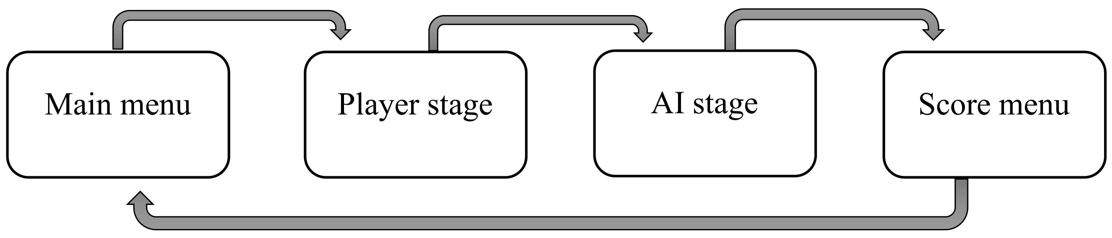


Figure 4 Game loop

### 2.1 Player Time Trail

With the game loop activated, the first phase begins. In this phase, the player is in control of a car positioned at the start of a racetrack. The player's objective is to reach the finish line as fast as possible to set the highest score, so that the artificial intelligence has a harder time trying to beat it in the later phase.

To control the car, the player can use arrow keys on keyboard [2]. The up and down keys accelerate and decelerate the car and left and right arrow keys steer the car in that direction. On the screen there are indicators for speed and turning values, giving information to the player about the current state of the car.

### 2.1.1 Unexpected steering

As it turns out, humans have a great initial understanding of controlling a virtual car using only a few buttons, so to make the game more challenging and intense, the car experiences unexpected steering. This simulates the difficulty of controlling a vehicle at high speed, giving the player an option to have full control over the car at lower speeds, sacrificing time, or have considerably less control at high speeds, but reducing its time to the finish line.

Implementation of unexpected steering is done with few steps:

1. Generate a random value between set boundaries
2. Calculate unexpected steering multiplying acceleration value to linear interpolation of generated random turn value and current target steering angle
3. Add newly calculated steering angle to the user inputted angle

$$maxSteering \ = 0.6$$
$$randomSteering = [-maxSteering \ , maxSteering \ ]$$
$$input = [-1{,}1]$$
$$randomSteering = Lerp(randomSteering, targetSteering, lerpTime) * acceleration$$
$$lerp = (1 - lerpTime) * randomSteering + lerpTime * targetSteering$$
$$steering = input + randomSteering$$

Figure 5 Calculating unexpected steering

### 2.1.2 Car Movement

After calculating the value of unexpected steering, we have all of the information needed to position the car to the corresponding position and orientation in the scene. For the position, linear interpolation is used to calculate the end position of the car in the current frame, with some level of smoothing so that the movement looks more natural. This newly calculated value is added to the car's position component, which adds the velocity to the current position and moves the car.

Furthermore, to rotate the car to the appropriate rotation, a new vector with its y component set at the turning value and some constant, is added to the cars Euler angles component, which in turn rotates the car to its current rotation angle.

This calculation is repeated after every frame, moving the car across the racetrack.

### 2.1.3 Collision detection

If the car tries to go off the racetrack, its controls instantly stop working, the player now can't move the car, the crash sound plays and the car is repositioned at the starting line, with its score back to zero.

To achieve this effect, both the car and the racetrack need an invisible hitbox, and every frame the game runs a calculation to see if the respective hitboxes are colliding.

A hitbox is an invisible box, or a series of boxes, which follow a rough shape of an object, with the purpose of collision detection [3]. Complex objects in the game scene could have many polygons defining their shape, which would in turn make collision detection an expensive task performing every frame of the game. Hitboxes make that complex and computer intensive process easier and faster, but in turn reduces the accuracy of collision detection. This error is mostly not large enough to affect the gameplay, which is why hitboxes are so widely used when detecting collisions.

The model of the car is made of a small number of polygons, given its style name being "low poly" [4], but adding hitboxes to it also effects performance in a positive way. It has only one hitbox, which is precisely positioned to encapsulate all vertices in as tight of a space possible. This is a trivial problem, being that the car is made from one large rectangle. This in return gives an accurate representation of the object, but now for collision detection, the game doesn't need to check all of object's vertices, only one rectangular hitbox.

The model of the racetrack is more complex. Car needs to freely ride on the racetrack, without any collision detection interruption. Collision detection needs to detect when the car exits the racetrack. Surrounding the whole track, there are invisible walls, hitboxes, made with extrusion of the racetrack model, and lifting the edges to be higher than the car, so it cannot leave the track without being detected and repositioned at the start.

Racetrack is designed such that the start and end position are the same, completing the loop. But, in this scenario, the game only allows the car to make one lap, after which it adds few points to the final score as a reward and resets the car to its starting position with score at zero, ready to set a new personal best score. This is also done using collision detection and an invisible hitbox placed at the end of the lap.

## 3 Artificial intelligence phase

After the players' time has run out, a new phase is initiated, in which artificial intelligence learns to drive the car with the goal of beating the players' best score in the set number of generations.

One of the easiest and most intuitive ways to make artificial intelligence learn to perform a task is using State Space Search [5]. This is a way of representing all the possible states in which the game can be in. With defined start and end states, a minimax algorithm can be performed to search through the game tree and find the optimal input combinations by minimizing the possible loss in the worst-case scenario. This approach is used only in games with perfect information like chess or tic-tac-toe [6].

State space search is not appropriate for this use case because of the complexity of the game, there are many possible states in which the game can be in in a certain time. This makes gathering all possible states next to impossible. Unexpected steering described in earlier chapter could also cause problems using this approach, but it is implemented such that it is activated only when the player controls the car, so the game for artificial intelligence has in fact perfect information.

After crossing out the option of representing all the possible states of the game, next option is to construct a model that represents the structure of the human brain and its biological neural networks, which make decisions according to what the person sees, touches, feels and so on. This model is called an Artificial Neural Network [8].

## 3.1 Artificial Neural Network

Biological neural networks are made up from an absurdly large number of neurons, an electrically excitable cell that fires electric signals. The brain of an adult human contains approximately 86 billion neurons [7], with each one connected to many other neurons. With this complex network of interconnected neurons, every input from all the senses gets processed and we as humans get the result of that calculation, an action that we want to perform.

To simulate the actions of the biological neural networks, each part of the network can be simplified and connected to construct an artificial neural network, capable of performing complex calculations that methods such as state space search wouldn't be able to do [8].

Artificial neurons are a representation of the biological neuron. It takes a vector of input values and returns an output value. Let X be the input vector, and $x_1$, $x_2$, ..., $x_n$ each value of that vector. Each value of the input vector is connected to the neuron with a weight value. This value represents the magnitude of each input element. If the weight of a connection is set to 1, that input value strongly corresponds to the output of the neuron. On the other hand, if the weight of the connection is set to 0, the value of the output doesn't rely on the input value.

Mathematically, this is represented with a summation function that adds all products of input value and weight value multiplication. Each neuron also has a bias value that is added to the sum no matter the input. Completed equation looks like this:

$$y = \sum_{i=1}^{n} x_i * w_i + \text{bias}$$

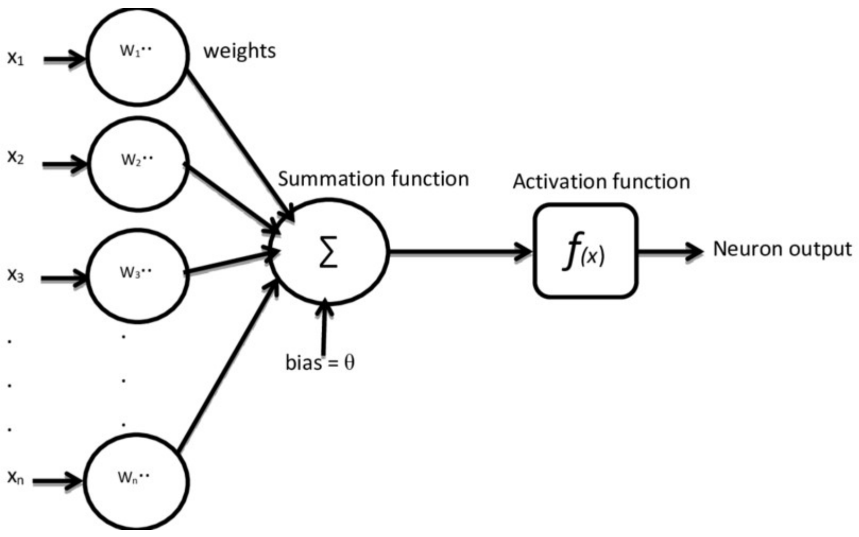


Figure 6 Artificial neuron model [9]

After putting the input vector through the summation function, output value y needs to be passed through an activation function.

Activation function is a function that calculates the output of the neuron based on the output of the summation function [10]. There are many activation functions, each with different use cases and properties. They can be divided into three categories: ridge functions, radial functions and fold functions.

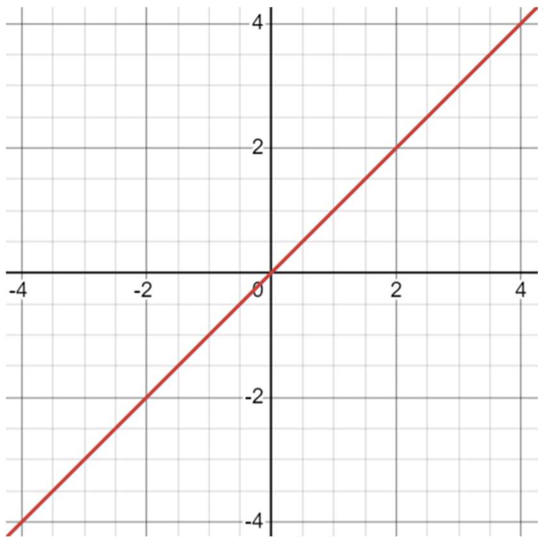


Figure 7 Identity function

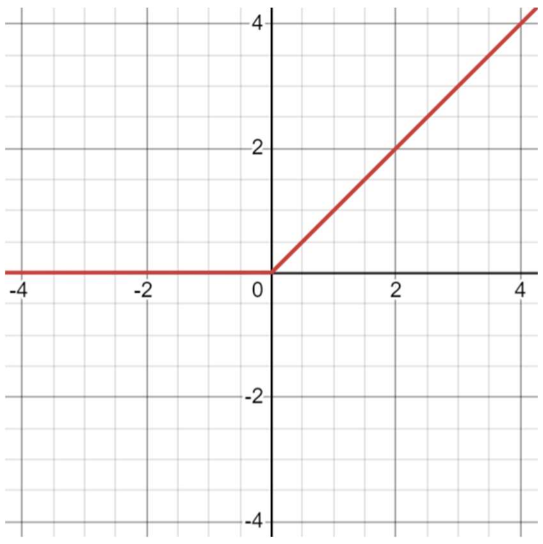


Figure 8 Binary step function

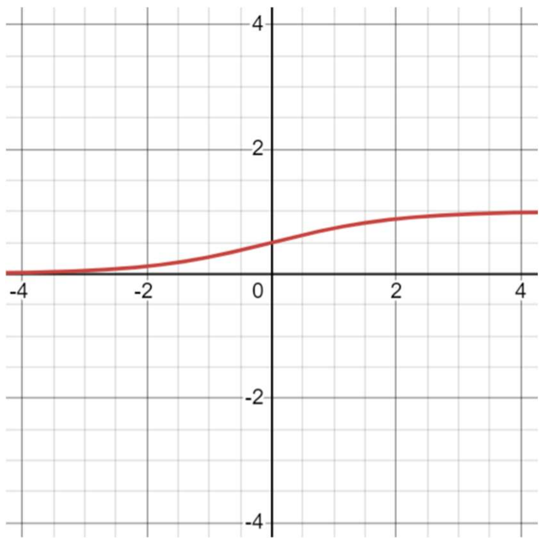


Figure 9 Sigmoid function

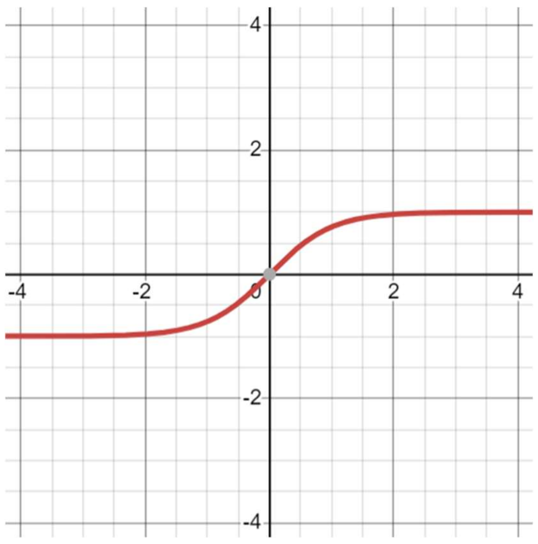


Figure 10 Hyperbolic tangent function

Now that the basic neuron is defined, by connecting multiple neurons together a large artificial neural network can be constructed.

Artificial neural networks are made from separate layers of neurons. The first layer is called the input layer, and it represents plain input vector, without any modifications to it. The last layer is called the output layer, and its output is the output of the entire neural network. Every layer in between is called the hidden layers. They are hidden because neural networks work like black boxes, meaning that given the input, it is difficult to determine the output of the network without computing every calculation for every neuron.

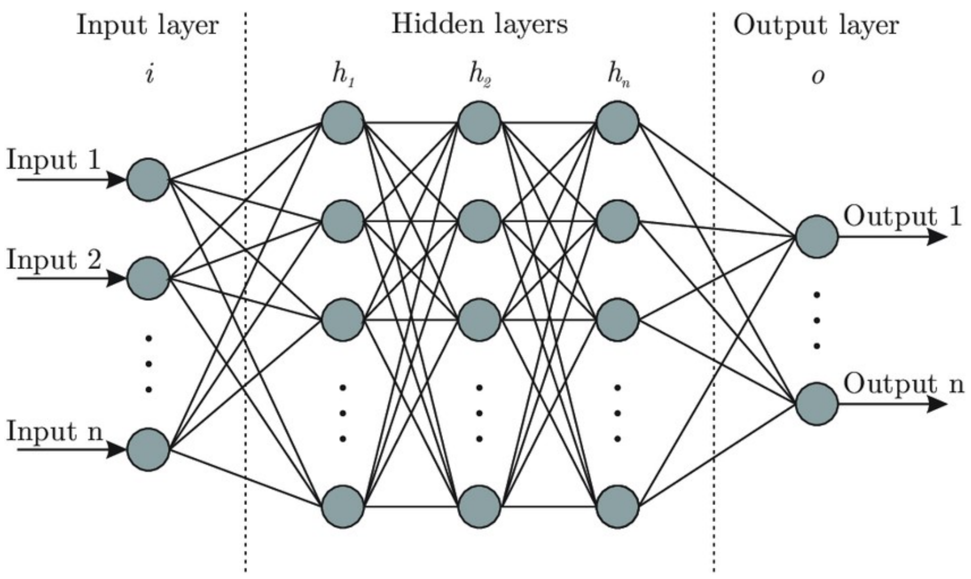


Figure 11 Artificial neural network model [11]

To calculate the output of a neural network. Each neuron needs to compute its output equation layer by layer, and after computing all layers, the output is calculated. To simplify the equation, weight values are in the form of a matrix $W_i$ , with each row representing weight values from neurons at the previous layer. Furthermore, bias values and layer output values are in the form of a vector. This simplification allows for easy calculations with matrices and vectors. Now, neural network needs to calculate output of each layer one by one, starting at the first hidden layer, labeled as $h_1$. This calculation is the same as calculating each neuron output one by one and putting the result into a vector. For hidden layers, output values are passed through the activation function. This doesn't apply to the last layer, the output layer. [12]

$$\vec{h_1} = f(W_1 * \vec{x} + \vec{b_1})$$

Generalization of this equation for the rest of the hidden layers:

$$\vec{h_i} = f(W_i * \overrightarrow{h_{\iota-1}} + \vec{b_i})$$

Output is calculated as such:

$$\vec{y} = W_i * \overrightarrow{h_{\iota-1}} + \vec{b_i}$$

With these equations in place, any inputted vector into the neural network will result in an output vector. This model is perfect to make artificial intelligence control the car.

To control the car, artificial intelligence has a limited amount of information, compared to the player playing the game. Players gather a lot of information from seeing every pixel on the screen and intuitively decide what action to take. In theory, the input of the neural network could be a vector of all the pixels on the screen, and then the artificial intelligence would have as much information about the current state of the game as would the player. But this would be extremely expensive and time-consuming for a regular computer, given that the neural network would have to do the calculation in every frame.

Instead, the car has 3 sensors that point to the left, to the center and to the right of the car, giving the artificial intelligence enough information to understand where the road hitboxes are and calculate its next move. A sensor is placed on the car, and it returns the distance from the car to the nearest hit point with the racetrack hitbox on the line in one of 3 directions. The goal is that artificial intelligence could make a good move with only a limited amount of information, instead of having to have every pixel of the screen as input.

The output of the artificial neural network is a vector with two values. The first value represents the acceleration, and the second represents the steering angle.

When the network is first created, all the values need to be initialized. This is implemented by iterating trough every layer, and generating a random value between -1 and 1, which is set to a value in the weight matrix of the layer. This process repeats through all values of the weights and all the bias vector values, each time generating a new random number.

With every layer initialized, the neural network is ready to take in the input from three car sensors and give an output about the acceleration value and steering direction. A car controlled by this neural network is going to move seemingly at random. It's choices of acceleration and steering angle pair is not going to lead to good scores and most of the time, the car will crash after only a few meters travelled. This is happening because now, the network is filled with random values. It did not modify its values according to the track. It does not know what to do better or how to do it. To start performing better, artificial intelligence needs to learn from its mistakes, modify its internal values of the weights and biases in a way that it's final score is higher than it was previous iteration. This process is called machine learning [13].

## 3.2 Machine Learning Phase

Machine learning is a process in which artificial intelligence learns to perform a given task better than the last iteration [13]. It learns from data and from past experiences, modifying the values in its artificial neural networks, with the result being a better final score.

There are different methods of machine learning, each with their advantages and disadvantages, and compatible with different tasks. Three main types of machine learning are:

1. Supervised Machine Learning
2. Unsupervised Machine Learning
3. Reinforced Machine Learning

Supervised Machine Learning is based on learning with a dataset that has pairs of input and output values [13]. Output values are called "labels", and the algorithm modifies networks' internal values to decrease networks' error on the training dataset. Each data pair is sent through the network, and the output is compared to the label of the current data pair, and the values are modified to produce a better network. To see the networks' accuracy, another dataset is used. A new set of test values is given to the model, and after going through every testing data pair, an error value is calculated to determine the accuracy of the network. This type of machine learning is perfect for problems such as object classification or spam detection.

Unsupervised Machine Learning is a type of machine learning that uses an unlabeled training dataset to train [13]. The main goal of unsupervised learning is to find hidden patterns in the dataset, and group categorize them according to those patterns, similarities and differences. This method is used for applications as identifying plagiarism and copyright and recommendation systems.

Reinforced Machine Learning works on finding the optimal strategy based on a feedback-based approach. The artificial intelligence agent acts, and gets rewarded if the action is good,

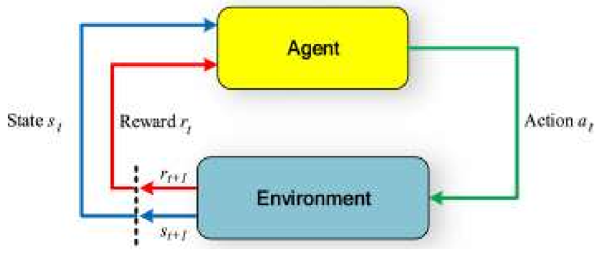


Figure 12 Reinforced learning model [14]

or gets punished if it made a bad action. The goal is to maximize rewards. As in Unsupervised Learning, given training dataset is not labeled, the agent learns only by its' past experiences.

In order for this simulation to use supervised machine learning, a training dataset would need to be constructed, with every input value of the three sensors labeled with a correct output of acceleration and steering angle. This approach is not ideal because sensor output values are float numbers, so the dataset would have to consist of a large number of input/output pairs. Furthermore, labels for each pair would have to be hand-selected, meaning that every input would have to be classified with the best action to take in that position, which is a huge task. Out of the other types of machine learning, reinforced learning is the best fit for this problem, but it is still not ideal. To make the agents learn in groups and then pick the best ones and fill in the rest with new agents with traits of the best agents, the best option is the subset of machine learning, called Evolutionary Computation [14].

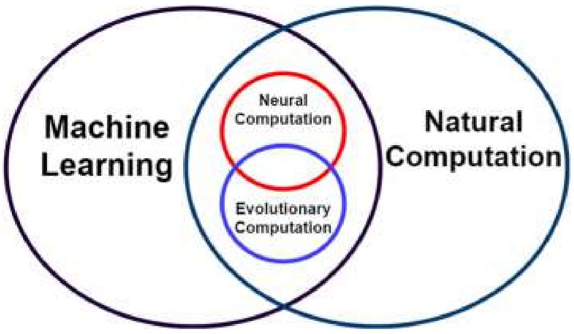


Figure 13 The intersection of natural and evolutionary computation in the context of machine learning and natural computation [14]

Evolutionary Computation are methods of problem solving inspired by the principles of natural evolution. Starting with the initial population of several agents, each agent runs its' neural network, takes action, and at the end outputs its' fitness value, a score indicating the success of the network. After every agent finishes its' calculation, using genetic operators such as crossover, selection and mutation, a new population is made. Newer generations have better performance results because they are based on the best agents from previous generations.

There are several different approaches of evolution computation. One of the most popular population-based algorithms are Ant Colony Optimization, Particle Swarm Optimization, and Genetic Algorithm. In this simulation, genetic algorithm is used to make artificial intelligence learn to control a car in a simulation.

## 3.3 Genetic Algorithm

Genetic algorithm is a heuristic search population-based algorithm that simulates natural evolution, where only the fittest individuals are selected to produce the next generation [12]. At the start, the initial generation of agents with neural networks are created. Then, in this simulation, every agent simultaneously starts controlling the car. After every agent receives its' fitness value, a new generation starts to form. This process happens in three steps: selection, crossover and mutation.

Selection is a process of selecting a predefined number of the agents with the highest fitness values. These agents had the best performance out of the whole generation. For this process, the generation needs to be sorted according to the fitness values of every agent. The best agents are then copied into a new generation.

After selecting the best performing agents, the next process into creating a new generation is Crossover. Crossover uses agents in the current generation to create new agents to put in the new generation. Firstly, the method picks two agents from the current generation as parents. There are a lot of different methods of picking the parents, some work by returning a random agent from the current generation with all agents having the same chance of being picked, and another work by having the better performing agents have a bigger chance of being picked as parents, called “proportional selection”. With parent agents picked, a new child neural network is initialized. Again, crossover can be implemented in many ways, but one of the more intuitive ones is by going through all of the layers of the child network and setting its weights and biases to the average of the first and the second parent. After a child is crossed with parents, it is added to the new generation, and the process is repeated until the new generation reaches the initial size of the generations.

Now that the new generation is filled with the top performing agents and the children of agents in the previous generations, the next step is a process called Mutation. The simulation has a mutation probability value, that determines if the neural network will be mutated or not. Iterating trough every layer, a random value between 0 and 1 is generated. If that value is lower than the mutation probability, the weight matrix mutates. Same goes for bias vector, a new value is generated and if it is lower than the mutation probability, it gets mutated. Mutation can be implemented in many ways. In the simulation, it is implemented by generating a random value between -0.01 and 0.01 for every element in the matrix or the vector. This value is then added to the current element, and the resulting value is clamped between -1 and 1, to make sure that the weights and biases stay in appropriate range.

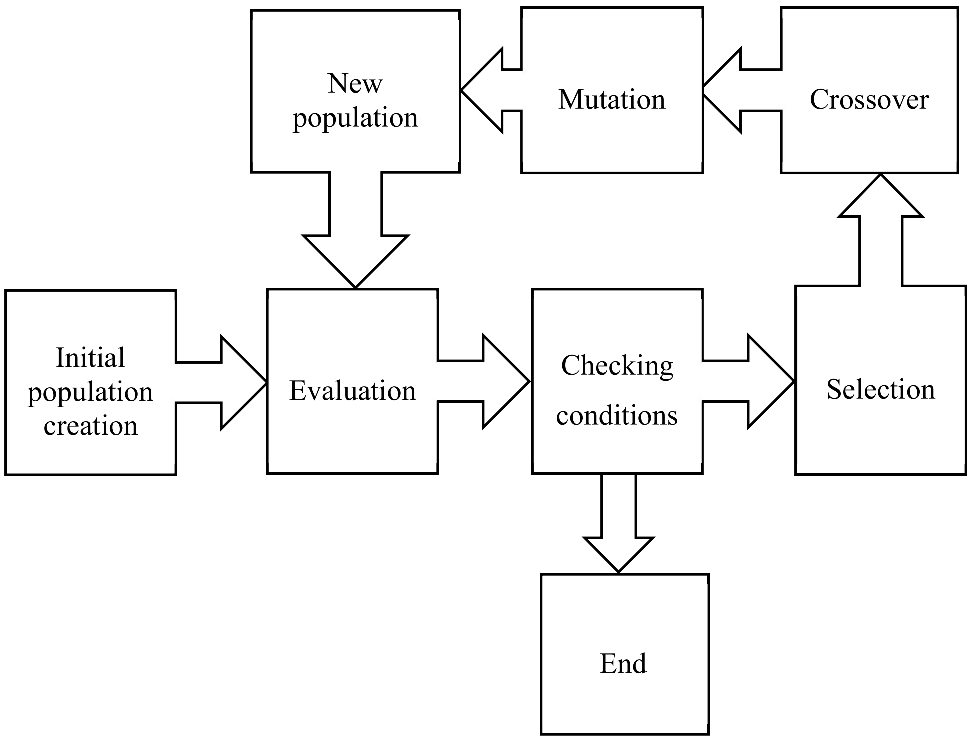


Figure 14 Genetic algorithm model [12]

Genetic algorithm has conditions to determine if it should continue to learn and create new populations, or if the current population meets the requirements. The condition that is implemented in the simulation is a timer that starts counting down at the start of the evaluation phase, where each agent is simultaneously controlling its car and getting final fitness values. After the timer runs out, all of the agents in the current generation get destroyed and the overall best score is the score that the artificial intelligence managed to achieve.

## 3.4 Simulation

With the implementation of the genetic algorithm, the simulation is ready to start understanding how to control the car and how to set the best score. When the learning phase starts, the population is filled with agents that use random neural networks. Each of these agents controls a separate car. They all start from the same starting position and learn at the same time. After every frame, each agent runs its input from the three car sensors into its neural network to determine the next move. After the calculation, the car is moved according to the acceleration and steering values.

After each frame, every agent calculates its fitness values according to the total distance that the agent has travelled and the average speed that it had while moving. This way of calculating overall fitness for agents prioritizes faster agents that travelled further on the racetrack.

$$\begin{aligned} overallFitnes = {} & totalDistanceTravelled * distanceMultiplier \\ & + avgSpeed * avgSpeedMultiplier \end{aligned}$$

Once all the agents calculated their next move and the corresponding fitness values, the simulation finds the best performing agent and changes its color to green, to have a better understanding of the current best agent in the generation. The simulation can be observed from a first person perspective and with a top-down view, while following the current best performing agent. To better improve the simulation understanding, at every hit point of the three sensors of the best car, there are red spheres to further visualize the data that the agent takes trough its neural network.

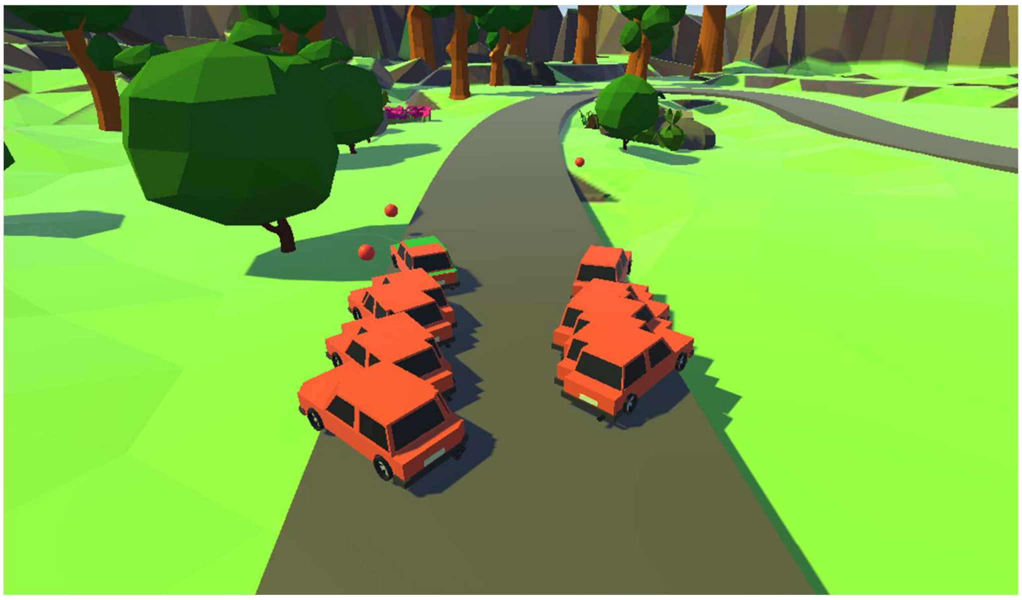


Figure 15 Initial population of agents

## 4 Results

Artificial neural networks and genetic algorithms both have a number of different variables used for their initialization and for further calculations. When looking for the best configuration of the simulation, every variable can be changed to see the resulting generations and the best achieved fitness values. To visualize the data from different configurations and to conclude what is the best configuration for the simulation, the simulation needs to save useful data for further analysis.

After starting the simulation with the wanted configuration, every time that the last agent in the population crashes, the average fitness of the population is stored in a text tile that is used for data analysis. The text file is made from rows, and each row corresponds to a population, from the first population to the last. If the same configuration is used to run the simulation, row with the corresponding generation number gest appended with the current average fitness values. After running the simulation with the same configuration's multiple times, large quantity of data is stored, which can be visually represented with a graph that has the x axis representing the generation number, and the y axis representing the average values of average fitness values for each generation.

To configure the simulation, these are all of the values that can be modified to achieve the preferred balance between accuracy and computational cost:

- neural network hidden layer count
- neural network neuron count
- population size
- mutation rate
- elitism

Changing any of the listed values will result in different outcomes of the simulation. In general, increasing the neural networks layer and neuron count would improve the simulations accuracy and final score, but would in turn increase the computation cost of running the simulation. Same goes for the population size. However, increasing the mutation rate and elitism values would not necessarily increase the computational cost. They are responsible for the genetic algorithm and would affect the accuracy of the model.

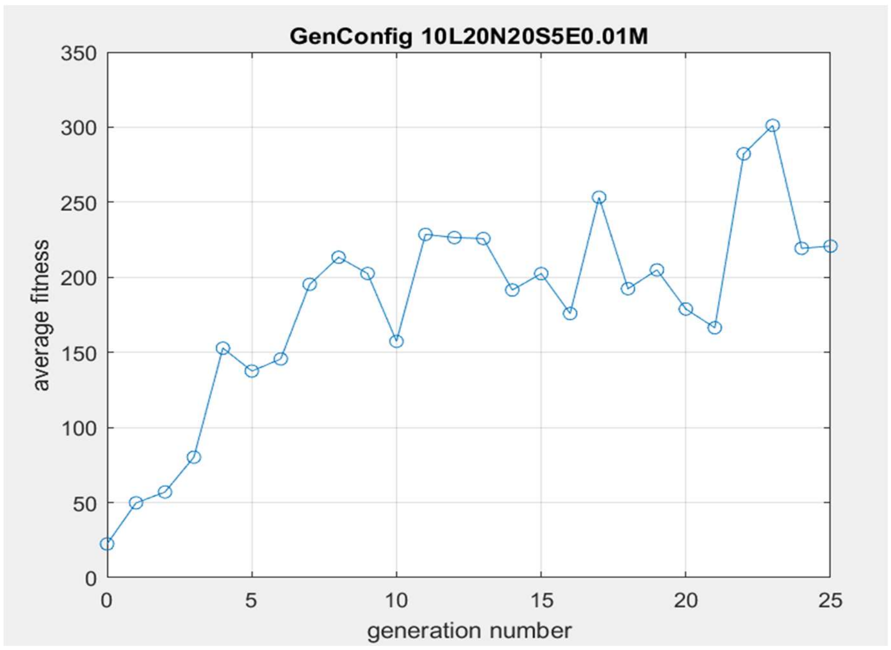


Figure 16 Final result of configuration 10L20N20S5E0.01M

To visualize the data, the text file is connected to a MATLAB program which iterates trough every row, sets the first value as the x position in the graph, and the rest of the values are used to calculate the average fitness of all generation in the file.

The configuration is represented with a name giving the information about every value of the configuration. To represent the configuration that has neural network layer count set at 10, neuron count at 20, population size at 20, elitism value at 5 and mutation value at 0.01%, the name could be "*GenConfig 10L20N20S5E0.01M*", as seen at the graph above.

## 4.1 Configuration comparison

There is no mathematical way to determine the best combination of these five parameters in this simulation. To find a sufficient configuration, the simulation needs to be ran with different parameter values, observing the output and determining the best configuration.

To visualize how each parameter changes the result of the simulation, its result is compared to the result of another simulation result which had the same configuration except that one parameter that is being tested to see its effect to the overall accuracy of the artificial intelligence.

### 4.1.1 Neural network hidden layer count

Firstly, how does the number of hidden layers in a neural network affect the outcome of the simulation? A small number of hidden layers in the neural network is sufficient if the given problem has a linear or relatively simple outcome. For example, if artificial intelligence is used to determine if an input number is positive or negative, the network wouldn't need a large number of hidden layers because of the simplicity of the problem. But the problem of controlling a car in this simulation cannot be solved without hidden layers because the calculation of acceleration and steering is not linear.

This would imply that the higher the hidden layer count is, the better the outcome of the simulation would be. This can be tested by running the simulation twice, once with a small number of hidden layers, and once where the hidden layer count is high.

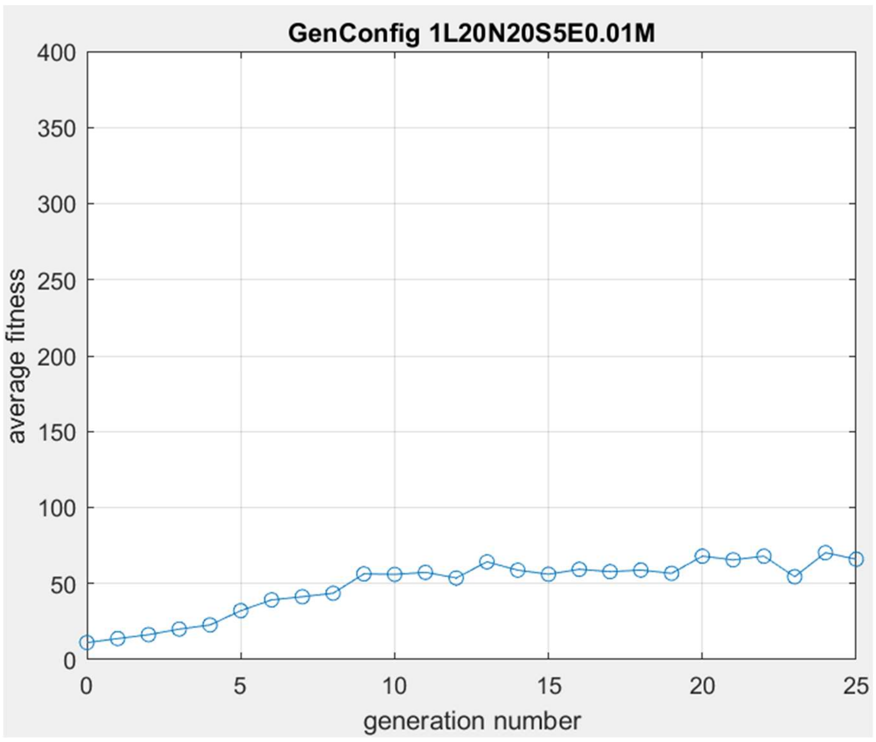


Figure 17 Final result of configuration 1L20N20S5E0.01M

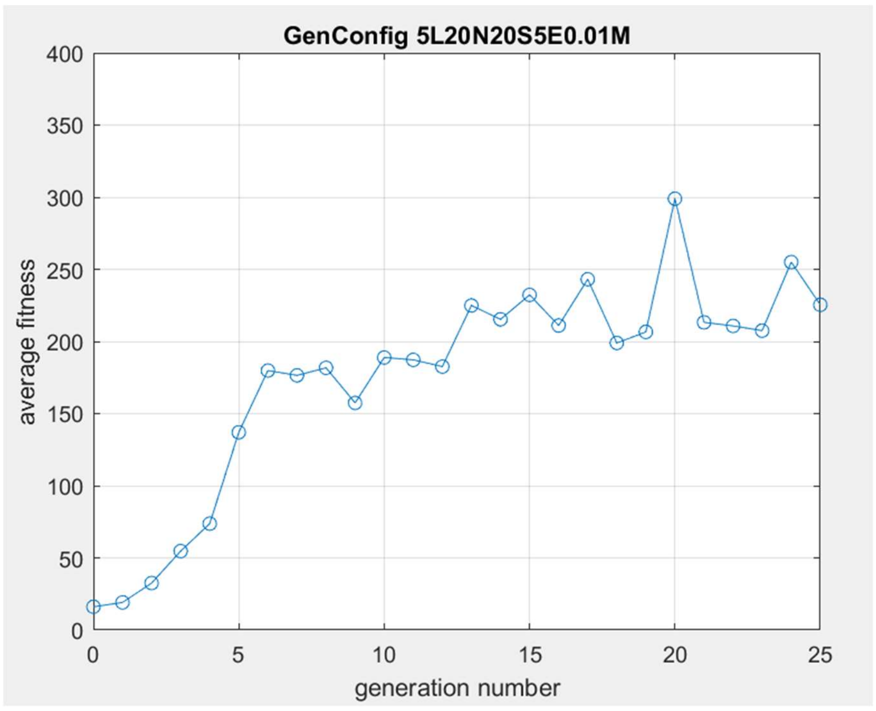


Figure 18 Final result of configuration 5L20N20S5E0.01M

After running the simulations and visualizing the results, it can be determined that the network with a higher number of hidden layers performed significantly better that the one with only one hidden layer.

### 4.1.2 Neural network neuron count

The last parameter of neural networks is its neuron count. This determines how many neurons each of the hidden layers have. There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers [14], such as:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Number of neurons in input and output layers are set by the simulation needs and cannot be modified. There are three neurons in the input layer that represent each of the sensor values, and two neurons in the output layer, one representing acceleration, and the other representing steering angle. With these two numbers in mind, following the rules, the ideal neuron count should be somewhere between 3 and 5.

It is to be expected that if the neuron count in hidden layers is too low, the network won't be able to calculate the right output values correctly. The higher the count is, the more accurate the network should be. This can be visualized with running a simulation with the configuration that has a low neuron count, set at 2 neurons, and another simulation with the same configuration except the neuron count, which is set at a higher number, let’s say 5. The results are the following:

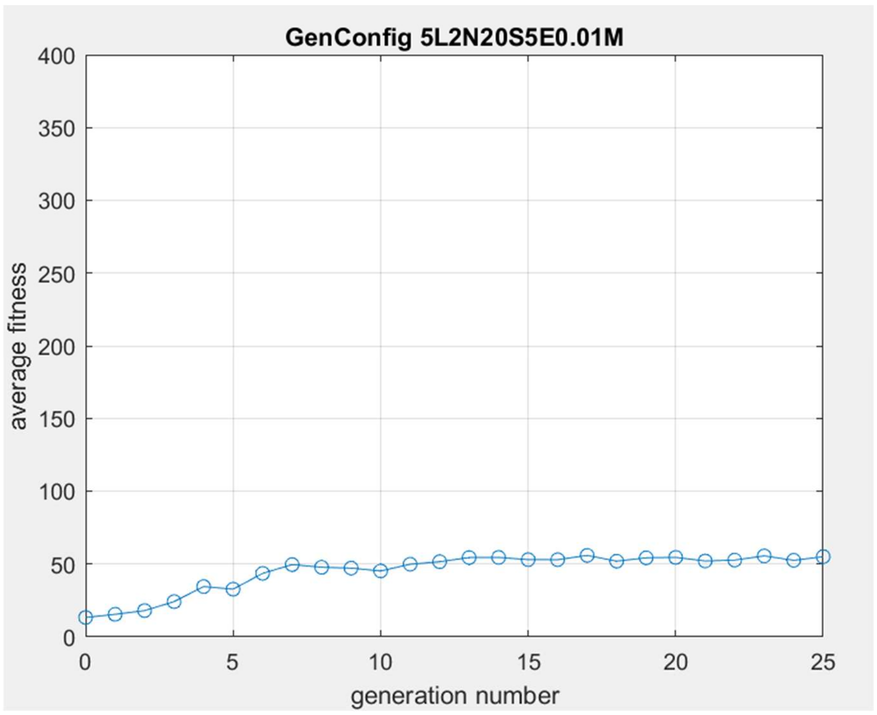


Figure 20 Final result of configuration 5L2N20S5E0.01M

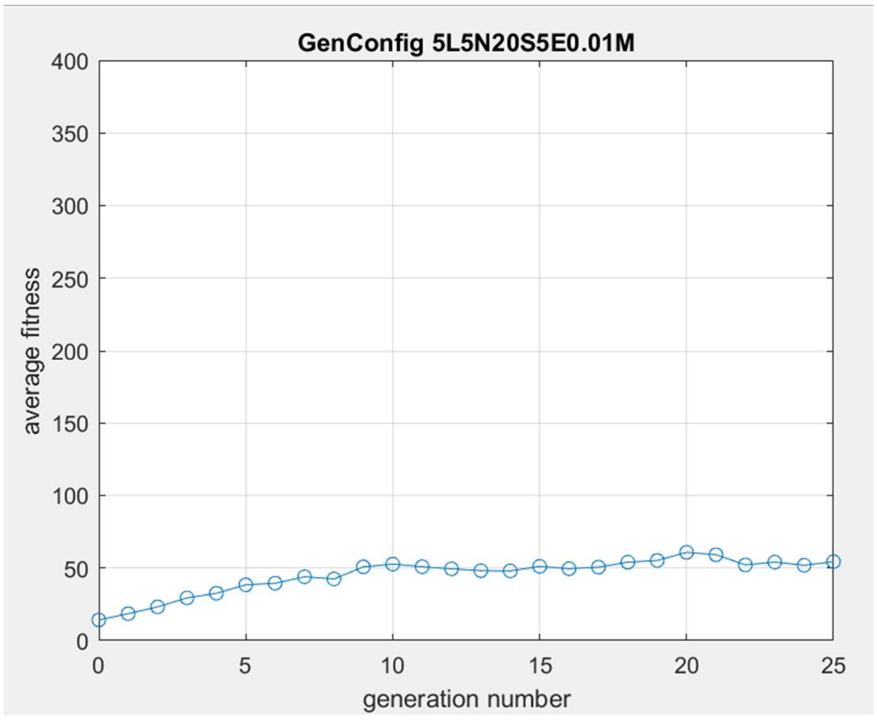


Figure 19 Final result of configuration 5L5N20S5E0.01M

Results of both simulations are very similar, meaning that the problem of controlling the car requires more neurons in each hidden layer to compute the correct calculations. Furthermore, the racetrack is designed to have a sharp right turn at the start of the race. This is a large reason why many simulations with relatively low number of hidden layers and neurons reach a score around 50, because at that distance from the start a sharp right turn is placed and most of the agents fail to complete the turn without crashing.

Further increase of neuron count in hidden layers will improve performance of agents and will be able to drive around sharp turns on the racetrack. Let's run the simulation with the number of neurons set at 10:

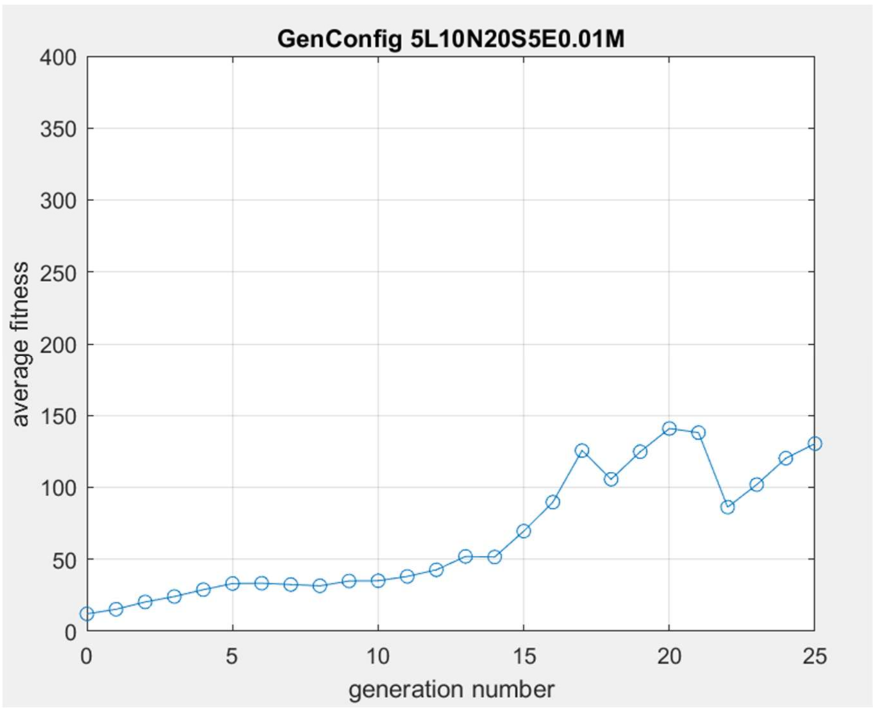


Figure 21 Final result of configuration 5L10N20S5E0.01M

With genetic algorithms, sometimes even the best configurations could produce an underwhelming result because the initial population generation determines how the next population is going to be created. If the initial population has a bad performance, there is a chance that none of the genetic operators (selection, crossover and mutation) will optimize the next population result, because the operators rely on randomness to find agents to crossover, to determine if the layer should be mutated and so on. This is the reason why all simulations with the same generations need to be ran multiple times and find the average of all of them, to make sure the result is stable and not luck based.

### 4.1.3 Population size

To further optimize the simulation, after finding a solid configuration of neural network parameters, genetic algorithm parameters can also be tested to find the best configuration. Population size is the main parameter that determines how many agents does the population have. If the number is high, initially there will be a large population of randomly generated agents that will start learning. A bigger population means that there is a bigger chance of one of the agents performing exceptionally well, and those agents will be selected for the next population. When performing crossover and mutation, there is a bigger chance that the newly created agents gain good traits, because these operations are based on random numbers. The larger the population size is, the larger the chance of producing a great agent is.

To test this hypothesis, simulation will be run with a small population size (10), and with a large population size (30):

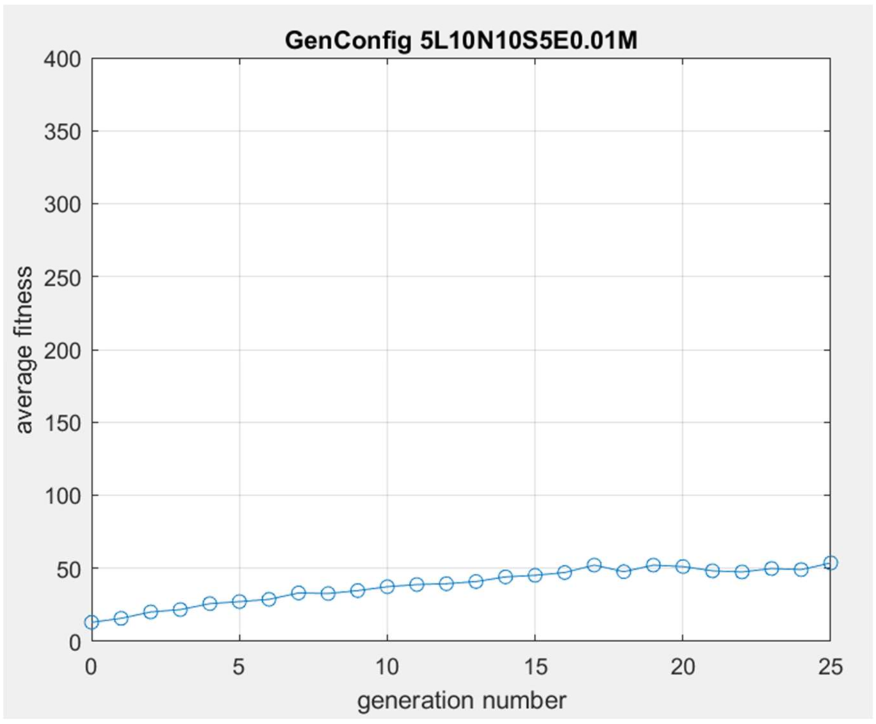


Figure 22 Final result of configuration 5L10N10S5E0.01M

With a small population size, the result is as expected. Each new generation of agents perform slightly better than the last generation, but the learning is happening in small increments. If the simulation is not stopped at 25th generation, every other generation would still be improving, and eventually the agents would have a really good performance. This small of a population size slows the learning process, and needs to be increased. But there is a fine line between a good population size and an excessive population size, one that makes running the simulation too computationally expensive. This is why it is important to test different configurations through trial-and-error.

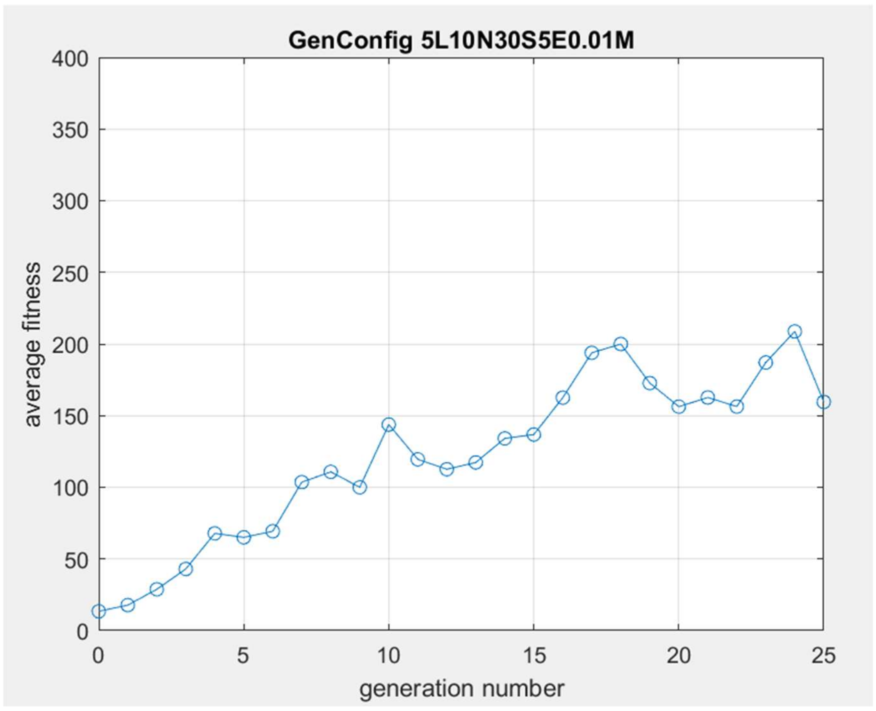


Figure 23 Final result of configuration 5L10N30S5E0.01M

### 4.1.4 Mutation rate

Mutation rate determines how often weight and bias values get modified. The goal of mutation operator is to change the values just enough to make the agent perform better in the next generation.

If the mutation rate is close to 1, mutation is going to happen almost all of the time, and if the mutation rate is close to 0, weight and bias values are rarely going to be modified. Neither of these extremes are ideal for this simulation. If the mutation operation never happens, the agents won't be able to modify their values from the initial initialization, except for swapping the values in the crossover operation. If the mutation happens to every single value every single time, this will add too much chaos into the generations, and it will make any beneficial traits of the agent be lost, which will in turn result in less optimal agents. To get the best performance, mutation should be a low number, but not zero. This way the generations can inherit beneficial traits of previous agents, and at times modify the values to try and optimize the solution.

To test this hypothesis, the simulation will be run with the same configuration, except one will mutation rate set at a small value (0.005), and another will have the rate set at a high value (0.8).

From this results, high mutation rate doesn't seem to affect the overall performance of the agenst like expected. The simulations are run only a handful of times to get the approximate results, but there is little to no difference between these two configurations. A configuration with the high mutation rate still finds good traits and propagates them to next generations.

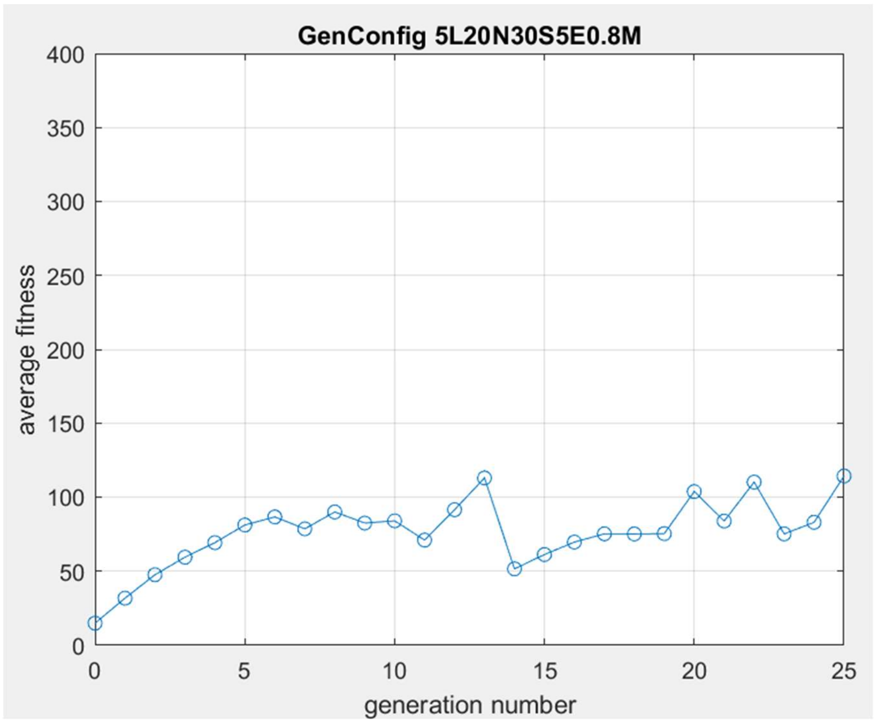


Figure 25 Final result of configuration 5L20N30S5E0.8M

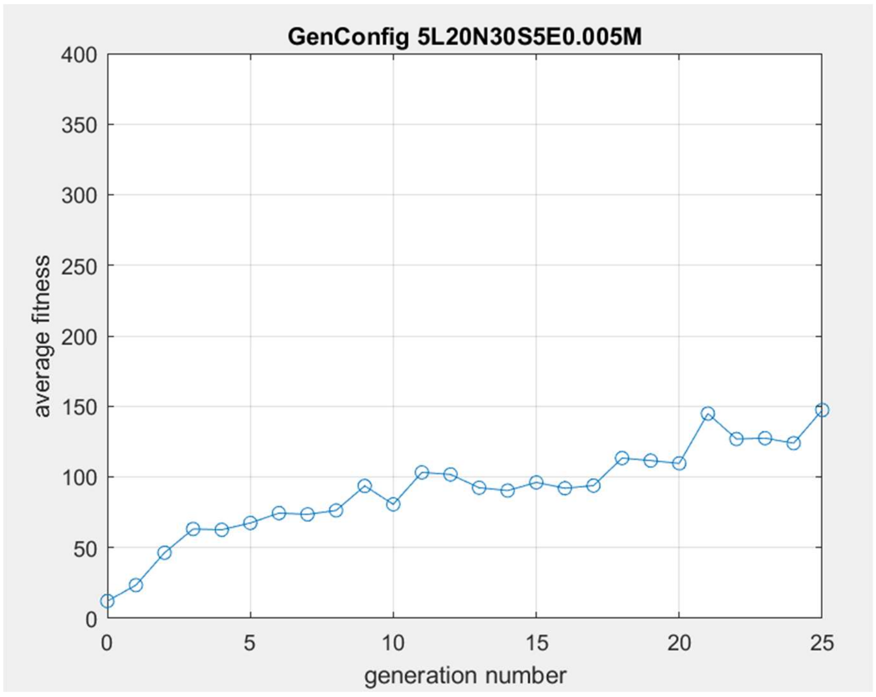


Figure 24 Final result of configuration
5L20N30S5E0.005M

This is in part because of other values in the configurations, layer and neuron count, population size and so on. It these values are optimal, mutation rate doesn't affect the simulation in a major way. Except if it is set at 0, which would result in slower learning and the agents could not escape from a local maximums to find the global maximum.

### 4.1.5 Elitism

Elitism is a number that determines how many best agents should be sent into the next generation without changing their values. If elitism is set to the same value as the population size, the model would push every agent from the initial population into the next generation, leaving no room for new agents from crossover operation. This means that every generation would have the same agents, and consequently, the same results.

On the other hand, if elitism is set to 0, none of the best agents from the current generation would be sent into the next, which could mean that the best agent in the next generation would have worse performance from the best agent in the current generation. Sending the best agents to the next generations without changing their values ensures that newly made agents will have parents with good traits, which improves simulation results.

Running one simulation with elitism set at 6, which is 20% with population size set at 30, and the other with 24 (80%), it is expected to see the best performance from the simulation with a smaller elitism value, due to the combination of passing the best agents to the next generation, and leaving enough room for new agents to be created.

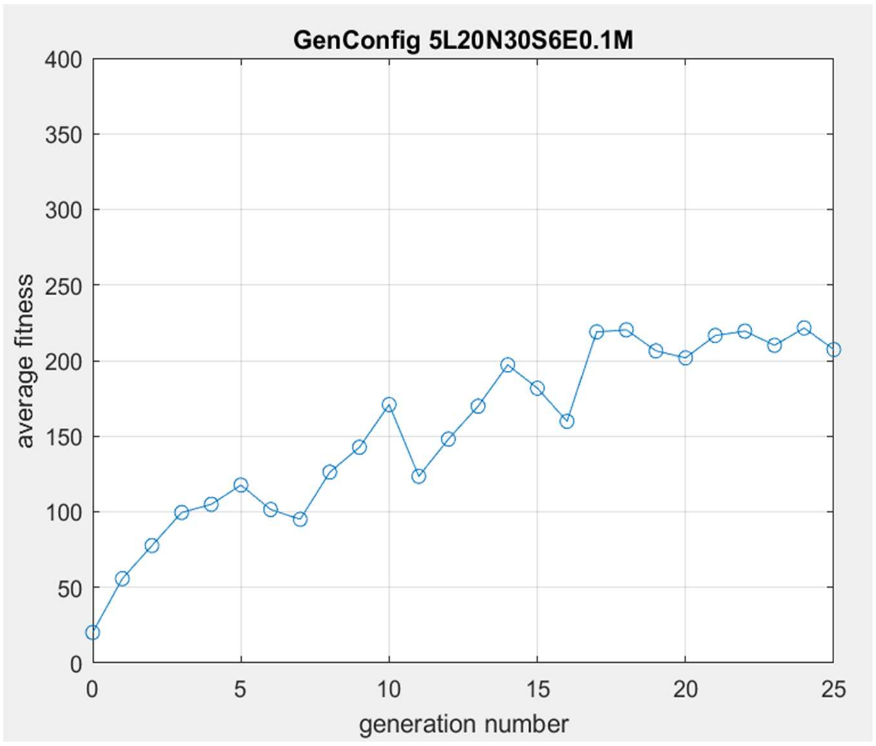


Figure 26 Final result of configuration 5L20N30S6E0.1M

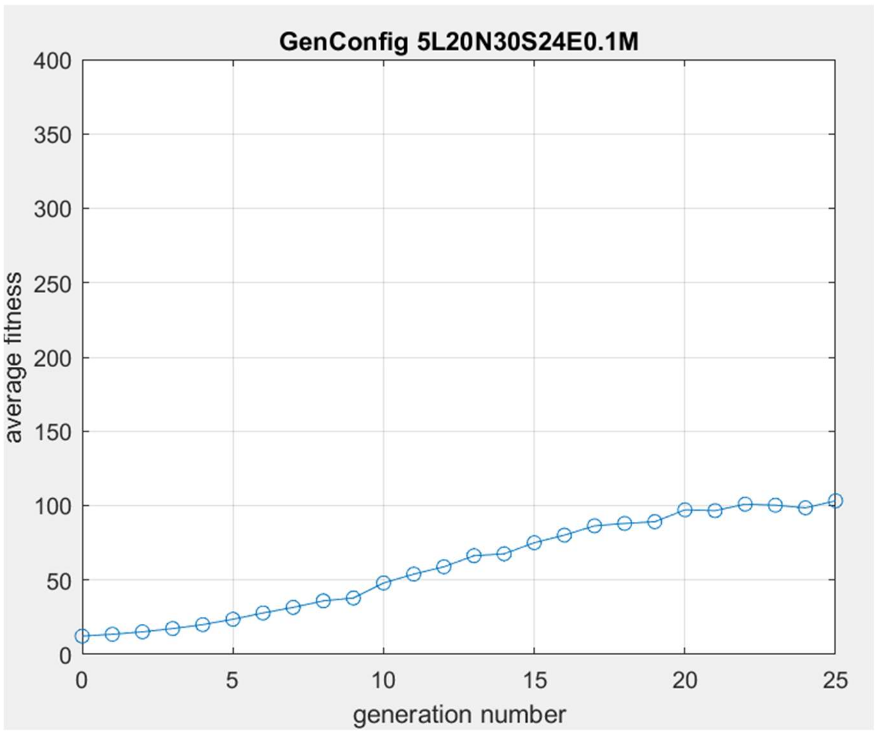


Figure 27 Final result of configuration 5L20N30S24E0.1M

As expected, smaller elitism value results in better performance. The simulation with a large elitism is in fact learning but given that it has only 6 new agents every generation, the process is slower.

### 4.1.6 Selection

Other than the genetic hyperparameters, different algorithms of selection can result in different performances. When performing selection operation, the goal is to select two agents that after crossover operation produce the best performing child.

Until now, every simulation was run with a fitness proportional selection algorithm [16]. This algorithm picks the agents according to their fitness score. The agent with a high fitness score will have a bigger chance of being picked, while the agent with a small score has a lower chance of being picked as a parent.

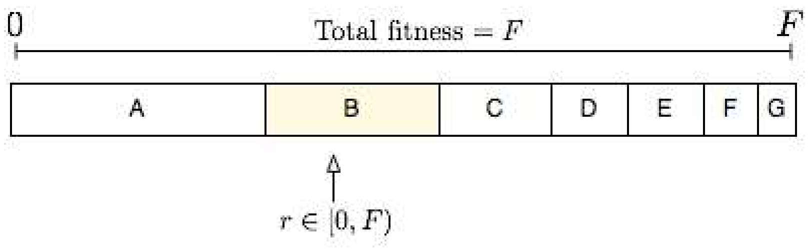


Figure 28 Fitness proportional selection model [16]

To determine if proportional selection improves the performance of the simulation, two simulations need to be run with the same configuration of genetic parameters, one needs to implement the proportional selection, and the other will use the regular selection that picks an agent from the population at random, with every agent having the same chance of being picked.

Results of the simulation using proportional selection with the configuration "*5L20N30S6E0.1M*" can be seen on Figure 26, and the same configuration is run using the regular selection, and its results are the following:

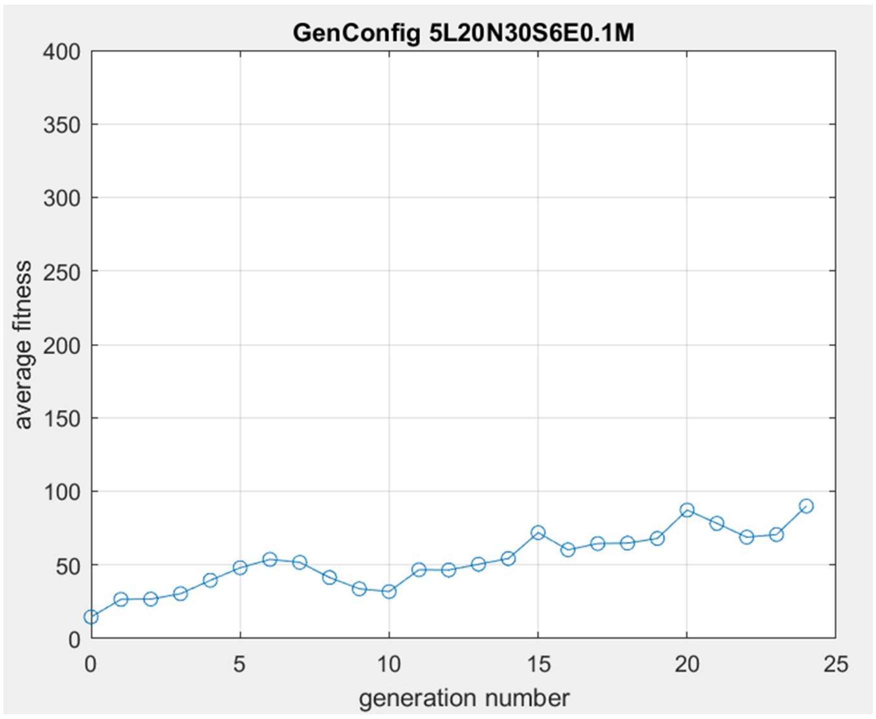


Figure 29 Final result of configuration 5L20N30S6E0.1M

There is a clear difference between the selection algorithms. It is important to note that the points in the graphs are values of average fitness for the whole population. The simulation using regular selection had agents that were able to finish the track with a good score, but other agents performed worse, hence lowering the average fitness.

This is to be expected. Fitness proportional selection gives the top performing agents a bigger chance of being picked, which results in more newly generated agents inhabiting their good traits. Agents that crash into a wall near the starting point get a small fitness value, which gives them a small chance of being picked as parents, meaning that their bad traits don't get passed on to next generations. Regular selection gives the bad agents the same chance of being picked as the top performing agents, resulting in lower average fitness and a slower learning process.

# 5 Model designs

The game uses a lot of models to represent an environment and to make the game look more aesthetically pleasing. Models of trees, bushes and rocks are free models from the Unity Asset store. Models of the car, racetrack and the terrain are made specifically for this game in a modeling software called Blender.

The car model consists of the car chassis, windows, wheels, lights, bumpers and a license plate. Car chassis is made from a single cube and is modeled with basic Blender tools such as: extrude region, inset faces, bevel and loop cut. To make space for tires, a Boolean modifier is used set at Difference mode, to combine mashes of the car and the area that the tire will occupy by subtracting them. Tires are made from a cylinder that has a low number of vertices to achieve the low polygon look to fit the rest of the game's art style. Tires have a Mirror modifier set to the x axis, to duplicate the model to the other side of the car. Lights, windows and bumpers are made using insertion of faces and are colored in appropriate colors.

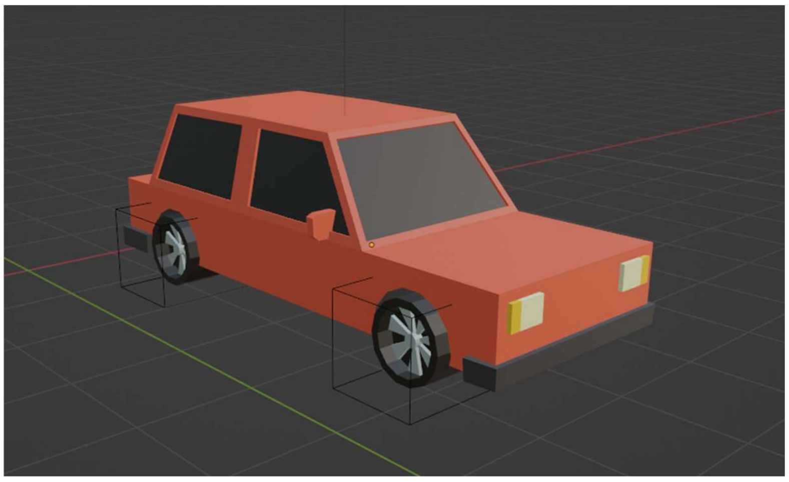


Figure 30 Car model

Terrain is made in the Shading workspace inside Blender. First, a flat plane is created and split into many vertices. Using various brushes, the height of each of the vertices is manipulated in a way to recreate an accurate representation of hills, mountains and meadows. Terrain is colored based on the slopes of each face, ones that are mostly flat should be colored in green to represent grass and if the slope is big then the color should be brown to represent dirt. This is done in the Shading workspace using multiple nodes to change the base color of the terrain. Firstly, a normal map node is added to get normals for every face in terrain. To filter which face should be colored green or brown color, a Greater than node is used, which sets a threshold at which point should the brown face be colored in green. After the desired look is achieved, the texture is baked, and the model can be imported to the Unity engine with correct textures.

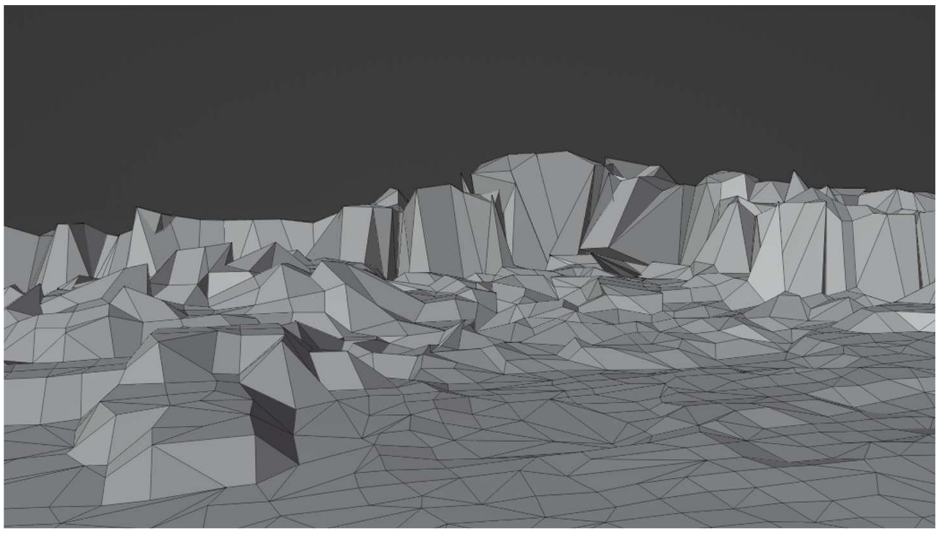


Figure 31 Terrain vertices

To model the racetrack with smooth turns and easy to manipulate trajectories, it uses an Array modifier, that is set to fit the curve. The path that the racetrack will follow is made with NurbsPath object. It allows easy modification and is responsible for the final shape of the road.

Using a racetrack model, extrusion and loop cut tools, a hit box can be modeled to perfectly fit the racetrack borders. Although the hit box won't be seen, it is important to model it right so that the collision detection in the game works as intended.

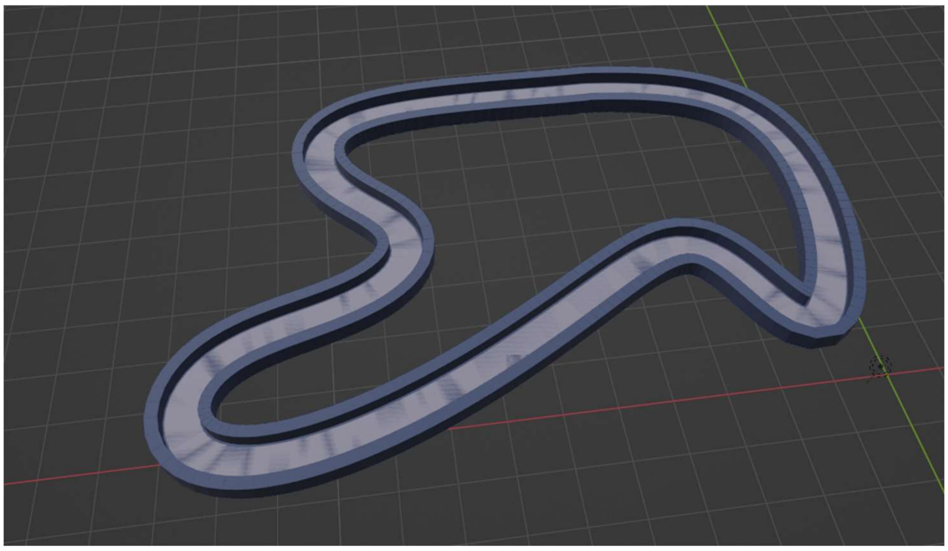


Figure 32 Racetrack and hitbox model

## References

1. Michael Nielsen (December 2019). Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/
2. https://brackeys.com/
3. https://developer.valvesoftware.com/wiki/Hitbox
4. https://www.youtube.com/watch?v=Dv4U-X-0Pms&ab_channel=KeelanJon
5. https://www.scaler.com/topics/artificial-intelligence-tutorial/state-space-search-in-artificial-intelligence/
6. https://www.cs.rit.edu/~lr/courses/ai/lectures/topic%204.pdf
7. https://www.nature.com/scitable/blog/brain-metrics/are_there_really_as_many/
8. https://www.researchgate.net/publication/5847739_Introduction_to_artificial_neural_networks
9. https://www.researchgate.net/publication/328733599_Impact_of_Artificial_Neural_Networks_Training_Algorithms_on_Accurate_Prediction_of_Property_Values
10. https://en.wikipedia.org/wiki/Activation_function
11. https://www.researchgate.net/publication/321259051_Prediction_of_wind_pressure_coefficients_on_building_surfaces_using_Artificial_Neural_Networks
12. https://www.fer.unizg.hr/predmet/uuui
13. https://www.javatpoint.com/types-of-machine-learning
14. https://dl.acm.org/doi/fullHtml/10.1145/3467477#:~:text=Evolutionary%20computing%20can%20be%20applied,even%20the%20rules%20for%20learning.
15. https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3
16. https://en.wikipedia.org/wiki/Fitness_proportionate_selection