Evaluating the performance of a neuroevolution algorithm against a reinforcement learning algorithm on a self-driving car

Author:
Kristián Kovalský

Supervisor:
George Palamas

AALBORG UNIVERSITY
DENMARK

Copenhagen
June 2020

Aalborg University Copenhagen

Semester: 10^{th} semester, Medialogy (MSc)

Title:
Evaluating the performance of a neuroevolution algorithm against a reinforcement learning algorithm on a self-driving car

Project period: February 1^{st}, 2020 - June 11^{th}, 2020

Semester theme: Master thesis

Supervisor:
George Palamas

Group members:
Kristián Kovalský

Finished: June 11^{th}, 2020
Pages: 43

Aalborg Univeristy Copenhagen.
A.C. Meyers Vænge 15, 2450.
Copenhagen SV, Denmark.
Secretary: Lisbeth Nykjær
Mail: lny@create.aau.dk
Phone: 99 40 24 70

Abstract
Exploring various machine learning methods revealed that the traditional gradient-based learning algorithms such as the ones used in reinforcement learning might not be as efficient with certain tasks as they might seem. By drawing inspiration from multiple state of the art examples, the idea of this project is comparing non-gradient-based algorithm such as neuroevolution with gradient-based reinforcement learning on an irregular task of training a car to self-drive around circuits with varying complexity. The raw quantitative data collected during evaluation show that neuroevolution is capable of producing solutions to this problem with great speed when compared to the reinforcement learning approach. However, when the reinforcement learning approach is allowed to train for long enough, it manages to train models that are outperforming even the ones created by neuroevolution. Further statistical research is required to see whether the differences in the performances are significant.

Contents

1 Introduction **1**

2 Background **2**
2.1 Tools . . . 2
2.1.1 Neural networks . . . 2
2.1.2 Reinforcement learning . . . 4
2.1.3 Genetic algorithms . . . 6
2.1.4 Neuroevolution . . . 8
2.2 State of the art . . . 9
2.2.1 Deep neuroevolution . . . 9
2.2.2 Neuroevolution of Augmented Topologies (NEAT) . . . 10
2.2.3 Neuroevolution as a game mechanic . . . 11
2.2.4 AlphaGo . . . 13
2.2.5 Learning to walk with deep reinforcement learning . . . 14
2.2.6 Experimenting with reinforcement learning in financial markets 15
2.3 Summary . . . 15
2.4 Final problem statement . . . 16

3 Methods **17**

4 Implementation **18**
4.1 Environment . . . 18
4.2 Neuroevolution . . . 23
4.3 Reinforcement learning . . . 25

5 Evaluation **28**
5.1 Preliminary evaluation . . . 28
5.2 Parameters . . . 29
5.3 Collection of data . . . 30
5.4 Results . . . 30
5.4.1 Training results . . . 30
5.4.2 Trained model results . . . 32

6 Discussion **35**
6.1 Results . . . 35
6.2 Biases . . . 35

7 Conclusion **37**

8 Future works **38**
8.1 Brakes 38
8.2 Recurrent NN 38
8.3 Evaluate generalization 38
8.4 Neuroevolution for reinforcement learning 38
8.5 NEAT 39

List of Figures

1 Structure of a neural network with one input, one hidden and one output layer and their connections . . . 3
2 Visual description of a population, genotype and a gene of a GA . . . 7
3 Visualization of a random crossover point between two members of a population . . . 8
4 Training screen of EvoCommander . . . 11
5 Battle between player and NPC in EvoCommander . . . 12
6 Visualization of how AlphaGo (black) decides what move to make next while playing against European champion Fan Hui . . . 14
7 Minitaur robot walking . . . 15
8 Front view of the car model used for evaluation . . . 18
9 Rear view of the car model used for evaluation . . . 19
10 Car placed on a circuit with its five raycasts displayed for debugging purposes . . . 19
11 The first circuit used for the evaluation . . . 20
12 The second circuit used for the evaluation . . . 21
13 The third circuit used for the evaluation . . . 22
14 Side view of the third circuit used for the evaluation showing various levels of elevation . . . 23
15 Snippet of a code displaying MoveCar method of the NE algorithm . 24
16 Snippet of a code displaying CalculateFitness method of the NE algorithm . . . 24
17 Snippet of a code displaying RePopulate method of the GA script . . 25
18 Snippet of a code displaying CollectObservations method of the RL script . . . 26
19 Snippet of a code displaying WallHit and CheckpointHit methods of the RL script . . . 27
20 Snippet of a code displaying PlaceCar method of the RL environment script . . . 27
21 Fitness scores across genomes during training of the NE on the first circuit . . . 31
22 Fitness scores across genomes during training of the NE on the second circuit . . . 31
23 Speed of the car trained with NE at each frame during three laps in play mode on the first circuit . . . 32
24 Speed of the car trained with NE at each frame during three laps in play mode on the second circuit . . . 33
25 Speed of the car trained with RL at each frame during three laps in play mode on the first circuit . . . 33

26 Speed of the car trained with RL at each frame during three laps in play mode on the second circuit 34
27 Entire configuration used for training with the RL algorithm 43

1 Introduction

Artificial intelligence (AI) has been present all over the world for some time now. One of the first mentions of intelligent computer systems that try to imitate human learning come from an article by Alan Turing in 1950 [32]. Ever since then, there have been multiple milestones in the development of the sub-field of AI called machine learning (ML). As its potential was established, the practice became more popular and made its way into various fields. In everyday life, AI can be for example found in charge of GPS systems that observe and evaluate a situation on the road and suggest the most optimal path for the driver or in self driving cars that use image recognition and decision making in order to function. In the field of entertainment, ML is capable of defining game logic and mimicking the actions and behaviour of real players. AI plays an integral part of video games where it is used for controlling non-player characters (NPCs). These can be very simple such as ghosts in Pac-Man acting according to a certain pattern at various stages of the game [33] to a more complex examples such as neural network (NN) controlled drivatars in the racing series of Forza games that learn to mimic the driving style of other players and both their strengths and weaknesses [21]. However, some of the most powerful AI bots such as AlphaGo developed by DeepMind require several days of training on extremely powerful hardware [25].

The issue with traditional ML algorithms that use gradient-based learning such as backpropagation is that they only work well when presented with big enough training data and sufficient computing resources [12] [19] [23]. The solution to this might be the introduction of a genetic algorithm (GA). One evolutionary approach called neuroevolution (NE) uses GAs for optimizing NNs instead of using stochastic gradient descent methods to train the weights. History shows that there are cases in which NE can outperform traditional ML algorithms such as reinforcement learning (RL) [20]. GAs might seem simple at first, thus creating the illusion that they are not capable of solving complex tasks efficiently.

This assumption is going to be investigated in the paper by testing the performance of a NE algorithm against a traditional RL algorithm. A series of circuits is going to be designed and the agent will be presented with a task of training a car to drive around the circuits.

2 Background

The following sections are first going to describe in detail how the ML approaches used in this paper work. Strengths and weaknesses of them will be discussed as well as their potential applications. Afterwards, notable and interesting projects and works in the field are going to be presented. Finally, a final problem statement will be established.

2.1 Tools

There are multiple approaches that can be utilized when developing a machine learning system. Some of them are going to be described in the following sub-chapters.

2.1.1 Neural networks

Artificial neural networks are based on how a real biological brain functions. What that means is that in the same fashion as real brain has millions of small neurons that are connected to each other through extensions called dentrites, the artificial neural networks also consist of computational units called neurons or nodes that have connections between each other. The amount of input and output connections that each node can have varies. The nodes are split into three main categories: *input*, *hidden* and *output* nodes. These nodes are grouped in layers and each NN consists of at least one input, one hidden and one output layer. When the number of hidden layers is greater than one, the NN is then referred to as a deep neural network. The amount of hidden layers and also the number of nodes within each of these layers can vary and this variation can have either positive or negative effect on the efficiency of the NN. Some tasks can be very simply solved with just one hidden layer and a few nodes whereas more complex tasks usually require tens of different hidden layers [13].

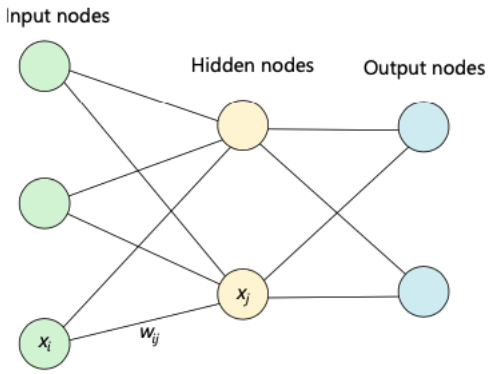

Figure 1: Structure of a neural network with one input, one hidden and one output layer and their connections [11]

Just as their names suggest, the function of the input nodes is to receive some sort of external input and pass its values through the outgoing connections to the hidden nodes. The hidden nodes then receive these incoming values, process them and send out an output value through the outgoing connections. Once the information passes through all the hidden nodes, it is sent to the output nodes that process all the incoming connections and return an output value after processing it.

Each of the connections of a NN has also weight associated to it. Weights in NNs are numerical values used to multiply the input that is passing through the given node and thus yield weighted inputs. The weighted inputs are added together with biases which are 'fixed' numerical values that are not connected to the input and this sum is called net input [9].

$$I_j = \sum_i w_{ij} O_i + \Theta_j \tag{1}$$

In the equation of net input I_j, the current node is denoted as j, incoming node is i, the weight between them is w_{ij}, the output node is O_i and Θ_j is the bias [9].

After the net input is calculated, an activation function is used on it which in return gives the final output value of the node. There are several activation functions that can be used and each of them is suitable for different type of performed task. Some of the most commonly used activation functions are Linear, Sigmoid, Rectified Linear Unit (ReLU) or Tanh [22].

During the training mode of a NN, the values of weights on the connections of the neural network are constantly being updated and changed using a certain optimizer such as the backpropagation algorithm and gradient descent.

Originally, NNs were most commonly used as classifiers, training on complex sets of training data. NNs proved to be quite good at this provided that the training data set was large enough and that there was some knowledge about what the optimal

topology of the NN should look like for the given problem. Currently, NNs can be seen performing more complex tasks as well, however their requirement for extensive training time prevents them from being used in real time. This issue can partly be solved by combining NNs with other machine learning tools [11].

2.1.2 Reinforcement learning

Another widely used machine learning technique is reinforcement learning. The explanation of RL can be extremely simplified and compared to to theory of trial and error. There are three main components of every reinforcement learning algorithm: *agent*, *environment* and *reward*. The agents that are present in the realm of RL have to face and learn from data that is completely unlabelled. The agents are placed in an environment and they can interact with it by observing the current state, taking an action and getting a reward (positive or negative) for their action. After the reward is given to the agent, it is again presented with the new state of the environment and it needs to decide on its next action. After certain time, the agent starts to develop a certain set of rules according which it acts. This strategy that is constantly being updated is called the policy [29].

Since the agents are receiving rewards after taking an action, all the actions have an expected future reward associated with them. Two things that the agent needs to take into consideration when deciding what action to take is the most immediate reward it will receive and also what is the next state it will go into. RL agents are most usually aiming to achieve as high long-term reward as possible. This means that the agent must sometime decide to take an action with smaller immediate reward in order to try to survive for longer and therefore accumulate higher long-term reward. If the agent would constantly be trying to accumulate actions with the highest immediate reward, it could end to an early fail of the agent [5].

The process of calculation of the future rewards is done by summing up the rewards that the agent acquired when it took a similar action at some point before. This sum is also multiplied by a variable called *discount factor* that is predefined by the developer. However, value of the discount factor (between 0 and 1) needs to be picked carefully, since too low value might be greatly disadvantageous for the future reward as the immediate reward would be too high. On the other hand, if the discount factor would be set to too high of a value, it would cause the opposite. This dilemma of whether immediate or future reward should be the main focus of the agent is called the credit assignment problem [5].

One way to solve this problem are value functions. The main two value functions are: *state-value function* and *action-value function*. The state-value function only looks at the current state of the agent within the environment when calculating the expected return whereas the action-value function needs to consider the action as well. The policy π which could be explained as probability $\pi(a|s)$ of the agent deciding to take a certain action a when being in a state s is taken into consideration as well [29].

$$V_\pi(s) = E_\pi\left[\sum_{k=0}^{\infty} \gamma + R_{t+k+1}|S_t = s\right] \quad (2)$$

The above equation shows an example of a state-value function. In the equation, $E_\pi[]$ represents the expected return value the agent would receive if it would act according to policy π.

Another problem that is present in RL is the one of exploration versus exploitation. The agent is facing this dilemma from the very beginning of training. Since the RL algorithm only sees state vectors and based on those it outputs certain action vectors that yield a reward, it sometimes might lead to the algorithm finding solutions which are far from what is desired. However, since the agent is getting rewarded, it keeps doing the same actions. This means that the agent is getting stuck in a local optimum. Example of such behaviour can be a self-driving car on a track that is getting rewarded based on the distance it travels. The expected behaviour is for the agent to try and drive the car as far as possible following the track. However, the car might through random exploration find out that making circles in one place will also generate positive rewards and thus it will 'burn in' that behaviour into the policy. There have been several approaches attempting to solve this issue, however, none of them are performing consistently on every possible task [3].

Last but not least, RL algorithms have to also deal with the problem of overfitting. This problem occurs when the agent performs at sufficient level during the training process, however it fails to generalize properly and fails to perform when introduced to a new environment. The main causes of overfitting are either if the data used for training are not sufficient to train the agent properly, or the agent is too complex for the given task and finds patterns in the training data that might simply be just a noise [5].

Markovian decision processes

Markovian decision processes (MDP) are integral part of RL algorithms. They come into place when an agent is forced to act according to a certain level of randomness since it cannot accurately predict the potential future reward for certain actions. The MDP is therefore also classified as discrete time stochastic control algorithm. The RL agents develop policy in state-action space. The policy says that at any given timestep, the agent that is in state S is provided with a certain set of possible actions to choose from. Once the agent chooses one of the actions based on its current state, it is provided with a new state S' from the environment. This means that every next state S' is completely based on the action that the agent takes while being in the previous state. This memorylessness of decision making process is the basis of the Markovian property: Every future state S' is solely dependent on the current state S and action A. This also means that it is conditionally independent from previous states and actions [2].

2.1.3 Genetic algorithms

The idea of mimicking natural process of evolution as a way of optimizing engineering problems first emerged in 1950s. The basic idea was to pick the most optimal solution from a population of all possible solutions by using *natural selection* and *natural genetic variation* operations. Both of these processes can be widely observed in nature where multiple living organisms interact with each other and their surroundings. The process of natural selection is the main mechanism responsible for variation of generations as it is the process of reproduction and survival of various individuals [18].

Genetic algorithms are a part of bigger group of algorithms referred to as *evolutionary algorithms* that belong to a class of *evolutionary computation*. The original genetic algorithm was first introduced by John Holland in 1960. There are 5 main phases of a genetic algorithm:

- Initialization of the initial population
- Fitness assignment according to fitness function
- Selection of the most optimal solutions from the current population
- Crossover
- Mutation

At the beginning of every GA, an initial population is generated. This is a random set of possible solutions. The set is random so that the initial variation is sufficient enough in order for the algorithm to go through many different possible solutions and hopefully finding the optimal one as quickly as possible. The members of the population, genotypes, consist of genes that are typically arrays or lists of strings or numbers. However, the form of genes can change depending on the current problem [18].

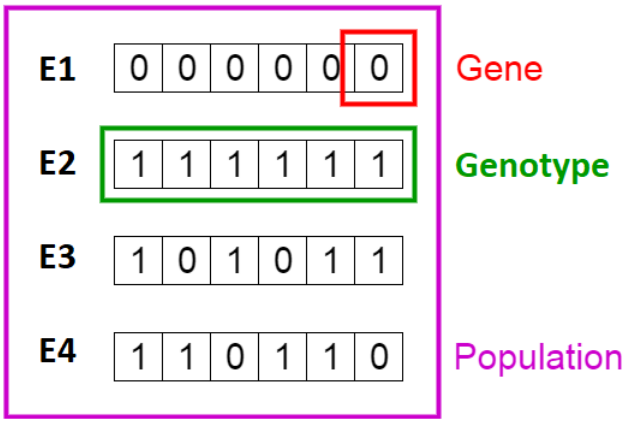

Figure 2: Visual description of a population, genotype and a gene of a GA [15]

After the initial population is generated, the algorithm assigns a fitness value or fitness score, which is a numerical value, to each member of the population based on how well it is suited as an optimal solution to the current problem.

After the fitness is assigned to all of the members of the population, the algorithm selects the members with the highest fitness value to be the donors of their genetic material for the next generation of the population. Usually, two fittest individuals are picked to be the parents, however this number can be changed by the developer, where in some cases even all of the members that have fitness higher than 0 can be selected to form the genes of the next generation. This way it is ensured that the most suitable individuals will contribute the most of the genetic material while the poorest performing individuals will only add slight change to the next generation [18].

The selection phase has two sub-phases: *crossover* and *mutation*. The crossover phase is the most crucial of every GA. This is the process when genes from the parental members of the population are exchanged to create new members of the next generation. This gene exchange is limited by a crossover point.

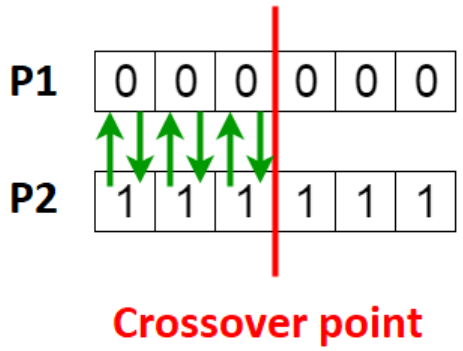

Figure 3: Visualization of a random crossover point between two members of a population [15]

The last step between crossover and re-population with the next generation is the mutation. Based on a predefined probability, each child member of the new population is created with some number of random genes. This random change exists so that there is still some variation present within each new generation. By doing so, the problem of the algorithm reaching a state of convergence before exploring all the possible solutions and probably also the most optimal one is partially eliminated [18].

2.1.4 Neuroevolution

When dealing with a problem where the optimal topology of the NN is unknown or when there is no training data available to train it with the traditional method of backpropagation, evolutionary algorithms can be used instead as an alternative. This way, the entire topology and weights of a NN can be evolved simultaneously without needing to know what specific setup to use beforehand. By assigning fitness values to the NNs according to how well they perform in the current scenario and further comparison of their performance can via the survival of the fittest technique eliminate the ones that perform poorly and then improve only on the better performing ones. However, sensible amount of mutation rate or a process where small amount of the poorly performing NNs is also kept alive for some amount of time can be beneficial for finding an optimal solution quicker and maybe even finding new, before unknown topologies that would outperform all of the previous ones [10] [26].

Because of the nature of NE and it searching for the right behaviour instead of a value function, it tends to be more suitable for problems where state space is continuous and high-dimensional [7] [6].

2.2 State of the art

GAs in combination with artificial neural networks have been used for many purposes. Some of the interesting examples of practical implementation of GAs as well as RL algorithms will be described in this section.

2.2.1 Deep neuroevolution

In their paper, Such et al. [28] describe how a non-gradient-based evolutionary algorithm can replace already established gradient-based learning algorithms such as backpropagation during the training phase of Deep artificial neural networks (DNNs). Their GA was used to evolve weights of a DNN. The evaluation of the approach was performed by comparing the performance of the GA to other contemporary algorithms such as Q-learning, random search (RS) or novelty search applied to deep RL. There were two different scenarios in which the GA was tested against the other algorithms, one environment being very well known set of Atari games, out of which 13 were picked, and second was a continuous problem in which a humanoid-like robot had to learn to walk.

When it comes to the Atari games, where pixels are directly mapped to actions, GA was performing very well when compared to DQN, A3C and ES. In 3 out of 13 games, the GA won in terms of the high score produced. There were some games in which the GA performed significantly worse which only goes to prove how some families of algorithms are more suitable to be used in deep RL for certain tasks than others. When the GA was allowed to run 6 times longer than the other algorithms, the score kept rising up and GA ended up outperforming A3C, ES and DQN in 7-8 games out of 13. Another thing that needs to be mentioned is how quick the GA found high performing solutions. The generations reached throughout the 13 games were ranging from 348 to 1834. What is interesting is that the GA was able to find a better solution to many games than DQN in one or few tens of generations [28].

Afterwards, the GA was tested against RS to confirm that that the GA is doing something more than just plain random search. The results showed that GA outperformed RS in every single game. This proves that GA optimizes efficiently with every new generation. However, RS ended up outperforming both DQN and ES on 3 games and A3C on 6 games, sometimes producing very sophisticated policies [28].

Unfortunately, when the GA was applied to the humanoid locomotion problem, it did manage to achieve satisfactory results, but it took approximately 15 times longer than in case of ES and yet the GA was performing slightly worse. The reason why this particular experiment failed is unclear, since GAs proved to be capable of performing well on locomotion tasks in the past [28].

2.2.2 Neuroevolution of Augmented Topologies (NEAT)

Neuroevolution of Augmented Topologies (NEAT) was first presented by Stanley and Miikkulainen in 2002 [27]. The idea behind NEAT is that it is an algorithm that attempts to evolve not just weights of NNs but also their topology at the same time. This enables NEAT to perform exceptionally well when faced with problems with limited domain knowledge and it also makes NEAT very good at generalizing. The only aspects that need to be specified about the NN beforehand are number of input and number of output neurons and a fitness function that is used to assess the performance of any given possible solution.

The process of optimizing NN with NEAT starts by producing a population of networks with no hidden layers and weights and connections that are chosen randomly. As the algorithm progresses through epochs and generations, hidden layers are added through the mutation and crossover processes of the GA. This results in the fact that NEAT is capable of producing solutions to problems with NNs that are as minimal as possible, ensuring that there are no bloated networks with parts that are not used at all. Another advantage of such network is performance gain of the algorithm [27].

One of the most prominent issues that had to be solved when developing NEAT was that of *Competing Conventions Problem* or *Permutations Problem* [27]. This problem arises when the network's hidden neurons are encoded. When looking at a simple network with three hidden neurons A, B and C, there are $3! = 6$ different permutations that can represent the same solution. The problem occurs when networks containing one of these permutations crossover, essential information ends up getting lost. When two networks have hidden units represented as A,B,C and C,B,A respectively, the resulting parent network after crossover could end up with hidden units C,B,C. The way NEAT deals with this issue is by keeping a track of which NNs are compatible with each other. These *historical markings* are then used to re-align genes prior to the crossover step so that no information is lost.

Performance evaluation of NEAT was performed on two different tasks: simple building of an XOR network and more complex task of balancing two poles on a cart by applying forces to it. In terms of the first task, NEAT produced very satisfactory results without any trouble. The networks produced very minimal topology and the algorithms has not failed once in 100 conducted simulations. The second task showed more noticeable advantage of NEAT. NE methods have proved to be able to outperform standard RL methods applied to the double pole balancing problem [20]. The algorithm had to balance upright two poles that are connected to a moving cart. The task was marked as complete when the algorithm managed to keep the poles between -36 and 36 degrees from vertical for at least 100000 time steps. Two versions of the second task were created: one where velocity inputs were available and one without velocity inputs. The first version was compared to four other NE systems and the second version was compared to two other systems. For the version with velocities, NEAT was able to complete the task in the fewest amount of evaluations with fewer average amount of hidden nodes than the other four systems. The more

complex version without velocities also added requirement that the algorithm has to balance the poles for 100000 time steps starting from 625 different test in order to prove its ability to generalize properly. At least 200 initial states need to be marked as completed in order to say that the algorithm is able to generalize. The results show that NEAT was again the fastest method for finding a solution with speeds up to 25 times faster than the other two systems [27].

2.2.3 Neuroevolution as a game mechanic

Interesting example of a practical application of NE is the game called EvoCommander developed by Jallov et al. [10]. In this particular project, a C# version of NEAT called SharpNEAT was ported into Unity and named UnityNEAT.

In EvoCommander, NE is again used to evolve artificial NNs, however, these trained networks are then not used for controlling NPCs, but they are given to the players and they can use these networks to take control of their character/robot. There are several behaviours that players need to train their agent to do first, such as ranged attacks, melee attacks, fleeing, etc. and thus prepare them for battle with other robots [10].

Figure 4: Training screen of EvoCommander where players are able to choose which hyperparameters are important to them for certain brain [10]

This novel concept of so called "brain switching" was then tested on a sample of 48 players as a single-player game. This test was aiming to collect data about how the players understood the concept and what did they focus on while training

their own ANNs. The results showed that more than half of the test participants finished the first mission of the game (from the total of 5). The missions served as beginner's guide to the game with each mission allowing the players to test their trained networks. After completing one mission, the next one unlocks together with a new trainable feature. The last mission consists of a boss fight after which the players can compete in online game modes [10].

Figure 5: Battle between player and NPC in EvoCommander [10]

Afterwards, another test was performed on a sample of 20 participants where EvoCommander was played as a multiplayer game. During the total of 143 played games, the players performed 4858 brain switches with at least two switches per match. Players who only switched their brain only two times per match won 29% of the matches. However, players who switched the brain three times during one match increased their win rate to 67%. The participants of the multiplayer test were also asked to fill out a questionnaire asking about their subjective experience with the game. The total of 14 answered questionnaires was gathered. The general theme of the answers was that the players found the online gameplay mode in combination with the brain switching game mechanics especially engaging. Despite this, many of the players reported that the initial training process of the ANNs was a bit too complex and lengthy [10].

All of this goes to show that NE can be used as an interesting and engaging game

aspect which allowed players to come up with many different and unique strategies to play the EvoCommander game and could potentially be expanded to other games [10].

2.2.4 AlphaGo

One of the finest examples of RL is the already mentioned bot trained to play the game of Go [24]. The game has been vastly difficult for AI to master because of its enormous search space and different combinations of positions and moves on the board. However, by combining value networks for keeping the track of positions on the board and policy networks for picking the actions, a novel approach for conquering computer Go has been introduced in 2016.

The system uses deep neural networks that are trained by combining supervised learning and reinforcement learning. The dataset for supervised learning contains data about predictions for moves from human expert players of the game Go. This first part of the system was trained on 30 million different positions and in the end reached accuracy of 57% compared to the previous best of 44.4%. This policy was called *SL policy network* [24].

The reinforcement learning part called *RL policy network* was trained by self-play which means that the current policy network was playing games against a randomly selected previous iteration of the same policy. The win-rate of RL policy was evaluated at 80% when it played against SL policy network.

Lastly, the position evaluation was trained by estimating a value function. This value functions serves as an indicator of what the outcome can be from the current position [24].

Both policy networks and value network were combined in a Monte-Carlo tree search algorithm, thus creating new version of computer Go called AlphaGo. The AlphaGo was evaluated during an internal tournament against several other computer Go programs that were considered as one of the strongest at that time. The outcome of the tournament was that AlphaGo won 494 out of 495 games it played, thus reaching a win-rate of 99,8% [24].

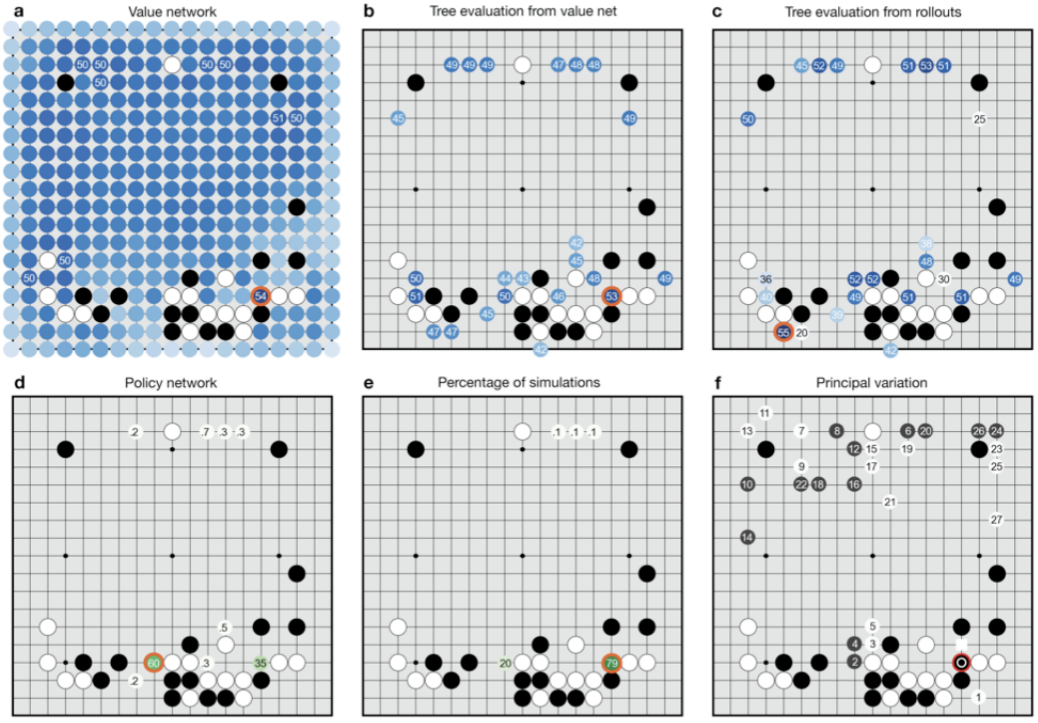

Figure 6: Visualization of how AlphaGo (black) decides what move to make next while playing against European champion Fan Hui [24]

2.2.5 Learning to walk with deep reinforcement learning

Another fine example of a practical implementation of a deep RL algorithm is project by Haarnoja et al. [8]. This project aimed to address the issue with transitioning from digital simulation space to real world space. The primary goal was to train deep RL algorithm so that it can map sensory inputs to low-level actions. The problem with applying RL to real life robots is that the system is greatly sensitive to small changes of hyperparameters and also suffers due to poor sample complexity.

Haarnoja et al. [8] present a novel sample-efficient algorithm based on entropy-constrained reinforcement learning. This algorithm is then applied on a real-world Minitaur robot and its task is to learn a pattern of steps in order to be able to walk. This method was able to train back-to-back, minimizing risks that come with fine-tuning hyperparameters in real world that might result in robots getting damaged because of the trial-and-error. The algorithm was able to develop a stable gait after mere two hours of training and achieved state of the art performance in simulated benchmarks. The learned policy was also able to generalize without any issues. The robot was first trained on a flat surface and then tested on a slope or on flat surfaces

that had obstacles placed in the way of the robot.

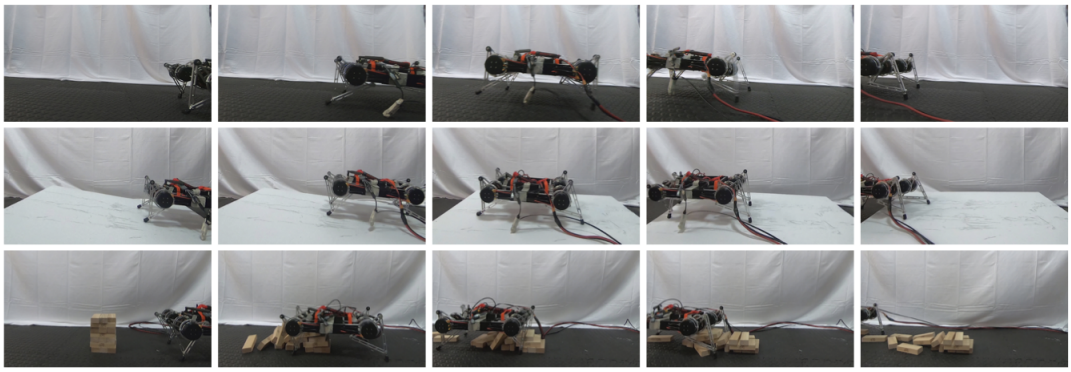

Figure 7: Minitaur robot walking on a flat surface as seen in the first row of images. The other two rows show terrain with a slope and terrain with wooden blocks on it, but the policy generalized properly to these novel situations [8]

2.2.6 Experimenting with reinforcement learning in financial markets

Examples of RL being used for computer bots and also for real life robots were already described. The following project explores the possibility of using RL in the area of financial markets [16].

AI has been making its way int the world of trading and financial markets quite rapidly in the last couple of years. Certain sophisticated machine learning methods are already being used for stock/forex predictions. However, when RL was used as the primary prediction method, there were many parameters such as transaction costs that were greatly affecting the accuracy of the algorithm and the lack of these parameters sometimes rendered the predictions made by RL completely unusable. The conclusion of all the comparisons between RL and already established deep learning models was that the RL is still very early in the development as a stock/forex trading tool and needs to be made more robust in order to be useful in real world [16].

2.3 Summary

All of the above findings show that GA tends to perform better in spaces that are irregular and poorly characterized. On the other hand, RL algorithms feel more comfortable at dealing with tasks that can be solved by creating a grid which maps states to actions. This knowledge combined with the findings of the papers presented in the State of the art section (see Section 2.2) leads to the formulation of the following final problem statement.

2.4 Final problem statement

Does a neuroevolution algorithm perform better at training a self-driving car to drive around circuits with varying complexity in terms of training time, average and maximum speed of the car and time it takes the car to complete a lap than a reinforcement learning algorithm?

3 Methods

The process and tools used in the evaluation of the Final Problem Statement (FPS) are going to be described in this section.

As it is clearly defined in the FPS, the aim of this project is to compare the performance of NE algorithm and RL algorithms on the task of training a car to drive around various circuits. In order to assess their performance, various metrics are going to be collected from the training phase of the algorithms and also from the final trained models. The metrics that are going to be collected during training are:

- Training time,
- Development of the fitness score/cumulative reward during training.

Once such policy is found so that two consecutive genomes complete at least three laps around a circuit, the training will be stopped. The trained models of both algorithms are then going to be used on the circuits that they were trained on to collect more quantitative data:

- Maximum speed the car reaches on a circuit,
- The average speed the car reaches on a circuit,
- The fastest time it takes the car to complete one lap,
- Average lap time.

After all data will be gathered, raw data will be visualized and statistical analysis will be performed on the results in order to determine whether there is a difference in the performance between the two algorithms and if yes, whether the difference is statistically significant.

4 Implementation

The following subsections are going to describe how all the machine learning techniques as well as the environment settings in which they were evaluated were implemented.

4.1 Environment

The entire project was put together using Unity3D game engine [30] and all the code was written in C# programming language. The project consists of three different circuits. The design of the circuits is identical for both GA and RL algorithms.

The circuits are placed on a simple plane object that has a grass texture applied to it. No special attention was put into making the surroundings of the circuits in any way pleasing since it could potentially harm the training time of the algorithms and the levels are not intended to be used as a part of any game, but only to showcase the performance of the algorithms.

The circuits themselves were created by using Bézier Path Creator asset from Unity Assets Store [14]. This lightweight tool allows to create custom paths in Unity by simply determining waypoints by clicking a mouse in the editor. After the waypoints are determined, the road tile objects were placed on their positions and connected together. The same waypoints were also used to spawn simple cube prefabs that were then moved to the edges of the road in order to make walls surrounding the entire path. A small green rectangle was put at the same position as the starting position of the car in order to determine the start/finish line.

The vehicle used in this project was taken from Car AI Tutorial series from EYEmaginary channel on YouTube [4]. The car consists of a simple 3D model with four wheel colliders that are used for controlling the speed and steering of the car.

Figure 8: Front view of the car model used for the evaluation [4]

Figure 9: Rear view of the car model used for the evaluation [4]

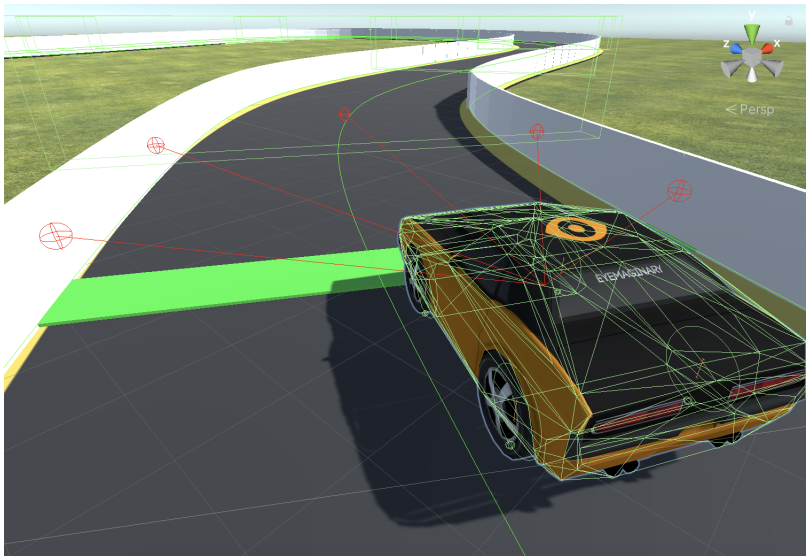

Figure 10: Car placed on a circuit with its five raycasts displayed for debugging purposes. Bright green line in front of the car is the start/finish line

The first circuit (Circuit 1) is the simplest of the three. The surface is completely flat with no elevations and the shape of it is a plain circle.

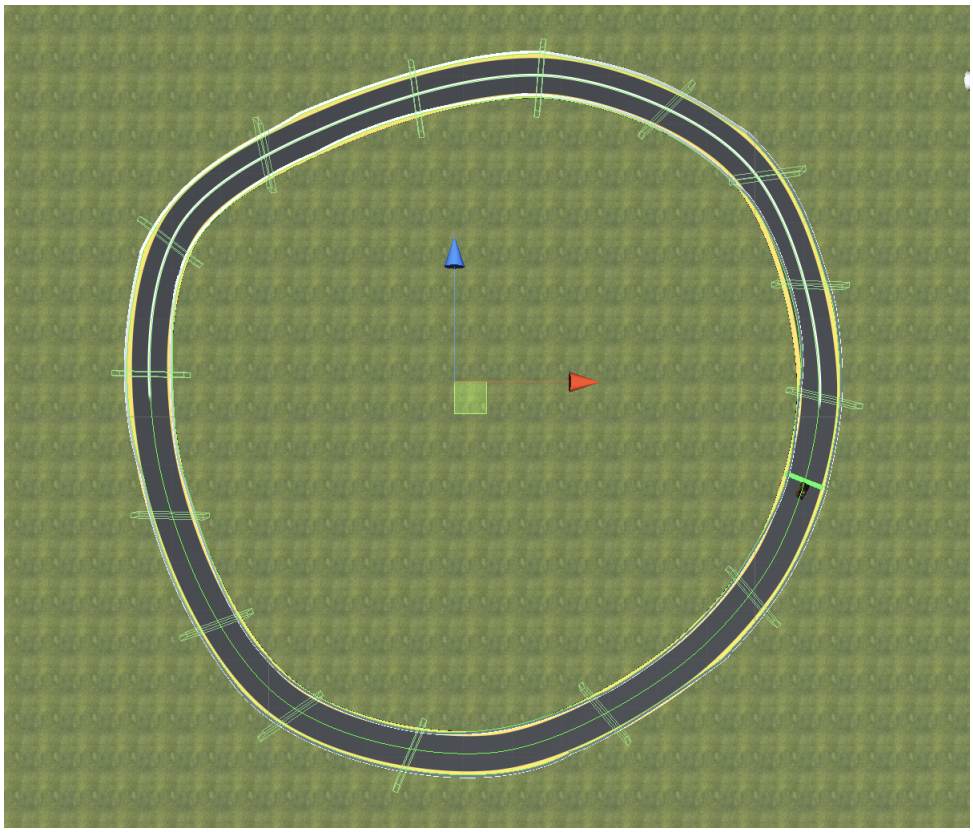

Figure 11: The first circuit used for the evaluation. Checkpoints used for RL algorithm are visible as hollow rectangles with green outline only for visualization purposes. Bright green line marks the start/finish line

The second circuit (Circuit 2) is also flat but it consists of multiple turns that were freely drawn by hand. The turns vary between left and right turns as well long and fast corners to slow and almost 180 degree hairpins.

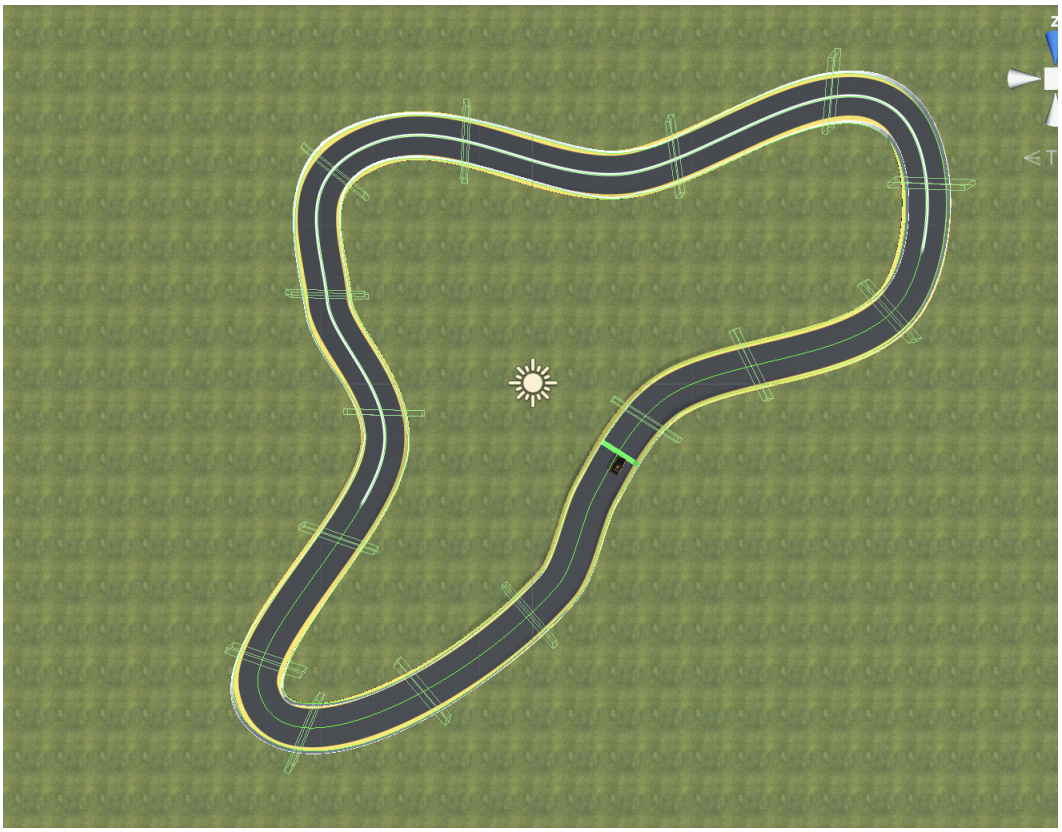

Figure 12: The second circuit used for the evaluation. Checkpoints used for the RL algorithm are visible as hollow rectangles with green outline only for visualization purposes. Bright green line marks the start/finish line

The third and last circuit (Circuit 3) is also the most complex one. There are only four turns but the first part of the circuit has a hill with a slight climb followed by a slight descent. The second part of the circuit has three successive hills with considerably steeper incline and decline than the hill in the first part of the circuit.

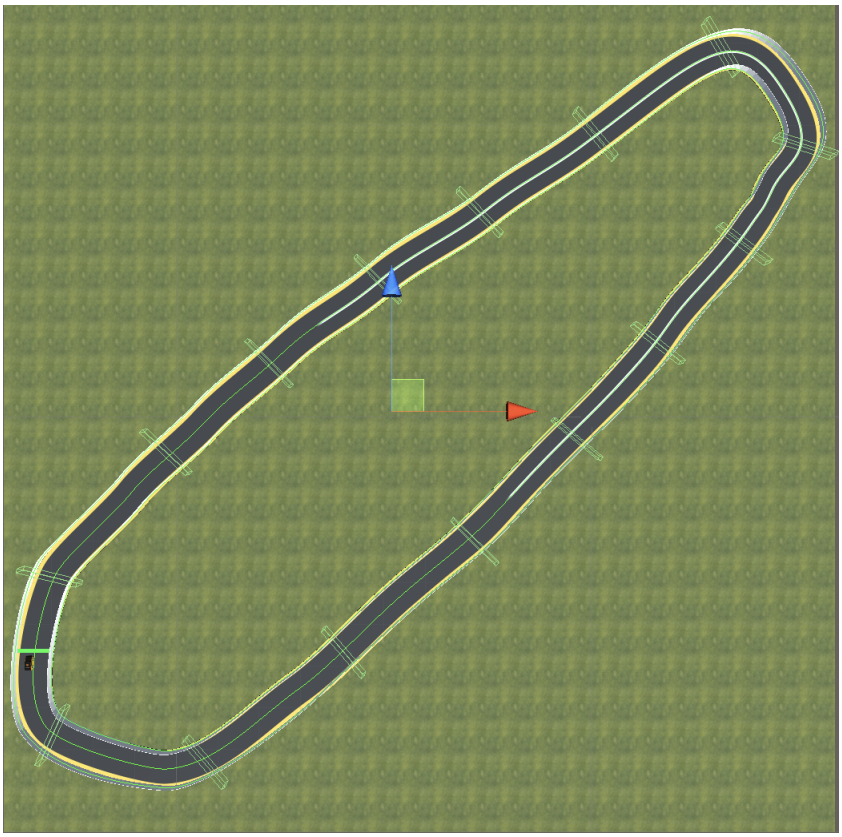

Figure 13: The third circuit used for the evaluation. Checkpoints used for the RL algorithm are visible as hollow rectangles with green outline only for visualization purposes. Bright green line marks the start/finish line

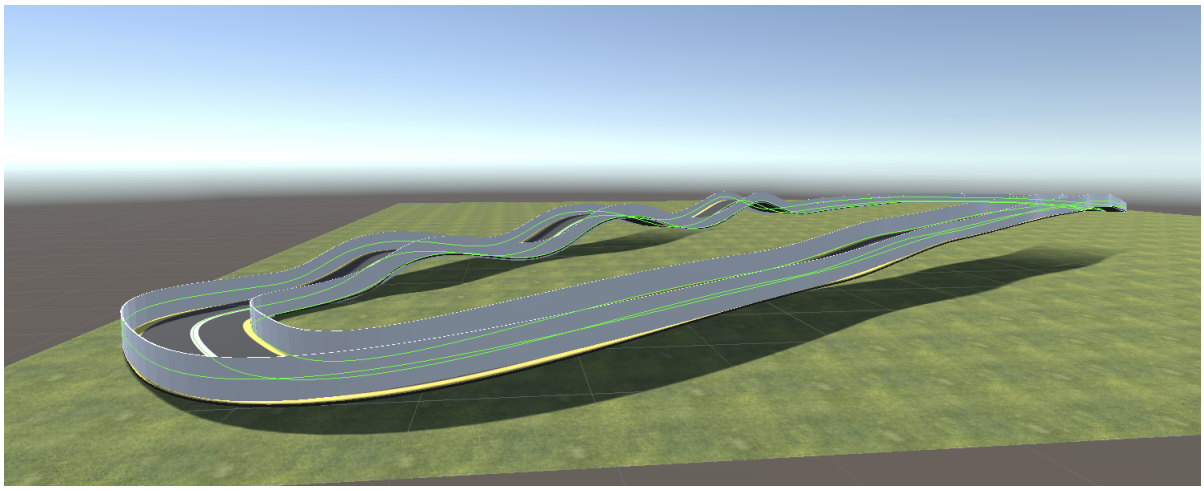

Figure 14: Side view of the third circuit used for the evaluation showing various levels of elevation

4.2 Neuroevolution

The skeleton of the neural network and the genetic algorithm were inspired by a project series called Self Driving Car [1]. For this project, a standard NE algorithm was used instead of more complex one such as NEAT. The reason behind this was the fact that the aim of this project was to compare training approaches. Both NE and RL use different methods for optimizing the weights of the NN which is the core of both algorithms. By using NEAT, the topology of the NN of GA used would be changed during the process as well, possibly creating unwanted biases.

There are three scripts controlling the NE part of the project: one script controlling the car, one script containing the code of neural network and one script for genetic algorithm which takes care of evolving the neural networks.

The car controlling script contains methods for resetting the car and its properties when it collides with a wall, method for applying forces to the car's wheel colliders in order to control the acceleration and steering of the car, method for placing the raycasts on the car and lastly a method for calculating the fitness of the car.

```
254     public void MoveCar (float v, float h) {
255         currentSpeed = rb.velocity.magnitude;
256
257         wheelFL.motorTorque = v * maxMotorTorque;
258         wheelFR.motorTorque = v * maxMotorTorque;
259
260         targetSteerAngle = h * maxSteerAngle;
261
262         wheelFL.steerAngle = targetSteerAngle;
263         wheelFR.steerAngle = targetSteerAngle;
264     }
```

Figure 15: Snippet of a code displaying Move method of the NE algorithm

The fitness function is based on 3 variables: the distance traveled by the car, average speed of the car and distance readings from the raycasts. Each of these variables has also their own multiplier assigned to it. The multiplier makes it possible to assign higher or lower importance to certain variables. By doing so, it is possible to control what kind of behaviour the car will aim for, whether it will focus on traversing as much distance as possible or whether it should try to reach as high speed as possible instead of opting for the traversed distance.

```
174     private void CalculateFitness() {
175
176         currentDistance = Vector3.Magnitude(transform.position);
177         startDistance = Vector3.Magnitude(lastPosition);
178         distanceTravelled = startDistance + currentDistance;
179
180         totalDistanceTravelled += Vector3.Distance(startPosition, transform.position)/10;
181         avgSpeed = totalDistanceTravelled/timeSinceStart;
182
183
184         overallFitness = (totalDistanceTravelled*distanceMultipler)+(avgSpeed*avgSpeedMultiplier)+(((aSensor+bSensor+cSensor+dSensor+eSensor)/5)*sensorMultiplier);
```

Figure 16: Snippet of a code displaying CalculateFitness method of the NE algorithm

The fitness calculation function also contains two if statements. The first if statement checks the time that has elapsed since a genome started existing and if the fitness score of that particular genome is smaller than a predefined value of 350, it will kill the genome and spawn a new one instead. The second if statement sets the upper bound for the fitness value and thus stopping the genome from exploring the environment once it has reached a sufficiently high enough fitness score.

Once the car collides with a wall, properties of that particular genome are saved, it is subsequently killed and new genome is spawned.

The script for the NN builds a functioning neural network from scratch. The base of the script is matrix multiplication. The input, hidden and output layers as well as weights between the neurons are all stored in matrices. The methods in the script are: method for initializing a neural network, method for randomizing weights and method for running the network which outputs the values for actions that the car makes: acceleration and steering. The acceleration value is constrained to values

between 0 and 1 and steering value is constrained between -1 and 1. The activation functions used are sigmoid for acceleration and tanh for steering. The methods of the NN script are constructed in a way that changing the topology of the NN requires very little to none changes to the code itself.

Lastly, the script for GA contains methods for creating an initial random population, creating new population of children, sorting and picking the best members of a population, crossover, mutation and method for when a genome dies and calls the reset method of the car controller script. Attributes such as number of members of a population, mutation rate and crossover rate can be specified in this script as well in order to enable as much control over the algorithm as possible.

```
77      private void RePopulate()
78      {
79          genePool.Clear();
80          currentGeneration++;
81          naturallySelected = 0;
82          SortPopulation();
83
84          NNet[] newPopulation = PickBestPopulation();
85
86          Crossover(newPopulation);
87          Mutate(newPopulation);
88
89          FillPopulationWithRandomValues(newPopulation, naturallySelected);
90
91          population = newPopulation;
92
93          currentGenome = 0;
94
95          ResetToCurrentGenome();
96
97      }
```

Figure 17: Snippet of a code displaying RePopulate method of the GA script

4.3 Reinforcement learning

A toolkit called *The Unity Machine Learning Agents Toolkit* (ML-Agents) developed by Unity Technologies was used for the implementation of the RL algorithm [31]. This toolkit is open-source and contains state of the art algorithms for training intelligent agents for various purposes such as game NPCs or running simulations.

The agent in the RL environment collects various observations about its current state and chooses actions based on that state. In this particular example, the car is presented with information about its immediate velocity on all three axes X, Y and Z, its local position and immediate angle of its front wheels. It also receives observations from five raycasts that are cast from the middle of the car. One raycast is aimed directly in the front of the care and the remaining ones are placed two on each side with 30 degrees angle between them. These raycasts are checking for the terrain, walls and checkpoints in front of and around the car. There are two actions that the agent

can perform: *acceleration* and *steering*. The form of actions is continuous, meaning that the action is presented to the agent in an array of floating point numbers. This means that the agent can finely regulate how much acceleration and steering is applied by using values between 0 (no acceleration) and 1 (full acceleration) for acceleration and between -1 (left) and 1 (right) for steering. The final acceleration force applied is calculated by multiplying the output of the acceleration action with a property of the car's wheel collider called motorTorque. The steering angle is calculated by multiplying the output of the steering action with a maximum steering angle allowed for the wheels which is -45 degrees to the left and +45 degrees to the right in this particular example.

The RL part of the project consists of four scripts: one script for controlling the actions of the car, one script for controlling the environment and two scripts for checking for collisions: one for checkpoints and one for walls.

The script for controlling the agent includes methods for collecting observations, choosing actions to take based on the observations as well as methods for awarding and punishing the agent. The agent is awarded 0.2 points every time it collides with any of the 15 checkpoints evenly distributed around each circuit. Positive reward is also given to the agent based on its immediate velocity on Z axis (forward and backward) divided by 2000. A tiny negative reward is given to the agent at each step in order to motivate it to move forward and seek higher reward. A big negative reward of -1 point is given to the agent when it collides with any of the walls. Hitting walls also ends the current episode, resets the agent to its initial starting location, reward is set back to zero and new episode is started.

```
130     public override void CollectObservations(VectorSensor sensor)
131     {
132         sensor.AddObservation(localVel);
133         sensor.AddObservation(transform.localPosition);
134         sensor.AddObservation(wheelFL.steerAngle);
135         sensor.AddObservation(wheelFR.steerAngle);
136     }
```

Figure 18: Snippet of a code displaying CollectObservations method of the RL script

```
170         public void WallHit()
171         {
172             AddReward(-1f);
173             EndEpisode();
174             distance = 0f;
175             diff = 0f;
176         }
177
178         public void CheckpointHit()
179         {
180             AddReward(0.2f);
181         }
```

Figure 19: Snippet of a code displaying WallHit and CheckpointHit methods of the RL script

The environment script contains methods for placing the car, resetting the entire environment and continuously updating current value of the rewards that the agent has accumulated. At the start of each episode, the agent is placed on the same position coordinates, however the rotation of the agent is picked randomly from a range of 0 to 60 degrees from its initial rotation in order to support exploration and introduce some variation.

```
33      private void PlaceCar()
34      {
35          Rigidbody rigidbody = carAgent.GetComponent<Rigidbody>();
36          rigidbody.velocity = Vector3.zero;
37          rigidbody.angularVelocity = Vector3.zero;
38          carAgent.transform.position = carAgent.startPosition;
39          carAgent.transform.rotation = Quaternion.Euler(0f, UnityEngine.Random.Range(0f, 60f), 0f);
40      }
```

Figure 20: Snippet of a code displaying PlaceCar method of the RL environment script

5 Evaluation

This section is going to describe in detail the test that was conducted in an attempt to answer the final problem statement. Afterwards, raw results are going to be presented as well.

5.1 Preliminary evaluation

Before the actual training and evaluation could have been conducted, a series of smaller tests was done in order to fine-tune different parts of the ML algorithms as well as the agents and the environment themselves.

One of the things that was tested was the number of raycasts the agents use to check for objects in its surroundings. At first, the agent was using three raycasts, one pointing forward and two pointing to the sides at the right angle to the forward facing raycast. This configuration caused issues for times when agent was trying to stay as close to the wall as possible and the path suddenly took a sharp turn and the agent crashed into a wall with either front left or front right part of the car. Afterwards, two more raycasts were added at 30 degree angles from the forward facing one to each side of it. This improved the behaviour of the agent as it could collect more observations around itself. Lastly, a configuration with ten raycasts was tested where the raycasts were surrounding the entire 360 degrees around the car object. It was discovered that increasing the number of raycasts above 5 and including raycasts that were cast from middle towards the back of the car did not have a significant impact on the performance of the training algorithm.

Next, the amount of force that is applied to the car's wheel colliders in the form of variable called motorTorque was explored. Too small value resulted in a car that was moving way slow, whereas too high value of the motorTorque resulted in a car that could achieve great speed but was not able to make it past some sharper turns as the wheels lost grip with the surface of the road and the car just slid into a wall. A value of 500f has proved to be working just fine when combined with the output of the NNs that is multiplied with the motorTorque value and ranges between 0 and 1.

A property of Unity's rigidbody component called centerOfMass had to be used in order to make the car more stable. Before the center of mass was lowered by 0.2f on the Y axis, the car would get very unstable when exiting corners at higher speed which very often resulted in an uncontrollable spin that ended in a wall or in special cases even with the car rolling over on its roof.

One very well known problem with RL algorithms that was already mentioned in the subsection called Reinforcement learning (see Section 2.1.2) is the one of exploitation. This problem also appeared in the early stage of developing the RL algorithm for this project. The agent ended up finding parts of the circuit that were wide enough for it to turn around and drive back to the start where it would turn around again and head back to the point where it turned around initially, thus creating a policy that

kept driving in a loop on one part of the track. This problem was solved by adding checkpoints to the circuits used for the training of the RL algorithm. By decreasing the reward for the distance traveled and instead giving the agent a reward for driving through checkpoints that was substantial enough to outweigh the reward for its speed, the agent learned that in order to get higher long term reward, it needs to keep driving forward and keep collecting rewards from checkpoints instead of driving around on the same place of the circuit indefinitely.

5.2 Parameters

Each of the two algorithms has its own parameters that need to be set before the training process can begin.

Table 1: Training settings of the NE algorithm

Initial population	50
Mutation rate	0.055
Best agents for crossover	8
Worst agents for crossover	3
Number to crossover	39
Distance multiplier	1
Average speed multiplier	0.5
Raycast multiplier	0.1
Number of raycasts	5
Number of hidden layers	3
Number of hidden neurons	15

Table 2: Training settings of the RL algorithm

Vector space size	8
Action space type	Continuous
Action space size	2
Number of raycasts	5
Trainer	PPO
Number of hidden layers	2
Number of hidden neurons	128
Learning rate	0.0003
Maximum steps	9.0e6

5.3 Collection of data

Both algorithms had the same goal to reach during training: the agent needs to go around the circuit at least three complete laps for two successive genomes. After this condition was met, the training was stopped and metrics about the training were collected. When it came to training with the NE algorithm, the model was trained 10 times until the condition of two successive genomes completing 3 laps was reached and the average training time across the 10 tests was calculated.

Once the trained model was generated, it was then used to drive around the same track in order to collect additional metrics that would help to assess the performance of the training algorithms. During this secondary data collection, the car was first let to complete one full lap from a standstill position in order to acquire some speed. As soon as the car crossed the start/finish line at the end of its first lap, data about the car's speed and elapsed time started being recorded. The car was then let to drive for three more laps. After the third lap, the data collection was stopped and all the collected values were saved in a text file.

5.4 Results

Raw data results are going to be presented in the following sections together with graphs visualising certain collected variables.

5.4.1 Training results

First of all, it should be mentioned that neither NE nor RL algorithms were able to meet the criteria for successful training on the Circuit 3 with elevation changes. Therefore, only the results from Circuits 1 and 2 are going to be presented.

The average training time of the NE algorithm on Circuit 1 was 43.42 seconds. The condition for successful training was met on average during generation 7. Figure 21 displays the progression of the fitness function during one of the fastest runs that performed well already during generation 2 genome 12. However, the performance dropped rapidly immediately after that genome and the successful training condition was met only after generation 4 genome 32.

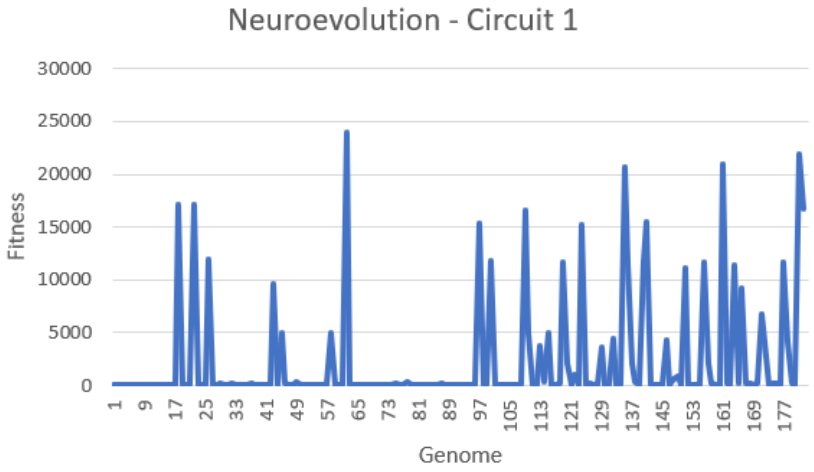

Figure 21: Fitness scores across genomes during training of the NE on the first circuit

The average training time of the NE algorithm on Circuit 2 was 1 minute and 21 seconds. The condition for successful training was met on average also during generation 7. Figure 22 again displays run that found the right solution quite soon during generation 4.

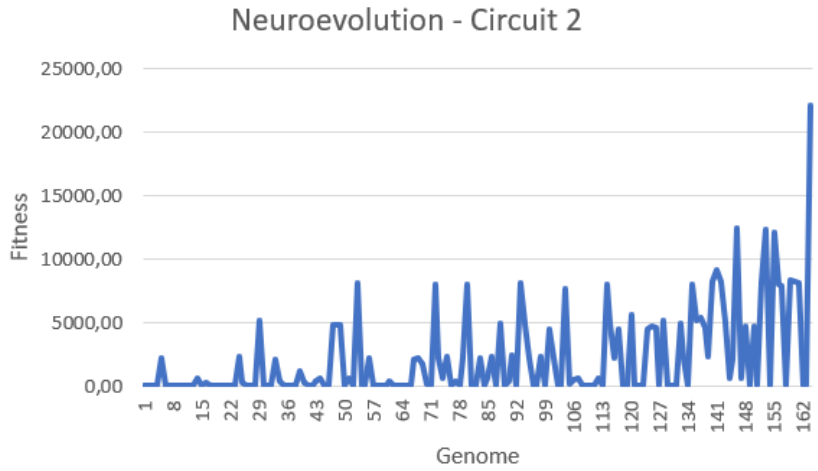

Figure 22: Fitness scores across genomes during training of the NE on the second circuit

When it came to training with the RL algorithm, the criteria for successful training on Circuit 1 was met after 8 hours and 25 minutes. The model went through 3950000 steps in order to reach the solution.

Training the model with the RL algorithm on more complex Circuit 2 took 15 hours and 10 minutes. The model completed 7050000 steps until the right policy was found.

5.4.2 Trained model results

Data collected from the trained models show that the NE algorithm on Circuit 1 reached the best lap time of 17.52 seconds. The average time after three laps was 20.16 seconds. Additionally, as it can be seen on Figure 23, the maximum speed reached on Circuit 1 was 27.57 and the average speed was 20.83.

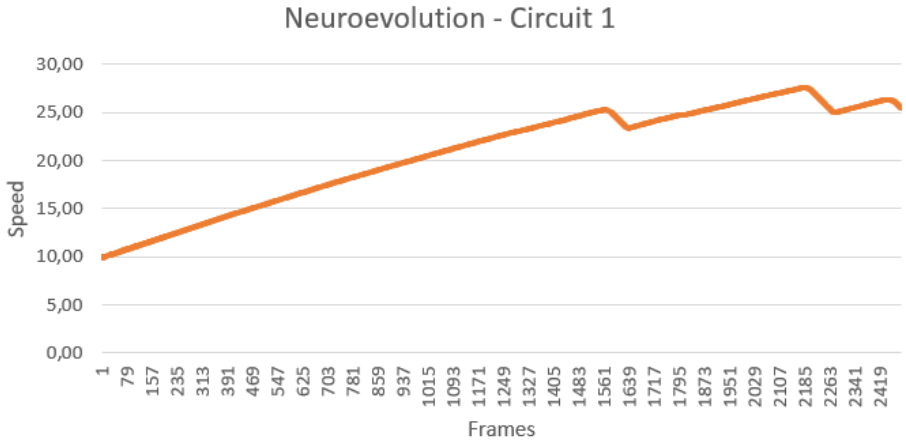

Figure 23: Speed of the car trained with NE at each frame during three laps in play mode on the first circuit

The model trained on Circuit 2 achieved the best lap time of 31.18 seconds and the average lap time after three laps was 33.63 seconds. The top speed reached on Circuit 2 was 15.35 and the average speed during three laps was 12.47 (see Figure 24).

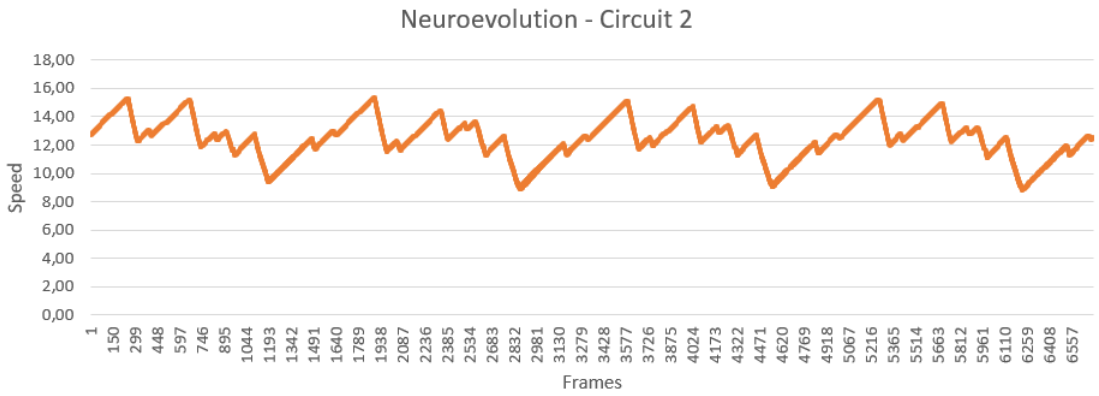

Figure 24: Speed of the car trained with NE at each frame during three laps in play mode on the second circuit

When the model trained with RL algorithm was tested on Circuit 1 it achieved the best lap time of 15.66 seconds and the average lap time of 17.15 seconds. Moreover, the top speed on Circuit 1 was measured at 30.57 with the average speed over all frames being 25.66 (see Figure 25).

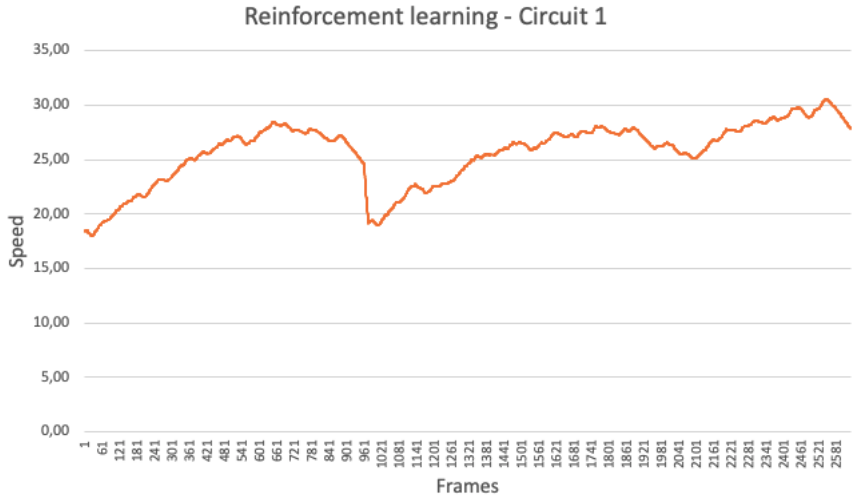

Figure 25: Speed of the car trained with RL at each frame during three laps in play mode on the first circuit

Finally, the RL trained model on Circuit 2 achieved the best lap time of 32.01 seconds and the average lap time over three laps was 32.82 seconds. The maximum speed the car reached on Circuit 2 was 18.29 with the average being 12.52 (see Figure 26).

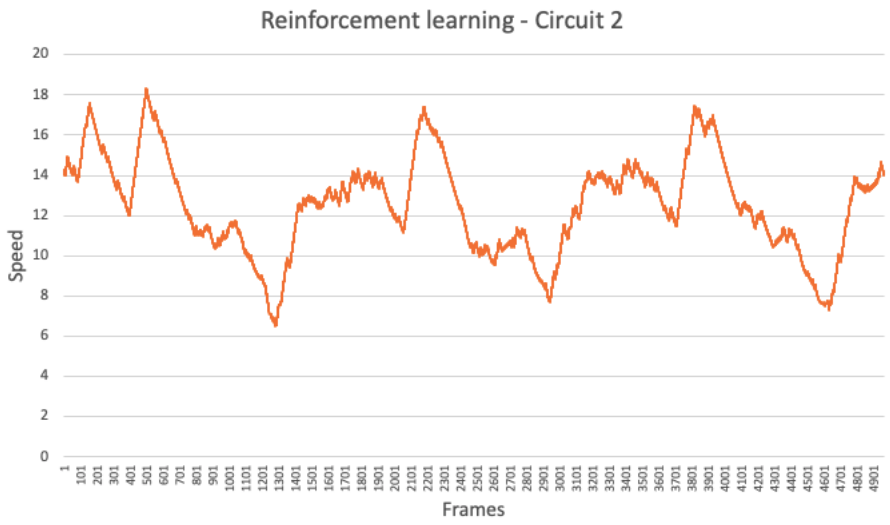

Figure 26: Speed of the car trained with RL at each frame during three laps in play mode on the second circuit

Now that all the raw data were presented, it allows for their discussion and reasoning of why certain algorithms behaved the way they did.

6 Discussion

This section is going to critically look at the gathered results and talk about causes of potential advantage of one training algorithm over the other.

6.1 Results

The first thing that is obvious when looking at the results is the difference in training times. The GA and in turn NE was expected to perform better than the RL, however the difference is very substantial and noticeable. Training of the NE on Circuit 1 took 43.42 seconds on average compared to 8 hours and 25 minutes of the RL. Training of the NE on Circuit 2 took 1 minute 21 seconds on average compared to 15 hours 10 minutes of the RL. The nature of the NE algorithm also enabled it to sometimes, through complete accident, find a network that managed to drive several laps around a circuit on its first attempt. This only goes to support the claims and findings stated in the Background section (see section 2) that NE is more suitable for problems that are irregular and not so clearly defined.

On the other hand, comparing the lap times and speed achieved by the agents show that the RL trained model was able to outperform the NE model in all four measured examples. On Circuit 1, the NE model took on average 20.16 seconds to complete a lap compared to 17.15 seconds of the RL model. Average speed on Circuit 1 was also around 20% higher for the RL model at 25.66 compared to 20.83 of the NE model.

On Circuit 2, the differences between the two models were not as noticeable. The average lap time of the NE model was 33.63 seconds and the average lap time of the RL model was 32.82 seconds, making the difference between them less than 1 second. The difference is even smaller for average speed, where NE model reached value of 12.47 and RL model reached value of 12.52.

6.2 Biases

There are several factors which if were treated differently, they might have affected the way the results turned out for both training and play parts of the algorithms.

First and foremost, the difference in topologies of the NNs used in both algorithms could have caused an unfair advantage of one over the other. As it was stated in the Parameters subsection of the Evaluation section (see section 5.2), the NN used in NE algorithm consisted of 3 hidden layers and 15 hidden neurons, whereas the NN used in the RL algorithm consisted of 2 hidden layers and 128 hidden neurons. Experimenting with finding a middle ground between the two setups could have resulted in different performance of either of the two algorithms.

Additionally, the mutation rate and crossover rate of the NE have a great impact on the ability of the NE to find optimal solutions. Looking at Figures 21 and 22 shows

how unstable the outputs of the NE algorithm are during the training. Lowering the mutation rate or increasing the number of better performing individuals to be used for crossover could potentially improve the stability of the NE algorithm.

Another difference between the two algorithms is the way they are awarded for their actions. In case of the NE algorithm, the agent is awarded the fitness score based on the sum of various weighted variables: distance traveled, average speed and raycast readings. On the other hand, the RL agents gets higher positive reward the faster it is moving forward and it is also driven by the fact that there are checkpoints placed at regular intervals around the track fueling its need to continue moving forward. As it was already mentioned in the section 5.1, the checkpoints were added to the RL algorithm to fight the well known issue of RL algorithms called exploitation. However, adding them on the NE circuits or adding rewards to the RL algorithm based on the same way as they are given to the NE algorithm would make the comparison more fair.

Lastly, to make the training of the agents easier, the acceleration action of both algorithms was constrained to be always within the ream of positive numbers. What this means is that the agents were basically told from the very beginning that moving forward is the right and the only direction they should be moving. If the agents would be able to counter the torque added to the front wheels by either braking or even being able to reverse, the results would almost definitely look different. However, with extra action to take, the problem of exploitation of the RL algorithm becomes more prominent again. The lack of braking force also contributed to the fact that neither of the two algorithms managed to successfully train on Circuit 3. As the car kept rolling downhill towards a turn, even though the agent was not accelerating by adding torque to the front wheel colliders, the speed of the car was increasing. Some agents found out that changing the turning angle of the front wheels rapidly between -45 and 45 degrees creates friction that does slow the car down slightly, however it was not enough to decrease its speed enough to take a turn at a safe speed that does not result in a crash.

7 Conclusion

The aim of this project was to see whether a neuroevolution algorithm can outperform a traditional reinforcement learning algorithm when applied on a task of training a car to drive around various circuits in terms of training time, maximum and average speed reached as well as their lap times. The results of different metrics collected both during training and after the training during play mode showed that while the neuroevolution is capable of finding the optimal solution extremely faster than reinforcement learning, the solution found by reinforcement learning performs better during play mode. The speeds that were reached by the model trained with the reinforcement algorithm as well as the lap times were consistently better than the ones reached by the model trained with the neuroevolution algorithm. However, further statistical analysis of the gathered results is required in order to be able to draw definitive conclusions whether the differences between the algorithms are significant.

Due to some inconsistencies in the implementation, neither of the two algorithms managed to solve the most complex circuit that was presented to them. Potential solutions to this problem as well as biases caused by differences in the topologies of the neural networks and the way algorithms were awarding their agents are going to be presented in the following section. There is definitely room for improving both algorithms either by doing minor adjustments to the parameters of the algorithm or by introducing more complex and robust techniques such as NEAT or recurrent neural networks.

8 Future works

The following paragraphs are going to describe what would be the necessary steps to take in the future of this project.

8.1 Brakes

The very first step at improving the performance of the algorithms in the current state of the project would be to introduce one more action to the agents and that would be braking. By doing so, the agents would need to learn how and when to apply the brakes properly which could possibly extend the training times. However, it would most definitely benefit the agents in the long run as they would then be able to control their speed more precisely and achieve even better lap times and high speeds. Braking would be also an essential part for conquering Circuit 3 with various hills and descents.

8.2 Recurrent NN

Next technique that would need to be implemented and evaluated would be recurrent neural networks. Recurrent neural networks apply to each layer of the network weights from all the other neurons in the network instead of just the ones from the previous layer. By doing so, agent is capable of remembering several past observations and therefore understanding the concept of time better. A network is considered fully recurrent when every single neuron in the network is fed input from every other neuron in the network [17].

8.3 Evaluate generalization

Another interesting property of the models that should be evaluated is how well they adapt to new environments. This process is called generalization and it is greatly essential for a network to be able to learn just enough patterns during training so that it can use this information when faced with completely new and unknown environments. Models that fail to generalize properly very often suffer from overfitting [5]. In this project, a model could be trained on Circuit 2 which has medium difficulty and the trained network could then be applied to cars driving on Circuit 1 and Circuit 3 and their performance would be evaluated.

8.4 Neuroevolution for reinforcement learning

One way to possibly improve the performance of the RL algorithm could be to combine it with NE algorithm which could take care of optimizing the neural network of the RL. Since NE has proved to be a great exploratory method, combining it with a

traditional method of RL which is capable of producing very robust and effective models could lead to very interesting findings.

8.5 NEAT

Last but not least, a more complex version of the NE algorithm such as already mentioned NEAT could be implemented [27]. By using an approach that instead of only training the weights also updates the structure of the network, new and perhaps more optimal topologies could be discovered that would be even more suitable for solving the task of a self-driving car.

References

[1] AJTech. Self Driving Car Series. `https://github.com/AJTech2002/Self-Driving-Car-Series`, 2020. Retrieved 28/04, 2020.

[2] Maruan Al-Shedivat, Trapit Bansal, Yuri Burda, Ilya Sutskever, Igor Mordatch, and Pieter Abbeel. Continuous adaptation via meta-learning in nonstationary and competitive environments. *arXiv preprint arXiv:1710.03641*, 2017.

[3] Michael O Duff. Q-learning for bandit problems. In *Machine Learning Proceedings 1995*, pages 209–217. Elsevier, 1995.

[4] EYEmaginary. Car AI Tutorial (Unity 5). `https://www.youtube.com/watch?v=o1XOUkYUDZU`, 2016. Retrieved 10/05, 2020.

[5] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[6] Faustino Gomez and Risto Miikkulainen. Learning robust nonlinear control with neuroevolution. Technical report, Technical Report AI01-292, Department of Computer Sciences, The University . . . , 2001.

[7] Faustino J Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, pages 1356–1361, 1999.

[8] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*, 2018.

[9] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[10] D. Jallov, S. Risi, and J. Togelius. Evocommander: A novel game based on evolving and switching between artificial brains. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(2):181–191, 2017.

[11] Daniel Ingmer Jallov. Evolve-introducing a novel game mechanic based on the indirect control of evolving neural networks. 2014.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] P Kshirsagar and N Rathod. Artificial neural network. *International Journal of Computer Applications*, pages 12–16, 2012.

[14] Sebastian Lague. Bézier Path Creator. https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082, 2019. Retrieved 10/05, 2020.

[15] Vijini Mallawaarachchi. Introduction to genetic algorithms. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3, 2017. Retrieved 21/05, 2020.

[16] Terry Lingze Meng and Matloob Khushi. Reinforcement learning in financial markets. *Data*, 4(3):110, 2019.

[17] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[18] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[20] David E Moriarty and Risto Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine learning*, 22(1-3):11–32, 1996.

[21] Jorge Muñoz, German Gutierrez, and Araceli Sanchis. A human-like torcs controller for the simulated car racing championship. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 473–480. IEEE, 2010.

[22] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[23] Frank Seide, Gang Li, and Dong Yu. Conversational speech transcription using context-dependent deep neural networks. In *Twelfth annual conference of the international speech communication association*, 2011.

[24] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[26] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[27] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[28] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[30] Unity Technologies. Unity. `https://unity.com`, 2020. Retrieved 31/05, 2020.

[31] Unity Technologies. Unity ML-Agents Toolkit. `https://github.com/Unity-Technologies/ml-agents`, 2020. Retrieved 25/05, 2020.

[32] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[33] Mark Wittkamp, Luigi Barone, and Philip Hingston. Using neat for continuous adaptation and teamwork formation in pacman. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 234–242. IEEE, 2008.

Appendices

```yaml
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e5
    memory_size: 128
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 10000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99

CarLearning:
    summary_freq: 5000
    time_horizon: 128
    batch_size: 128
    buffer_size: 2048
    beta: 1.0e-2
    max_steps: 9.0e6

CarLearning2:
    summary_freq: 5000
    time_horizon: 128
    batch_size: 128
    buffer_size: 2048
    beta: 1.0e-2
    max_steps: 9.0e6
```

Figure 27: Entire configuration used for training with the RL algorithm