Neuroevolution vs Reinforcement Learning for Training Non Player Characters in Games: The Case of a Self Driving Car

Kristián Kovalský(✉) and George Palamas

Aalborg University, A. C. Meyers Vænge 15, 2450 Copenhagen, Denmark
aau@aau.dk
https://www.cph.aau.dk

Abstract. The aim of this project is to compare two popular machine learning methods, a non-gradient-based algorithm such as neuro-evolution with a gradient-based reinforcement learning on an irregular task of training a car to self-drive around 3D circuits with varying complexity. A series of 3D circuits with a physics based car model were modeled using the Unity game engine. The data collected during evaluation show that neuro-evolution converges faster to a solution when compared to the reinforcement learning approach. However, when the reinforcement learning approach is allowed to train for long enough, it outperforms the neuro-evolution in terms of car speed and lap times achieved by the trained model of the car.

Keywords: Neuroevolution · Reinforcement learning · Neural network · Evolutionary algorithm · Autonomous systems · Self driving car · Unity · Games · Non player character · NPC

1 Introduction

Autonomous systems are capable of observing and evaluating a situation on a complex and unstructured environment [3] 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 [2]. In the field of entertainment, machine learning (ML) is capable of defining game logic and mimicking the actions and behaviour of real players. Artificial intelligence (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 [30] to a more complex examples such as neural network (NN) controlled drivatars in the racing series of Forza games [18]. However, some of the most powerful AI bots such as AlphaGo developed by DeepMind require several days of training on an extremely powerful hardware [23].

N. Shaghaghi et al. (Eds.): INTETAIN 2020, LNICST 377, pp. 191–206, 2021.
https://doi.org/10.1007/978-3-030-76426-5_13

The issue with traditional ML algorithms that use gradient-based learning such as back-propagation is that they only work well when presented with big enough training data and sufficient computing resources [13,16,21]. The solution to this might be a global optimizer such as the evolutionary algorithm. Neuroevolution (NE) is an evolutionary approach that uses a genetic algorithm (GA) for optimizing a set of weights describing a NN instead of using stochastic gradient descent methods to train these weights. Several studies have shown that there are cases in which NE can outperform traditional ML algorithms such as reinforcement learning (RL) [9,17,24].

This assumption is going to be investigated in this paper by testing the performance of a NE algorithm against a traditional RL algorithm. A simulation is going to be performed on a series of 3D racing circuits, in Unity3D game engine, with a task of training a car to autonomously drive around these circuits. Their performance will be compared in terms of training time, average and maximum speed the car reaches on these circuits and average and shortest lap time.

2 Background

2.1 Reinforcement Learning

There are three main components of every reinforcement learning algorithm: *agent*, *environment* and *reward*. 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 [27].

Agent always needs to consider the most immediate reward it will receive and also what is the next state it will go into. RL agents are usually aiming to achieve the highest long-term reward 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 periods [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 multiplied by a variable called *discount factor* that is predefined by the developer (between 0 and 1). 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 [27].

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

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

Another problem associated with RL is the one of exploration versus exploitation dilemma [31]. 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. There have been several approaches attempting to solve this issue, however, none of them are performing consistently on every possible task [4].

Last but not least, RL algorithms have to also deal with the problem of over-fitting. 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 over-fitting are either that 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 be just noise [5].

2.2 Neuro-Evolution

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 back-propagation, 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 [12,24]. Neuro-evolution algorithm follows the basic steps of the genetic algorithm as seen on the Fig. 1 where the genomes are weights of a NN [29].

Because of the nature of NE, 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 [6,7].

Genetic Algorithms. Genetic algorithms are a part of bigger group referred to as *evolutionary algorithms* that belong to the class of *evolutionary computation*. The original genetic algorithm was first introduced by John Holland in 1960s [10]. At the beginning of every GA, a random set of possible solutions with high variability, called initial population, is generated. The algorithm then assigns a fitness value to each member of the population based on how well it is suited as an optimal solution to the current problem. Reproduction promotes the survival of the fittest through a selection mechanism which favours solutions with higher

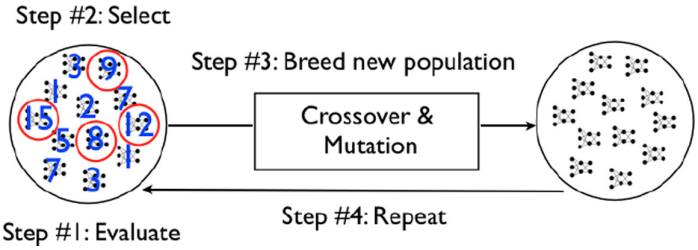

Fig. 1. The basic neuro-evolution loop. The GA is used to evolve both neural network topologies and weights [29].

fitness values [15]. A mutation operation ensures that the offsprings will be significantly different from their parents, thus avoiding local minima stemming from a premature convergence to a solution.

2.3 State of the Art

Deep Neuro-Evolution. In their paper, Such et al. [26] describe how a non-gradient-based evolutionary algorithm can replace already established gradient-based learning algorithms 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, against a set of Atari games. The GA was performing very well when compared to DQN, A3C and ES. 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. What is interesting is that the GA was able to find a better solution to many games than DQN much quicker. Afterwards, the GA was tested against RS to confirm that the GA is doing something more than just plain random search. The results showed that GA outperformed RS in every single game [26].

Neuroevolution of Augmented Topologies (NEAT). This was first presented by Stanley and Miikkulainen in 2002 [25]. The idea behind NEAT is 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 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, hidden layers are added through the mutation and crossover processes of the GA [25].

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. In terms of the first task, NEAT produced very satisfactory results without

any trouble. The networks produced very minimal topology. 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 [17,25].

Neuroevolution as Game Mechanics. A notable example of a practical application of NEAT is the game called EvoCommander developed by Jallov et al. [12]. In EvoCommander, NE is used to evolve 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. There are several behaviours that players need to train their agent to do first, such as ranged attacks, melee attacks, fleeing, etc. [12]. This approach, called "brain switching", showed that the players found the game mechanics engaging in both single-player and multi-player game modes [12].

Reinforcement Policy Learning. One of the finest examples of RL is the bot trained to play the game of Go [22]. The game has been notoriously difficult for AI to master. However, by combining supervised learning and reinforcement learning, AlphaGo has been able to reach win-rate of 99,8% by winning 494 of 495 games played against other computer Go programs that were considered as one of the strongest at that time [22]. Jaderberg et al. [11] used 3D game Quake III Arena to concurrently train multiple independent RL agents. They demonstrate how a RL agent can achieve human-level performance by training on pixels and game score as inputs. Pan et al. [20] proposed a novel method for transitioning from virtual space to real space when it comes to developing driving policy learning with RL, and show promising results of the RL adapting to real world driving. A project by Haarnoja et al. [8] also aimed to address the issue with transitioning from digital simulation space to real world space. Their approach was able to train a real-world Minitaur robot to learn a pattern of steps in order to be able to walk and generalize without issues. Moreover, different NE controllers, based on the concept of pro-prioception, have been compared for efficiency in balancing 3D biped characters in the complex and dynamic environment of a game [1].

All of the above indicate 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.

3 Experimental Setup

The entire project was created using Unity3D game engine. The circuits were created by using the Bézier Path Creator asset from Unity Assets Store. 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 model consists of a simple 3D car with four wheel colliders that are used for controlling the speed and steering.

The problem of exploitation over exploration also appeared in this project where the agent ended up finding parts of the circuit that were wide enough for the car to turn around and drive back to the start where it would turn around again and head back, thus creating a policy that kept driving in a small loop. This problem was solved by adding checkpoints to the circuits used for the training of the RL algorithm. Adding a sequence of sub-goals has been an efficient method for global optimization problems such as autonomous navigation [19]. By decreasing the reward for the distance traveled and instead giving the agent a reward for driving through checkpoints, 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 in the same place of the circuit indefinitely (Fig. 2).

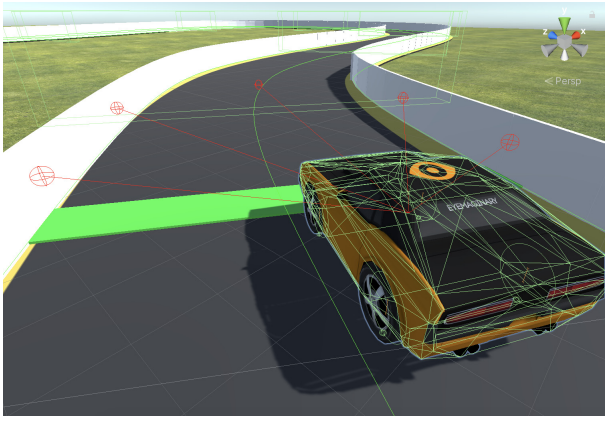

Fig. 2. The car placed on a circuit with its five ray-casts displayed for debugging purposes. Bright green line in front of the car is the start/finish line (Color figure online)

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. 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 as long and fast corners to slow and almost 180° hairpins (Fig. 3).

The third and last circuit (Circuit 3) is the most complex. There are only four turns but the first part of the circuit has a variety of successive hills with a steep inclines and declines (Fig. 4).

3.1 Neuroevolution of an Autonomous Car Controller

For this simulation, a standard NE algorithm was used instead of more complex such as the NEAT. The reason behind this was the fact that the aim of this

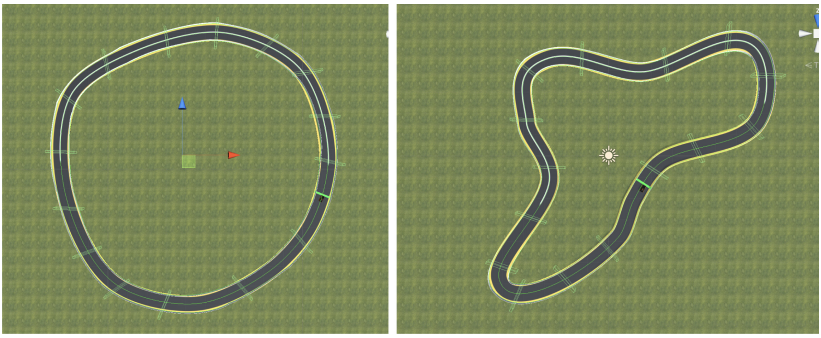

Fig. 3. The first (left) and second (right) circuits used for the evaluation. Checkpoints used for RL algorithm are visible as hollow rectangles with green outline only for visualization purposes. Bright green line marks the start/finish line (Color figure online)

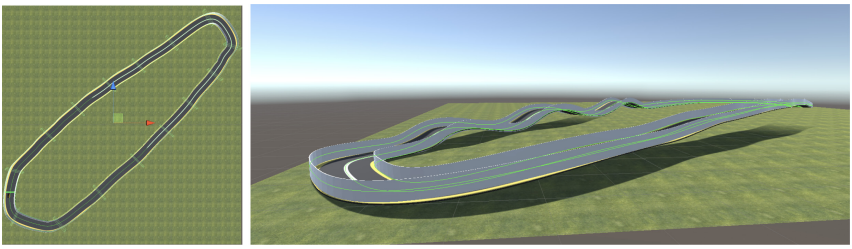

Fig. 4. The third circuit used for the evaluation. Checkpoints used for the RL algorithm are visible as hollow rectangles with green outline only for visualization purposes. Bright green line marks the start/finish line. The right image shows various levels of elevation (Color figure online)

project was to compare training approaches. Both NE and RL use different methods for optimizing the weights of a NN. By using NEAT, the topology of the NN would be changed during the process as well, possibly creating unwanted biases.

The car controller 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 ray-casts on the car and lastly a method for calculating the fitness of the car.

The fitness function is based on 3 variables: the distance traveled by the car, average speed of the car and distance readings from the ray-casts. Each of these variables has also their own multiplier that makes it possible to assign higher or lower importance to certain variables. 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 output of the NN are 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. 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. Training settings of the NE can be seen in Table 1.

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

3.2 Reinforcement Learning of an Autonomous Car Controller

The Unity Machine Learning Agents Toolkit *ML-Agents*, developed by Unity Technologies, was used for this implementation [28]. 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 ray-casts that are cast from the middle of the car forward and to the sides at 30° angle steps. 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 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 the motor torque of the wheels. 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° to the left and +45° to the right.

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. 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° from its initial rotation in order to support exploration and introduce some variation. Training settings of the RL can be seen in Table 2.

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

3.3 Data Collection

Training times and rewards/fitness scores were collected at the end of training session. The trained models were then used to drive around the same circuit in order to collect additional measurements. During this secondary data collection, the car was first let to complete one full lap in order to acquire some speed. At the start of the second lap, data about the car's speed and elapsed time started being recorded. The car was then let to drive for three more laps.

4 Results

First of all, it should be mentioned that neither NE nor RL algorithms were able to complete the Circuit 3. 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 s. The condition for successful training was met on average during generation 7. The left graph in Fig. 5 displays the progression of the fitness function during one of the fastest runs that performed well already during the second generation. The average training time of the NE algorithm on Circuit 2

was 1 min and 21 s. The condition for successful training was met on average also during generation 7. The right graph in Fig. 5 displays run that found the right solution quite soon during generation 4. When it came to training with the RL algorithm, the criteria for successful training on Circuit 1 was met after 8 h and 25 min. The model went through 3950000 steps in order to reach the solution. The top graph in Fig. 6 displays how the entropy of the model was changing. It can be seen that the randomness of the choices was generally decreasing during the training. The bottom graph displays the increasing rewards achieved by the agent at certain steps. Training the model with the RL algorithm on the more complex Circuit 2 took 15 h and 10 min. The model completed 7050000 steps until the right policy was found. The top graph in Fig. 7 shows slightly unstable entropy in the first half of the training, however it starts to decrease more stably in the second half. The bottom graph shows the overall increase of the accumulated rewards of the agent, mainly in the second half of the training period.

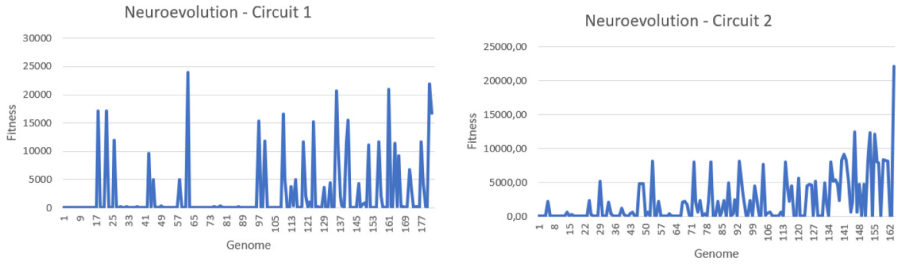

Fig. 5. Fitness scores across genomes during training of the NE on the first (left) and second (right) circuit

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 s. The average time after three laps was 20.16 s. Additionally, as it can be seen on Fig. 8 (top), the maximum speed reached on Circuit 1 was 27.57 and the average speed was 20.83. When the model trained with RL algorithm was tested on Circuit 1 it achieved the best lap time of 15.66 s and the average lap time of 17.15 s. Moreover, the top speed on Circuit 1 was measured at 30.57 with the average speed over all frames being 25.66 (see Fig. 8 bottom).

The NE model trained on Circuit 2 achieved the best lap time of 31.18 s and the average lap time after three laps was 33.63 s. The top speed reached on Circuit 2 was 15.35 and the average speed during three laps was 12.47 (see Fig. 9 top). The RL trained model on Circuit 2 achieved the best lap time of 32.01 s and the average lap time over three laps was 32.82 s. The maximum speed the car reached on Circuit 2 was 18.29 with the average being 12.52 (see Fig. 9 bottom). Now that all the raw data were presented, it allows for their discussion and reasoning of why certain algorithms behaved the way they did.

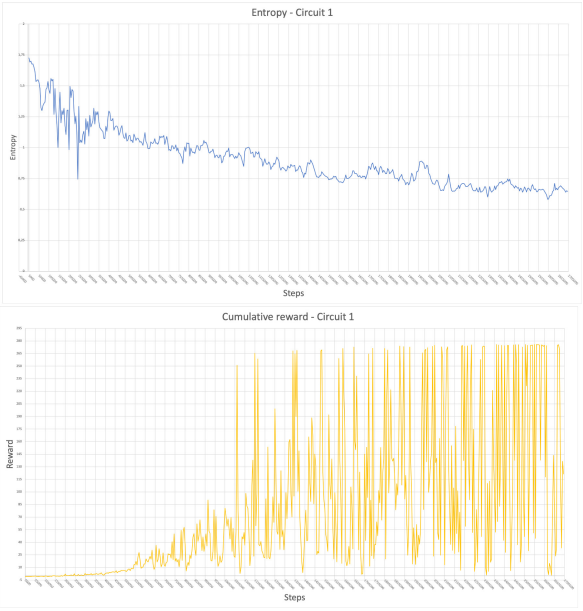

Fig. 6. Top graph shows the change in entropy during training of the RL model on the first circuit while bottom graph shows the change in accumulated reward

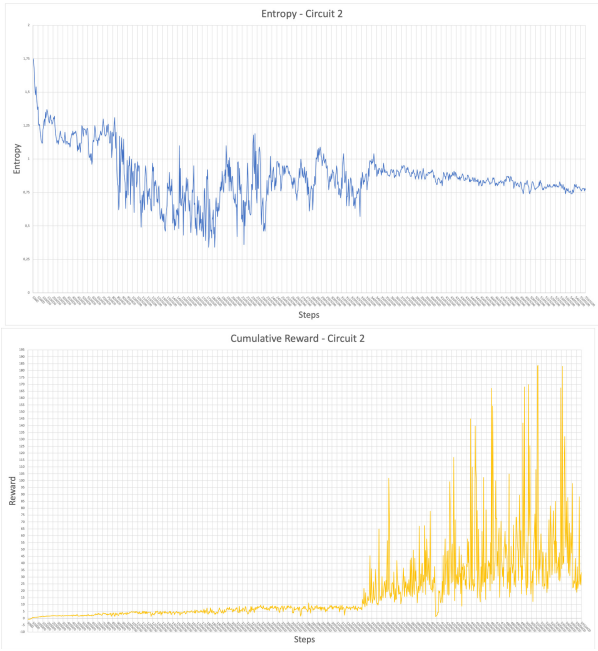

Fig. 7. Top graph shows the change in entropy during training of the RL model on the second circuit while bottom graph shows the change in accumulated reward

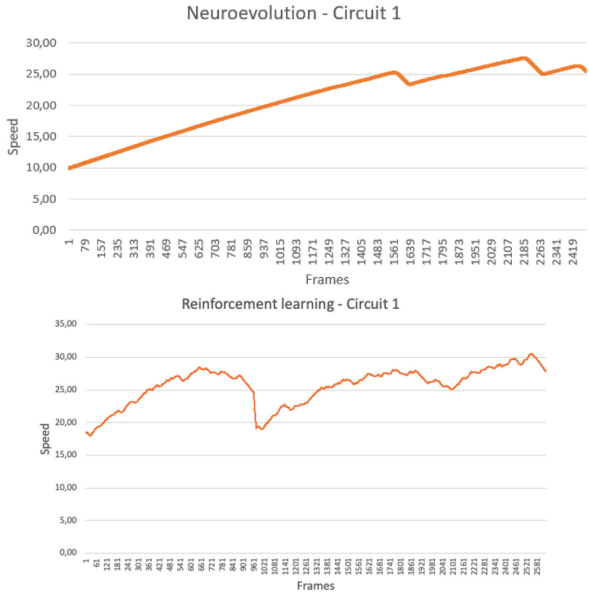

Fig. 8. Speed of the NE (top) and RL (bottom) trained car models at each frame during three laps in play mode on the first circuit

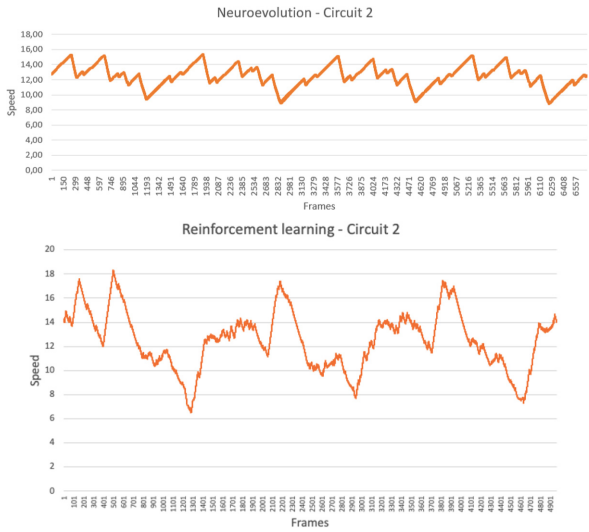

Fig. 9. Speed of the NE (top) and RL (bottom) trained car models at each frame during three laps in play mode on the second circuit

5 Discussion

The difference in training times is quite obvious. 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 s on average compared to 8 h and 25 min of the RL. Training of the NE on Circuit 2 took 1 min 21 s on average compared to 15 h 10 min of the RL. The NE algorithm, through complete accident, happened to 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 Sect. 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 s to complete a lap compared to 17.15 s 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 s and the average lap time of the RL model was 32.82 s, making the difference between them less than 1 s. The difference is even smaller for average speed, where NE model reached value of 12.47 and RL model reached value of 12.52.

5.1 Biases

There are several factors that might have affected the way the results turned out for both training and play parts of the algorithms. The difference in topologies of the NNs used in both algorithms could have caused an unfair advantage of one over the other. 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. Figure 5 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 ray-cast readings. On the other hand, the RL agents gets higher positive reward the faster it is moving forward and passing through checkpoints. 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, the acceleration action of both algorithms was constrained to be always within the realm of positive numbers. This means that the agents were basically told that moving forward is the right and the only direction they should be moving. If the agents would be able to reverse, the results would almost definitely look different. However, 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.

6 Conclusion

The aim of this project was to see whether a neuro-evolution 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 neuro-evolution is capable of finding the optimal solution much 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 neuro-evolution algorithm.

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.

7 Future Works

The very first step at improving the performance of the algorithms would be to introduce braking to the agents. This could possibly extend the training times, but it would most definitely benefit the agents in the long run and help in conquering Circuit 3. Next step would be to introduce a recurrent neural networks. Recurrent neural networks are capable of remembering several past observations and therefore they can deal better with temporal series of events [14]. Another property that should be evaluated is how well the trained models adapt to new environments. Generalization is greatly essential for a network when faced with completely new and unknown environments. Models that fail to generalize properly very often suffer from over-fitting [5]. Last but not least, a more complex version of the NE algorithm such as NEAT could be implemented [25].

References

1. Carlsen, C.S., Palamas, G.: Evolving balancing controllers for biped characters in games. In: Rojas, I., Joya, G., Catala, A. (eds.) IWANN 2019. LNCS, vol. 11507, pp. 869–880. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20518-8_72
2. Chen, S., Zhang, S., Shang, J., Chen, B., Zheng, N.: Brain-inspired cognitive model with attention for self-driving cars. IEEE Trans. Cogn. Dev. Syst. **11**(1), 13–25 (2017)
3. Cui, Y., Ge, S.S.: Autonomous vehicle positioning with GPS in urban canyon environments. IEEE Trans. Robot. Autom. **19**(1), 15–25 (2003)
4. Duff, M.O.: Q-learning for bandit problems. In: Machine Learning Proceedings 1995, pp. 209–217. Elsevier (1995)
5. Géron, A.: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, Sebastopol (2019)
6. Gomez, F., Miikkulainen, R.: Learning robust nonlinear control with neuroevolution. Technical report, Technical Report AI01-292, Department of Computer Sciences, The University (2001)
7. Gomez, F.J., Miikkulainen, R.: Solving non-Markovian control tasks with neuroevolution. In: IJCAI, vol. 99, pp. 1356–1361 (1999)
8. Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., Levine, S.: Learning to walk via deep reinforcement learning. arXiv preprint arXiv:1812.11103 (2018)
9. Hausknecht, M., Lehman, J., Miikkulainen, R., Stone, P.: A neuroevolution approach to general atari game playing. IEEE Trans. Comput. Intell. AI Games **6**(4), 355–366 (2014)
10. Holland, J.H.: Genetic algorithms: computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand. Sci. Am. **267**, 1992 (2005)
11. Jaderberg, M., et al.: Human-level performance in 3D multiplayer games with population-based reinforcement learning. Science **364**(6443), 859–865 (2019)
12. Jallov, D., Risi, S., Togelius, J.: EvoCommander: a novel game based on evolving and switching between artificial brains. IEEE Trans. Comput. Intell. AI in Games **9**(2), 181–191 (2017)
13. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
14. Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Eleventh Annual Conference of the International Speech Communication Association (2010)
15. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
16. Mnih, V.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
17. Moriarty, D.E., Mikkulainen, R.: Efficient reinforcement learning through symbiotic evolution. Mach. Learn. **22**(1–3), 11–32 (1996). https://doi.org/10.1023/A:1018004120707
18. Muñoz, J., Gutierrez, G., Sanchis, A.: A human-like TORCS controller for the simulated car racing championship. In: Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, pp. 473–480. IEEE (2010)

19. Palamas, G., Ware, J.A.: Sub-goal based robot visual navigation through sensorial space tesselation. Int. J. Adv. Res. Artif. Intell. **2**(11), (2013)
20. Pan, X., You, Y., Wang, Z., Lu, C.: Virtual to real reinforcement learning for autonomous driving. arXiv preprint arXiv:1704.03952 (2017)
21. Seide, F., Li, G., Yu, D.: Conversational speech transcription using context-dependent deep neural networks. In: Twelfth Annual Conference of the International Speech Communication Association (2011)
22. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484 (2016)
23. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
24. Stanley, K.O., Clune, J., Lehman, J., Miikkulainen, R.: Designing neural networks through neuroevolution. Nat. Mach. Intell. **1**(1), 24–35 (2019)
25. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002)
26. Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O., Clune, J.: Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567 (2017)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
28. Unity Technologies: Unity ML-Agents Toolkit (2020). https://github.com/Unity-Technologies/ml-agents. Accessed 25 May 2020
29. Whiteson, S.: Evolutionary computation for reinforcement learning. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, vol. 12, pp. 325–355. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27645-3_10
30. Wittkamp, M., Barone, L., Hingston, P.: Using neat for continuous adaptation and teamwork formation in pacman. In: 2008 IEEE Symposium On Computational Intelligence and Games, pp. 234–242. IEEE (2008)
31. Yogeswaran, M., Ponnambalam, S.: Reinforcement learning: exploration-exploitation dilemma in multi-agent foraging task. Opsearch **49**(3), 223–236 (2012). https://doi.org/10.1007/s12597-012-0077-2