Neuroevolutionary Training of an Autonomous Racecar

Pouya Tobias Strand Nikoui s165946, Marius Cristian Mic s193282, Francisc Vlad Loghin s193204

January 4, 2021

Abstract

Darwin inspired Neuroevolutionary methods are implemented to accelerate the training of an autonomous racecar. Curriculum learning is applied iteratively to improve the ability of the agent to navigate more difficult environments. It was found that the performance of the agents was highly sensitive to the applied penalty functions, which are responsible for the fitness determination of the subsequent generations. Furthermore, it was found that the training improved when put in context of previous states which were provided by a LTSM network.

I. Introduction

Self-driving cars are a major area of research from both universities and the private sector nowadays. They will revolutionize transportation in a similar way that the first Ford cars did in the 19^{th} century. Some of the important areas where they will have an impact are road safety, where driver behaviour or error are a factor in 94 percent of crashes, higher independence for disabled people, reduced congestion and environmental benefits [1].

Two methods of achieving self-driving cars are backpropagation-based reinforcement learning algorithms (RL) and genetic algorithms (GA). This report will focus on GAs in combination with curriculum learning. The basis for this project was provided Deloitte Denmark with whom we have collaborated throughout the project.

II. Theoretical Background

A. Genetic Algorithms

1) How Genetic Algorithms work

Genetic Algorithms are methods for solving optimization problems that draw inspiration from Darwinian Evolution and are used to design artificial intelligence and machine learning applications. They start from a set of randomly generated agents and then, after observing each agent's performance, the algorithm selects the most successful ones to "pass on their genes" and be the basis for future agents. The agents in this case are the individual cars, and each of them having a neural network model with some weights. The selection process happens repeatedly until, after evaluating multiple iterations, the models converge towards an optimal solution.

Each iteration of the algorithm is called a generation. Each agent inherits its solution from its parents, which are combined randomly. In each generation the algorithm will select two parents that perform the best and use their chromosomes to create children. Children are then multiplied and mutations are applied randomly to all children, at random times. This adds stochasticity and thus more variety in the "gene pool". This process is visualised in Figure 1.

Figure 1: The process of evolution

2) Genetic Algorithms vs. Traditional Reinforcement Learning

Both GA and RL can be used to solve similar tasks, they both use neural networks to create a mapping between a state and an action. However, in RL the neural network's weights are being updated by the backpropagation algorithm and using a loss function. In a GA, the weights are not being updated directly. Instead of slowly incrementing a weight to converge towards the desired one, the GA just takes the agents with the best weights and uses them as a basis for the next generation. It prioritizes selection instead of refinement. Genetic algorithms are a strong contender to backpropagation-based reinforcement learning, because they are able to better avoid local minima issues due to the diversity generated by crossover and mutations. Also, they have a better mean learning time due to not needing to

compute all the derivatives necessary for backpropagation. Variance can be a problem with GAs, as they depend on the starting point of the randomly generated weights. However, Ziyi Xiang has shown that despite their higher variance genetic algorithms are well-suited for navigation tasks [2].

B. Curriculum Learning

In school children learn following a highly organized curriculum that is meant to build on existing knowledge. Curriculum learning in the context of machine learning does exactly the same thing. The basic idea is to start small, learn easier sub-tasks and slowly increase the difficulty. Researchers have found that neural networks can learn faster with a curriculum in place [3].

For GAs the initial weights of the agents are random, and all optimization comes from a selection from those weights, therefore it is easily seen why too complex tasks can become a problem. It is reasonable to train them on easier sub-problems first in order for the agents to learn, for example, how to drive in a straight line towards the goal or to learn how to take a curve. Once those are solved, it can focus on higher-level problems.

For this project the training starts off with a zero-turn map. It saves the final weights and loads them to the agents as a starting point for a one-turn map, then a two-turn map and so on. This way the simpler features can be learned in the easier maps and transferred to the more complex ones.

C. Reward System

A crucial part of any reinforcement learning algorithm, including GAs, is the reward system. The agent needs to know whether the actions taken are right or wrong, and since the learning is unsupervised there is no default good or bad. The developer needs to create a reward system that will push the agent in the right direction. This is a part that can make or break a reinforcement learning model. Of course there is a primary reward, such as hitting the goal in our case. But with only that in place it will take the agent many iterations to converge, if ever. But with a robust reward system it can have rules of thumb to guide it through the learning process. In the following paragraphs the rewards used in this project are presented. They are called penalties here, because they are applied negatively.

To start off, the best score an agent can get is 0. That means it hit the goal, within the maximum amount of steps. Hence the problem becomes about minimizing the bad penalties. The fitness is thus the total accumulated penalty and the child generation is then selected by ranking the results from least to greatest from a chosen threshold of parent agents. This threshold is denominated as the top limit and was chosen to be ten for this project. The penalty functions act as the sorting criteria for which agents are passed on the next generation and therefore have a significant effect on the performance of the system.

III. Methodology

A. Environment

A virtual environment was created using OpenAI's gym package in combination with code provided by Deloitte Denmark. The language is Python. The environment defines the main functions which call upon the simulation and racetrack. The simulation is based on simple kinematic equations of motion and the racetrack is created using a library called shapely. The genetic algorithm script contains the training function which calls upon the crossover and mutation functions. The algorithm was modified to use curriculum learning, whereby weights are saved to a dictionary and loaded as the starting point for the next task. Seeding has been utilized to introduce stochasticity during training by randomizing the map orientations. The agents are then run, evaluated, and their rewards sorted for the following generation [4].

The software was set to run 100 agents 3 times per generation giving each one a maximum of 250 steps to hit the goal. The training was consistently run for a total of 51 generations, adding up to a maximum of 15300 goals to be hit. This value was used to attain a success rate used to qualify the performance. The key hyper-parameters of the experiment are stated in Figure 2.

Number of Agents	Number of Generations	Crossovers	Mutation Power	Top Limit
100	51	20	0.1	10

Figure 2: Hyperparameters of the algorithm

B. Model Architecture

Various model architectures and methods were experimented with in order to observe their effect on the training process. The two models that performed best are shown below. The original model architecture consisted of four fully connected layers each with a ReLU activation function on the output. An LTSM network was later implemented; both models are shown in Figure 3.

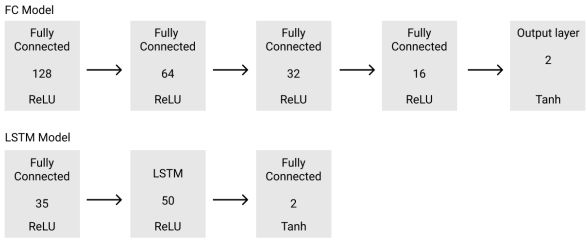


Figure 3: Fully-connected vs. LSTM architecture

C. Penalty functions

In this report penalties have been utilized to determine whether the subsequent state following a given action is desirable. Agents are originally penalized for either hitting a wall or neither hitting a wall or the goal. This penalty is represented in Equation 1. Additional penalties have been implemented and will be detailed in this section.

1) Distance Penalty

Firstly, the distance based penalty is applied if the agent's action results in a distance further from the goal the previous position. A very simple function called closer, which takes two distances as inputs, was used to determine whether the reward would be penalized. This function is described in Equation 2.

$$R_f = R_i + (Goal_x - Car_x)^2 + (Goal_y - Car_y)^2 \cdot 10 \cdot 3 \quad (1)$$

$$R_f = R_i + (D_{previous} - D_{update}) \cdot 10 \cdot 3 \quad (2)$$

2) Angle Penalty

The angle penalty must be introduced to account for the physical limitations of the car when testing the algorithm. The car's wheels have a maximum turning radius and therefore actions resulting in larger angles will be penalized. Additionally, the probability of hitting a wall is larger if the car turns with large angles. The correct value for the turning radius of a physical racecar varies from car to car depending on the hardware. Due unforeseen circumstances physical testing of the racecar was not possible. For the sake of the experiment, this angle was set to be 90°.

3) Lidar Penalty

The lidar penalty was initially implemented to penalize the agent from deviating from the center of the racetrack. In theory, however, it is not problematic if the car deviates from the center of the track as long as it completes the task. Therefore, it was attempted instead to keep the car parallel to the walls by using the sensor measurements. The simulated lidar measurements were generated via intersections with the shapely geometry used to construct the track. Thirty measurements are taken in total over a 270° angle which corresponds to increments of 9° each time. It was then calculated that the 5^{th} and 25^{th} values correspond to the left and right perpendicular lidar rays. The difference between the left and right lidar rays should be zero when the car is perfectly in the center of the track. A threshold of 0.3 was then introduced to select agents with minimal differences.

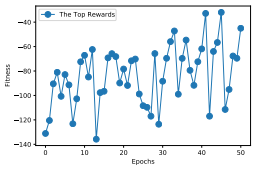
4) Step Penalty

Finally, a penalty was added based on the amount of steps taken by the agent in a given run. The idea was to penalize bad momentum, meaning that if the amount of steps taken by the agent decreased every run it would receive a penalty. This was done programmatically by predefining a zero vector the length of the number of runs and then simply tallying each run. Conditional statements were utilized to check the momentum of the agents' runs.

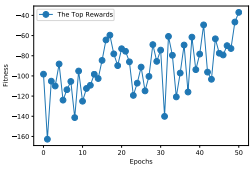
IV. Results

A. FFNN

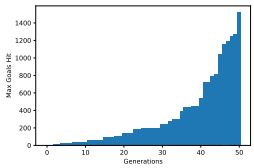
The top rewards from running the original model without any modifications can be seen in Figure 4, sub-figure 4a. The effect of the addition of a distance based penalty function is then seen in sub-figure 4b. The corresponding goal tallies are presented in sub-figures 4c and 4d, respectively. It is important to mention that the goal tallies are cumulative. With the number of goals hit, it can be seen more accurately that with the distance penalty the agents manage to get over 2000 goals, while without it they only go over 1400. It is important to mention that these results are on 0-turn maps, which are essentially straight lines, except sub-plots 4e and 4f.



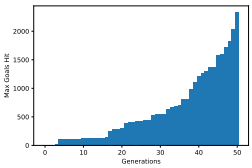
(a) Fitness without modification



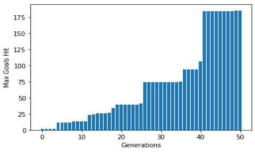
(b) Fitness with distance penalty



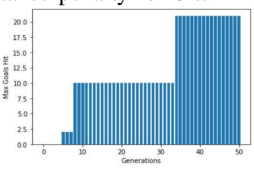
(c) Goal tally without modification



(d) Goal tally with distance penalty for 0-turn



(e) Goal tally with distance penalty, for 1 turn



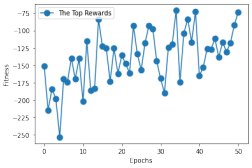
(f) Goal tally with distance penalty for 2 turns

Figure 4: Original FFNN

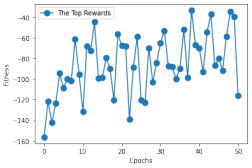
B. LTSM

Following the same procedure as with the previous model, the results obtained by the LTSM network is seen in Figure 5. The figure contains graphs for 0-turn maps and 1-turn maps.

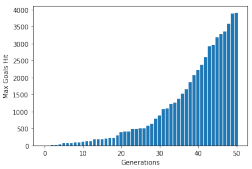
Comparing the LTSM model with the FFNN, it can be seen that the total numbers of goals hit is much higher in Figure 5c then in Figure 4d. They are both for 0-turn maps so the comparison holds. The LTSM hit around 4000 goals in total while the FFNN only hit around 2000. Therefore the LTSM became the model of choice from then on.



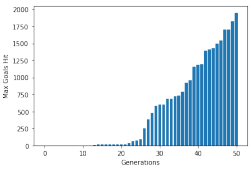
(a) Fitness 0-turn



(b) Fitness 1-turn



(c) Goal tally 0-turn

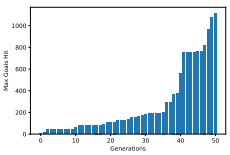


(d) Goal tally 1-turn

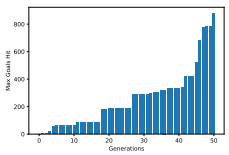
Figure 5: LTSM

C. Effect of Hyperparameters

The hyperparameters selected during training were investigated to observe their effects on the performance. In Figure 6, a comparison of a different amount of crossovers is made, where 6a had 40 and 6b had 80 crossovers.



(a) 40 Crossovers 0-turn

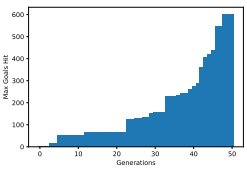


(b) 80 Crossovers 0-turn

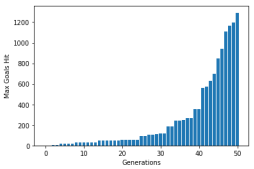
Figure 6: Effect of Crossover

D. Angle Penalty

A model was run with the angle penalty and one without in order to see the improvement. It seems from Figure 7 that the angle penalty approximately doubled the total number of goals hit from 600 to 1200. It is important however to mention that every time the GA is run, the results change a bit. That is the variance mentioned earlier.



(a) LTSM without angle penalty

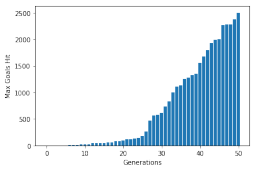


(b) LTSM with angle penalty

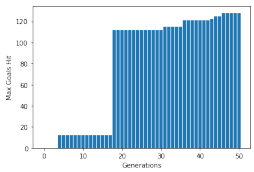
Figure 7: Angle Penalty

E. Model Generalization

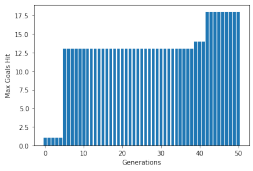
The model was tested on a map with over double the amount turns that it had been trained on. It was trained progressively on 0 to 3-turn maps, and then tested on a 7-turn one. In Figure 8a, it is seen that the model is only able to hit around 20 goals, a far cry from the approximately 150 on a 3-turn map.



(a) 1-turn



(b) 4-turn



(c) 8-turn

Figure 8: Model Generalization to higher complexity.

V. Discussion

A. Penalty Functions

The fitness determination plays a crucial role in the efficiency and effectiveness of the exploration in the search domain. Poor fitness functions can result in the solution being confined to a local minima and the loss of discovery power [5].

From Figure 4 it can be deduced that the addition of the distance penalty causes a drastic improvement to the car overall performance. From the fitness plots, one sees that the penalty reduces the variance of the rewards and delivers a faster convergence toward zero in 4a when compared with to 4b. Furthermore, the cumulative goals hit are significantly increased, seen in 4d, in comparison to Figure 4c. The success rate in either case is approximated to be 9.2% (1400 goals) and 13.1% (2000 goals) meaning that the distance penalty yielded almost a 4% increase during training. This result justifies the usage of the implemented distance penalty. It may be important to note, however, that this penalty will need to be modified for a more complex task, where the car must travel away from the goal to reach the target. In Figures 4e and 4f, plateaus begin to appear indicating that the training is stagnating and having difficulty learning when the task becomes more complex.

B. Model Architecture

When comparing Figure 4 and Figure 5 it can be seen that the LTSM outperformed the FFNN in terms of goals hit. The difference is approximately 4000 (Figure 5c) goals vs 2000 (Figure 4d), so around twice as many. This corresponds to a success rate of approximately 26% which is a substantial improvement, that likely stems from the model having information about the previous states due to the LTSMs memory capacity. Additionally, it was found that the LTSM architecture eliminated the plateaus of the original model (Figure 4e and 4f). This indicates that the new model increased the training speed, where the original model stagnated. The LTSM is well-suited for data that comes in series as it provides a context for the following data. The steps an agent takes in the track can be interpreted as a series, and thus it is concluded that the context of the previous actions improves the performance and justifies the new model architecture.

C. Physical Constraint

The angle penalty was originally intended for testing the model on a physical racecar. However, even in virtual space this turned out to be useful, as seen in Figure 7. The intuition behind this is that the car should never turn more than a certain amount, because then it has a high chance of hitting a wall. For this project, the amount was 90°. Naturally, if some turns in the track were to be sharper than this, the penalty would need to be modified to take this into account.

D. Lidar Penalty

The lidar penalty did not work as expected, which there could be several reasons for. Assuming that the penalty implementation is correct, one possibility is that the threshold value is low. This could effectively constrict the model's unpunished actions to the narrow subset of actions that remain within this range. The track is constructed to have a total width of two units (meters), one to each wall, and the wall hit flag is raised within 30 cm to the wall. The lidar threshold then leaves 40 cm of space where the agent no longer can go unpunished. This could very well be the reason that far fewer goals were hit, as the car likely enters this territory even on success runs.

Another possibility, is that the single left and right lidar values may not be accurately representing the behaviour at a given point. This may specifically be the case through turns where the distances on either side of the car are not equal. A potential resolution to that problem could be to

take a range of left and right values instead of just one for each. This in conjunction with a suitable lidar threshold should improve the results significantly.

E. Step Penalty

The step penalty did not return a significant enough increase in performance to be included. This could be because the condition was too stringent and so not many agents fulfilled the criteria. Conversely, to try to resolve this, one could try applying a bonus for the reverse situation instead. This would method would reward and highlight agents with good momentum instead of punishing bad ones.

F. Variance in GAs

As it can be seen from the fitness plots in Figure 4 and Figure 5, the GAs have quite a high variance. It is also apparent, from figures pertaining to goal hits, that the final goal tally is not consistent with every run. This is best exemplified in Figure 5c and 7a where the model is the same however the tally varies between the runs. The underlying reason for this behaviour is thought to be due to the randomization process. The seed alters the map orientation and there are thus likely to be certain orientations which prove more difficult to learn. Another possibility is that the initial randomized weights of the agents create this difference. This is something that could have been addressed better by having a much larger number of generations for each turn.

Furthermore, the hyperparameters also have an effect on the randomness of the training process. The number of crossovers, for instance, was found to decrease the overall performance when making more crossovers. The result is shown in Figure 6 and 5c for 20; there is an evident drop in performance and the chosen value for all trials was thereby 20 crossovers. This performance reduction suggests that there is a risk of introducing too many permutations to the model which could change otherwise good agents to poor ones when passed down. From this information, it is therefore assumed that the mutation power will have a similar effect. The number of agents and generations would surely have significant effects, however due to a lack of time they were kept constant.

G. Model Generalization

The results in Figure 8 show large plateaus arising demonstrating the significant reduction in learning speed. The fact that the model is able to hit any goals at all, albeit at a success rate of around 0.14%, indicates that the model is able to apply what it learned to some extent. Moreover, it was also confirmed that curriculum learning made a significant difference as the weights learned from the previous map significantly improve the success rate on higher turn maps. Improving model architecture may be necessary in order to capture the required features in the data. An idea could be to use a convolution on the simulated lidar data. For a similar problem, Trasnea et al. combine a CNN with an LTSM to learn the optimal state trajectory by extracting relevant spatial features [6].

VI. Conclusion

The goal of this project is to try a variety of GAs in order to make a virtual car navigate a maze. Several useful things were discovered such as the distance and angle penalty and the memory characteristic of the LTSM. Other things are thought to be good in principle, like the lidar penalty, but require more refinement in practice. It is important to mention that other, more complex network architectures were also tried, but with little effect.

The LTSM model with the added penalties did quite well when it was trained on few-turn maps but when it came to generalizing to a 7-turn map it encountered some problems. Avenues of exploration for future research are refining the lidar penalty, discovering more penalties if possible and experimenting with a deeper neural network trained over many more generations[1].

References

[1] S. Singh, "Critical reasons for crashes investigated in the national motor vehicle crash causation survey," 2015.
[2] D. Project, I. N. Technology, and F. Cycle, "A comparison of genetic algorithm and reinforcement learning for autonomous driving," 2019.
[3] V. Cirik, E. Hovy, and L.-P. Morency, "Visualizing and Understanding Curriculum Learning for Long Short-Term Memory Networks," pp. 41–48, 2016.
[4] P. T. S. Nikoui, "Racecar repository." https://github.com/PNikoui/Racecar, 2021.
[5] W. Fan, E. Fox, P. Pathak, and H. Wu, "The effects of fitness functions on genetic programming-based ranking discovery for web search," *Journal of the American Society for Information Science and Technology*, vol. 55, pp. 628–636, 05 2004.
[6] S. M. Grigorescu, B. Trasnea, L. Marina, A. Vasilcoi, and T. Cocias, "Neurotrajectory: A neuroevolutionary approach to local state trajectory learning for autonomous vehicles," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3441–3448, 2019.

[1]The Github repository link is the fourth reference.