Neuroevolution for Automated Driving

Mika Skjelnes and Alexander Sellström

14th December 2024

Abstract

Imitation learning is used in leading companies to employ autonomous driving. However, it requires a large amount of high-quality reference data to train, which costs time and money to acquire. Reinforcement learning does not require any input data for training, and instead relies on user-provided learning policies to train an agent to drive, therefore it should contest imitation learning in autonomous driving. Current success stories with reinforcement learning in real-life applications are however sparse, although it is popular in small-scale projects in simulated environments. Neuroevolution has been shown to outperform reinforcement learning in many problems, and should therefore be considered for autonomous driving. We present a python implementation using python-NEAT to communicate with Unity in order to train a vehicle agent to navigate a racing track with set sub-goals. Our results show that neuroevolution of augmenting topology is well suited for a simplified autonomous driver problem within the limits of a racing track, achieving good solutions in ten generations for a racing track with ninety degree turns.

Contents

1 Glossary

Cross-over: The process of combining two genomes in sexual reproduction. Gene: A unique pair of nodes. Gene pool: The union of all genes in the population.
Mutation: The process of introducing new genetic material to an individual.

2 Introduction

Recent progress in machine learning (ML) and artificial intelligence (AI) has shown to become ever more useful in society, where tasks traditionally maintained by humans now are replaced by autonomous systems, as seen in warehouse packing [5], manufacturing [15], transport [1], driving [13], among others. Especially automated driving has become a hot topic, partially due to companies like Tesla that have invested significant amounts in development in automated systems for driving their cars [9]. There is no single best approach to design an autonomous system for driving, although some have been more successful than others in certain contexts. For instance in virtual systems with a smaller network of roads, and with complete data of the surroundings of the virtual car, imitation learning has been potent [4]. In the real world, in actual cars, a combination of real-time image analysis and data from the drivers for imitation learning seems promising as that is part of what Tesla is integrating into their autopilot [19]. The main problem with this type of learning is that it requires plenty of (good) data to be generated before the autonomous system can be trained, which costs time and money. Other approaches, such as using Reinforcement Learning (RL), do not require any training data to start training therefore more time can be spent on training and tuning the system for the task. Research exists that uses RL for automated driving, but it seems that successful integration into physical vehicles like what Tesla has done is rare. Although RL is suitable for autonomous driving, there are only few success stories, and little literature with real-world autonomous driving applications [12].

Genetic algorithms (GA) is a branch of AI that focuses on developing methods for training a system through simulated evolution. It maintains a population of individuals, each corresponding a solution to the system, and performs parallel-search. The population is replaced after each generation, with bias towards individuals that were considered more fit. Neuroevolution (NE) is a class of GA that evolves artificial neural networks (ANN) and have been shown to outperform RL in many problems [18, 17]. Neuroevolution of augemented topologies (NEAT) is an extension of NE that allows the topology of the ANNs to differ, and this has lead to NEAT outperforming NE in certain benchmarks [17]. In this project we examine the idea of training an autonomous system to drive a vehicle on an enclosed race-track using NEAT. This approach requires no data to commence training, unlike imitation learning, and explores multiple areas of the search space in parallel, unlike reinforcement learning.

3 Background

NeuroEvolution of Augmented Topologies (NEAT) [17] is an extension of ***NeuroEvolution*** (NE) that evolves ***Artificial Neural Networks*** (ANN) of non-fixed topologies. NEAT solves the problem of crossing differently sized ANNs in a meaningful way.

3.1 Innovation number

Each gene in a neural network is associated with a global innovation number. These are mapped incrementally during training when a new gene is discovered in an individual. New genes can be introduced to the gene pool by mutation. Innovation numbers allow for cross between individuals of different topology to be made that preserves the structural integrity of the parents. An individual is described by a sequence of genes.

3.2 Crossover

Crossover is the search operator of NEAT, using the genes of two parents to construct a new individual. The parents have genes in common if they share genes with the same innovation number. The child will inherit any disjoint or excess genes from the fitter parent and randomly inherit the gene configuration of genes that both parents have in common.

Example Let two individuals N_1 and N_2 have genetic sequences $S_1 = [G_0,\ G_2,\ G_3,\ G_5,\ G_7]$ and $S_2 = [G_0,\ G_1,\ G_4,\ G_6]$, where G_i has innovation number i, and weights $W_1 = [0.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0]$ and $[1.0,\ 1.0,\ 1.0,\ 1.0,\ 1.0]$. For simplicity, we leave out activation function and whether a gene is activated or not. Assuming that N_1 is more fit, crossover of N_1 and N_2 yields the sequence of genes $[G_0,\ G_1,\ G_4,\ G_6]$ with weights [random(1.0,\ 0.0),\ 0.0,\ 0.0,\ 0.0], where random(A,B) randomly selects one from (A, B) with equal chance.

3.3 Mutation

Mutation consist of two mutating operations that could introduce new genetic information into the gene pool.

- ***add connection*** adds a connection between two nodes in an individual, essentially constructing a gene.
- ***add node*** adds a node n between a connection of two nodes a, b. This operation deactivates the affected gene responsible for the connection. Two genes are created a, n and n, b

NEAT-Python [14] implements NEAT in the programming language Python. It introduces two more mutating operations

- ***remove connection*** reverses the add operation.
- ***remove node*** removes a node and all connections to and from it.

Mlagents [10] is a toolkit for communicating with agents in the game engine Unity [8], and implements reinforcement learning to train an agent to drive through a track while maximising rewards. It uses the Unity Karting Microgame template [11] for training. The karting template provides with the necessary vehicles, input/output-handling, physics computation that mlagents can use to run the simulations.

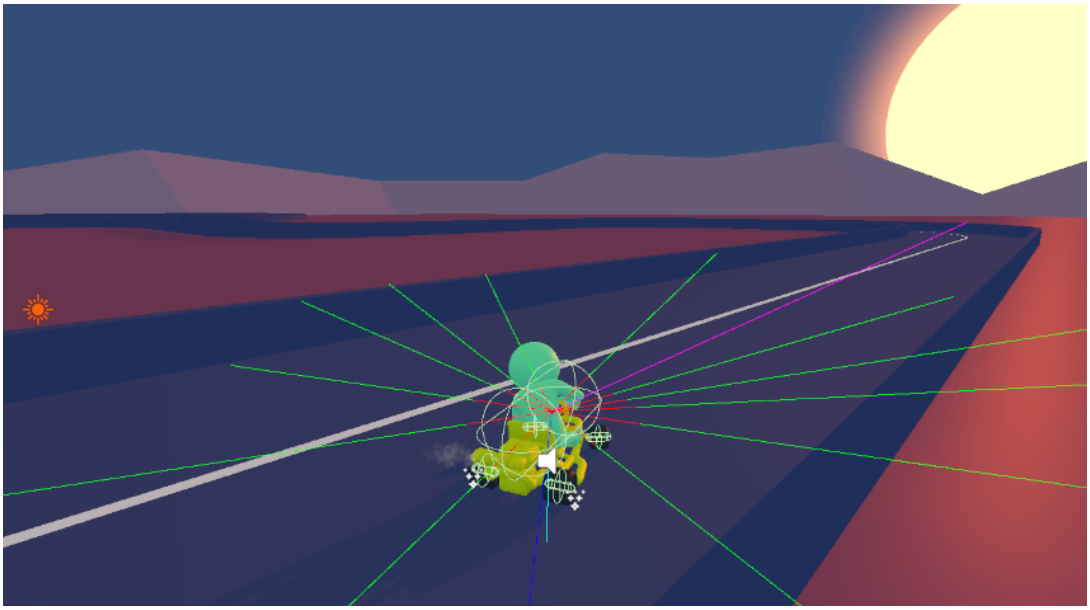

Figure 1: Visualisation of the spacial awareness of the agent.

4 Implementation

4.1 The Agent

Inputs The kart agent have 12 ray casting sensors pointing in different directions from the centre of the cart. These sensors return the ***hit distance*** of the ray casting, if within reach of the maximum hit distance of the particular ray sensor. The sensors are rotated, allowing the agent to get some idea of its surroundings, as visualised in figure 1. The distance values from the sensors are sent to the trainer, together with ***local velocity***, ***forward velocity*** and ***acceleration*** of the cart.

Outputs The available controls of the kart agent are

- Gas (Boolean)
- Reverse (Boolean)
- Left Turn (Float)
- Right Turn (Float)

These instructions can be fed to the agent by an positive integer array of size 2.

Rewards A subgoal is a region defined on the track. Rewards are computed from how the agent drives, increasing by covering distance towards the next subgoal and reaching the subgoal, and decreasing by driving away from the next subgoal and by crashing into a wall.

4.2 NEAT Implementation

NEAT is used to find an ANN that manages to navigate through racing tracks without crashing. A trained network must be able to respond appropriately to the sensory inputs so that it manages to navigate through racing tracks. It has more time to accumulate rewards if it maintains a high velocity. but it must also be able to handle the curves without crashing into the wall.

4.2.1 Hyperparameters in config

Initial network complexity of created individuals

- ***num_hidden*** is the number of hidden nodes.
- ***num_inputs*** is the number of input nodes.
- ***num_outputs*** is the number of output nodes.

Network alteration of individual during mutation

- ***node_add_prob*** is the probability that a mutation adds a node.
- ***node_delete_prob*** is the probability that a mutation removes a node.
- ***conn_add_prob*** is the probability that a mutation adds a connection.
- ***conn_delete_prob*** is the probability that a mutation removes a connection.

Gene alteration of individual during mutation

- ***weight_mutate_power*** is the ratio of which the weight can be altered in a mutation.
- ***weight_mutate_rate*** is the probability that a mutation alters a weight.
- ***bias_mutate_power*** is the ratio of which the bias can be altered in a mutation.
- ***bias_mutate_rate*** is the probability that a mutation alters a bias.

All individuals have 15 input nodes and 4 output nodes, corresponding to the 15 observations received from the agent, and two binary strings used to compute the action fed to the agent. An ***initial population*** of randomly constructed neural networks is generated, limited to ***num_hidden***, ***num_input*** and ***num_output***. Each ***individual*** in the population is evaluated by the reward it gets from controlling the agent for t seconds, or by d decisions steps. Once all individuals have been evaluated, they are ranked by reward and perform ***cloning***, ***crossover*** and ***mutation*** with regards to probabilities, some of which are based on their ranking.

4.3 Cloning

Cloning passes over the genes of an individual to the next generation without alteration. Given a ranking of the individuals in a population, the ***elite*** consist of the k highest ranked individuals. The elite are cloned, and cloning may occur of the remaining population if they are not selected for crossover or mutation.

4.4 Crossover

Unlike conventional crossover, NEAT crossover only generates a single individual. To compensate for any would-be population loss, each individual mates with two other individuals so that the next generation contains the same number of individuals.

4.5 Mutation

Mutation is done proportional to the rank of an individual. The less fit an individual is, the more desirable we find it for the individual to change their genes, in hope that it will attain something more competitive.

4.6 Generating New Population

In the following sections two different population generation algorithms will be described. These algorithms were developed in tandem for experimental purposes using fundamentally different approaches.

4.6.1 Purging The Weak

The population generation algorithms is heavily inspired by a genetic selection algorithm called GNS detailed in [16]. Our population generation algorithms starts off by ranking our individuals by fitness. The surviving population is selected so that their survival chance is proportional to their fitness rank. Survivors will mate to create offspring for the next generation using crossover. The mating process is done by iterating through the list of survivors, starting with the fittest. For each iteration another parent is chosen for crossover. The second parent must be less fit than the first parent and is chosen randomly with a weighted chance proportional to its rank. Offspring will also have a chance of mutating proportional to the rank of the first parent. This is because the child will resemble the first parent more.

4.6.2 No Purging

This method does not create any new individuals other than those of the initial population. The probability of crossover is fixed, and is performed downwards, meaning that an individual with rank r selected for crossover breeds with the individual of rank $r+1$. Mutation probability increases with rank, having less fit individuals mutate by higher chance. An individual is either cloned by elitism, crossed, mutated else cloned. Note that cross and mutate operators are not applied on the same individual in one generation. Essentially, for each individual G_i we have a crossover probability p_c and a mutation probability p_{mi}, optimally subject to $p_c + p_{mi} \leq 1$. For each individual, other than the elite, we randomise a number $n \in [0, 1]$.

- if $n < p_c$ then G_i is crossed
- if $p_c \leq n < (p_c + p_{mi})$ then G_i is mutated.
- else G_i is cloned.

5 Evaluation

All experiments discussed in this section use the population generation algorithm with purging. It was determined that the purging algorithm had vastly superior performance. Initial testing of the non-purging algorithm did not show a positive trend in average fitness over time. Due to this, no further experimentation was carried out using the non-purging algorithm.

All experiments were run on: Windows 11 on a Ryzen 9 3900X 24-thread 142-core CPU with an NVIDIA GTX 1080 GPU Initial network complexity of created individuals had 15 input nodes and 4 output nodes. The remaining hyper-parameters were chosen by various experimentation, from which the following gave the most promising results.

- ***num_hidden*** $= 10$
- ***node_add_prob*** $= 0.02$
- ***node_delete_prob*** $= 0.02$
- ***conn_add_prob*** $= 0.5$

- ***conn_delete_prob*** $= 0.5$.
- ***weight_mutate_power*** $= 0.1$
- ***weight_mutate_rate*** $= 0.9$
- ***bias_mutate_power*** $= 0.1$
- ***bias_mutate_rate*** $= 0.1$

5.1 Results

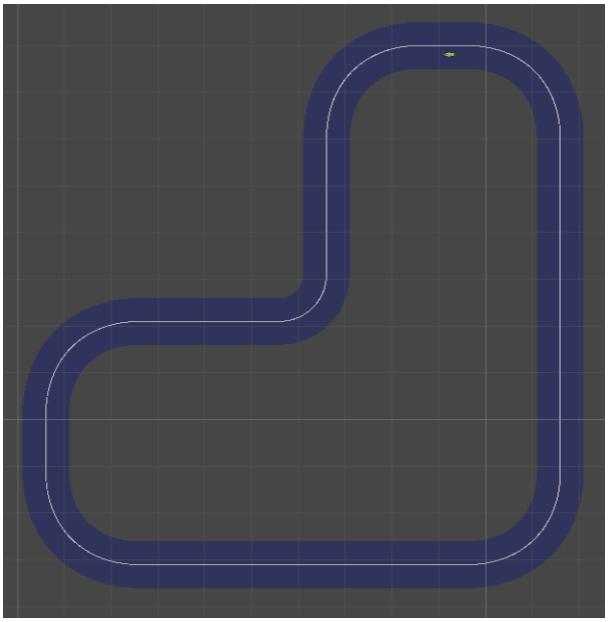

Figure 2: The track that was used for evaluation

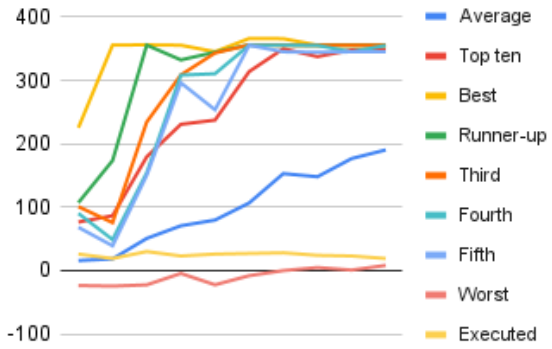

Figure 3: Metrics for a run with 50 in population, over 10 generations

With the selected generation population and configuration parameters, we managed to evolve individuals that could drive an entire lap through the track without crashing into the wall. A theoretically optimal individual would barely finish a lap within our configured timeout, equating to approximately 380 in fitness. As seen in figure 3, the top performers managed to hit optimum after circa 6 generations. Also, around that time training seemed to converge as the average performance started to stall.

5.2 Problems current scoring

Our rewards calculation is based on how many checkpoints the agents can reach withing a set amount of ***decisions frames***. An agent will send a decision every ***decision frame*** which occurs every fifth physics calculation step. An agent progressing towards the next checkpoint also give partial credit, to differentiate individuals that reach the same number of checkpoints. Finally, agents are heavily penalised if they hit a wall. But, this scoring method is not enough as we get a mixture of individuals that score well but that are not necessarily desirable. We divide individuals into ***safe*** and ***risky*** individuals, where safe ones (i) tend to follow the road without oscillations and (ii) take curves without getting close to the walls. Risky individuals are those who break any of those two conditions. The risky individuals are not only undesirable due to subjective reasons we may construct, such as the driving being ugly or unsafe, but we have noticed that risky individuals are less stable than safe ones. By this, we mean by the chance that the individual will be able to reproduce the same outcome in successive runs. Risky individuals have a greater chance of crashing or getting stuck in a wall.

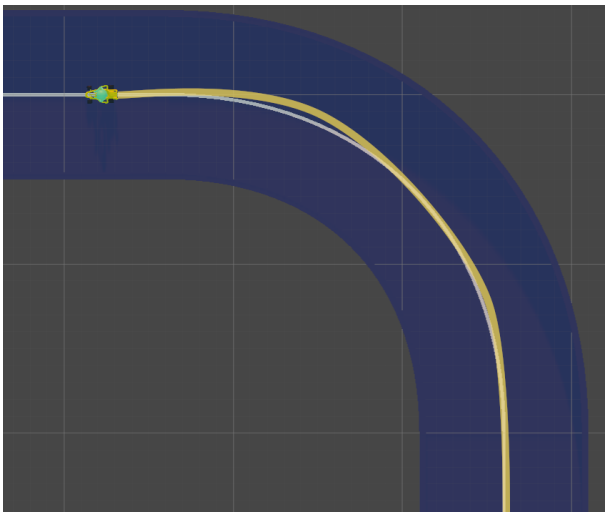

Figure 4: Example of how a safe individual handles a curve

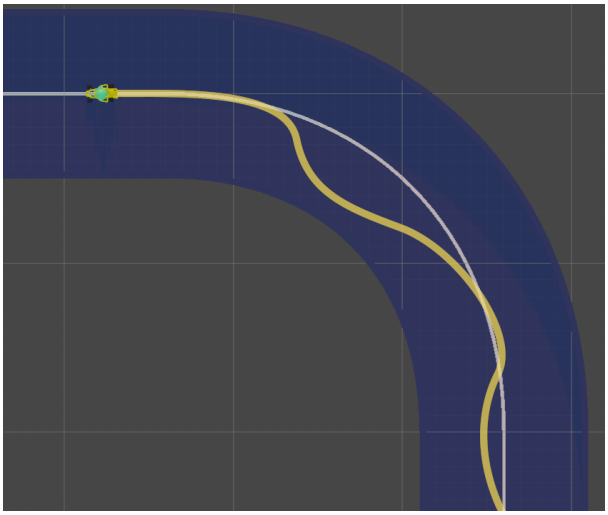

Figure 5: Example of how a risky individual handles a curve

6 Related Work

NEAT is used for usage with Unity to train a ANN to orient a vehicle to follow a leading vehicle in [7]. They use visual input to find the vehicle to follow, using chromatic data for navigation. They use their trained network in a physical AC-car. They do not use Python-NEAT for the training backend, but communicate with Unity using mlagents, therefore their configuration were of some value for us.

A ***Genetic Natural Selection*** (GNS) algorithm that concern ***fitness*** specification, ***elimination*** algorithm and ***repopulation process*** is proposed in [16]. The most significant part of GNS is that it eliminates half of the population with a weighted random distribution, based on rank. The surviving population reproduces randomly in order to fill out the gaps of those who were eliminated. This elimination and repopulation method is integrated into out first population generation algorithm.

Another approach [2] uses a virtual camera as the sensory input in a world simulated in Unity to compare the performance of various Neuroevolution algorithms to Double Deep Q-Learning (DDQN). The performance is determined by how close the car stays to the middle of the lane and the distance driven. The results show that the genetic algorithms outperform DDQN in all performance metrics.

A paper [6] introduces a new way to use search-based algorithms in the field of self-driving by procedurally generating roads to benchmark self-driving car models in a simulated environment. The object of the road-generating model is to reveal flaws in a self-driving model relating to lane-keeping. A fitness value based on how far the car drives from the middle of the lane determines which genetic operations will be used to create a new generation of roads to test. The results conclude that search-based procedural generation can be used to create test suites for exposing safety flaws related to lane-keeping.

Using theory from image analysis and computer vision, [3] uses image-data from a camera to identify certain properties of the road such as middle, side lanes, and other painted lines, in order

to navigate efficiently and at high speeds through paved roads. Techniques from image analysis such as edge detection and classification are done in real-time during driving. The autonomous system is tested on a virtual environment called ***Pro-Civic*** that allows for running multi-vehicle simulations. The results show that their method can control a vehicle using simple fuzzy-logic laws, and is sufficiently stable up to speeds of 70 kilometres per hour in sharp turns.

7 Discussion

We often encountered the issue that unaltered individuals would perform vastly different between generations. This was clear in some plots seeing that the top performer, whom is cloned due to elitism, would follow up with a lower score and sometimes even by a substantial amount. Initially we suspected that the genetics of the top performer would not be preserved to the subsequent generation, but was falsified upon verification that the genetics were indeed unchanged. Upon closer inspection, it was discovered that a network would not always act exactly the same after resetting the simulation. Although it tended to act in an expected fashion, being how it acted previously. But by no means was this behaviour fully deterministic, having sporadic uncalled diversions out of the blue. Our method of combating this consisted of

- Forcing the car to reset all of its properties to default values on episode start
- Resetting the sensory inputs
- Ensuring that the game always run at 50 or more fps to not throttle the physics computations.

Although the unintended behaviour have been minimised, we have no way of ensuring that this never occurs below our radar. We suspect that this still happens from time to time, which results in some random unfair worsening of one or more individuals, potentially slowing down convergence. For instance, a fairly potent individual N is unfairly affected by this performance anomaly, resulting in a lower ranking by the end of the generation evaluation, therefore it is either crossed with a lesser individual or is directly altered by mutation, whose rate increases by rank, thus leading to offspring which we can not expect to perform equally to N. Of course, this could by chance cause the offspring of N to become better than before, but since N either crossed with a significantly less fit individual, or it mutated, whose effects can go in any direction, we assume that the more probable case is that it turns out for the worse.

We also wanted to each run of an individual to be as fair as possible. Initially, we let each individual run for a fixed number of seconds. However, CPU throttling during training would cause individuals to run for a different number of decision steps, resulting in unfair advantages for those who were not affected negatively by the CPU. We made timeout not dependent on system time, and instead on number of ***fixedUpdate()*** calls. Unity calls fixedUpdate() each physical computation frame, which we used as our unit time. Each individual now have a number n of decision steps it can do in its run. A decision step is performed each fixedUpdate().

8 Conclusions

NEAT proved to be a suitable genetic algorithm for evolving non-static ANNs to drive a car using spatial awareness. Training would converge faster if individual performance was consistent. The non-deterministic simulation in Unity presents a challenge for this sort of project.

9 Future Work

The main contribution of our work consisted of showing that NEAT indeed is appropriate for the automated driving problem within the limits of a racing track. It is left for future work to make a fair performance assessment and comparison of RL, NE and NEAT for this task. Also, we also want to see this being used for vehicles in real-life, and not just in virtual worlds where the majority of projects reside. But for that to happen, we would probably have to combine it with image analysis so that more detailed observations could be made from the sensory inputs of the vehicle, much like how its done for imitation learning.

Training multiple agents to race each other. Get to the finish line as fast as possible without crashing into the wall, or any other opponent.

It would also be desirable to design a scoring function that penalises risky individuals and rewards safe individuals so that they are not scored equally even though if they make it equally far on the track. This would not only fix the problem visualised in figure 4 and 5, but it would also push the trainer to tune the ratio of risky and safe behaviour. Although figure 1 shows a safe individual, it is probably not the best way to handle the curve, in the context of racing for instance.

We probably want an individual that does not oscillate, but that also drives a bit closer to the wall in the inner curve to save time. An idea to promote more safe driving is to add some wall distance check at every checkpoint, a ***safeness score***, which is included in the rewards computation. Checkpoints can be located anywhere on the track, therefore by placing some in curves could ensure that risky individuals do not gain the same reward from that particular checkpoint. On the other hand, we must also tune the rewards gained from reaching a checkpoint and progressing towards the next checkpoint to promote incentives of driving a bit riskier than of those who gain maximum score of safeness at each checkpoint. If we do

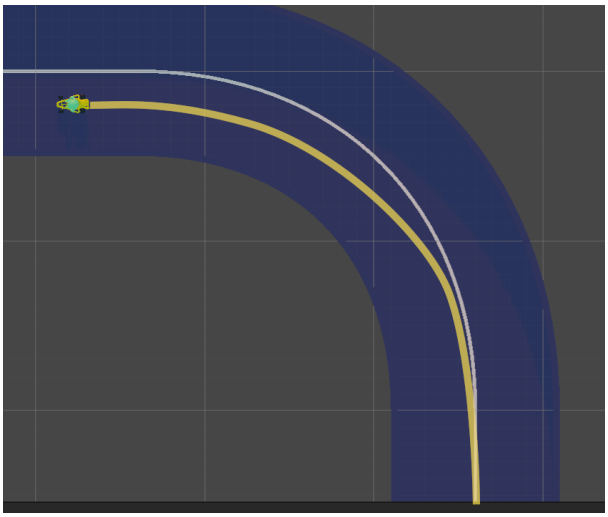

Figure 6: This individual still drives desirably safe, but subjectively takes the curve much better than the individual in figure 4

not want to directly penalise the individuals for driving near the walls as we would do in the previous solution, another idea would to penalise the oscillating driving behaviour, essentially an incentive for individuals to change in turning direction as little as possible. This could minimise the oscillations without dictating how individuals should navigate the track.

References

[1] Rusul Abduljabbar et al. “Applications of artificial intelligence in transport: An overview”. In: *Sustainability* 11.1 (2019), p. 189.

[2] Ahmed AbuZekry et al. “Comparative study of NeuroEvolution algorithms in reinforcement learning for self-driving cars”. In: *European Journal of Engineering Science and Technology* 2.4 (2019), pp. 60–71.

[3] Farid Bounini et al. “Autonomous Vehicle and Real Time Road Lanes Detection and Tracking”. In: *2015 IEEE Vehicle Power and Propulsion Conference (VPPC)*. 2015, pp. 1–6. DOI: 10.1109/VPPC.2015.7352903.

[4] Felipe Codevilla. *CARLA Challenge Track 2 Baseline - Conditional Imitation Learning*. URL: https://github.com/felipecode/coiltraine/blob/master/docs/carla_challenge_coil_baseline.md (visited on 22/06/2022).

[5] Hieu Dinh. “The Revolution of Warehouse Inventory Management by Using Artificial Intelligence: Case Warehouse of Company X”. In: (2020).

[6] Alessio Gambi, Marc Mueller and Gordon Fraser. “Automatically testing self-driving cars with search-based procedural content generation”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 318–328.

[7] Joseph Ricardo González Núñez. *Design of a self-driving mini-robot for indoor navigation using evolutionary artificial intelligence algorithms*. B.S. thesis. 2020. URL: https://201.159.223.86/bitstream/123456789/184/1/ECMC0028.pdf (visited on 27/05/2022).

[8] John K Haas. “A history of the unity game engine”. In: (2014).

[9] Shantanu Ingle and Madhuri Phute. “Tesla autopilot: semi autonomous driving, an uptick for future autonomy”. In: *International Research Journal of Engineering and Technology* 3.9 (2016), pp. 369–372.

[10] Arthur Juliani et al. “Unity: A General Platform for Intelligent Agents”. In: *CoRR* abs/1809.02627 (2018). arXiv: 1809.02627. URL: http://arxiv.org/abs/1809.02627.

[11] *Karting Microgame*. URL: https://learn.unity.com/project/karting-template (visited on 26/05/2022).

[12] B Ravi Kiran et al. “Deep reinforcement learning for autonomous driving: A survey”. In: *IEEE Transactions on Intelligent Transportation Systems* (2021).

[13] Enrique Marti et al. “A review of sensor technologies for perception in automated driving”. In: *IEEE Intelligent Transportation Systems Magazine* 11.4 (2019), pp. 94–108.

[14] Alan McIntyre et al. *neat-python*. URL: https://github.com/CodeReclaimers/neat-python (visited on 21/06/2022).

[15] László Monostori. “AI and machine learning techniques for managing complexity, changes and uncertainties in manufacturing”. In: *Engineering applications of artificial intelligence* 16.4 (2003), pp. 277–291.

[16] Jihene Rezgui, Clément Bisaillon and Léonard Oest O’Leary. “Finding better learning algorithms for self-driving cars: An overview of the LAOP platform”. In: *2019 international symposium on networks, computers and communications (ISNCC)*. IEEE, 2019, pp. 1–6. DOI: 10.1109/ISNCC.2019.8909159.

[17] Kenneth O. Stanley. "Efficient Evolution of Neural Networks Through Complexification". PhD thesis. Department of Computer Sciences, The University of Texas at Austin, 2004. URL: http://nn.cs.utexas.edu/?stanley:phd2004 (visited on 26/05/2022).

[18] Felipe Petroski Such et al. "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning". In: *arXiv preprint arXiv:1712.06567* (2017).

[19] Tesla. *Tesla Autonomy Day*. URL: https://youtu.be/Ucp0TTmvqOE?t=8021.