Neuro-Evolution for Car Racing

Kristinn Runarsson
Student Number: 4314565
Magnus Snorri Bjarnason
Student Number: 4321456

Delft University of Technology

Abstract

Driving a car is mostly a reactive process where driving decisions is based on the eyes, experience and hearing. In this paper we show how simple neural network is trained with evolutionary methods can turn sensor data from a racing environment to meaningful actions that result in agent that is able to drive decently through a racing track. The agent is also tested on various tracks which were not used during training and different neural network architectures are compared.

I. Introduction

In this project our aim is to design and train an agent suitable for racing in an environment with realistic physical simulation. Our agent will operate in the TORCS environment which is state-of-the-art open-source racing environment which suits this project perfectly. TORCS comes with various tracks with different complexity and physical properties and our goal will be implementing an agent that is able to drive through, without colliding, different tracks within respectable amount of time. TORCS also comes with a simple driver which is hard-coded whose only goal is to stay in the middle of the track. Because of it's nature the simple driver is truly incompetent in the sense of driving fast but it is consistently able to drive through various tracks without colliding. Therefore the simple driver is perfect for comparison to our own agent where our agent should perform significantly better.

Each agent has access to several sensors from the environment and should make use of the sensor data to output meaningful actions. The problem suits evolutionary neural networks well since the sensors are the input in to the network while the actions are the output of the network. The training of each agent is unsupervised so we will make use of evolutionary methods to optimize our agent's performance.

Beneath are the research questions we seek to answer with our experiments.

1. Can simple evolutionary neural network train an agent that can significantly outperform the simple driver?
2. How will agents trained on different tracks perform compared to each other when the tracks are switched.
3. How do different network architectures affect the performance of the agent?
4. How to train an agent that is able to perform on tracks of various complexity and how does that agent perform compared to an agent that is only optimized for one track.

The report is sectioned in the following way. Section II goes over related work in the field of racing agents and evolutionary neural networks for simple games. Section III gives a brief introduction to the TORCS environment and it's sensors and actuators. In section IV evolutionary neural networks are introduced and our motivation for choosing this technique. Details on the implementation of our evolutionary neural network can be found in section V while section VI describes the experimental setup. Finally results are presented in section VII and sections VIII and IX discuss future work and conclude the project.

II. Related Work

For the application domain considered in this report, a number of researches have investigated the application of computational intelligence methods for car racing games and simulations. In 98' Pyeatt and Howe [1] used reinforcement learning to make agents to race in RARS, an open source racing simulator. More recent work has focused more on evolutionary computation to increase the performance of car racing agents. One approach that seems to appear often in the literature on this subject is evolutionary neural networks[2].

Evolving neural networks or neuro-evolution have been used to train agents playing games with good results. [3] showed how four different neuro-evolution approaches were able to play Atari games with decent results. They showed that evolved policies beat human high score on at least three different Atari games and were able to discover opportunities for infinite score in other three games. One of the techniques used in this research is Neuro-evolution of Augmenting Topologies (NEAT)[4] which has also been successfully used to evolve car racing agents as NEAT is one of the most successful and widely applied neuro-evolution approach. NEAT has been applied in on-line neuro-evolution[5] to evolve drivers in TORCS, an open source racing simulator, to evolve a neural network driver that can autonomously navigate a track without crashing in a car racing simulator [6] and in [7] Cardamone, Loiacono and Lanzi applied NEAT to evolve car racing driver in TORCS with similar goals.

The field of intelligent car racing agents has been active and it is apparent in the game industry as well where Artificial Intelligence(AI) drivers in racing games have improved significantly over the last decade. Most recent innovation in the game industry is the AI concept used in the game Forza Motorsport 5. Drivatar[8] is a novel form of learning AI where the goal is to make more human-like computer opponents in a racing game using information about players racing behaviors.

In our work we did not emulate, strictly follow or extend any of the already made researches in this application domain. But, some of the researches and experiments performed in the literature cited in this section gave us inspiration and ideas how to solve our problems.

III. TORCS

TORCS[9] is an open source car racing simulator that provides physics engine, graphical 3D visualizations, race tracks, car models and a few game modes(e.g, practice, quick race, etc). The car dynamics are accurately simulated and the physics engine takes into account many aspects of racing cars such as traction, aerodynamics, fuel consumption, etc. Drivers or bots in the simulator are able to access the current game state at each control step or game tick to get various information about the car and the environment. This information includes car position relative to

Sensor	Description
angle	Angle between the car direction and the direction of the track axis. degrees, spanning clockwise from -90 degrees up to +90 degrees with respect to the car axis.
gear	Current gear of the car.
rpm	Number of rotations per minute of the car engine.
speedX	Speed of the car along the longitudinal axis of the car.
speedY	Speed of the car along the transverse axis of the car.
speedZ	Speed of the car along the Z axis of the car.
track	Values of 19 range finder sensors, each sensor returns the distance between the track edge and the car within a range of 200 meters. The sensors sample the space in front of the car every 10 degrees, spanning clockwise from -90 degrees up to +90 degrees with respect to the car axis.
trackPos	Distance between the car and the track axis.

Table 1: *Description of main sensors that drivers use in experiments reported in this report. For more details see [10].*

the track, position of other cars, fuel status, speed, etc. The driver can control the car using break and gas pedals, the gear stick and steering wheel.

Experiments in this report used version 1.3.4 of TORCS using the Simulated Race Car Championship(SCR) platform[10]. The SCR platform abstracts the driver from the TORCS simulators by running the driver program as a client that is connected with UDP connection to a server hosting the TORCS engine. This creates a physical separation between the game engine(TORCS) and the drivers. Thus, to develop a driver no knowledge about the TORCS engine is required. All information about the drivers perceptions and available actions are defined by a sensor output and action input which the client gets/makes at each game tick. Information about the current state of the environment are sent to the client which calculates an action that is sent back to the simulator. This allows many clients to connect to a server that runs TORCS and is very convenient for competitions, like the Simulated Car Racing Championship[11][1]. This abstraction makes developers able to select other programming languages than C/C++ for driver implementation and today Java and Python clients are available.

I. Sensors

The SCR platform receives number of sensor readings which provide information both about the game environment (the track, the speed, etc) and the current state of the race (the current lap time, the current damage, etc). [10] provides details about available sensor reading the driver can access at each control step. Table 1 give overview of what sensors are used in experiments reported in this report.

II. Actuators

To control the car the driver can send actions to the TORCS engine at each control step. The driver controls the car through typical set of actuators and are these actions showed in Table 2. The

[1]This competition has been held each year for the last few years, but this was the most recent one we were able to find paper about.

accel	Gas pedal, 0 no gas and 1 full gas.
break	Break pedal, 0 no break and 1 full break.
gear	Gear value.
steering	Steering value, -1 and +1 means respectively full right and left.

Table 2: *Description of main actuators that drivers use in experiments reported in this report. For more details see [10].*

driver computes his actions at each control step by assessing the sensor readings and decides a value for each action in Table 2 by some set of rules or more complex mechanism like neural network. The actions are then sent to the TORCS engine and the client waits for next control step.

IV. Neuro-Evolution

Artificial Neural Networks are used to solve many problems in machine learning and pattern recognition. The basic idea is that the Neural Network calculates output values based on the input. Evolutionary methods are powerful for solving complex optimization problems based on a calculated fitness function where the search space is large. Neuro-Evolution is the combination of Neural Network and Evolutionary methods. In this section the decision to use these methods is motivated.

I. Neural Network

The Torcs racing environment has many meaningful sensors which give data that can be used to decide what action should be taken. Driving a car is basically a reactive process where decisions are taken based on what we see and hear so for instance if the eye sees an obstacle on the road the relevant action should be to break or steer around the obstacle. Similarly if the sensors sense that the car is too close to the edge of the track the car should turn to avoid going off the track. Because Neural Networks map input values to output they fit our project and experiments perfectly. Our agent makes use of over 20 sensors which need to be mapped appropriately to output vector with 5 values. With one ore more hidden layers the network can therefore be quiet complex with a lot of links between nodes and somehow these links need to be trained in order to get meaningful results.

To hand design an agent to make decisions based on particular sensor data is a very time consuming task. Such design could use many conditional statements or be hand designed Neural Network. Our solution aims to minimize time consuming decision phase allowing the Neural Network to evolve it's architecture, subject to constraints. To achieve this our implementation allows the user to specify number of node in input layer, each hidden layer and the output layer and the network then creates appropriate amount of nodes and one link between every node in adjacent layers. Every link has a weight which needs to be optimized and a boolean value determining if that link is active or not. By activating and deactivating links the network can change it's architecture without changing the number of nodes. Every node can have different transfer function which need to be optimized as well.

II. Evolutionary Methods

To optimize the links and nodes in the network we make use of evolutionary methods. In a supervised environment the network could be trained with back-propagation by observing the result of each action taken. The agent would get input values and calculate corresponding output

values before getting feedback right away if the action taken was good or bad. However this is not feasible approach for our problem since it is non-trivial to determine if taken action in car driving is good or bad. Better approach is to make use of fitness function that is calculated after the agent has performed for some predefined time. This makes the problem unsupervised which promotes the use of evolutionary algorithm. The first generation is filled with randomized networks and each individual is given the same time to drive on the track before the fitness function is calculated based on distance driven and damage to the car. With some constraints on the agent's initial knowledge this guarantees that the distance driven will be optimized which in turn leads to lap-time being optimized. Since the Neural Network is quiet complex with many nodes and links to optimize the search space is huge and evolutionary methods and using evolutionary algorithm we avoid getting stuck in local optima.

V. Implementation of Neuro-Evolution

The implementation is inspired by the lecture slides on neural networks and evolutionary methods and our experience and knowledge in programming. The implementation is also inspired by the literature on evolutionary neural networks [2]. We started with implementing our own evolutionary neural network which can be used to solve any similarly defined optimization problem. We used our own implementation mainly to have better understanding of how the network performs and to give us the control to adapt it better to our domain. This section gives details on the implementation and motivation for different design choices. Our description assumes that the reader is familiar with neural networks and evolutionary methods.

I. Neural Network

Each neural network is built with node and link objects where the nodes are divided into different layers of nodes and each link connects node from layer i to a node in layer $i+1$. Every node in layer i has link connecting it to every node in layer $i+1$. Every link has a weight value and a boolean value determining if it is active or not. This way our network has as many links as possible for network that only connects nodes in adjacent layers but still the freedom to evolve different architectures depending on what links are active or not. Each node can have different transfer functions describing how it's inputs should be treated and sent forward in the network.

The neural network class can be constructed from an integer array representing number of nodes in each layer or from a chromosome describing the state of each node and link. When the network is constructed with array of integers the weight and activation of links is randomized along with the transfer function of the nodes. The network can also create a chromosome from it's architecture, nodes and links. When receiving an array of input values the network initializes the values of it's input layer and then proceeds to calculate through the network until it reaches the output layer. Our implementation is really straightforward and promotes the use of evolutionary methods to optimize the network.

II. Evolution

The fact that the neural network object can be constructed with chromosome and output it's own chromosome sets it up perfectly for evolutionary algorithm to optimize the network. The

evolutionary algorithm has list of chromosomes, called generation, and every chromosome has fitness value attached to it so it is very easy to sort the list based on fitness value in order to decide which individuals get selected for crossover and reproduction. To keep future generations as diverse as possible only n-best unique fitness values determine which individuals get to pass their genes on. The user can specify how many individuals should be considered for reproduction.

When the top n individuals have been selected based on their fitness value the evolutionary algorithm proceeds to create a new generation. For each individual in the new generation it's parents need to be selected from the list of fittest individuals. Our implementation uses roulette wheel selection where the probability of an individual being selected as a parent depends on it's fitness divided by the sum of all fitness values. This ensures that the fittest individuals are selected more often than less fit individuals.

III. Crossover

After the two parents have been determined their genes need to be mixed to create the new individual. The chromosome contains node genes and link genes. The transfer function of each new node is randomized from the two transfer functions corresponding parent nodes use. Similarly the activation of each link is randomized from the activation of the corresponding parent links. However the weight of each link is converted to a bit steam and then a crossover point is randomly determined. Bits appearing before the crossover point are used from parent A while the remaining bits are used from parent B. Since the weight is a double value we made sure that the new weight is not infinite or null which can happen on very rare occasions. The new chromosome can now be created using the new links and nodes.

IV. Mutation and cloning

Before the crossover can take place each individual has a chance of being cloned from one of it's parents. The parent selected for cloning is the fitter parent. Before the new individual can be added to the new generation it has to undergo the mutation phase. For every link and node there is a slight possibility of mutation happening. In the case of nodes the transfer function is simply uniformly randomed from every possible transfer function. The link activation is also randomed, either true or false, while the weight is adjusted based on Gaussian distribution. The current weight value has number added to it based on a Gaussian distribution with multiples of it's current value as standard deviation.

V. Extension allowing evolution of number of hidden layers and nodes

In order to allow the neural network to evolve it's architecture more freely it should be able to evolve the number of hidden layers and number of nodes in each hidden layer. To allow this the crossover function of the two chromosomes needed to be extended. First the architecture of the new individual needs to be determined. The new architecture is either mutated or derived from the two parents. Subsequently the nodes and links of the new individual need to be created and there are 3 scenarios that can happen.

1. Both of the parents have the node/link and the new node/link is formed by simple crossover
2. Only one parent has the node/link and the new node/link is directly copied.
3. Neither parent has the node/link resulting in a new randomized node/link.

Because each individual has only a few hidden layers the chance of mutation is very low. For example if each individual has 1 hidden layer and the chance of mutation is 0.1% then the expected amount of individuals with mutated number of layers in each generation is below 1. Therefore we introduced new mutation probability variable only for mutations on the network architecture to promote changes in the architecture.

Unfortunately early experiments with agents that were allowed to evolve the number of nodes in the network proved unsuccessful as is discussed in the results section.

VI. Experimental Setup

I. Initial knowledge and constraints

Since the training is extremely time consuming our agent has some initial knowledge and constraints in order to speed up convergence. When the agent wants to go wrong-way or is stuck automatic hard-coded correction kicks in and overrides the neural network. This correction scheme is taken from the simple driver. This prevents our agent to exploit the fitness function. Furthermore our agent has knowledge about when it should shift the it's current gear up or down. This knowledge is based on RPM value from the RPM sensor and it's current gear. We argue that gear changing is general over different tracks and that the convergence gain outweighs the possibility of the agent finding better gear shifting scheme.

II. The fitness function

The environment comes with a sensor that measures total distance raced which we use as the main component for our fitness function. It is quite simple, we want to maximize the distance raced and since the agent is not able to turn around and go the wrong way this gives an accurate estimate of the performance of our agent. At one point we ran into problems with agents exploiting walls surrounding the track to boost the total distance traveled and to prevent that from happening we introduced damage as the second component of our fitness function. The environment has sensor measuring total damage to the car, damage is received when colliding with walls and obstacles, so we simply subtract the damage received from the distance raced promoting agents that are able to finish the track without crashing into the walls.

The reason we go for maximization on distance raced rather than minimization on lap time is because the agents in the first couple of generations are not able to finish one lap making the best lap time metric irrelevant. What we could do is initially train with the fitness function of distance raced and then at one point switch it to minimizing the lap time when most of the agents are able to finish one lap. Since distance raced favours a zik-zak behaviour a bit it would be interesting to see if the agents would become more direct when the goal is to minimize lap-time. Ultimately the best lap time will also result in the most distance raced which was the reason we stuck to the fitness function of distance raced.

III. Parameters

The evolutionary algorithm is initialized with population size of 100 and number of fittest individuals is set to 10%. So the algorithm will consider the top 10 unique individuals (unique fitness value) for reproduction. The cloning probability is set to 20% and the mutation probability

is 0.5%. The number of nodes in each layer of the network is tweaked between experiments along with the track the agent is trained on.

VII. Results

We trained the agent with various number of hidden layers and nodes in each hidden layer. Each training session took several hours which meant we could not run as many experiments as we wanted to. The experiments aim to answer the research questions presented in the first section.

Figure 1: *Overview of track 1.*

Figure 2: *Overview of track 2.*

I. Experiment 1

The first experiment was to demonstrate that our method results in significant improvement over the the hard coded simple driver. Since the simple driver is really slow, especially in turns, we expect a simple well-trained neural network to be able to outperform the simple driver by big margin.

Table 3 shows the results of this experiment. Agent 1 has one hidden layer with 15 nodes while Agent 2 has one hidden layer with 30 hidden nodes. The agents are specifically trained for each of the tracks they are tested on. The lap time is measured in the second lap because that is considered as "‘a flying lap"’ (when the agent starts a lap on full speed). The results show that agents trained

Figure 3: *Overview of track 3.*

Agent	Track number	Lap time
Simple Driver	1	1:25:19
Simple Driver	2	1:55:19
Simple Driver	3	2:31:02
Human Driver	1	39:54
Human Driver	2	52:49
Human Driver	3	2:31:02
Agent 1	1	48:47
Agent 1	2	1:07:06
Agent 2	1	46:48
Agent 2	3	1:22:67

Table 3: *Results from experiment one where lap times of different agents are compared.*

with evolving neural network are significantly better than the simple driver that came with the environment but compared to a driver controlled by a human player the agent is way off.

II. Experiment 2

It is also interesting to compare agents trained on different tracks when they are asked to perform on another track. For this experiment we used varying network architecture and for each architecture we trained two agents on different tracks before comparing them by letting them drive on three tracks.

The results from Tables 4, 5, 6 show that some agents are able to finish tracks they are not trained on much faster than the simple driver while other agents are not able to finish the tracks at all. This suggests that the agents might suffer from overfitting to the track they are trained on. It is worth noting that usually the agents damaged the car on the tracks they were not trained on which means that the automatic mechanism to fix the agent from a "'stuck"' position most likely kicked in for a short period of time to assist the agent. Interestingly the agent that was trained on track number 3 was able to finish track number 1 and track number 2 with a decent performance compared to the agents trained specifically on those tracks. The agent even finished track number 2 without damaging the car. Looking at figures 1, 2 and 3 we can see that track number 3 is

Agent	Trained on	Track number	Lap time
Agent a	1	1	48:47
Agent a	1	2	did not finish
Agent a	1	3	did not finish
Agent b	2	1	1:42:48
Agent b	2	2	1:07:06
Agent b	2	3	2:25:04

Table 4: *Results from experiment 2 where agents are faced with the challenge of driving through tracks they have not seen before. The agents in this table have one hidden layer with 15 nodes.*

Agent	Trained on	Track number	Lap time
Agent a	1	1	46:48
Agent a	1	2	1:12:92
Agent a	1	3	did not finish
Agent b	3	1	1:09:22
Agent b	3	2	1:13:19
Agent b	3	3	1:22:67

Table 5: *Results from experiment 2 where agents are faced with the challenge of driving through tracks they have not seen before. The agents in this table have one hidden layer with 30 nodes.*

Agent	Trained on	Track number	Lap time
Agent	1	1	53:25
Agent	1	2	1:24:10
Agent	1	3	1:47:55

Table 6: *Results from experiment 2 where agents are faced with the challenge of driving through tracks they have not seen before. The agent in this table has two hidden layers with 15 nodes each.*

Agent	Architecture	Track number	Lap time
Agent 1	15	1	48:47
Agent 2 15 - 15	30 1	1 53:25	46:48heightAgent 3
Agent 4	0	1	1:14:82
Agent 5	15	2	1:07:06
Agent 6	0	2	did not finish

Table 7: *Results from experiment 3 where agents with various number of hidden layers and number of nodes in each hidden layer are compared. The architecture column represents the number of nodes in each hidden layer, 0 means no hidden layers and x - y means two hidden layers with x and y number of nodes.*

considerably more complex than the other two tracks. This indicates that an agent trained on a complex track is more likely to be able to perform on a more simple track than the other way around but we obviously need more data to back up this claim.

III. Experiment 3

This experiment focuses on how different number of hidden layers and nodes in each hidden layer affect the performance of the agent. As discussed in the implementation section the evolutionary neural network was extended so it could handle evolving number of layers and number of nodes in each layer. However early experiments which allowed the network's architecture to evolve more freely proved unsuccessful resulting in the agent to converge very quickly on one architecture and not deviating away from it. The reason for this is most likely because once a new node or a new layer is added through mutation every link that is not present in the parents must be randomed giving the new individual very little chance of actually improving from the parents. Therefore we deviated from our plans to allowing the network to evolve the number of nodes in each hidden layer along with the number of hidden layers and instead trained agents with $[0; 2]$ hidden layers. As mentioned before, each training session took several hours so therefore we could not experiment with all the architectures we wanted to try.

The results from table 7 suggest that number of nodes in each hidden layer are critical to the results while the number of hidden layers seem to be more important. For instance agent with no hidden layers is not good at all and the agent with no hidden layer trained on track number 2 wasn't even able to finish the track. More experiments with various architectures need to be made before any meaningful conclusions can be drawn.

IV. Convergence

One interesting thing we saw throughout the experiments was that some agents tended to converge on a good solution very fast and did not improve much over subsequent generations after reaching the initial peak. Figure 4 shows this very clearly. It is surprising how fast the agent converges but also interesting to see no further improvements after some point. This may be the result of poor mutation function which is not able to get out of local optima or this shows that this method needs some clever addition to push the agent's performance further.

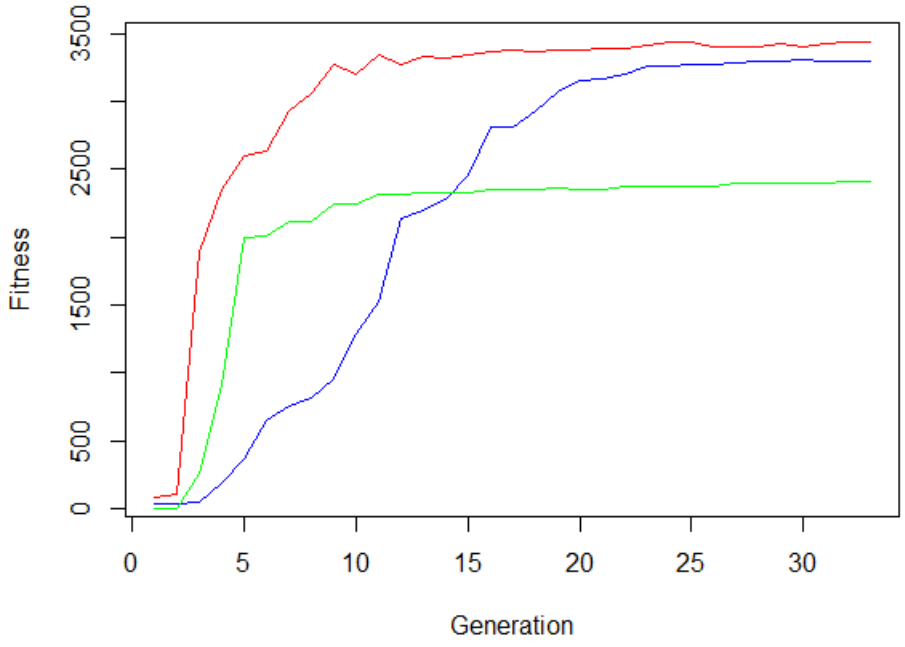

Figure 4: *The evolution of the fittest individual over generations.*

V. General remarks on agent behaviour

The agent always showed some tendency to zik-zak a bit, especially on the long straight sections of the track. This is probably a result from the fitness function which rewards distance driven without damage to the car. This might also be a result of the agent's design which might make it very sensitive to various readings from the sensors. The agent also failed to achieve the same top speed as the human drivers which made the agent considerably slower on the straight sections than humans. However in the turns the agent showed great performance often resulting in smooth turns. This is a bit contradicting to human behaviour because humans find it very easy to drive a car as fast as possible on a straight section while often taking sub-optimal turns or simply crashing out of the track in steep turns.

It is not unlikely that the agent is stuck in a local optima or that different number of hidden layers and nodes would do the trick. It would be interesting to hand design the network and only evolve the weights and see if that would give a better performance. That way not all sensors would be mapped to all outputs.

We were not fully able to address research question 4, on how to train a general agent that could drive through various tracks it had not encountered before without colliding. One way, but very time consuming, would be to train each individual on a randomized subset of tracks and then calculate the average fitness value over those tracks. This would take a couple of days to train but should eventually result in a decent agent that is able to handle various tracks.

VIII. Future Work

This project can be taken forward in many different ways and we have identified three possible directions for future work which we will present in 3 different sub-sections.

I. Further optimize lap time for specific track

Experiment 1 showed that the agent was significantly better than the simple driver while also being significantly worse than a human driver. There is a lot of work that can be done in order to further increase the agent's performance with the goal of behaviour which matches human controlled drivers. To achieve this the mutation function could be changed to a more standard bit-flipping scheme, different architectures tried or even hand-designing the agent and evolve only the weights. Another approach would be to use different networks for different sections of the track but training an agent with many different networks would be very time consuming.

II. Train a general agent able to drive through various tracks

Some racing games give the user a track-creation tool which allows the user to create his own tracks of various complexity. In that case an agent that is optimized on one track will be pretty useless when presented to the new track. Therefore it would be very interesting and highly practical to develop an agent which can perform consistently on various tracks without knowing anything about them. One possible solution would be to make use of an algorithm which could calculate a good racing line and incorporate the racing line into the agent's behaviour in a way that would encourage the agent to stay on the racing line. This approach however requires some knowledge about the track's structure in order to calculate the optimal racing line and a better solution would require no such racing line in order to complete the track in a swift fashion. As discussed in the results section one possible approach would be to train each individual in a generation on a random subset of tracks with the aim of improving the average fitness over those tracks.

III. Train the agent in a race environment with other agents

Key feature in most racing games include a mode where it is possible to race against computer controlled opponents. Training an agent that can drive in a race with other agents increases the complexity significantly. The environment has sensors that gives track position, relative to our own agent, of other agents which could be used to determine when to overtake and when to stay in line. Crashes do occur in such environments resulting in other cars possibly being obstacles on the track which need to be avoided without loosing too much time. The agent also needs to be able to defend itself against overtaking threat from other agents. The search space and training time required to achieve good results is enormous so using evolving neural networks might not be the correct solution. To make a successful overtake the agent needs to be able to reason that it is faster than the opponent in front of it, know in which section an overtake is favourable and how to execute the overtake.

IX. Conclusion

In this project we implemented evolving neural network aimed at training agents for a racing environment. We showed that simple network can be trained with evolutionary algorithm that gives decent results without much initial knowledge. Agents trained with this method were able

to significantly outperform a simple hard coded agent but failed to give a human controlled driver much competition. This method is not very good at producing general agents that can drive through tracks not presented during training. Early experiments with different number of hidden layers and nodes in each hidden layer show that the architecture of the network influences the agent greatly. There is a lot of room for improvements and we identified three main areas for future work.

References

[1] Larry D. Pyeatt and Adele E. Howe. *Learning to race: Experiments with a simulated race car*, 1998.

[2] Dario Floreano,Peter Durr and Claudio Mattiussi. *Neuroevolution: from architectures to learning*, 2008.

[3] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen and Peter Stone. *A Neuroevolution Approach to General Atari Game Playing*, 2013.

[4] K. O. Stanley and R. Miikkulainen, *Evolving neural networks through augmenting topologies*, 2002.

[5] Luigi Cardamone, Daniele Loiacono and Pier Luca Lanzi. *On-line Neuroevolution Applied to The Open Racing Car Simulator*, 2009.

[6] Kenneth Stanley, Nate Kohl, Rini Sherony and Risto Miikkulainen. *Neuroevolution of an Automobile Crash Warning System*, 2005.

[7] Luigi Cardamone, Daniele Loiacono and Pier Luca Lanzi. *Evolving Competitive Car Controllers for Racing Games with Neuroevolution*, 2009.

[8] Author Unknown, *Drivatar*, http://research.microsoft.com/en-us/projects/drivatar/

[9] Bernhard Wymann, Eric Espie, Christophe Guionneau, Christos Dimitrakakis, Remi Coulom and Andrew Sumner. *TORCS, the open racing car simulator*, v1.3.5., http://www.torcs.org, 2013.

[10] Daniele Loiacono, Luigi Cardamone and Pier Luca Lanzi. *Simulated Car Racing Championship Competition Software Manual*, 2013.

[11] Daniele Loiacono and Pier Luca Lanzi and Julian Togelius and Enrique Onieva and David A. Pelta and Martin V. Butz and Thies D. LÃűnneker and Luigi Cardamone and Diego Perez and Yago SÃąez and Mike Preuss and Jan Quadflieg. *The 2009 Simulated Car Racing Championship*, 2009.