Computational Intelligence - TORCS Project Report

Bryan Eikema 10422804[1], Sander Lijbrink 10371109[2] Minh Ngo 10897402[3]

December 20, 2015

[1] University of Amsterdam
MSc. Artificial Intelligence
{bryan.eikema,sander.lijbrink,minh.ngole}@student.uva.nl

1 Introduction

Development of new learning techniques for computer games is a very promising research direction. Insights of these works can be used to improve user experience in similar tasks or even to apply obtained knowledge to agents that interact with the real world.

Prior research has been done to develop a strong, reliable AI for the The Open Racing Car Simulator (TORCS). Onieva et al. (2012) have introduced their approach for solving this problem by proposing a set of heuristics for gear, pedal and steering control and in addition an approach for coupling with opponent's cars. Kim et al. (2012) used an Artificial Neural Network in combination with Linear Regression and heuristics. Furthermore, the use of an evolutionary approach is proposed to make a car able to compete with opponents (Cardamone et al., 2009; Seong Kim et al., 2012).

In our project we are presenting an approach to build an autonomous AI for TORCS. Our controller solution is called *Caffeine*[1] where the requirement was to build a car decision making system that can drive reliably on unseen tracks and compete with other opponents. Our method has shown promising results in competition against other controllers (**6th place** in the UvA contest).

2 Data collection

We've collected data from several tracks (CG Speedway Number 1, CG Track 2, CG Track 3, Oval A-Speedway, Oval D-Speedway, Aalborg) needed to train our neural network. We've attempted to include a wide variety of tracks, ones with lots of sharp corners where caution is necessary and easy ones where little steering is required. This way, we hope our neural network will be able to handle difficult tracks, while still being competitive on easy tracks as well.

We've used heuristics from Seong Kim et al. (2012) to create a decent driver to collect data from. The hyperparameters have been manually tuned for each of the tracks to be completed as fast as possible, while not crashing.

We've collected the current speed, angle to the track axis, distance from the track center, RPM and the vector of 19 sensors measuring the distances to the track edges, oriented every 10 degrees in front of the car. These values can be used as inputs to our neural network. We've also collected the acceleration, braking, steering and gear as the expected outputs of our neural network. We've measured these values for a single round on the mentioned tracks, both with normal sensors and noisy sensors using the `-noisy` command-line option in TORCS. Data from the noisy sensors may allow us to train our neural network more robustly. We've measured every track three times using noisy sensors and once using normal sensors.

We used cross-validation to evaluate our models. For this purpose the collected dataset has been shuffled and divided into 10 parts. In each iteration 9 parts have been used for training the model, and

[1] Inspired by an amount of drunk coffee during the project.

1 has been used for evaluation. Mean squared error between a predicted output and a desired output has been used as an objective function.

3 Neural networks

We experimented with several Neural Network architectures and applied the approach introduced in the previous section to pick the best model for our purpose. Each model has been run for 1000 iterations with a learning rate of 0.1 and then a model with the lowest mean squared error score is chosen for the final version. An average mean squared error on the cross-validation task is reported in the Table 1.

Model	Mean Squared Error
Single Perceptron	0.163
Echo State Network	0.152
Multi-MLP	0.154
Single-MLP	**0.02**

Table 1: Average mean squared-error on the cross-validation task

It's valuable to point out that each of our experimented models **have their implementation done from scratch** (with the use of Apache Commons Math library for linear algebra primitives). Consequently, a brief description of each models will be presented below:

- In the Single Perceptron model (Rosenblatt, 1962) a prediction is computed according to the equation 1.

$$y = \alpha(\sigma(\mathbf{w}^T\mathbf{x} + b)) \tag{1}$$

  where $\mathbf{w}$ is a weight vector, $\mathbf{b}$ is a bias, $\mathbf{x}$ is an input vector, $\alpha(x)$ is a de-normalization function that scales a sigmoid output $\sigma(z)$ (that is from 0 to 1) to the output range. Cross-entropy error back-propagation is used for training (Sadowski, 2013). Separate perceptrons are used for acceleration and steering prediction. Weights are initialized with a Gaussian noise.

  The output range is determined by minimal and maximal possible values of the car control. Experiments have been done with normalization/denormalization on values based on the training data, but it didn't produce better results.

- Echo State Network (Jaeger, 2010) is used in the same manner as the Single Perceptron by applying the equation 2 for prediction.

$$\begin{aligned} x(n+1) &= \sigma(\mathbf{W^{in}u(n+1)} + \mathbf{Wx(n)} + \mathbf{W^{back}d(n)}) \\ y(n) &= \alpha(\sigma(\mathbf{W^{out}}[\mathbf{u(n)}; \mathbf{x(n)}])) \end{aligned} \tag{2}$$

  where $\mathbf{W^{in}}$ is a matrix from an input layer to the reservoir layer, $\mathbf{W}$ is a weight matrix for connections inside the reservoir, $\mathbf{W^{back}}$ is a weight matrix from the output layer to the reservoir and $\mathbf{W^{out}}$ is a weight matrix from the input layer and the reservoir to the output layer; $\mathbf{u}$ is a state of the input layer, $\mathbf{x}$ is a state of the reservoir and $\mathbf{y}$ is an output vector. Only $\mathbf{W^{out}}$ is trained via back-propagation of the cross-entropy error.

  Separated reservoirs are used for predicting acceleration and steering. The weight matrix $\mathbf{W}$ is normalized by a spectral radius of $\mathbf{W}$ multiplied by β, where $\beta < 1$ to satisfy an echo state property.

- In the Multi-MLP model (Rosenblatt, 1962) we expand the Single Perceptron model by an additional hidden layer:

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W_{in}}\mathbf{x} + \mathbf{b_{in}}) \\ y &= \alpha(\sigma(\mathbf{w}^T\mathbf{h} + b)) \end{aligned} \tag{3}$$

  where $\mathbf{h}$ is a state of the hidden layer, $\mathbf{W_{in}}$ is a weight matrix from the input layer to the hidden layer, $\mathbf{b_{in}}$ is a bias vector. We again used separate models for steering and acceleration and back-propagation of the cross-entropy error for training.

- In the Single-MLP model we expand our previous Multi-MLP model by introducing multiple nodes in the output layer and use 1 single model to predict steering and acceleration.

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W_{in}}\mathbf{x} + \mathbf{b_{in}}) \\ \mathbf{y} &= \alpha(\sigma(\mathbf{W}^T\mathbf{h} + \mathbf{b})) \end{aligned} \tag{4}$$

  where $\mathbf{W}$ is a weight matrix, $\mathbf{b}$ is a vector of biases and $\mathbf{y}$ becomes an output vector.

Consequently, in our final controller we used a single multilayer perceptron, with the following configuration:

- 21 input sensors (Table 2)
- 2 outputs (Table 3)
- Hidden layer of 25 neurons
- Sigmoid activation function

Sensor	Description
Speed	Current speed of the car (in km/h)
Angle to track axis	Current angle to center axis of the track
Track edge sensors (19)	Sensors providing distance to track edge for 19 angles

Table 2: Neural network inputs

Actuator	Description
Acceleration	Acceleration of car (0 to 1)
Steering	Steering angle

Table 3: Neural network outputs

All inputs and outputs are normalized to the range [0,1]. Experiments have been done with normalized inputs in the range [-1, 1], but it didn't provide any better results. We have also experimented with the *hyperbolic tangent* activation function instead of sigmoid, without notable improvement. Note that we do not use neural network predictions for braking, but instead apply heuristics as described below.

4 Heuristics

Using the trained MLP as discussed above, our driver was capable of completing tracks safely, but with a low average speed. This is explained by the fact that our training data was limited and gathered from a low-speed bot. To improve the race performance, we added a rule-based braking and acceleration policy to make it faster. These rules are shown in table 3 below. If applicable, they override the output for acceleration of the neural network. This was found to be more effective than using the neural network for braking predictions. Although these heuristics significantly improved our driver performance, they were tuned by hand, and could certainly be improved.

Speed (km/h)	**Front distance (m)**	**Action**
< 50	-	Accelerate (full)
-	> 150	Accelerate (full)
> 140	< 150	Brake
> 100	< 60	Brake
> 80	< 30	Brake

Table 4: Heuristics used to increase driver aggressiveness

Heuristics for opponents overtaking (Onieva et al., 2012) have been considered. Shortly, in the case of opponents detection closer than the threshold distance a car will try to overtake by moving to the right (or to the left) side and then back depends on the condition on the track. Our experiments have shown that increasing a speed improves results better than trying to overtake an opponent, therefore it has been decided to exclude the described heuristic in the final version.

5 NeuroEvolution of Augmenting Topologies (NEAT)

A big area of research is the use of neuroevolution for creating of neural networks. One well-known example of neuroevolution is the NEAT algorithm, which has been successfully used to develop AI for games (Munoz et al., 2009). We decided to use an implementation of NEAT from the Encog Machine Learning Library (enc) for our project.

The NEAT algorithm is used to create a network topology by evolving genomes. Each individual in the population is described by the genome, corresponding to a multilayer perceptron. Rather than just tuning the weights, NEAT changes the network structure by adding nodes and removing connections. The algorithm requires a single fitness function to optimize. Because random initialization of our NEAT network is unlikely to produce a driver that could finish a track or produce meaningful results, we first evolved an initial genome by using NEAT on the training data. Our configuration looks like this:

- Initial layout of 21 input and 2 output neurons
- Population size of 5 individuals
- Mutation and crossover: default NEAT parameters as used in Stanley and Miikkulainen (2002)
- Fitness value: the mean squared error on the training set, which consists of the data collected from 6 tracks.

After 10K generations, we obtained a driver that could complete most tracks. However, as we expected, the performance was worse than the non-evolutionary MLP approach trained with error back-propagation. The next step in evolving the NEAT topology was to use actual racing results as a fitness function, instead of training on the collected data. A suitable fitness function would be:

$$Fitness = -(C * Damage) - LapTime \tag{5}$$

This fitness function favours a low lap time and penalizes damage. Starting with the driver genome obtained in the previous step, we tried to run several races where all individuals race against each other to obtain a fitness score. Unfortunately, issues with TORCS prevent us from completing this part of evolution. We were not able to run a reasonable number of NEAT iterations because the non-GUI mode of TORCS crashed on our system.

6 Swarm Intelligence

We've considered a simple approach for applying swarm intelligence in our controller by letting two cars race together as a team. In this approach one car would be doing its best to get to the finish as fast as possible, while the other would hinder competitors by, for example, braking in front of them or ramming them of the road. This role selection could be made arbitrarily at the start of the race and possibly switch during the race if, for example, the car aiming to win the race ends up crashing.

The hindering behaviour could either be performed by some learning algorithm, such as a neural network. However, this would require good training data containing the opponent distance sensors (36 sensors all around the car, measuring the distance of opponents within a 10 degree sector) as inputs for the learning algorithm, and the behaviour (steering, acceleration, braking, gear switching) as the outputs. Acquiring this training is hard, though, so using a heuristic would be a better option. An example heuristic would be a rule-based system in which the opponent sensors are used to detect cars behind the controlled car, after which the controlled car can steer in a way such that it's exactly in front of the opponent car. And when the opponent car would be within some minimum distance behind the controlled car, the controlled car can brake to force the opponent to also slow down. This behaviour would need to balanced such that it doesn't overdo this and ends up standing still or falling behind entirely on the other racers.

Creating and tuning the exact heuristics turns out to be hard, however, and can easily cause the car to crash. For sake of the competition, we decided to drop the swarm intelligence ideas, so we can guarantee a stable driver. Both our drivers now race to win and have no sense of one another. Future work will have to find a good swarm intelligence approach and look into the benefits of using this.

7 Results

We have run our TORCS controller on different tracks and made it compete with other state-of-the-art bots that are included in TORCS, *berniw 1* and *berniw 10*. Results are presented in the table 5. For each of the three selected tracks, we measured the lap time and damage.

Track	**Caffeine**	**berniw 1**	**berniw 10**
Aalborg	01:51:39 / 58	**01:38:10 / 1680**	01:47:00 / 2011
1-Seoul	55:48 / 16	01:17:68 / 1940	**39:97 / 146**
E-track 3	02:18:48 / 16	**01:35:75 / 0**	02:02:83 / 0

Table 5: Measurements of lap time / damage for Caffeine with our final setup, and two TORCS built-in bots

These results show that our controller was not able to win the race against the berniw bots. However, it completed the tracks with little damage. In the track Aalborg Caffeine was not far behind.

8 Final Remarks

We built a working autonomous AI for The Open Car Racing Simulator (TORCS). We've experimented with Computational Intelligence techniques such as reservoir networks, evolutionary algorithms (NEAT) and looked into applying swarm intelligence. We found, however, that the best results were

obtained using a simple feed-forward neural network. A large portion of the trouble we encountered was due to the difficulty of working with TORCS.

Our final controller was able to finish all tracks we tested on with little damage, compared to other existing bots. However, our controller was significantly slower than the existing bots. Nevertheless we did end up sixth place at the UvA racing contest. In order to make our car driver faster, we would need better training data obtained from a more sophisticated driver, or implement more advanced heuristics. We would have liked to implement an evolutionary approach to improve our driver, such as NEAT which has shown to be successful in previous TORCS championships (Loiacono et al., 2008). Unfortunately, our issues with TORCS prevented us from fully implementing NEAT during this project.

References

Encog Machine Learning Framework. http://www.heatonresearch.com/encog/. Accessed: Dec 2015.

L. Cardamone, D. Loiacono, and P. L. Lanzi. Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1179–1186, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: 10.1145/1569901.1570060. URL http://doi.acm.org/10.1145/1569901.1570060.

H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks âĂŞ with an Erratum note. 2010. URL http://minds.jacobs-university.de/sites/default/files/uploads/papers/EchoStatesTechRep.pdf.

K.-J. Kim, J.-H. Seo, J.-G. Park, and J. C. Na. Generalization of torcs car racing controllers with artificial neural networks and linear regression analysis. *Neurocomput.*, 88:87–99, July 2012. ISSN 0925-2312. doi: 10.1016/j.neucom.2011.06.034. URL http://dx.doi.org/10.1016/j.neucom.2011.06.034.

D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds, and Y. Saez. The wcci 2008 simulated car racing competition. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 119–126. IEEE, 2008.

J. Munoz, G. Gutierrez, and A. Sanchis. Controller for torcs created by imitation. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 271–278. IEEE, 2009.

E. Onieva, D. A. Pelta, J. Godoy, V. Milanes, and J. Perez. An evolutionary tuned driving system for virtual car racing games: The autopia driver. *Int. J. Intell. Syst.*, 27(3):217–241, Mar. 2012. ISSN 0884-8173. doi: 10.1002/int.21512. URL http://dx.doi.org/10.1002/int.21512.

F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Washington, Spartan Books, 1962.

P. Sadowski. Notes on Backpropagation, 2013. URL https://www.ics.uci.edu/~pjsadows/notes.pdf.

T. Seong Kim, J. Chae Na, and K. Joong Kim. *International Journal of Advanced Robotic Systems*, 2012. URL http://cdn.intechopen.com/pdfs/39283.pdf.

K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.