Sociopathfinder

A fast and competitive virtual car controller

Dana Kianfar[1], Jonas Köhler[2], and Jose Gallego-Posada[3]

University of Amsterdam
{dana.kianfar[1], jonas.koehler[2], jose.gallegoposada[3]} @ student.uva.nl
December 16, 2016

1 Introduction

In this report we describe our efforts in applying methods from different paradigms of Computational Intelligence to develop controllers for virtual car racing on The Open Racing Car Platform (TORCS) [1]. State-of-the-art controllers employ a variety of techniques ranging from optimizing physics-based heuristics using evolutionary algorithms [2] [3], to training multi-layer perceptrons to imitate existing controllers [4]. We propose a novel method based on swarm communication, and describe our controller **Sociopathfinder**. After explaining our design process, we elaborate our controller and conclude with quantitative results. Our controller achieved the 1st position in the 40-race general competition and the 4th position in the finals of the Computational Intelligence course tournament at the University of Amsterdam in December 2016.

2 Design

We designed our controller in three stages; first by achieving fast driving and precise steering using a combination of neural networks and heuristics, followed by implementing overtaking heuristics, and finally introducing swarm communication for our two controllers.

Neural networks Inspired by [4], we generated training data from the `Inferno` built-in controller in a competition setup with other built-in controllers on all tracks. These controllers have access to global information, and are able to compute the optimal spline within a track. After experimenting with different architectures of multi-layer perceptrons (MLP), Extreme Learning Machines, Spiking Neural Networks and Echo State Networks, we chose the MLP with which we achieved better approximation of the behaviour with respect to the steering, acceleration and brake. We observed a strong tendency to over-fitting even for small architectures. The model generalizes poorly for unseen scenarios such as being off-track and crashing into opponents or track edges. Moreover, our MLPs were susceptible to noise which resulted in unexpected steering or acceleration.

Genetic algorithms To improve on that we applied NEAT [5] for both off and on-line learning. For the supervised task, we used the mean squared error (MSE) as the objective function. Given the large number of hyper-parameters, specifically the initial topology, this did not result in a better performance than the MLPs. For the online task, we used the race position as a fitness measure and gave full driving autonomy to the neural network. To this end, we created a parallel execution environment of TORCS allowing an efficient evaluation of the fitness. Although this latter approach seemed promising in the beginning, the computational cost of the evaluation of the fitness function and the large number of hyper-parameters restrained the applicability of this approach within the time scope of the project.

Improved heuristics To achieve better behavior in critical scenarios involving crashes and competition, we implemented heuristics for speed, acceleration, brake, and recovery as described in [2], and referred to [6] for ambiguities regarding technical implementation of said heuristics. We improved the overall performance as our controller became more robust. In addition, we added competitive behaviour using our own overtaking heuristic. However, we noticed excessively prudent behavior in curves.

Swarm behaviour Simple rule-based approaches for collaboration, such as using one car for blocking competitors, do not exploit the full potential of the swarm intelligence. We devised a simple bottom-up strategy based on pheromone dropping inspired by Ant Colony Optimization techniques. This method enabled the cars to leverage their shared experiences to reduce the lap time on each track.

Ensemble fine-tuning Our final ensemble-based model can be described by 46 real valued parameters consisting of model mixture coefficients, heuristic values and swarm strategy behaviour. We used two different evolution strategies, namely *Differential Evolution* and a hand-crafted genetic algorithm, to optimize the values for these parameters using the race position of the driver against several TORCS bots as an objective function. While we could observe some improvement after several hours of training, the resulting drivers showed unstable behavior. Finally, hand-tuning based on the values obtained from the genetic optimization yielded a good mixture of some innovations learned throughout the evolution and the designed behaviour that we derived by deduction.

3 Description of the Controller

Components and architecture Our driving algorithm can be broken down into three steps: computing the steering, computing a target velocity, and computing corresponding acceleration and brake. For each of the for tasks we use an ensemble of different methods to return a final action. A general illustration of the control flow is presented in Figure 1.

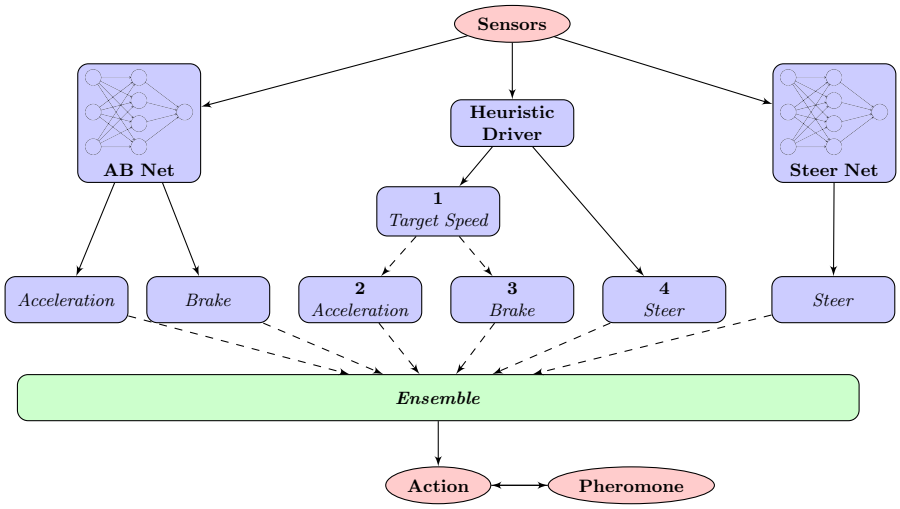


Fig. 1: Flow diagram for the implemented system.

Target velocity The target velocity is a mixture of two different heuristics together with the result of the swarm communication. All predictions are combined using a weighted harmonic mean. We chose the latter rather than the arithmetic mean since it is more robust to outliers.

Steering For the steering we combine the output of a heuristic together with the prediction of a neural network using a weighted average.

Acceleration and brake Braking and acceleration are computed according to the current target velocity. Braking and acceleration are a combination of three components: a constant term, a heuristic term and a neural network controller. The returned values are combined using a weighted average.

Heuristics COBOSTAR [2] employs a set of optimized on-track and off-track strategies that relies only on local sensor information. The controller computes two characteristic quantities: α, the angle with respect to car pointing to the direction of largest track sensor, and v_t, the target velocity computed from a polynomial combination of *alpha*, the largest track sensor, and a set of optimized parameters. Naturally, v_t is measured differently when the car is off-track.

On-track strategy The controller computes α and v_t as shown below:

$$\alpha = \alpha' - 0.5 + \frac{d - d_l}{2d - d_l - d_r} \tag{1}$$

where d is the largest track sensor, d_l and d_r are respectively the sensors on its left and right, and α' is the angle of the largest track sensor. Note that the car initializes the track sensors on every 10 degrees from $[-\pi, \pi]$. Also note that the

angle α is a value within the range $[0, 18]$ representing the 19 track sensors from left to right. For a detailed analysis of the parameters indexing p and θ refer to [2].

The steering angle τ is calculated by normalizing α to the steering direction and scaling by a constant parameter. The acceleration and brake are determined by comparing the current velocity of the car with the target velocity. The car accelerates fully if it is below the target velocity. If the current velocity is within a range of the target velocity, no acceleration or brake is applied. In the case where the car is going faster than the target velocity, the brake b is computed as shown below.

$$v'_t = p_1 + p_2 d + p_3 \max\{0, \frac{d - \theta_1}{\theta_2 - \theta_1}\}^{p_4} - p_5 \left(\frac{\alpha - 9}{9}\right)^{p_6} \tag{2}$$

$$\tau = p_{10}(9 - \alpha) \tag{3}$$

$$b = \min\{1; p_9(v_c - p_8 v_t)\} \tag{4}$$

Off-track strategy In order to compensate for crashes, mistakes or slippage on low-friction tracks, the car adopts a more prudent acceleration and braking regime. A target angle β with respect to the track axis is computed and used to estimate the target speed v_t:

$$\beta = sgn(t)(|t| - q_1)q_2 \tag{5}$$

$$v_t = q_3 + q_4(\max\{0, 1 - q_5|\gamma - \beta|\}) \tag{6}$$

where γ representing the track angle sensor, and t representing the track position sensor.

Artificial Neural Networks We use two multi-layer perceptrons with different architectures and training. The first one is responsible for predicting the acceleration and brake (AB-Net) for the given situation. The second one predicts the steering (Steer-Net).

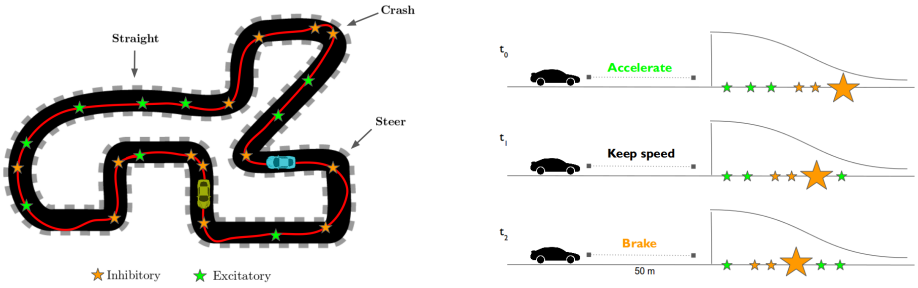
AB-Net The AB-Net has two hidden layers both having 100 units and uses a hyperbolic tangent activation. It is trained using ADAM[7] on approximately 1.6M datapoints generated from the TORCS built-in controller *inferno*, using mini-batches of size 200. We used a 10% validation set to monitor the training and achieved a 0.0067 training squared-loss. We use the `angle, rpm, speedX, speedY, track, track position, opponent` sensors as input and apply whitening to normalize our data. The network has two outputs for acceleration and brake.

Steer-Net The Steer-Net has three hidden layers with respectively 100, 100, and 70 hidden units and a hyperbolic tangent activation. It is trained on the same dataset and in a similar fashion to AB-net, achieving a 0.0044 training squared-loss. For the input layers, we feed the `speedX` and `speedY` sensors, and two features that measure the curvature: `max-angle`, which is the angle of the largest track sensor and `curvature`, measured as $\frac{d - d_l}{2d - d_l - d_r}$ for each sensor reading in the data, similar to Equation 2.

Swarm communication We designed a new method to optimize the acceleration and braking of our agents. Our result is an unsupervised learning procedure using implicit communication between all involved robots based on a *pheromone* trail left behind while racing. A pheromone p contains three real numbers: an *emission time* $t_p \in [0, \infty)$, an *intensity* $I_p \in (-\infty, \infty)$ and its *position* $d_p \in [0, \infty)$ on the track. The position is given as the distance to the start since the agents have no global spatial information available. To make the time independent of potential network delay we use the tick count.

Every pheromone dropped by one of the agents can be sensed by all other agents (using a file-system based message queue). The intensity value in each pheromone encodes information regarding whether the agent should accelerate ($I_p > 0$) or brake ($I_p < 0$) in the area. After a certain number of ticks a pheromone is deleted to give an emphasis on newer information. At each step the agent drops pheromones based on the current sensor data and adjusts its action depending on the currently sensed pheromone intensity close to it.

We use three simple rules for the agents to drop pheromones. First, every 20 ticks the agent drops a pheromone based on its current steering action $S \in [-\pi/2, \pi/2]$ with intensity $I_P = \theta_0 S^2$. Here $\theta_0 \in (0, \infty)$ is a free parameter controlling the amount of pheromone dependent on the steering. Second, whenever the front distance sensor senses a value below a threshold $\theta_1 \in (0, \infty)$, it drops a pheromone with a high intensity $\theta_2 \in (0, \infty)$. Finally, whenever the car experienced a crash (identified by a drop of velocity higher than a threshold $\theta_3 \in (0, \infty)$), it drops a pheromone with intensity $\theta_4 \in (0, \infty)$). The first encodes a rough estimate about the current track curvature. The second and the third model previous failure situations. These rules are illustrated in Figures 2a and 2b.



(a) Pheromones dropping (b) Reaction to pheromones

Fig. 2: Swarm intelligence strategy

The intensity of a single pheromone p, sensed by an agent a, is weighted using an exponential decay:

$$w(a, p) = \exp\left(-\theta_5 \left(\min(d_p - d_a, d_T - d_p + d_a) - \theta_6\right) / \theta_6\right) \tag{7}$$

where d_T denotes the length of the track[1] and d_a the current position of the agent. Taking the weighted sum over all pheromones gives the total sensed intensity:

$$I(a) = \frac{\sum_p w(a,p) I_p}{\sum_p w(a,p)} \tag{8}$$

Based on this, we estimate the new target velocity by:

$$V = max\left(\theta_7, \theta_7 + \theta_9\left(\theta_{10} - I(a)\right)\right) \tag{9}$$

where θ_7 denotes a baseline velocity, θ_{10} a bias term at which pheromone level we want to increase or decrease the velocity, and θ_9 a scaling factor that controls the influence of the pheromone level.

Implementation In the design phase we worked with Python using the numerical library *numpy* [8]. For training the MLP we used the implementation of *scikit-learn* [9]. The feedforward stage of the MLPs and the PCA has been written by us, using a fast matrix multiplication in Java [10]. The pheromone system is self-designed and written from scratch. The heuristic components are re-implemented and improved based on the referred articles. For NEAT we relied on *python-neat* [11] and *Encog* [12]. For the genetic optimization we used the *SciPy* implementation of differential evolution, and the Java library *Jenetics* [13] for the hand-crafted genetic algorithm. The other mentioned neural network models have been implemented by ourselves.

4 Results

To quantitatively evaluate our controller's performance, we selected a set of benchmark tracks on the basis of their uniquely difficult characteristics, such as sharp and unexpected curves, low-friction, and jumps. We collected lap times from `SimpleDriver`, `Inferno` and our controller. To demonstrate the effectiveness of of our swarm communication, we present the first and fifth lap times on each race.

We took the lap time of our car and the race position against a random mix of built-in bots as fitness measures. Table 1 shows, that our final model clearly beats the baseline and competes with `Inferno`.

After the fifth lap our driver performs better on the majority of tracks. This is due to the incremental optimization strategy of pheromone sensing. On every round the agents improve their acceleration and braking, resulting in an overall decline in lap time. Accidents that occurred during the first laps led to stronger braking in those areas, and more confident driving in others in which the heuristic would act cautiously.

These promising results motivate further investigations on which other types of information encoded in the pheromones could improve the performance of autonomous car controllers.

[1] This can be estimated by the *distance to start* sensor in the moment when the race is started, as all cars are behind the finish line.

Track	`Simple Driver`	`Inferno`	`Sociopathfinder Lap 1`	`Sociopathfinder Lap 5`
Dirt-4	2:52	1:23	1:20	**1:15**
E-Track 5	1:10	0:38	0:34	**0:29**
Michigan	1:15	0:42	0:43	**0:37**
Dirt-2	3:12	**1:27**	2:00	1:28
Aalborg	3:15	**1:20**	1:25	**1:20**
Corkscrew	2:56	**1:28**	1:47	1:33
Alpine-1	5:29	**2:16**	2:42	2:24
Dirt-3	2:09	1:09	1:07	**1:02**

Table 1: Comparison of lap time between SimpleDriver, Inferno and Sociopathfinder

References

1. B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, "TORCS, The Open Racing Car Simulator," 2014. [Online]. Available: http://www.torcs.org
2. M. V. Butz and T. D. Lonneker, "Optimized sensory-motor couplings plus strategy extensions for the torcs car racing challenge," in *2009 IEEE Symposium on Computational Intelligence and Games*, Sept 2009, pp. 317–324.
3. M. Bonyadi, Z. Michalewicz, S. Nallaperuma, and F. Neumann, "Ahura: A heuristic-based racer for the open racing car simulator," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. PP, no. 99, pp. 1–1, 2016.
4. C. Athanasiadis, D. Galanopoulos, and A. Tefas, "Progressive neural network training for the Open Racing Car Simulator," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, Sept 2012, pp. 116–123.
5. R. Stanley, Kenneth O.Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.
6. huiwq1990, "Genetic algorithms," https://github.com/huiwq1990/GeneticAlgorithm/tree/master/COBOSTAR, 2013.
7. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980
8. E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/
9. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
10. R. Sedgewick and K. Wayne, *Matrix.java*. Princeton University, 2016. [Online]. Available: http://introcs.cs.princeton.edu/java/22library/Matrix.java.html
11. Code Reclaimers, LLC, *neat-python*, 2016, r0.1. [Online]. Available: http://neat-python.readthedocs.io/
12. J. Heaton, "Encog: Library of Interchangeable Machine Learning Models for Java and C#," *Journal of Machine Learning Research*, vol. 16, pp. 1243–1247, 2015. [Online]. Available: http://jmlr.org/papers/v16/heaton15a.html
13. F. Wilhelmstötter, "Jenetics," 2016, v2.0. [Online]. Available: http://jenetics.io/