Computational Intelligence
Practical Assignment - Report

Erwin Huijzer[1], Jan-Willem Feilzer[2], and Mathijs Mul[3]

[1] Vrije Universiteit,
Studentnummer 2551603
[2] Vrije Universiteit,
Studentnummer 2596978
[3] Universiteit van Amsterdam,
Studentnummer 6267939

Abstract. This report describes the TORCS controller ranked 1st place in the Computational Intelligence Course 2016-2017 TORCS Competition. The controller uses heuristics for steering and a Multilayer Perceptron Neural Network for acceleration and brake. After initial training of the network, the resulting weights are optimised using differential evolution. Additionally, a heuristic recovery module is implemented in order to improve performance.

Keywords: machine learning, neural network, multilayer perceptron, evolutionary computing, differential evolution

1 Introduction

The goal of this assigment is to apply computational intelligence techniques in a practical setting. The platform on which this takes place is called TORCS, an open racing car simulator. The assignment consists of three different stages. In order to pass these stages a controller is built. This controller should be able to keep the racing car on the track (stage 1), race against opponents (stage 2), and lastly cooperate with a team member to gain a competitive advantage (stage 3). The following sections will describe the approach chosen in order to tackle the different problems faced for each stage.

2 Approaches and underlying rationale

Prior to coding, relevant literature was reviewed. An overview of the most important sources used can be found in the section References.

The next step was building a rule-based controller of which the data was saved. These data were used to train a Multilayer Perceptron Neural Network (MLP). The resulting weights were evolved using a differential evolution algorithm. Additionally, some specific routines were hardcoded. In the subsections below details of the chosen approach are described.

2.1 Multilayer Perceptron Neural Network

An MLP is a feed-forward neural network. This is to say that there are no cycles between the units, as happens in recurrent neural networks (e.g. LSTM). One of the design principles for the network was to limited the number of nodes and associated number of weights to avoid a high dimensional solution vector during evolution. After experimenting with both LSTM and MLP, it was concluded that the best results were obtained using an MLP neural network; therefore this type of network was used for the final controller.

As mentioned earlier, the first step was to build a rule-based controller. The data provided by this driver were used to train the network. Below, the 6 input variables that are fed into the MLP are listed:

- `speed`: resultant speed, calculated from the default speed (`speed_x`) and the lateral speed (`speed_y`) as follows: $\sqrt{(\texttt{speed\_x}^2 + \texttt{speed\_y}^2)}$
- `track_position`: Distance between the car and the track axis, -1 when the car is on the right edge of the track and +1 when it is on the left edge of the car.
- `angle_to_track_axis`: Angle between the car direction and the direction of the track axis.
- `radius`: the radius of curvature, i.e. the radius of the approximating circle at a position of the track. If we let $y = f(x)$ denote the curve, then the radius of curvature at any point x can be expressed as stated in formula 1.

$$\frac{(1 + (\frac{dy}{dx})^2)^{\frac{3}{2}}}{\frac{d^2y}{dx^2}} \tag{1}$$

  To approximate this value, the sensor along the track axis is used, along with the sensors that are immediately adjacent. Based on their values, the circle radius is calculated, as illustrated in the Figure 1.

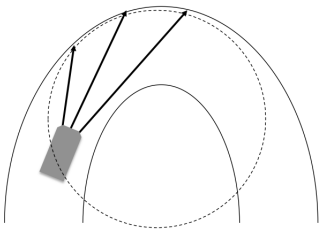

Fig. 1. Radius of curvature

- `corner_direction`: denotes the direction of a curve. This parameter is set to -1 ,1 and 0 for a left, right and no corner respectively.
- `trackedge_9`: the front-facing sensor, distance between the track edge and the car within a range of 200 meters.

The network has a single hidden layer consisting of 10 units. It has a single output variable, which corresponds to either acceleration or brake. The corresponding architecture of the network can found in Figure 2.

Training the network was accomplished through backpropagation using the DL4j Java library. As activation function, `tanh` is used. Thru some manual tuning, the values 0.005 for learning rate and 1480 epochs were found to give the best training results. Since further optimization is done using neuroevolution, no extensive training is done.

The acquired weights were then fed into the controller to predict whether the next action should be accelerate, brake, or neither of the two. Note that there is no output variable for steering. Steering was determined using a hardcoded method. Its characteristics can be found in section 2.3.

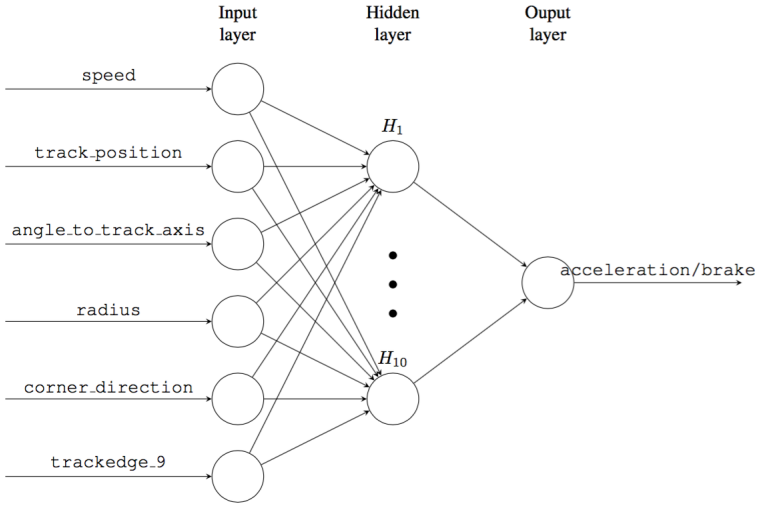

Fig. 2. MLP used for training

2.2 Differential evolution

The controller was evolved by applying differential evolution to the weights obtained from the neural network. Differential evolution dates from 1995 and is therefore a relatively new member of the evolutionary algorithm family. The main difference between differential evolution and other evolutionary algorithm variants lies in its reproduction method: differential mutation. Given a population of candidate solution vectors in $\mathbb{R}^n$, a new mutant vector $\overline{x}'$ is produced by adding a perturbation vector to an existing one,

$$\overline{x}' = \overline{x} + \overline{p} \tag{2}$$

where $\overline{p}$ is the scaled vector difference of two other randomly chosen vectors. The scaling vector F controls the rate at which the population evolves [2].

$$\overline{p} = F * (\overline{y} - \overline{z}) \tag{3}$$

Anthony (2013) has successfully applied differential evolution for neuroevolution of a TORCS controller using population size 10 and crossover rate 0.7. The same settings are used for this controller. The neural network is trained on a full data set ten times, each time the obtained weights (81) are stored in a vector, which represents one individual of a population. Thus resulting in a population of ten individuals. To complete initialisation of the population, each individual is evaluated. The evaluation function uses each individual vector as input to the controller, starts a race on a prespecified track, and stores its corresponding lap time. This lap time is then used as an individual's fitness value (lower lap time represents higher fitness).

When the evaluations limit is reached, the algorithm terminates. After termination, the individual with the lowest fitness value is selected. The corresponding vector is then used as input to the controller. A breakdown of the methods used for differential evolution can be found in Table 1.

Representation	Real-valued vectors
Recombination	Uniform crossover
Mutation	Differential mutation
Parent selection	Uniform random selection of the 3 necessary vectors
Survival selection	Deterministic elitist replacement (parent vs. child)

Table 1. Differential evolution breakdown

2.3 Heuristics

In addition to the neural and evolutionary learning described in the preceding subsections, the controller makes use of a few hardcoded heuristics for cases in which the learning procedures did not provide fully satisfying results. They are described below:

– *Recovery*

Recovery mode is activated when the car's speed equals zero. The routine distinguishes four possible situations, listed below as they appear in Figure 3, moving from the upper left corner in clockwise direction.

1. The car is located on the left side of the road, turned to the left. It then goes in reverse, and steers left.
2. The car is located on the left side of the road, turned to the right. It then moves forward and steers left.

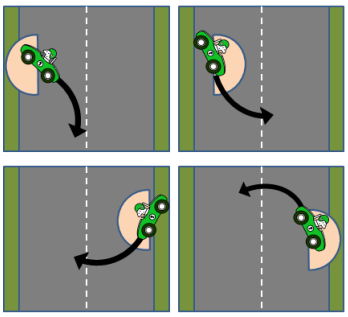

Fig. 3. Recovery mode: four situations

3. The car is located on the right side of the road, turned to the left. It then moves forward and steers right.
4. The car is located on the right side of the road, turned to the right. It then goes in reverse, and steers right.

In all of these situations, acceleration is continued until the other side of the track is reached or the car finds itself within 30 degrees of the driving direction.

– *Steering*

For the steering, four coded methods are implemented:

1. If drifting of the car is detected, steering is aligned with the axis.
2. If the car is located near the edge of the track, it drives back to the middle.
3. Under normal circumstances, the controller steers in the direction of the furthest-facing sensor.
4. Other situations are handled by correcting the car in such a way that the steering realigns with the axis.

– *Acceleration / Brake*

For each track, both the neural controller and the heuristic, rule-based controller were tested. The best result determines which controller was used to either accelerate or brake.

- *Neural Network.* If the prediction function outputs a value of less than -0.2, brake is set to 1.0. If it outputs a value larger than 0.2, acceleration is set to 1.0. These threshold values were adopted to normalize the acceleration. For values in between, acceleration and brake are both set to 0.0.
- *Rule-based.* The distance the controller of the car can look ahead is determined based on its speed. The target speed is based on the calculated radius of curvature. If the current speed is more than 5 units larger than the target speed, brake is set to 1.0. If the current speed is larger than the target speed, but less than 5 units, acceleration and brake both equal 0. If the current speed is less than the target speed, acceleration is set to 1.0.

3 Estimated performance

This section will deal with the performance of the different types of controllers; some performance measures and aspects that affect performance will be discussed. As noted in section 2.2, the fitness of a controller for a certain track is solely determined by the lap time it achieves on that track. Due to the nature of differential evolution (deterministic elitist survival), a more complicated fitness function is not required. In addition, a controller is evolved per standard track, and per type of track (road, dirt or oval) it is checked which type of controller had the best performance.

Next, the best controller for every type of track is selected and its resulting weights are used as a starting population for a next stage of differential evolution. The tracks that are typically selected for further evolving the controller are Alpine-2, E-Road and Dirt-1. In total almost 7000 evaluations on 35 tracks have been performed, resulting in multiple robust controllers.

The two main influences that have the biggest impact on the performance of a controller are the type of track it races on, and whether a guardrail is present on the selected track or not. The type of track has a big influence, since on dirt tracks a lot more drifting occurs than on road or oval tracks. The biggest difference when guardrails are present is the speed at which the car instructed by the controller can drive. Due to these difficulties that are taken into account, two different types of controllers are used during a race. The first lap of each race, a relatively ”cautious” controller is used. This is the controller that drifts more in order to stay on the track. The second lap, a different, faster controller is used, the ”aggressive” controller. This controller benefits from tracks where guardrails are present. Since damage of the car is not an issue for this assignment, this is a very useful controller. Driving into the guardrail is relatively harmless due to the speed the car drives at (i.e. after driving into the guardrail the car still possesses significant velocity).

The choice of controller can happen in two ways. One scenario is that the first controller is outperformed by the first, in which case the second controller is used for the remaining laps. The other scenario is that the second controller is slower than the first. If this is the case, the first controller will be used starting from the moment that it becomes clear that the second controller does not outperform the first. The main benefit of this approach is that it guarantees reasonable results. Of course, if the track type is known prior to a race, this approach will not be ideal. Since this is not the case, this more cautious method is used to compete and gives satisfying results.

To conclude this section, a small summary of some of the resulting values of the used approach is given. First of all, 5 controllers were tested on 10 different tracks. A controller’s performance is then measured by again minimizing the summation of the achieved lap times. In Table 2 the best lap times for the most used tracks can be found. For each track three lap times are displayed. These lap times are the results of a rule-based controller, a pre-evolved controller, and lastly an evolved controller. It can immediately be noted that the evolved controller has the best results, as expected.

Track	Rule-Based	Pre-Evolved	Evolved
Alpine-2	117.608	89.846	81.69
E-Road	79.014	77.918	75.274
Dirt-1	53.858	42.772	31.136

Table 2. Lap times per track for different types of controllers

4 Evaluation and interpretation of results

The results of the final controller are in line with the expectations: both the type of track and the presence of guardrails greatly influence performance. When guardrails are present, the best results are obtained using the more aggressive controller. Guardrails have a negligible influence on the relatively cautious controller. Obviously, if such information about the track is known in advance, the more cautious controller does not always have to be adopted. This would lead to a significant improvement on tracks without a guardrail.

More generally, the decision between the controllers, and the moment when this decision is made, always has a great influence on the performance of the driver. As described in the preceding section, this choice is now made at the earliest moment during the second lap where comparative evaluation of the two drivers is possible. Thus, even if the aggressive controller takes over in the second lap, a lot of time can be wasted on the cautious driver that always performs the first lap. Conversely, if the cautious controller performs better, it is preferable not to adopt the aggressive driver in the second lap. Hence it is worth investigating if the choice between the two drivers can be made at an earlier point in time. Also, biases towards driver should be critically assessed. Since, the car possesses no speed when the first lap is started, there will always be a bias towards the second controller.

Furthermore, it is likely that better results can be obtained when a neural network is also used to determine steering. Currently, the steering is completely rule-based. As performance greatly improved after the neural network was trained for acceleration/brake on the data of the initial rule-based controller, it is to be expected that teaching the network how to steer would also be beneficial.

If steering is taught to the network as well, it is possible to perform co-evolution on the weights for steering and those for acceleration/brake. Co-evolutionary algorithms excel at capturing the interaction between variables. Since steering and acceleration are very much related, this would be a promising approach to model and optimise their dynamics together.

As for the evolutionary algorithm that is adopted, many variations are possible. Currently, differential evolution is used, and although it performs very well, the decision to adopt it was a pragmatic one. There may be alternatives that are better suited for this context.

Many other suggestions can be made. For instance, the driver could make use of the track map that becomes available after the first lap on a circuit is completed. For unknown circuits, this could offer good guidance. Also, in racing against others, the

opponents' sensors could be used. As for the coded heuristics, these could also be optimised. In particular, a better recovery mode is imaginable. Yet, although there are many options to improve the driver's results, we can conclude that the controller presented in this paper can obtain satisfying results as demonstrated by its 1st place in the Computational Intelligence Course 2016-2017 TORCS Competition.

References

1. Onieva, E., Pelta, D. A., Alonso, J., Milanés, V., Pérez, J.: A Modular Parametric Architecture for the TORCS Racing Engine. IEEE Symposium on Computational Intelligence and Games. 256–266. 2009.
2. Perez, D., Saez, Y., Recio, G., Isai, P., Evolving a rule system controller for automatic driving in a car racing competition. IEEE. 336–342. 2008.
3. Anthony, P., e.a., Evolving controllers for simulated car racing using differential evolution. Asia-Pacific Journal of Information Technology and Multimedia. 57–68. 2013.
4. Eiben, A. E., Smith, J. E.: Introduction to Evolutionary Computing. Berlin: Springer, 2003.
5. Quadflieg, J., Preuss, M., Rudolph, G.: Driving as a human: a track learning based adaptable architecture for a car racing controller. Genet Program Evolvable Mach. 433–476. 2014.
6. Quadflieg, J., Preuss, M., Kramer, O., Rudolph, G.: Learning the Track and Planning Ahead in a Car Racing Controller. IEEE. 395–402. 2010.
7. Yap, K. L., Incremental evolution for simulated car racing with differential evolution. Own thesis. 2015.