Developing car driving agent in the TORCS virtual environment

(Tworzenie agenta kierującego pojazdem w wirtualnym środowisku TORCS)

Kacper Kulczak

Praca inżynierska

Promotor: dr Paweł Rychlikowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

11 stycznia 2019

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

Autonomous cars are a very fashionable subject. The idea has a lot of benefits. Most of accidents on roads, occurs due to human mistakes. Programmes don't get tiered or distracted while driving a vehicle. Also they can save a lot of fuel with optimal decisions. That's why companies around the globe put a lot of effort into making the car software for autonomous driving. It is very interesting task from artificial intelligence perspective. Description of the world is huge. It includes near objects, borders of the street, speed, turning force, engine rotation and much more. The output of such programme is quite simple in compared to the input size. We need only to specify new position of steering wheel and decision if we want to speed up or slow down. During my work I wanted approach the problem by building my own agent capable of drive safely on a racing track.

1.1 TORCS Environment

"The online racing car simulator"(TORCS) is highly portable, multi platform car simulator, with various cars and tracks. The simulation features a simple damage model, collisions, fuel consumption, tire and wheel properties (springs, dampers, stiffness), aerodynamics and much more [1]. It is designed to enable programed agents compete against each others. There is very ditalied instruction on developing your own bot. It has to be written as C++ loadable library and it is attached to main thread during program startup.

I have much more experience with programming in high level languages, so clean TORCS environment did not meet all my expectations.

Figure 1.1: Screen shot from TORCS race [1]

1.2 Simulated Car Racing Championship

SCRC competition took place between 2007 - 2015 with some breaks. It was organized by the University of Adelaide and the Politecnico de Milano. They used TORCS engine for competition, but organizers provided official patch which changed architecture of the programme. After patching, TORCS became client-server application which allows multiple bots communicate with game engine via UDP connections.

Server sends current sensor inputs (track border, speed, lap time, etc...) and waits for 20ms for the client action (gas, break, steer, etc...). API is details are descrived in table from manual. With that change participants cant choose whatever language they want. That's why I decided to use patched version of TORCS game engine in version 1.3.7

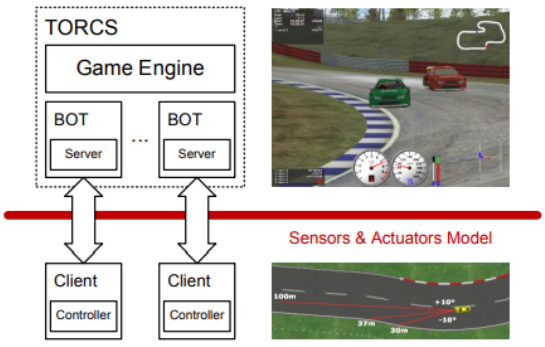

Figure 1.2: Simulated Car Racing Championship - architecture overview [2]

Organizers provided "clients" programmes only in Java and C++. I didn't want to implement communica-

tion interface on my own, beacuse it's time consuming and uninteresting task. After some research, I found `http://xed.ch/p/snakeoil` project, which provides Python Class handling communication with TORCS server. It allowed me to add layer of abstraction and focus directly on driving functionality.

1.3 Game sensors and actions parametes

Server provides very accurate description of virtual environment state. All parameters are described in 1.1 1.2 table attached to SCRC competition manual [2].

Name	**Range (unit)**	**Description**
angle	$[-\pi, +\pi]$ (rad)	Angle between the car direction and the direction the track axis.
curLapTime	$[0, +\infty)$ (s)	Time elapsed during current lap.
damage	$(0, +\infty)$ (point)	Current damage of the car.
distFromStart	$[0, +\infty)$ (m)	Distance of the car from the start line along the track line.
distRaced	$[0, +\infty)$ (m)	Distance covered by the car from the beginning of the race
fuel	$[0, +\infty)$ (l)	Current fuel level.
gear	$\{-1, 0, 1, \ldots, 6\}$	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.
lastLapTime	$[0, +\infty)$ (s)	Time to complete the last lap
opponents	$[0, 200]$ (m)	Vector of 36 opponent sensors: each sensor covers a span of 10 degrees within a range of 200 meters and returns the distance of the closest opponent in the covered area.
racePos	$\{1, 2, \ldots, N\}$	Position in the race with respect to other cars.
rpm	$[0, +\infty)$ (rpm)	Number of rotation per minute of the car engine
speedX	$(-\infty, +\infty)$ (km/h)	Speed of the car along the longitudinal axis of the car.
speedY	$(-\infty, +\infty)$ (km/h)	Speed of the car along the transverse axis of the car.
speedZ	$(-\infty, +\infty)$ (km/h)	Speed of the car along the Z axis of the car

track	$[0, 200]$ (m)	Vector of 19 range finder sensors: each sensors returns the distance between the track edge and the car within a range of 200 meters. When noisy option is enabled, sensors are affected by i.i.d. normal noises with a standard deviation equal to the 10% of sensors range. By default, 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. However, the configuration of the range finder sensors (i.e., the angle w.r.t. to the car axis) can be set by the client once during initialization, i.e., before the beginning of each race. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the returned values are not reliable (typically -1 is returned).
trackPos	$(-\infty, +\infty)$	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when car is on the 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. Values greater than 1 or smaller than -1 mean that the car is outside of the track.
wheelSpinVel	$[0, +\infty)$ (rad/s)	Vector of 4 sensors representing the rotation speed of wheels.
z	$(-\infty, +\infty)$ (km/h)	Distance of the car mass center from the surface of the track along the Z axis

Table 1.1: : Description of the available sensors.

Name	**Range (unit)**	**Description**
accel	$[0, 1]$	Virtual gas pedal (0 means no gas, 1 full gas).
brake	$[0, 1]$	Virtual brake pedal (0 means no brake, 1 full brake).
clutch	$[0, 1]$	Virtual clutch pedal (0 means no clutch, 1 full clutch).

gear	$\{-1, 0, 1, \ldots, 6\}$	Gear value.
steer	$[-1, 1]$	Steering value: -1 and $+1$ means respectively full right and left, that corresponds to an angle of 0.366519 rad.
focus	$[-90, 90]$	Focus direction (see the focus sensors) in degrees.
endRace	$\{0, 1\}$	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

Table 1.2: Description of the available action parameters.

Chapter 2

Standard programming approach

2.1 Line follower

The first approach to the problem was developing simple bot which follows the track axis. Every time, absolute value of trackPos sensor (normalized position of the car on the track between $[-1, +1]$) is bigger than given small threshold, agent turns towards axis of the track with turning force set to 35%. To avoid zigzag driving, very small course corrections are applied, whenever $abs(trackPos) < 0.2$. Speed is limited to 80 km/h. These configuration allows bot to safely drive on every track.

I am aware that limiting turing force, has huge negative impact on bot performance. However increasing that value leads to dangerous behaviors. I was surprised how easy it is to slip. During development line follower I haven't found any solution for getting out of the slip. For now limiting that value was necessary.

2.2 Speed limit signs

In real world, government for safety reasons, places speed limit sign on dangerous sectors of roads. Signs are placed before sharp turns, steep slopes, bridges, viaducts, and more. Drivers adjust the speed to specific environment conditions. Inspired by this observation, I tried to improve previous agent iteration.

I've split the track on 8 sections, using distFromStart sensor (distance of the car from the start line along the track line). Every part would have different speed limit set for specific track. The goal is to finish the lap as fast as we can while not falling out of the track. So we need to minimize function f, where l_i - speed limit for section i in km/h.

$$f(l_1, l_2, \ldots, l_n) = \begin{cases} \infty, & \text{when the car fell off the track} \\ \text{lap\_time}, & \text{otherwise} \end{cases}$$

I was trying to implement genetic algorithm for finding optimal limits for given track, but it appears that game engine, even with turned graphics off, isn't as fast as I anticipated. It takes almost 20 seconds, to complete one lap. Simple genetic algorithm would need long hours to find approximately good solution. I've used two following observations while developing my algorithm.

- $l_i \in [40, 300]$: Smaller values can lead to disqualification from race.
- true speed of car on sction i, depends on l_i and true speed at the end of section $i - 1$
- l_i is correct, when $l_{i+1} = 40$ and car didn't fell out from track
- $g_j(x) = f(l_1, l_2, \ldots, l_{j-1}, x, l_{j+1}, \ldots, l_n)$ is decreasing while car stays on the track

I've implemented algorithm based on divide and conquer idea. It finds maximal x, where $g(x) \neq \infty$. We run that procedure on every argument:

$l_i \leftarrow 40$
for $i \leftarrow 1, \ldots, n$ **do**
  $l_i = devide\_and\_conquer(func = g_i, min = 40, max = 300)$
end for

For every section it uses around 12 game engine runs. So whole algorithm ends execution after around half an hour.

2.3 Results and observations

In following table I've presented line-follower performance on specific tracks from TORCS environment. While speed limits are significant improvement,

they still do not deal with major flaw of the concept. Our bot is extremely reactive. Human driver see a turn from distance and prepares for it (reduces speed, drifts towards opposite side of the road). On the other hand my bot turns the steering wheel only when it drifts away from the middle of the track (this happens after entering the turn). It is basically to late for a perfect turning maneuver. We need a model, which predicts and reacts to the future events. That's why I resign from extending current concept.

track name	unexperienced driver[s]	line-follower[s]	speed limits[s]
forza	122.65	265.51	146.31
cg_track_2	79.64	148.84	84.68
cg_track_3	102.09	133.74	91.86

Table 2.1: Line follower performance for specific TORCS tracks

Chapter 3

Machine Learning Approach

3.1 Introduction

For very complex tasks creating an algorithm can be very challenging. Most of the time we have an input, we define requirements and develop an algorithm which solves our problem. But sometimes the task is to complex to be solved by an algorithm. In that case programmes can use machine learning. For example let's take email spam detection. Users get a lot off unwanted messages on their emails, most of the time they delete just them and that's a great thing. They unconsciously provide a labeled data sets, on which we can test our solution. Creating algorithm which predicts if a message is spam is extremely difficult. We can check for existence of specific keywords, measure length, check upper case letters appearance, but combining all this variables needs a lot of testing.

Instead of that, we can try to extract statistical relations in collected data. Machine learning algorithms can be applied when we have algorithm input and expected output for every example in the data, but we have no idea how it should be achived. We receive not optimal, but approximately good solution for our problem and that's exactly what we want from spam detector. It should help users in omitting unwanted messages, but it's not a problem if he is wrong.

Primary machine learning problem is classification. We want to assign every input to one of given classes. In our email example we've got two classes (SPAM, NOT-SPAM), but we can have much more of it. Model designed to solve that type of problems is called classifier. Quality of classifier can be measured by percentage of correctly predicted classes on the test data sets.

Problems where output is a single floating point number are called regres-

sion. Model which predicts price of car is much more useful thant one which assigns car to one of following classes (cheap, medium, expensive). Such models are called regressors and we can measure their quality by counting mean squared error between real and predicted values on the test data set.

3.2 Applying machine learning for car driving agent

Accuracy of machine learning models depends heavily on amount of collected data. Trains data set should describe whole space of problem variables. With plenty of data samples we can extract most meaningful dependencies and predict outputs on real life data samples.

Although !!!!!INSERT REFERANCE!!!! we can see some success with learning car driving model on quite small data set. I don't have access to huge database of car driving logs, but during agents development I became well TORCS driver. So I decided to record my performance on race tracks.

Unfortunately TORCS environment does not have an option to log sensors data and driver actions. It is planned to add this functionality in the future development. That's why I developed software which mimics TORCS car controls and saves data from all sensors and driver actions to json file. From game perspective it is normal agent, but it reacts only on user keyboard inputs (accelerate, break, left, right). With that infrastructure During 25 recording laps on *cg_track_2*, I've collected around 55000 data frames (description of car state and driver action, saved every 10ms).

3.3 Single model agent

First machine learning approach was to develop simple model which predicts action for every data frame during the race. Input consist of all state sensors described in 1.1 and normalized to fit in range $[-1, 1]$. To determine classes of actions, I've extracted all unique actions taken by the driver and then labeled them, to generate simple classification problem.

Action parameters

- **accel** $\in \{0, 1\}$

- **brake** $\in \{0, 1\}$
- **steer** $\in [-1, 1]$

After splitting the data on train and test data set, I've trained classifiers to predict the best action for current car situation. I've used algorithms implemented scikit-learn [4] library. Results of experiments are shown in table 3.1

I've started with simple Decision Tree. CART (classification and regression tree) algorithm produces binary tree which consist of decision nodes and terminal leafs. Every node holds a binary condition (for example $angle > 0.37$) which passes data frame to one of it's children. Every leaf is labeled by class which it represents. To classify specific input we traverse a path directed by decision nodes. To choose variable and value used in condition creation we are minimizing impurity function. More detailed description can be found in [3].

Second model used was Random Forest. It is based on multiple randomized decision trees. Final result is determined by averaging predictions of every tree included in the model.

agent	cg_track_2 result [s]	train score [%]	test score [%]
speed_limits	84.68	-	-
CART	61.18	0.81	0.65
Random Forest	61.18		

Table 3.1: Single Model:agents performance

!!!MORE CONCLUSIONS!!!

Performance was improved by 20% even with poor model accuracy. Classifier accuracy on test data set is equal 61%. Other models such as (Neural Network, Support Vector Machine, Random Forest) resulted in similar bad accuracy for test data set and all of them were getting out from the race track.

It is worth noticing, that using **distFromStart** (distance from start line according to track middle line) parameter isn't very practical for generalized driving agent. My models weren't learning how to drive a vehicle. The were rather memorizing right actions for specific sector of the track. We need model which can be applied also for unseen tracks. However without that parameter, all my models were crashing on the track. They were able only to apply similar actions in similar places.

3.4 Two models agent

Major flaw for previous one model agent was tha amount of classes used in classification. Data set is to small for such variety of decisions. To simplify classification task I've decided to use two separate models:

- **Steer Regressor** - responsible for determining the turn rate (floating number between [-1,1]).
- **Acceleration Classifier** - responsibrle for choosing one of three speed actions {accelerate, do_nothing, brake}

This time I've used MLP (Multi Layer Perceptron) implemented in scikit-lirary [4]. It is simple neural network model which need more time to train, but is much more powerful than simple decision trees. !!! DWSCRIBE NEURAL NETS!!!

Because of hardware limits I was able to train very simple networks regressor hidden layers (300,30); classifier hidden layers (200,20) both with hyperbolic tangent activation function). Input data have all parameters from single model agent, instead of **distFromStart** parameter. All of them are standardized in range [-1,1].

3.4.1 Architecture motivations

There were two observations in advantage of this architecture.

First of all, it significantly reduces amount of classes used in classifier. Three classes should be covered effectively by amount of collected data.

Secondly with steer parameter determined by classifier we encounter following problem with penalty for learning algorithm. Let's say we've got data sample with, action taken by human driver, to $steer = 0.5$. Consider two wrongly predicted answers by models:

a) $steer = -0.4$

b) $steer = 0.35$

For classification problem both predictions (a and b) are equally wrong. The class prediction is just missed. However for reconstructing steering actions case b) is much worse and more dangerous than a). When we are near edge

of the track, that kind of wrong prediction can lead as away from the track resulting in a crash. When we approach setting *steer* as regression problem we tends to choose values which are close to real ones. Regressor more accurately reflects steering actions which appears in the data set.

Chapter 4

Adidtonal Problems

- **Transmission** Choosen architecture didn't allowed me to use automatic transmission included with the torcs game. I had to build my own basic automatic transmission system, which is far from perfect. I haven't change it during development of different agents, so all results are comparable.

Bibliography

[1] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, A. Sumner. TORCS: The Open Racing Car Simulator, v1.3.6, 2014.

[2] Daniele Loiacono, Luigi Cardamone, Pier Luca Lanzi, "Simulated Car Racing Championship: Competition Software Manual", Technical Report 2011.06, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 2011.

[3] Introduction to Machine Learning

[4] Scikit learn