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

19 stycznia 2019

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

Autonomous cars are very trendy subject nowadays. 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 very accurate car simulator with 3D graphic engine. It allows to run races with various cars and tracks. The simulation features a damage model, fuel consumption, aerodynamics, wheel properties, aerodynamics, advanced slipping model and much more [1]. It is designed to enable programed agents compete against each others. There is very detailed 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 10ms 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

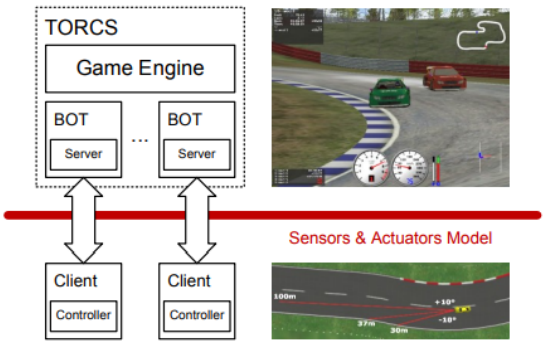

Figure 1.2: Simulated Car Racing Championship - architecture overview [2]

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 sensors used by ma agents are described in tables 1.1 1.2. Full list of parameters provided by game engine can be found in SCRC competition manual [2]. Sensors allow to determined exact car position on the track, and also predict close future shape of the track.

Name	**Range (unit)**	**Description**
angle	$[-\pi, +\pi]$ (rad)	Angle between the car direction and the direction the track axis.
distFromStart	$[0, +\infty)$ (m)	Distance of the car from the start line along the track line.
gear	$\{-1, 0, 1, \ldots, 6\}$	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.
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.
trackPos	$(-\infty, +\infty)$	Normalized distance between the car and the track axis. Values beetwen $[-1, 1]$ means that the car is on the track.
wheelSpinVel	$[0, +\infty)$ (rad/s)	Vector of 4 sensors representing the rotation speed of wheels.

Table 1.1: : Description of the available sensors. Cited from [2]

Name	**Range (unit)**	**Description**
accel	$\{0, 1\}$	Gas pedal
brake	$\{0, 1\}$	Brake pedal
gear	$\{-1, 0, 1, \ldots, 6\}$	Gear value
steer	$[-1, 1]$	Steering value: -1 and $+1$ means respectively full right and left

Table 1.2: Description of the available action parameters.
Cited from [2]

Chapter 2

Heuristic 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. Several configurations of constant parameters were tested and current 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 section 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 human[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 designed by human. 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 achieved. 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[3]. 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 regression [3]. 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.

In some problems, single action don't have huge impact on result. An action is good only as a part of good policy. Methods which focus on that approach are called reinforcement learning [3]. Car driving can be classified as such problem. Sometimes there isn't optimal action for current situation, it is rather optimal sequence of action which gives positive result.

3.2 Applying machine learning for car driving problem

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. However recent work by P. Dybiec [5, 2018], which focuses on autonomous driving developed for martian rover shows that, it is possible to successfully apply machine learning technics with relatively 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 classifier. 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 contains multiple decisions tree, constructed with randomized algorithm (in my experiment, 40 trees). To predict class of data point, we average predictions from all trees in the model [3].

agent	cg_track_2 result [s]	train score	test score
speed_limits	84.68	-	-
CART	63.25	0.81	0.65
Random Forest	65.39	0.99	0.66

Table 3.1: Single Model:agents performance

Despite poor accuracy on test models, performance was improved by almost 25%, compared to heuristic agent with speed limits. Difference between train and test score is large, so models are overfitted. Anyway, agent can recreate enough human actions, to stay on well known part of 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 too 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** - responsible for choosing one of three speed actions {accelerate, do_nothing, brake}

State vector for regressor is the same as on used in single model approach. Classifier input is expanded by **steer** value predicted by the regressor. Intuition behind passing value between two models is that, breaking and accelerating during turning the manoeuver, leads to dangerous behaviors. Humans tends to reduce the car speed before beginning of the turn. Acceleration classifier should depend on expected steering decision.

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 with 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.

3.4.2 Results

agent	cg_track_2 result [s]	train set	test set
double agent	DNF		
Steer regressor error [MSE]		0.0041	0.0043
Acceleration classifier score [%]		0.99	0.98
double agent - joined state	DNF		
Steer regressor error [MSE]		0.0026	0.0028
Acceleration classifier score [%]		0.99	0.98

Table 3.2: Double models agents performance

Models were trained on from single agent approach (without **distFromStart** sensor). Accuracy of models on testing set is quite reliable 3.2. That's the effect of reducing number of actions classes. However agent is unable to finish the lap on training track. It appears that steering regressor isn't effective enough.

!!!! Paagraph about prev state !!!!

One of ideas to improve these agent, was assumption that, maybe planning driving policy is impossible with only one data frame processed at once. To take into account longer period of time, agent remembers state and action from previous time frame and attaches it to both models inputs. Results of the experiment are shown in second part of table 3.2. New approach allows to reduce mean squared error twice of steering regressor on data sets. However, agent driving is very aggressive. It includes a lot of sharp steering wheel turns even on straight sections of track. It is still unable to reach the finish line without a crash.

Even with significant improvement in accuracy, models are still not capable to drive safely on the track. Racing cars, drives extremely fast and one sharp turn in opposite direction, can causes inevitable crash situation. There is no place for such mistakes.

3.5 Three model agent

I've been observing previous models performance and I've came to conclusion that biggest problem occurs when agent unexpectedly steer sharply in opposite directions. Steering regressor accuracy is measured on absolute distance between magnitude of output and real steering value. It does not take into account steering direction, so we need to put more effort in correctness of that decision.

3.5.1 Steering supervision

Steering supervisor classifier is supposed to reduce amount of missed steering directions decisions. Input for this one is the same as other models. Training output is extracted from data set by applying function f on **steer** value from driver actions.

$$f(steer) = \begin{cases} sign(steer), & steer \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

To predict **steer** value we compute m which is output of steering regressor and d which is output of steering supervisor. Then we apply function g for those values, and that's our final **steer** value. Function g passes m only when

both models agreed on turning direction. Also when supervisor predicts no turning action, g decreases m significantly by a constant.

$$g(m,d) = \begin{cases} \frac{m}{c}, & d = 0 \\ 0, & sign(m) \neq sign(d) \\ m, & \text{otherwise} \end{cases}$$

3.5.2 Minor improvements

One of the most important sensors is **angle**. It allows agent to determine car orientation in reference to the track. To extract more informations from sensor and take into account nonlinearity, the value in all inputs is replaced with sine and cosine of **angle** parameter. It should give more meaningful knowledge for the agent.

Training neural network is randomized process, which takes significant amount of time. Agents trained with same architecture can provides completely different performance. That's why for current architecture I've used simple reinforcement learning method. I've trained agents in a loop, ending an algorithm with success only when the current one was capable of safely finishing the race.

3.5.3 Results

Every single model was a neural network from scikit-learn library [4], with following parameters given in table 3.3.

Model	hidden layers	activation function
Steering regressor	(500,30)	tanh
Steering supervisor	(350, 30)	tanh
Acceleration classifier	(200,20)	tanh

Table 3.3: Triple model, neural networks parameters

Agent was tested on cg_track_2 and it's performance is shown in table 3.4. It is first successful agent, which doesn't use **distFromStart** parameter during models training.

Data used to train models was collected on cg_track_2. To check driving abilities of three model agent we need to drive it in unseen previously environment. I've tried to test it on tracks with similar road width. Agent was stable

agent	cg_track_2 result [s]	train set	test set
triple agent	63.44		
Steer regressor error [MSE]		0.0043	0.0046
Steer supervisor score		0.94	0.90
Acceleration classifier score		0.99	0.98

Table 3.4: Triple models agents performance

and was taking reasonable decisions, but weren't able to finish the race. Problem appears on long sharp turns. Agent see only information about the track exactly in front of him, there is no way to predict how long the turn actually is. Very often the speed at the beginning of the turn is to high. We would need a mechanism which provides desired speed on current section, because we are unable to extract that information from single state frame.

To improve agent performance on unseen tracks I've tried to use speed limits approach from chapter 2. It would help with previously mentioned high speed problems. However to use learning algorithm, our agent must be capable of finishing lap with minimal available speed (40 km/h) and it isn't. The problem is that our models have never drive with such small speed. It's enlivenment not covered by training set. Decisions taken by driver are completely random, and the race ends on the first turn. Amount of collected that covers only a small fraction of possible states, so agent has to mirror human driver actions and cannot make his own decisions.

Chapter 4

Conclusions

Both approaches to problem of car driving in virtual TORCS environment leads to success. Heuristic agent, after adjusting speed limits, in reasonable time can finish every race. Machine Learning agent is highly effective on the track used to collect data. However without huge data set it can't be generalized for driving on new tracks.

4.1 Future Work

While we developed agent which solves basic problem there still room for improvements. There was not enough time to try out ideas mentioned bellow.

- Implementation of neural networks in scikit-library is very basic. It can not use a graphic card for computation. With usage of libraries designated to neural networks (PyTorch, TensorFlow)

4.2

- **Transmission** Chosen 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.
- Python real-time system limitations

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] Ethem Alpaydin, Introduction to Machine Learning, The MIT Press, 2010

[4] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[5] P. Dybiec, "Veryfing the remote control capabilities of rover using neural networks", https://github.com/dyniec/thesis/blob/master/iithesis.pdf Wrocław 2018