Training a self-driving car through supervised learning

Francisco Javier Jurado, Xavier Timoneda

June 23, 2019

1 Introduction

1.1 Work description

This project covers the process of developing and training a neural network-controlled simulated car through supervised learning techniques, with the aim of obtaining a car model able to traverse simulated tracks while controlled by the neural network.
In order to achieve this we have implemented a system combining a simulation environment made in *Unity*, a cross-platform game engine which greatly simplifies the graphics and user interface setup, and a neural network in *R* which eventually drives the car through the simulated track. The project can be divided in three main blocks or phases:

1.1.1 Data Gathering

As its name suggests, this phase focuses in the collection of the data that will be used to train and test the neural network. The reason of being of this phase comes from the realization that, while we needed "*good*" driving data to train the neural network and generating it procedurally was not a trivial task, thanks to Unity's toolset it was surprisingly easy to simply modify the simulator required for the eventual car visualization to allow us to control the car and export our driving data into a *.csv* file (A simple diagram of the system can be seen in figure 1). By doing this we managed to collect the necessary data for the supervised learning while making sure that it represented a proper way of driving through the track, as it corresponded to ourselves driving (after some practice, we managed to get a reasonable amount of samples without crashing).

1.1.2 Training

The second phase uses the gathered driving data build a model by training a neural network (as seen in the second half of figure 1). All coding and training in this section is done in *R*, as required by the project statement.

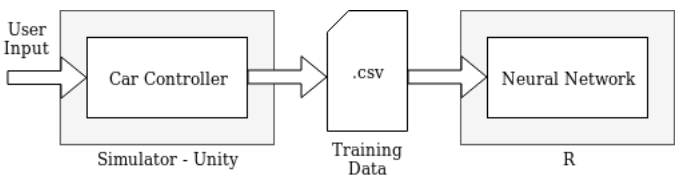

Figure 1: Diagram of the system in the data gathering & training phases

1.1.3 Autonomous Driving

The last phase corresponds to the testing of the network in the simulated environment as a way to validate that it actually learned how to drive the car. For this we use the same simulator that was used to record the data in phase I with the difference that this time the car is controlled by the trained neural network rather than by us. In order to do this we created an interface between the simulator and *R* which allowed the car controller to send data to the network and retrieve the driving "instructions" (the description of the data used to train the network is provided in section 1.3, *Available data*). A diagram of this configuration can be seen in figure 2.

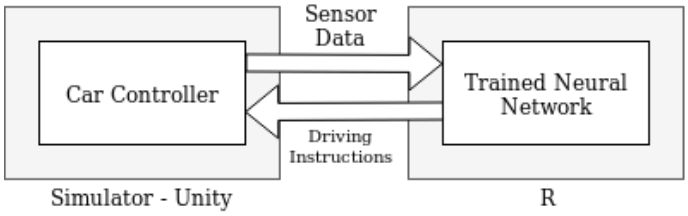

Figure 2: Diagram of the system in the autonomous driving phase

1.2 Goals

The main objective of this project is to train a neural network using supervised learning methods in order to make it able to successfully drive a car through a track in a simulated environment as long as possible. For this purpose we use data collected from (human) users driving the car in the same simulated environment (but not necessarily the same track). The definition of successful is the same as usual, eg. while avoiding crashing.
As enablers for the main objective we implement also the simulation framework as well as the interface between the simulator and *R* to allow the network to control the car once trained.

1.3 Available data

Compared to more analysis-oriented machine learning projects, our work has the peculiarity of not starting from a given data set but rather gathering the data it needs as part of the process. As described in the *Work Description* section, in the *Data Gathering* phase the data used for training and testing was collected through a driving simulator which registered a set of metrics of a (human) user driving the car through a track.
We now provide a description of the block in charge of generating the data (the simulator), together with some design considerations. A more extensive explanation of the decision process behind the data that was finally used is given in the *Pre-processing* section.

1.3.1 Description of the graphical interface

The graphical interface for the simulator is built over the platform of *Unity v.5.5.2*, which compiles an eases the production of executable games composed by a combination of scripts programmed in *C#* language. When executing the data collection simulator, we can observe a top view of a car represented by a rectangle, going through a path previously defined by the programmer. The path has growing difficulty as the car advances along the path. In the simulation it can also be seen the updated real-time parameters, and five crosses representing what each of the sensors is seen.

These sensors are situated in the front of the car, and provide an insight of what the car is seeing in front of itself, as well as in its lateral directions. They serve to collect the required input data which will be then introduced to the Neural Network when the data collection simulation has ended.

1.3.2 Simplifications of the simulation model

In order to simplify our model to the scope of our project, we decided to perform several assumptions which make the model deviate from a very realistic stage. This assumptions allow to simplify our model and let it be available for future enhancements.

The first assumption is that the car does not have speed control and therefore, we assume a constant velocity. This simplification was done in order to do not force the model to predict so many target variables, therefore allowing us to have a model with lower complexity. Another assumption is that the car starts the simulation always in the correct position, aligned in parallel with the walls of the first section of the path. This simplification was seeking to increase the chances of our model to succeed, at least, for the first sections of the path.

1.3.3 Controls

In the simulation for the data collection, the car is driven by the user through the left an right keys in the keyboard. At each frame, in a given specific time-stamp, the input of the user is at the same time applied to the simulation, as well as stored in a buffer to later be exported.

At the end of the simulation, when the user thinks that has recorded enough data, the output of the sensors is saved in a log file, as well as the input provided from the user.

2 Previous work

Although the idea of car-controlled neural networks is not novel (any search for "*Neural Network car*" on your favourite search engine will return hundreds of results), using supervised learning to train the network is somewhat rare. The vast majority of solutions proposed ([1]) are based in some flavour of reinforcement learning, this project itself being inspired by the video "*Deep Learning Cars*" ([4]), which manages to train a car to traverse a track using evolutionary algorithms. It is worth noting that we reused a big chunk of the simulator logic from the *Deep Learning Cars* project's GitHub to avoid spending excessive time in non-ML related topics.

Out of those who do use supervised learning to train the network, most of the effort is placed on computer-vision related methods ([3]), where the path followed by a human driver is recorded by a set of cameras in the car and the resulting images are then used to train the network which will then drive the car. In some sources ([2]) this process is referred to as "*Expert Guidance*" or "*Imitation Learning*". Although initially they might seem quite different, the basic idea between the image-based and our little toy problem is the same: to use human knowledge on how to successfully drive a car through the track to train the network to do it by itself.

3 Data extraction

3.1 Pre-processing

Due to the nature of the project there was little to no pre-processing to be performed. Because we were generating the data ourselves by recording keystrokes and the sensor values as we drove the car through the track, it did not have any missing values and was always in the right format. Nevertheless, during our exploration, we found out that the way of controlling the car in the first block and therefore, the way how the model was getting the training data had a huge influence in the performance of such model. For this, we considered two different approaches.

The first and most intuitive one, consisted on a discrete value that controls the turn with levels -1, 1 and 0, which resulted into applying a force to the left of the car, to the right and no force, respectively. The force was applied during the interval between that frame and the next. In this approach, the modelling problem turns into a multiple classification one, as we just have one target variable but it has three levels. First of all, the target variable *turn*, had to be declared in R as a factor. Then, it hat to be split into two binary indicator variables, *turn0* and *turn1*, which indicate whether the variable turn is 0 or 1, respectively, with the implicit constraint that they cannot be both 1 at the same row. This conversion was done making use of the function *model.matrix* in R. With such conversions performed, the modelling problem could be treated as a binary classification one, with two target variables.

A second approach was purposed by the authors of this work after the observed difficulties for the model to learn from the factorial target data. In this new model, the turning force was not a three-level discrete variable but it was a continuous value between the range of -1 and 1. The turning force was calculated at each frame by the simulator program in unity, and its magnitude depended on the number of frames since a key had been pressed. Therefore, for small turns, the user would only click a few times to do a small turn, which would result into a small turning value. On the other hand, for sharp turns, the user would click the same key for a long time, and the turning force would increase at each frame until reaching the maximum value. In this scenario, the modelling problem turns into a single target regression one, with a different model from the previous approach. Note that the single target variable is ranged symmetrically and zero-centered, so therefore, normalization was not required. The results in the third block showed that without not normalizing the target variable we obtained better results.

The former reader of this document can try himself the built data recorder programs, for each of the two approaches, called *DataCollection_discrete.exe*, and *DataCollection_continuous.exe*.

The next step before applying the model was to discern the data which was providing useful information to the model from the data which was only adding noise and ambiguity. Through experience, we deduced that it was very important that the data is captured only when the driver is doing it right. This implies stopping the recording of the data before the user hits a wall. Thus, we had to implement a new keyboard shortcut to save the data recorded until the moment. Therefore, this shortcut would be pressed when the user decides it has performed well until that point. In the same way, another shortcut had to be implemented to stop recording and discard such data.

Regarding the time-series nature of our data, it was purposed the possibility of turning the

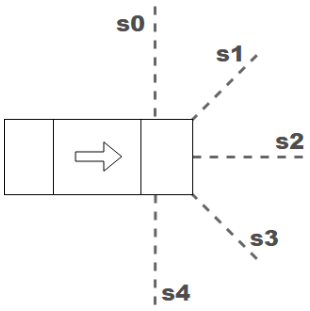

Figure 3: Schema of the sensors of the car, with the arrow indicating the forward direction of movement.

input data into a format in which time-series problems are typically converted. This format would imply creating new variables of the past samples for each of the original input variables. This possibility was discarded as it was out of the scope of this project, and it would hugely increase the complexity of the model as well as the training times.

3.2 Features rationale

The values for the sensors are float values ranged from 0 (which represents being very close to some wall) and 10, which means we are enough far away from any wall to assume we have free line of sight. The input variables of the collected data, which are called sensor0, sensor1, sensor2, sensor3 and sensor4, correspond to the values provided by the sensors from left to right, respectively, as it can be shown in Figure 3. Taking into account the values of all the sensors are ranged between 0 and 10, and their value saturate at the maximum, it does not make sense to apply any normalization process to the input data.

3.3 Visualization

For the visualization of the data recorded, it is very clear that typical multivariate analysis techniques for data visualization such as factorial analysis would not make any sense in our context. Therefore, the best visualization of our data is the playback of the recording of the simulation itself, where the user can see how the controlled car moved through the path, as well as the sensor position, and the turning force applied at each frame.

Nevertheless, we can also inspect the data in Rstudio in order to have an insight of the main structure. Figure 4, shows the summary of the data introduced used to train and test the model. We can observe that the turn variable is slightly biased towards the left, as its mean is negative. This makes sense as the track used for collecting the data has an approximately spiral shape. Furthermore, regarding the input data from the sensors, we can see that *sensor2*, which is situated in the front of the car, has the highest mean, whereas the sensors 0 and 4, situated on the perpendicular directions, have the lowest one. This can be easily interpreted as the sensor of the frond tends to have the maximum value the majority of the time, whereas the sensors on the sides are usually pointing closer to the walls.

```
> summary(logdata)
    sensor0          sensor1          sensor2
 Min.   :0.000   Min.   : 0.000   Min.   : 0.000
 1st Qu.:4.433   1st Qu.: 6.409   1st Qu.:10.000
 Median :4.952   Median : 8.764   Median :10.000
 Mean   :4.900   Mean   : 8.215   Mean   : 9.853
 3rd Qu.:5.187   3rd Qu.:10.000   3rd Qu.:10.000
 Max.   :9.406   Max.   :10.000   Max.   :10.000
    sensor3          sensor4            turn
 Min.   : 0.000   Min.   : 0.000   Min.   :-0.99400
 1st Qu.: 7.873   1st Qu.: 4.032   1st Qu.: 0.00000
 Median : 9.977   Median : 4.596   Median : 0.00000
 Mean   : 8.867   Mean   : 5.110   Mean   :-0.03896
 3rd Qu.:10.000   3rd Qu.: 5.875   3rd Qu.: 0.00000
 Max.   :10.000   Max.   :10.000   Max.   : 0.59833
```

Figure 4: Summary of the data gathered for the continuous turn approach

4 Modelling and validation protocol

4.1 Model

The model we have used in both approaches is a Feed-forward Multi-Layer Perceptron Neural Network. The R package used to build this model is neuralnet. Given the limitations of this package and in order to do not add an extra layer of complexity to the project making use of the tensorflow implementation for R, we had to set the *threshold* to 0.1 and the *stepmax* to 1e6 in order to allow the Neural Network to converge. An exploration over the possible number of hidden layers as well as number of neurons per layer showed that a single hidden layer provided better results than deeper multi-layer networks. Finally, for both approaches, the number of hidden layers was set to 1, and the number of neurons per layer was set to 5.

For the discrete turn approach, we needed to set the activation function of each layer to *logistic*, and set the *linear.output* parameter to false, as we want the neural network to apply the logistic activation function to the output neurons. This way, the target variables *turn0* and *turn1* will always provide a binary result.

In contrast, for the countinuous turn approach, the activation function was set to *Hyperbolic Tangent*, which was the kind of sigmoidal activation functions available that provided better results. Other types of activation functions such as *Rectified Linear Unit* were not considered, as they are most commonly used in deep learning models. The *linear.output* was set to true, as for this approach we do not want to apply any activation function to the output neurons.

The schema of the neural networks used for both approaches with their weights after training is shown in Figure 5.

4.2 Validation protocol

The validation protocol in our very special scenario was mainly based on performing an actual simulation that calls the trained neural network in every step, and evaluate its performance by a figure of merit such as the distance travelled. Visual inspection of this simulation is also very important in order to improve the network performance as well as to evaluate its generalization ability. The details of this simulation will be explained later in this section.

Despite the mentioned above, some validation techniques could be applied to the trained model in Rstudio in order to have a first insight about its performance, as well as they could be used to compare how these metrics agree or not with the performance found through simulation. First of

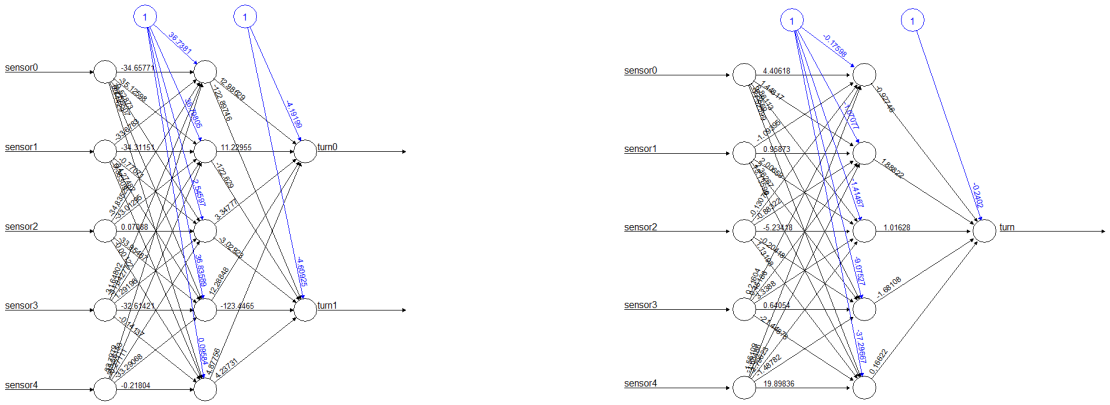

Figure 5: Schema of the neural networks for the discrete turn approach in the left, and for the continuous turn approach, in the right.

all, we need to mention that cross-validation is the most used method for selecting the parameters of models such as the ours. Nevertheless, we decided not to apply it because the configuration of the network in our case was mainly constrained by the limitations of the *neuralnet* package and its ability to converge. Furthermore, cross-validation makes sense in predictive models in which you cannot evaluate for sure with real data the generalization performance of the model.

For the discrete turn approach, we built confusion matrices for both *turn0* and *turn1* binary target variables. It has to be mentioned that before building such matrices, we needed to convert into 0 or 1 the predicted values which were very close but not exactly those ones.

For the continuous turn approach, we calculated both train and test Mean Square Errors as validation metrics.

The simulation needed to join the neural network with the graphical interface created by unity. Therefore, the unity project was split into two models, the data gathering one, which has been already explained in the pre-processing section, and the simulation one. This simulation, written in C# language, had to query the trained neural network at every at every step, process its output and apply it to the simulation in order to generate the next frame. As the neural network was created in an *Rstudio* environment, the unity code needed to execute a portable version of *Rstudio* called *Rscript*, by invoking the command line. In this invocation, the information from the sensors is passed as an argument through the command line, and then, a simple R script loads the saved neural network, evaluates its results, and generates and output which is retrieved by the unity script. Note that this process implies a high overhead mainly due to the execution of *Rscript*, and therefore, the simulation was quite slow. A video of the results is provided in order to ease the evaluation of the results to the former reader. This video has been speed up because of what we have just previously mentioned.

5 Results

In this section we present the best results obtained with each of the approaches already presented, providing also an insight of what was the tendency when modifying the parameters. For the discrete turn approach, the confusion matrices with the training data are shown in figure 6.

	tr.turn1_pred_1	tr.turn1_pred_0
true_1	40	371
true_0	26	12791

	tr.turn0_pred_1	tr.turn0_pred_0
true_1	11727	74
true_0	1260	167

Figure 6: Train confusion matrix of the binary indicator of turn1 (in which 1 means turn to the right), at the left, and confusion matrix of turn0 (in which 1 means staying straight), at the right.

	turn1_pred_1	turn1_pred_0
true_1	12	137
true_0	9	4251

	turn0_pred_1	turn0_pred_0
true_1	3878	26
true_0	445	60

Figure 7: Test confusion matrix of the binary indicator of turn1 (in which 1 means turn to the right), at the left, and confusion matrix of turn0 (in which 0 means staying straight), at the right.

The confusion matrices with the test data are shown in figure 7.

The confusion matrices show how unbalanced is the data. The rationale of this phenomena can be explained as while driving, due to the sharpness of our turning force, the user spends the most of the time not turning. A re-sampling of the non-zero values of the data was performed, but even it greatly increased the recall and precision of the model, it did not provide better results in the simulation.

According to the confusion matrices provided, the test accuracy for the turn1 variable was 96%, the precision was 57%, and the recall was 8%. Similar results but switching 1s to 0s were obtained for the turn0 variable.

For the continuous turn approach, the Mean Squared Error for training and test of the model configuration that provided better results was 0.025 and 0.026, respectively. The fact that the train and test results have similar values, makes us intuitively think that the model may not be suffering from over-fitting.

In both approaches, we could observe that gathering more data increased the training error and made it more difficult for the training algorithm to converge, but instead, at the same time, the test error slightly decreased. Regarding the results in the simulation, gathering more data generally produced longer lifetimes until the car hits the wall, even though in some particular parts of the path, the data gathered while traversing those regions was detrimental for the overall performance of the driving. In figure 8, we show a line with the path performed by the car before hitting any wall for both of the approaches. The simulation is also recorded in the videos provided *Train_track_car_discrete.mp4* and *Train_track_car_continuous.mp4*.

6 Final model

The approach that worked better was clearly the continuous turn one. This makes sense as when recording the data in the discrete mode, the user had to press the key intermittently with lower frequency if he wanted to do a small turn, and higher frequency or even continuously pressing it if he wanted to do a big turn. This, for the neural network it is very hard to learn as from the instant perspective, in some instants very close in time in which the sensors have similar values, the neural network would find out that in a small turn, some values are 1, and some values are 0,

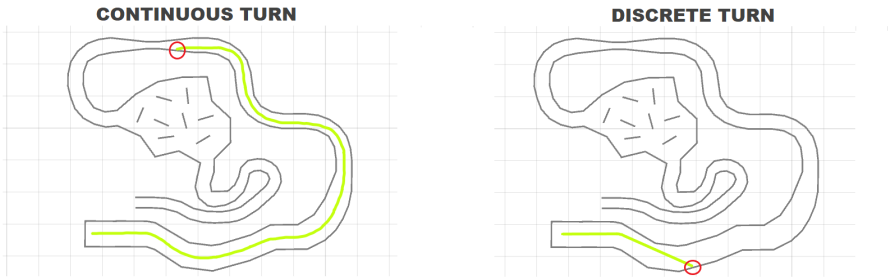

Figure 8: Picture of the path traveled by the autonomous car in the train track. The left plot corresponds to the continuous turn approach, and the right one corresponds to the discrete turn approach.

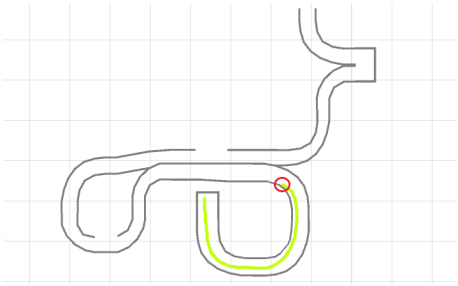

Figure 9: Picture of the path traveled by the autonomous car in the test track.

which is very uncertain and hard to learn.

Instead, in the continuous turn approach, we are implicitly pre-processing the data in such a way that we are implicitly including information about the previous pressed keys some frames before, as the value of the turn force provided to the neural network depends on the previous pressed keys. This could be seen as a particular method for dealing with a time-series problem approach, without having to add all the time-consuming complexity that implies replicating the past samples to create new variables.

In order to visualize completely the generalization ability of our model, we decided to perform an extra simulation in a totally different track, never seen during training process. In figure 9, we show a line with the path performed by the car in this new track. A recording of the simulation is also provided in the video *Test_track_car.mp4*.

7 Conclusions

In this project, we achieved the main goal of using data of (human) users driving a car through a track in a simulated environment to train a neural network capable of successfully driving through the same environment, not necessarily the same track, as long as possible. During the development process, we overcame all the issues and constrains due to mainly the inefficiency of the neuralnet package in Rstudio, and the difficulties risen when trying to communicate different environments that solve the problems of each phase.

The proper implementation of both data gathering and simulation environments, provided

a very useful graphical interface to first allow the user to control the car while generating the required data, and second, to visualize through simulation the performance of the neural network once trained.

Thanks to persistence, as well as all the knowledge obtained from the Machine Learning course, we could achieve that the neural network controlling the car could travel a distance equivalent to a 0.476 score of the evaluation function in the training track, as well as a score of 0.2098 in the testing track. These results were obtained with a very low complexity model, which consisted on a neural network with just 1 hidden layer of 5 neurons.

During the development process of the project, we observed that gathering more data leads to higher training error, but at the same time, we obtain lower test error.

Finally, we end the conclusions by stating that R studio is not the best environment for neural networks, and typical evaluation techniques are not enough to describe the performance of the neural network in our particular approach. For this reason, we purposed alternative methods to visualize the performance of the model. In the next section, we purpose possible extensions other than supervised learning, which could highly improve the performance of this self-driving car.

8 Possible extensions and known limitations

The number of possible extensions of this project is huge, as we could either improve the prediction with more complex models, improve the quality of the graphical interface, add new functionalities and parameters, or change the learning method.

The existing limitations of our project were clearly known by the authors from the beginning, since we have a limited computational power, limited time and fixed deadlines, as well as the constraint of using certain software platforms to embed the learning part of our project. In addition, after having gone deeply in analysis of this project and knowing with detail what is the information learnt for our model, we can state that supervised learning is not the best approach for this kind of problems.

Instead, other approaches such as semi-supervised learning, in which data can be either unlabeled or labeled, or reinforcement learning, in which feedback is only provided after several decisions have been taken by the model, seems more suitable. For example, if the car needs to go to the right, in order to have enough space to then do a sharp turn to the left, the action of going to the right will always be understood as a wrong decision in supervised learning, as the car is approaching to the wall. Semi-supervised learning techniques would allow to not define a label for some actions, and reinforcement learning would allow the model to explore new routes such as turning first to the right to do a sharp left turn between two checkpoints.

In addition, this project has made use of some knowledge and infrastructure of a previous work which was using unsupervised learning. That project was very interesting from the educational point of view, yet solving this problem through genetic algorithm exploration is very inefficient and therefore we would not consider unsupervised learning as a good implementation for this problem.

Finally, the authors of this work would purpose as a good next step, an implementation of a Deep Reinforcement learning model for this same approach, and making use not only of the actual data at each instant, but also the previous data from earlier time-stamps.

References

[1] Teigar, H. et al, *2D Racing game using reinforcement learning and supervised learning*, Institute of Computer Science & Neural Networs, University of Tartu, 2017.

[2] Azystev, A. & Gangwani, T., *Deep Learning for Manipulation and Navigation lecture slides*, University of Illinois at Urbana-Champaign, Illinois

[3] Markelic, et al, *Anticipatory Driving for a Robot-Car Based on Supervised Learning*, 4th Workshop on Anticipatory Behavior in Adaptive Learning Systems, Munich, 2008

[4] *Deep Learning Cars*,
$https://www.youtube.com/watch?v = Aut32pR5PQA$, Youtube
$https://github.com/ArztSamuel/Applying_EANNs$, Github
2008