Learning to drive based on multiple sensor cues in The Open Racing Car Simulator (TORCS)

Tim Bicker and Nimar Blume

***Abstract*—To implement an autonomous driver in The Open Racing Car Simulator (TORCS) we use a combination of a deep neural network (DNN) and a spiking neural network (SNN) based on multiple sensor cues. Specifically, the DNN predicts the current car displacement and angle to the road centre from a driver's view image. Based on the two values a SNN generates driving commands for the car. Subsequently, the car is put onto a new track and the driving performance is evaluated. The DNN is based on a Convolutional Neural Network and after training the mean absolute error for the displacement is XXXX and for the angle is XXX on an unseen test track.**

***Index Terms*—deep learning, TORCS, convolutional neural network, spiking neural network, autonomous driving**

I. Introduction

AUTONOMOUS driving is currently a controversial subject, especially regarding feasibility and performance. To explore the details of autonomous driving, an abstract prototype is build focussing on the implementation of deriving location data from images and using sensor inputs to drive a car. In this paper The Open Racing Car Simulator (TORCS) is used to simulate a car on a race track and collect relevant sensor data. The goal is to use multiple sensor cues to drive the car safely around the defined test tracks. Through the Robot Operating System (ROS) TODO: ADD CITATION the following sensors can be read from TORCS TODO: ADD ROS TORCS CITATION:

1) range finder
2) driver point of view (POV) image
3) displacement (distance from road centre to car centre)
4) angle (rotation angle between car and road)

Then, a deep neural network will be trained on the ground truth data to infer the angle and displacement from the provided POV image. Subsequently, a controller based on spiking neural networks (SNN) is trained to generate driving commands based on range finder, displacement and angle sensor inputs. Finally, the controller and the DNN are connected. So the overall model works only based on the POV image and the range finder data, which are directly taken from TORCS.

As a competing group is working on the same task, a race will be carried out at the end and the winner will be determined. Thus, the goal is to drive the car around the track as quickly as possible while maintaining a safe driving style.

Authors: Tim Bicker (12345678, tim.bicker@tum.de) and Nimar Blume (03638934, nimar.blume@tum.de) **Course**: Practical course: Computational Neuro Engineering Winter Semester 2017/2018 **Submitted**: January 7, 2018 **Supervisor**: Florian Mirus. Neuroscientific System Theory (Prof. Dr. Jörg Conradt), Technische Universität München, Arcisstraße 21, 80333 München, Germany.

II. A deep neural network for regression

The implementation of the deep neural network is done in `python` version 3.6.3. To facilitate development, the library `keras` [5] is used with the backend `tensorflow` [3]. To manipulate image data `openCV3` [4] is used and finally to load and manipulate data the library `numpy` [12] is utilised.

A. The testing deep neural network

To be able to chose hyperparametres for the convolutional neural network (CNN) at an early point, a basic CNN network architecture was chosen which has been proven before. The criteria for chosing the network architecture was, that it has to provide reasonable performance while being quick to train. The focus was rather on fast training performance, as the available resources are limited to us. Thus, after studying the architectures' training performances as seen here [6], AlexNet [8] was chosen as testing DNN.

AlexNet is a network designed for image classification tasks, such as the ImageNet challenge. Therefore, the last layer of AlexNet was altered to use it for multidimensional regression problems such as guessing the angle and displacement from an image. To achieve that, the number of output neurons of the last fully connected layer (FCL) was reduced from 1000 to 2, as there are two numbers to guess. Furthermore, the last FCL used a rectified linear unit (ReLU) as activation function.

$$f(x) = max(x, 0) \tag{1}$$

To prevent the activation function from cropping the output to values greater than zero, a linear activation function is used instead: $f(x) = x$.

B. Data set splitting

TORCS provides 19 tracks of which all are relevant. Therefore, the recorded data set is split into three parts:

1) Training set
2) Validataion set
3) Test set

First, the two tracks TODO:FIX TEST TRACKS were determined to be used for testing later on. As the goal is to train a general model, a prerequisite is that data recorded on either of the two test tracks is not used during training. In total XXX data points were recorded on the two test tracks.
The remaining tracks are used to train the DNN and to determine its hyperparametres. Therefore, the data set is randomly split into `train` and `val` (validataion) at a ratio of 90% to 10%. The validation set data itself was thus not used for traning, but it is recorded on tracks which are included in the training tracks.

C. Input images

The image data is acquired from TORCS [13] using ROS via the TORCS-ROS node [9].

1) Choosing a camera angle: TORCS provides several camera perspectives:

1) Driver's view with hood
2) Driver's view without hood
3) Third person perspective: far
4) Third person perspective: close

All perspectives are evaluated based on the DNN described in subsection II-A and as final perspective the driver's view without hood is chosen. The basis for that decision can be seen in Figure 1, which shows the mean absolute error for each view, with the driver's view without hood being the lowest.
TODO: INCLUDE EXAMPLE IMAGES

Figure 1. Mean absolute error of the same DNN trained with multiple camera angles

2) Choosing the image size: The images are provided by TORCS ROS [9] at a rate of 10 frames per second (fps) at a resoultion of $640\,\text{px} \times 480\,\text{px}$. Because that image size is too large to train a reasonably deep network in a reasonable time, the images are down-scaled prior to training as well as in the final application. To determine the image providing the best result, four different sizes were evaluated with the DNN described in subsection II-A:

1) $320\,\text{px} \times 240\,\text{px}$
2) $160\,\text{px} \times 120\,\text{px}$
3) $80\,\text{px} \times 60\,\text{px}$

First, the training images were collected from TORCS ROS at a resolution of $640\,\text{px} \times 480\,\text{px}$. Upon loading the images, `openCV` based downscaling was applied using the bilinear interpolation algorithm. Second, the testing DNN was trained with the images at the mentioned resolutions and the mean absolute error for the validation set was recorded, which is visualised in Figure 2.

Therefore, the image size $80\,\text{px} \times 60\,\text{px}$ is used for the DNN as it provides the smallest mean absolute error and additionally reduces the training time due to the small image size.

D. Choosing the optimiser

A plethora of different optimisers are available for use in the backpropagation step when training DNNs. The largest differences are in their abilities to exit large local maxima, training performance and accuracy. As the currently used DNN is managable in size, the training performance is not yet a critical property. Therefore, the optimiser was selected based on its ability to minimse the mean absolute error of the validation set. As illustrated in Figure 3, the `adamax` optimiser provides the best result and is thus chosen for future application. `adamax` is similiar to the `adam` optimiser while being more stable in certain conditions, see [7].

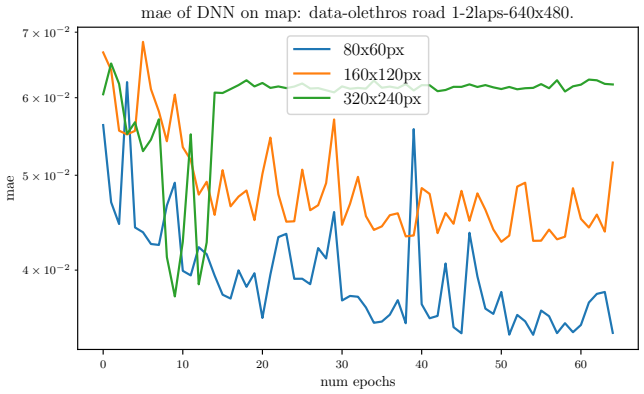

Figure 2. Mean absolute error of the same DNN trained with multiple image resolutions

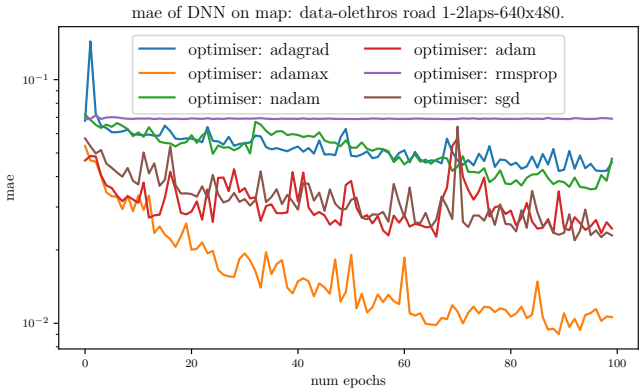

Figure 3. Optimiser comparison in the test DNN using with an image size of $80\,\text{px} \times 60\,\text{px}$

E. Building a deep neural network with keras

Keras is a python library hello keras [11]

III. Nengo Controller

The controller of the car is designed based on Spiking Neural Networks (SNN) which are implemented with Nengo [2], a python framework for building large-scale neural systems that is based on the Neural Engineering Framework (NEF) [10].
The controller is divided in several modular parts: acceleration, braking, steering, gear changing and clutching. We choose the modular approach, because it allows to test different modules independently from each other and also mix hard coded solutions with learned ones.
In order to discuss the architecture of the nengo model, we choose to first explain TODO: the setup?

a) Signals: The available input signals are as follows: The already mentioned lateral displacement, rotation angle relative to the street center line and the 19-dimensional range finder signals are used. Furthermore, we calculate the car speed

v as $v = \sqrt{v_x^2 + v_y^2}$, with v_x and v_y as the car speed in the front and side directions relative to the car.
As control signals an acceleration, brake and clutch signal, as well as a steering and a gear signal are given.

A. Controller Training

In this work a supervised learning approach for training the controller is chosen. This leads to several variables, that strongly influence controller performance. Those variables are discussed in the following.

B. Data Sources

a) TORCS-ROS default driver: The first and easiest to acquire is the default TORCS-ROS driver [9]. The advantage of this driver is that he drives a maximum speed of $149\,\mathrm{km}\cdot\mathrm{h}^{-1}$ and drives very carefully. Which means, that he always tries to stick to the middle lane of the road and brakes heavily when he comes close to a turn. We believe that this is an easy to learn and robust driving style. However, this driver is very slow, as can be seen in Table I.

b) TORCS drivers: Furthermore, there are several drivers that are implemented in TORCS. Those drivers have no speed limit and their driving style approximates that of a real race. The cars drive very close to the edge of the road and drive very fast through turns. We think that this is a hard to learn and error-prone driving style, because small errors lead to disastrous effects.

c) driving manually: Finally, there is the option to drive manually. As it turns out this is really difficult with a keyboard. Because fractions of seconds on the left and right arrows lead to very strong steering behavior and almost every time lead to a full turn-around or a far deviation of the track. This leads to a manual driving style that comes close to the default driver: very slow through turns and sticking close to the road center. However, because it is possible to drive as fast as one wants and in general one achieves a smoother driving of the car, we achieve a faster time than the default driver.

Because of the above mentioned characteristics of the different driver styles, it does make sense to first learn the TORCS-ROS default driver, because he seems to be the most robust one and data generation is simple.

TORCS-ROS default	85s
TORCS	43s
manual	69s

Table I
THIS TABLE COMPARES THE LAP TIMES OF DIFFERENT DRIVER TYPES.

C. learning methods

Besides the functionality of Nengo to directly encode and decode hardcoded functions according to the NEF [10], Nengo provides two different learning strategies: offline and online learning. With Nengo deep learning [1], an extension to implement classic deep learning methods exists as well.

a) offline learning: can be used for the classical supervised learning approach. Feature data and corresponding labels are given and Nengo uses a least squares approximation to solve for the neuron weights [10].

b) online learning: iteratively improves the initial neuron configuration in a supervised manner during runtime. However, this requires that for every timestep during runtime a goal value has to be known so Nengo can solve for it. In our problem this is not the case. Therefore online learning is not further considered in this work.

c) Nengo deep learning: allows to train classical deep learning methods e.g. feedforward neural networks with backpropagation and then implement the learned nets into Nengo. This work focuses on a mixture between hard coded functionality and offline learning.

D. Evaluation metric

To evaluate the controller's performance, we used the following two metrics: Firstly, we investigate if the controller is able to finish one lap in a reasonable time and how fast he is compared to the default driver. Secondly, we judge the overall robustness of the controller. For this we observed the actual controller during driving and evaluated how stable he drives, e.g. how close he actually drives to the edge of the street.

E. Nengo controller design

In the default driver's source code, the driving modules are based on different sensory inputs. The steering module is a function based on the car speed v_x, the lateral displacement d_{lat} and the rotation angle $/alpha$. Both, the acceleration and the braking module are based on three range sensor signals -5, 0 and 5 degrees as well as v_x. We chose to implement the gear changing as a hard coded solution without any learning involved, since it is a simple if-else condition. It is further possible to neglect the clutching module, because it did not show to make any difference for driver performance even though it is implemented in the default driver. The final model architecture can be seen in Figure 4
TODO: neuron type

IV. Experiments and Evaluation

A. Performance comparison between guessing angles vs. guessing displacement

For the final application, the same model is used to guess the car's displacement and its angle. During training, the loss function (MSE) and the metrics (MAE) are continueally evaluating the consolidated training performance. However, to determine the performance of predicting the individual values, two models were trained on each either predicting the angle or predicting the displacement of the car. Figure 5 shows the difference between angle and displacement prediction is shown. In training and validation set, the value of the displacement is in the range of -0.9 to 1.1 and the angle is in the range of -0.9 to 1.1. Thus, it can be concluded that the displacement prediction performance especially still needs to be worked on.

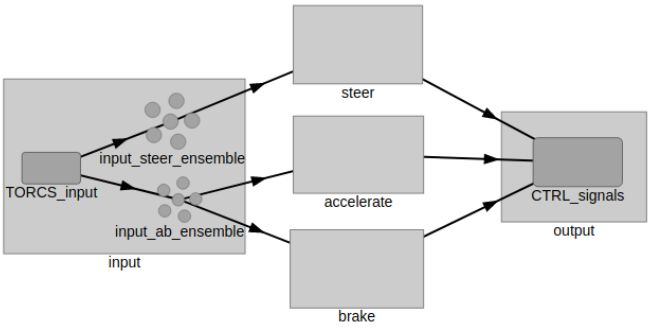

Figure 4. Nengo architecture with different modules. The input network contains an input node that has subscribed to the ROS topics and contains the CNN for inference of rotation angle and lateral displacement. The steer, accelerate and brake module contain one ensemble each. The output network contains an output node that publishes to the ROS topics. The gear module is not shown here.

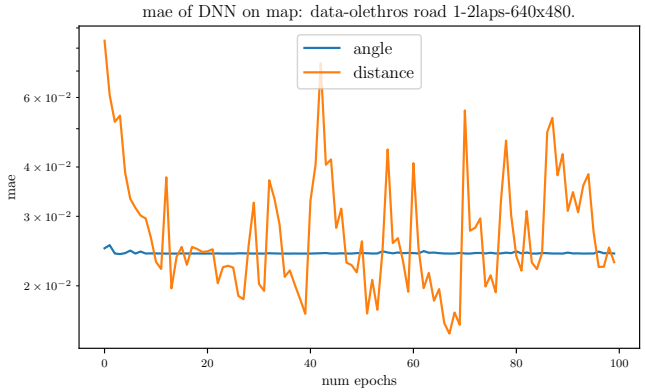

Figure 5. Mean absolute error of two test networks trained only to predict either the displacement or angle of the car

B. Controller performance

For finetuning the results we have basically three tuning possibilities: the neuron sizes, the amount of data and the composition of tracks the data was collected from.
We increased the neuron sizes and track data until we hardly discovered any improvements on the generalization task. As tracks for the training set we chose tracks with somewhat different shapes and turns in order to cover a vide variety of situations.
Further it played an important role to keep the forward propagation time of the controller relatively low. This is necessary because when the neuron size becomes too large, the controller signals are too far behind of the car and controlling becomes instable. Further we observed that braking needed much more neurons than accelerating and steering to work reliable. We assume the reason for this is that the braking function in the driver is highly non linear and consists also of a ABS implementation.
As the final neuron sizes for the different modules we chose $2,000$ neurons for each input signal. The steering and acceleration module consist of $2,400$ neurons and the braking module consists of $4,800$ neurons. The controller performance based on ground truth data is shown in Table II.

track	default driver	Nengo controller
cg track 2	1:58:15	1:59:76
wheel 2	4:21:45	4:26:80

Table II
THIS TABLE COMPARES THE LAP TIMES OF THE CONTROLLER AND THE DEFAULT DRIVER ON THE GENERALIZATION TASK.

C. Controller and DNN

V. Conclusion

A. Outlook

The network architecture can perhaps still be significantly improved upon, especially with an architecture specifically designed for regression tasks. The controller needs to be improved further and looked at more closely. Finally, the DNN and the controller will be connected in order to work together.

References

[1] Deep learning integratino for nengo. Accessed: 2017-12-13. 3
[2] The nengo neural simulator. Accessed: 2017-12-10. 2
[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 1
[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 1
[5] F. Chollet et al. Keras. https://github.com/fchollet/keras, 2015. 1
[6] J. Johnson. Cnn architecture benchmarks. https://github.com/jcjohnson/cnn-benchmarks, 2017. 1
[7] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 2
[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 1
[9] F. Mirus. torcs ros. https://github.com/fmirus/torcs_ros, 2017. 2, 3
[10] T. C. Stewart. A technical overview of the neural engineering framework. Technical report, Centre for Theoretical Neuroscience, 2012. 2, 3
[11] A. Toshev and C. Szegedy. Deeppose: Human pose estimation via deep neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014. 2
[12] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. 1
[13] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2016. 2