Self driving in DuckieTown environment (Önvezető autózás DuckieTown környezetben)

Ódor Dávid - **IFZYRQ**, Weyde Szabolcs - **DC6KRO**, Frey Dominik - **AXHBUS**

December 11, 2022

Abstract

The development and research of vehicles supporting full or partial self-driving now cover a very significant part of both industrial and academic projects. This task mainly involves the design, training and testing of algorithms and models based on machine learning. Training and testing such models is an extremely expensive process in a real-world environment. This is because huge annotated datasets are required for training, and testing is not safe in real traffic situations. As a result, the idea arises to train rudimentary models on generated data sets and test them in simulation environment. DuckieTown [8] is a platform for educational and research purposes, where we can test and compete our models in an environment developed for this purpose, on maps that can be configured as desired. The aim of this document is to present our work and experiences in the DuckeTown environment, designing, training and testing two models based on supervised learning. In the end of our work we will upload our best solution to the official DuckieTown Challenger site.

A teljes vagy részleges önvezetést támogató járművek fejlesztése és kutatása mostanra igen jelentős részét fedik le mind az ipari, mind az akadémiai projekteknek. Ez a feladat döntő részt gépi tanuláson alapuló algoritmusok, modellek tervezését, tanítását és tesztelését foglalják magukban. Az ilyen jellegű modellek tanítása és tesztelése rendkívül költséges folyamat valós környezetben. Ezt az okozza, hogy hatalmas méretű annotált adathalmazok szükségesek a tanításhoz, a tesztelés pedig nem biztonságos igazi forgalmi szituációkban. Ezekből kifolyólag felmerül az ötlet, hogy a kezdetleges modelleket generált adathalmazon tanítsuk és szimulált környezetben teszteljük. A DuckieTown [8] egy oktatási és kutatási célú platform, ahol egy erre a célra kifejlesztett környezetben tudjuk kipróbálni és versenyeztetni modelljeinket, tetszés szerint konfigurálható pályákon. A dokumentum célja, hogy bemutassuk saját munkánkat és tapasztalásainkat a DuckeTown környezetben, két felügyelt tanuláson alapuló modell tervezésével, tanításával és tesztelésével. A munkánk végén feltöltjük a legjobb megoldásunkat a DuckieTown hivatalos Challenger oldalára.

1 Introduction

The goal of this project is to train and test a self-driving AI vehicle in the Duckie Town World simulation. In this document, you will see how we are using two models, collecting training data, teaching the model, optimizing hyper-parameters and evaluating the results. The project is available at the teams git repository: https://github.com/fdominik98/DSD-DuckieTown

2 Related work

When we are speaking of end to end self driving models two approaches come into play RL (reinforcement learning) and IL (imitation learning). Imitation learning proved to be a powerful method as models can be trained on smaller datasets with relatively good results [3]. However it is very uncertain in case of errors [4]. Many researchers suggest reinforcement learning instead, [1][2] as it behaves better when the input is unusual or low probability.

3 Our approach

We decided to use an imitation learning and behavior cloning based approach, despite being aware of its limitations in self driving [4]. Our idea is to generate a reasonable amount of data while driving the duckiebot in the duckie simulator. The data consists of observations as snapshots of the scene, angular and linear speed values. Our models learn the angular and linear combinations for a given observation and that later in the process can be converted into PWN values. We were experimenting on two model architectures. One that is a pure volatile convolutional network and another with convolution combined with LSTM [10]. We will see that the LSTM network performs better as it relies also on previous observations.

4 Implementation

4.1 Dataset generation and preparation

To collect a proper amount of data for training our model we are using the Duckie Town Simulator available at Gym Duckie Town. We are generating data while driving the vehicle in the simulator, capturing images assigned with the appropriate action (linear and angular velocity).

4.1.1 Map generating

In order to run the simulator we needed a map to load into it. Although there are several preconstructed maps in the Duckie Town World repository we were advised to prepare our own. For this we used a map generator available at the map-utils repository. With this utility several configurations are supported e.g. map size, density of road object or side object. Our map is generated using the generator.py script with the following main configurations:

- Map width = 11
- Map height = 11
- Map density = dense
- Road objects = sparse
- Side object = dense

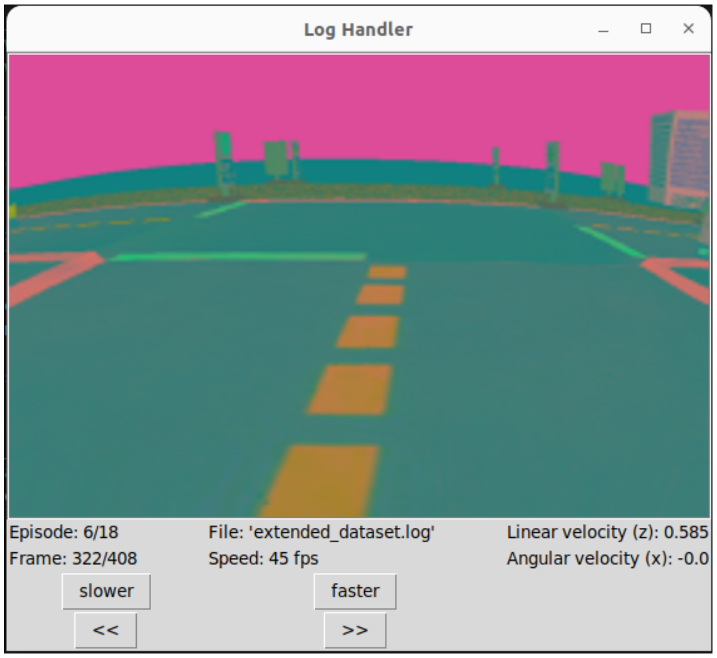
The output and the map we used to collect data from simulator is a .yaml file loadable from any python script.

4.1.2 Collecting training data

The next step was to use our map in a simulator with a script that captures information while driving the car. The script we used, human.py is originated at the Behavior Cloning repository. It allows the user to control the vehicle with an xbox controller and to decide which generated log is to be saved to the dataset after the session is finished. We used the script with the following main configurations:

- Our generated map as input
- Domain randomization for robustness
- extended_dataset.log as the output log file

We used an xbox ONE controller for the driving and modified a base code to personalize the controlling. After driving and generating for half an hour (keeping only the quality data), we managed to create a dataset of 500 MB which we expanded later to a 1.1 GB dataset.



Episode: 6/18 File: 'extended_dataset.log' Linear velocity (z): 0.585
Frame: 322/408 Speed: 45 fps Angular velocity (x): -0.0

Figure 1: A frame from the generated dataset.

4.1.3 File format and visualization

Briefly the log file contains a list of frames and actions. For each frame there is an action to take which is represented in linear and angular velocity. The size of each image is **150x200** pixels and the linear and angular velocities come in lists of floats with two element. The first version of our dataset now consists of 11651 list items.

For visualizing our data we used a tool, log_viewer.py originated from Behavior Cloning. With this utility we can see the recorded frames in order (as a video): Figure 1.

There are also a 1-minute recording of the captured log at link by the name of **logcapture.webm**. We modified the original **log_viewer.py** so that it shows the linear and angular velocities for each frame.

4.1.4 Creating train, test and validation sets

In code, the dataset can be read by the log_reader.py module also taken from Behavior Cloning. The output of its **modern_read()** function is a tuple frames, linear velocities, angular velocities. These lists can be easily split into train, and validation subsets by using **sklearn.model_selection** modules **train_test_split()** method. The validation split percent is 0.2-0.8 while for testing, we generated a completely new dateset on a different map to also check the robustness of the system.

4.2 Training the models

We have implemented two model architectures. At first a Convolutional Neural Network (CNN)[7], and after that a hybrid network, in which we have combined our first network with Long short-term memory (LSTM) layers [9]. To implement these networks we have used Keras [5] as we have found that it offers a consistent and simple APIs for Deep learning applications.

4.2.1 Describing the CNN

The first network we have implemented is a Convolutional Neural Network. The layer of the network normalizes the pixel intensities of the image. After the normalization we apply a 2D Max Pooling layer and some 2D Convolutional layers. After the Convolutional layers we have inserted another 2D MaxPooling layer. After that, we are flattening the output of the pooling layer and drive the data through some fully connected layers.

4.2.2 Describing the Hybrid Network

The hybrid network has a lot in common with our CNN. We have found the foundations of the first network was acceptable, but we were not satisfied with the result. To further improve the results of our network, we have complemented the network with LSTM layers. As the new network was not compatible with the data provided to the first network, we had also modified the data reading functionality. As we have overcame these obstacles the final network looks the following: a MaxPooling layer followed by some Convolutional layers and another MaxPooling layer. After that the output is flattened and some LSTM layers applied. At the end there are some Dense layers. With these changes applied on the model, we have achieved results that are better than the ones produced with the previous model. The LSTM uses 4 time-step windows.

4.2.3 Working with the two networks

As the differences started to grown between the two networks, we were facing a decision.If we wanted to be able to compare the results, we had to decide if we wanted to maintain two separate versions of our network or modify our program to be able to switch between the two.We have chosen the latter option, so the we could change

4.3 Hyper-parameter optimization

For hyper-parameter optimization we used Hyper Tune [6]. The first thing we need to do is writing a function, which returns a compiled Keras model. It takes an argument hp for defining the hyper.parameters while building the model.

4.3.1 Define the search space

When we want to tune the number of units in any of the layers we have to define an integer hyper-parameter with

```
hp.Int("linear_lstm_units", min_value=16, max_value=128, step=32)
```

, whose range is from 16 to 128 inclusive. When sampling from it, the minimum step for walking through the interval is 32. There are many other types of hyper-parameters as well and we can define multiple hyper-parameters in the function. In the following code, we also tune which activation function to use with hp.Choice().

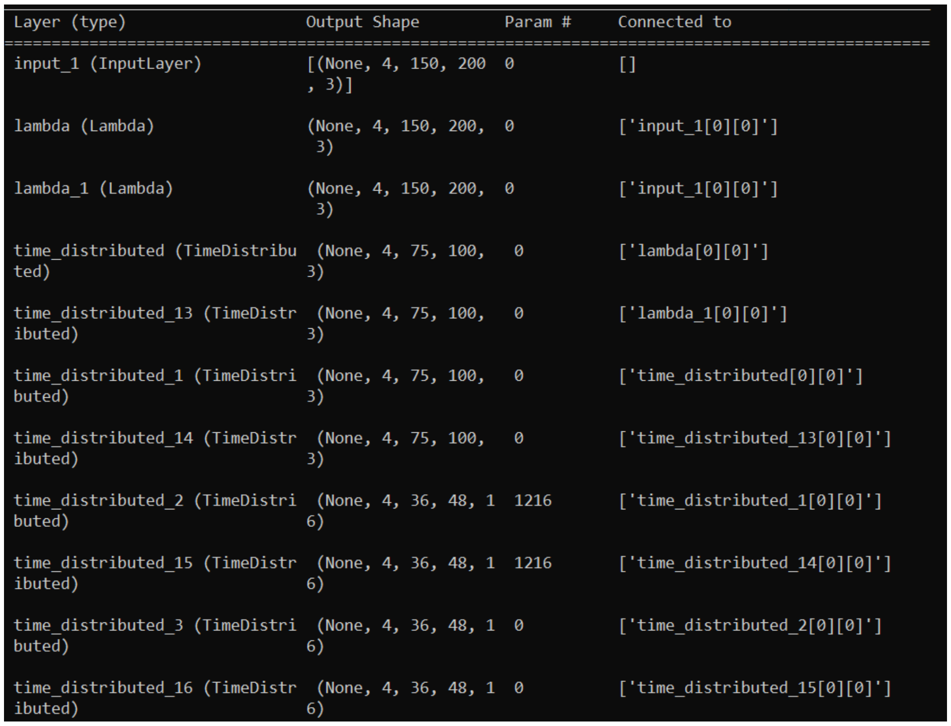
```
x = Dense(
          # Tune number of units
          units=hp.Int("linear_units", min_value=512, max_value=1200, step=64),
          kernel_initializer="normal",
          # Tune the activation function to use.
          activation=hp.Choice("linear_activation", ["relu", "tanh"]))(x)
```

Each of the hyper-parameters is uniquely identified by its name (the first argument). To tune the number of units in different layers separately as different hyper-parameters, we give them different names.

4.3.2 Start the search

After defining the search space, we need to select a tuner class to run the search. There are 3 options: RandomSearch, BayesianOptimization and Hyperband, which correspond to different tuning algorithms. Here we use RandomSearch. To initialize the tuner, we need to specify several arguments in the initializer:

- hyper-model: The model-building function, which is build in our case.
- objective: The name of the objective to optimize (whether to minimize or maximize is automatically inferred for built-in metrics).



```
Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)           [(None, 4, 150, 200  0           []
                               , 3)]

lambda (Lambda)                (None, 4, 150, 200,  0           ['input_1[0][0]']
                                3)

lambda_1 (Lambda)              (None, 4, 150, 200,  0           ['input_1[0][0]']
                                3)

time_distributed (TimeDistribu  (None, 4, 75, 100,   0           ['lambda[0][0]']
ted)                           3)

time_distributed_13 (TimeDistr  (None, 4, 75, 100,   0           ['lambda_1[0][0]']
ibuted)                        3)

time_distributed_1 (TimeDistri  (None, 4, 75, 100,   0           ['time_distributed[0][0]']
buted)                         3)

time_distributed_14 (TimeDistr  (None, 4, 75, 100,   0           ['time_distributed_13[0][0]']
ibuted)                        3)

time_distributed_2 (TimeDistri  (None, 4, 36, 48, 1  1216        ['time_distributed_1[0][0]']
buted)                         6)

time_distributed_15 (TimeDistr  (None, 4, 36, 48, 1  1216        ['time_distributed_14[0][0]']
ibuted)                        6)

time_distributed_3 (TimeDistri  (None, 4, 36, 48, 1  0           ['time_distributed_2[0][0]']
buted)                         6)

time_distributed_16 (TimeDistr  (None, 4, 36, 48, 1  0           ['time_distributed_15[0][0]']
ibuted)                        6)
```

Figure 2: Best hyper parameter configuration.

- max_trials: The total number of trials to run during the search.
- executions_per_trial: The number of models that should be built and fit for each trial. Different trials have different hyper-parameter values. The executions within the same trial have the same hyper-parameter values. The purpose of having multiple executions per trial is to reduce results variance and therefore be able to more accurately assess the performance of a model.
- overwrite: Control whether to overwrite the previous results in the same directory or resume the previous search instead. Here we set overwrite=True to start a new search and ignore any previous results.
- directory: A path to a directory for storing the search results.
- project_name: The name of the sub-directory in the directory. We named it „random_search_tuner”.

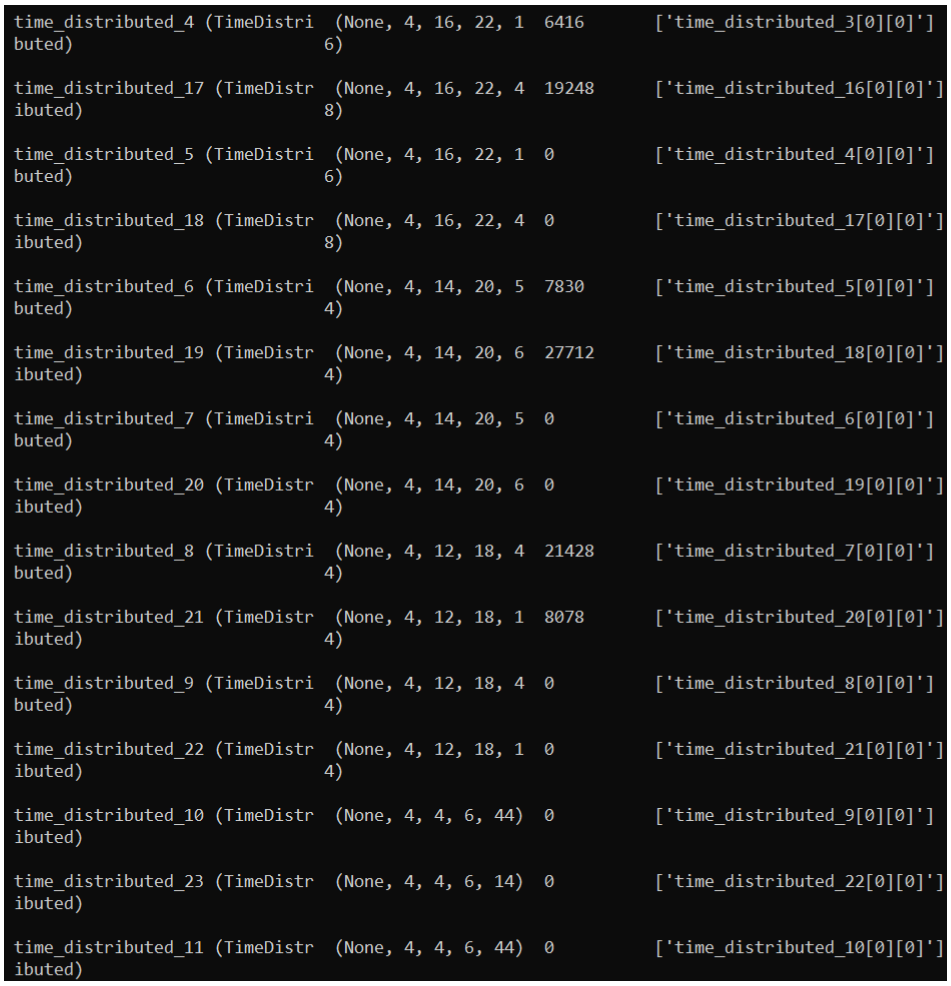
Then, start the search for the best hyper-parameter configuration. All the arguments passed to search is passed to model.fit() in each execution. During the search, the model-building function is called with different hyper-parameter values in different trial. In each trial, the tuner would generate a new set of hyper-parameter values to build the model. The model is then fit and evaluated. The metrics are recorded. The tuner progressively explores the space and finally finds a good set of hyper-parameter values.

4.3.3 Query the results

When the search is over, we retrieve the best model(s). The model is saved at its best performing epoch evaluated on the validation_data. We print a summary of the search results (tuner.results_summary()).

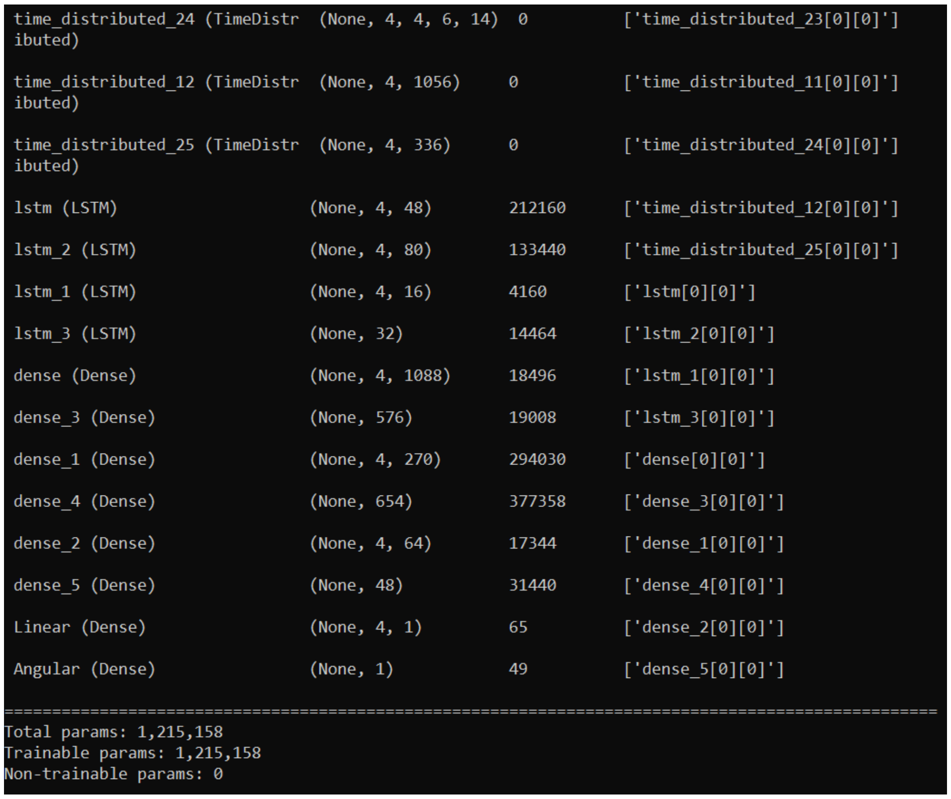
And the summary of the best model (best_model.summary()):

Here is the best model's summary in Figure 2, 3, 4. We see all layers output shape, params, and which layer is each layer connected to. At the end of the summary we see, that 1.215.158 total params is in the model and there is 0 non-trainable params.



```
time_distributed_4 (TimeDistri  (None, 4, 16, 22, 1  6416       ['time_distributed_3[0][0]']
buted)                          6)

time_distributed_17 (TimeDistr  (None, 4, 16, 22, 4  19248      ['time_distributed_16[0][0]']
ibuted)                         8)

time_distributed_5 (TimeDistri  (None, 4, 16, 22, 1  0          ['time_distributed_4[0][0]']
buted)                          6)

time_distributed_18 (TimeDistr  (None, 4, 16, 22, 4  0          ['time_distributed_17[0][0]']
ibuted)                         8)

time_distributed_6 (TimeDistri  (None, 4, 14, 20, 5  7830       ['time_distributed_5[0][0]']
buted)                          4)

time_distributed_19 (TimeDistr  (None, 4, 14, 20, 6  27712      ['time_distributed_18[0][0]']
ibuted)                         4)

time_distributed_7 (TimeDistri  (None, 4, 14, 20, 5  0          ['time_distributed_6[0][0]']
buted)                          4)

time_distributed_20 (TimeDistr  (None, 4, 14, 20, 6  0          ['time_distributed_19[0][0]']
ibuted)                         4)

time_distributed_8 (TimeDistri  (None, 4, 12, 18, 4  21428      ['time_distributed_7[0][0]']
buted)                          4)

time_distributed_21 (TimeDistr  (None, 4, 12, 18, 1  8078       ['time_distributed_20[0][0]']
ibuted)                         4)

time_distributed_9 (TimeDistri  (None, 4, 12, 18, 4  0          ['time_distributed_8[0][0]']
buted)                          4)

time_distributed_22 (TimeDistr  (None, 4, 12, 18, 1  0          ['time_distributed_21[0][0]']
ibuted)                         4)

time_distributed_10 (TimeDistr  (None, 4, 4, 6, 44)  0          ['time_distributed_9[0][0]']
ibuted)

time_distributed_23 (TimeDistr  (None, 4, 4, 6, 14)  0          ['time_distributed_22[0][0]']
ibuted)

time_distributed_11 (TimeDistr  (None, 4, 4, 6, 44)  0          ['time_distributed_10[0][0]']
ibuted)
```

Figure 3: Best hyper parameter configuration.



```
time_distributed_24 (TimeDistr  (None, 4, 4, 6, 14)  0          ['time_distributed_23[0][0]']
ibuted)

time_distributed_12 (TimeDistr  (None, 4, 1056)     0          ['time_distributed_11[0][0]']
ibuted)

time_distributed_25 (TimeDistr  (None, 4, 336)      0          ['time_distributed_24[0][0]']
ibuted)

lstm (LSTM)                    (None, 4, 48)       212160     ['time_distributed_12[0][0]']

lstm_2 (LSTM)                  (None, 4, 80)       133440     ['time_distributed_25[0][0]']

lstm_1 (LSTM)                  (None, 4, 16)       4160       ['lstm[0][0]']

lstm_3 (LSTM)                  (None, 32)          14464      ['lstm_2[0][0]']

dense (Dense)                  (None, 4, 1088)     18496      ['lstm_1[0][0]']

dense_3 (Dense)                (None, 576)         19008      ['lstm_3[0][0]']

dense_1 (Dense)                (None, 4, 270)      294030     ['dense[0][0]']

dense_4 (Dense)                (None, 654)         377358     ['dense_3[0][0]']

dense_2 (Dense)                (None, 4, 64)       17344      ['dense_1[0][0]']

dense_5 (Dense)                (None, 48)          31440      ['dense_4[0][0]']

Linear (Dense)                 (None, 4, 1)        65         ['dense_2[0][0]']

Angular (Dense)                (None, 1)           49         ['dense_5[0][0]']

==================================================================================================
Total params: 1,215,158
Trainable params: 1,215,158
Non-trainable params: 0
```

Figure 4: Best hyper parameter configuration.

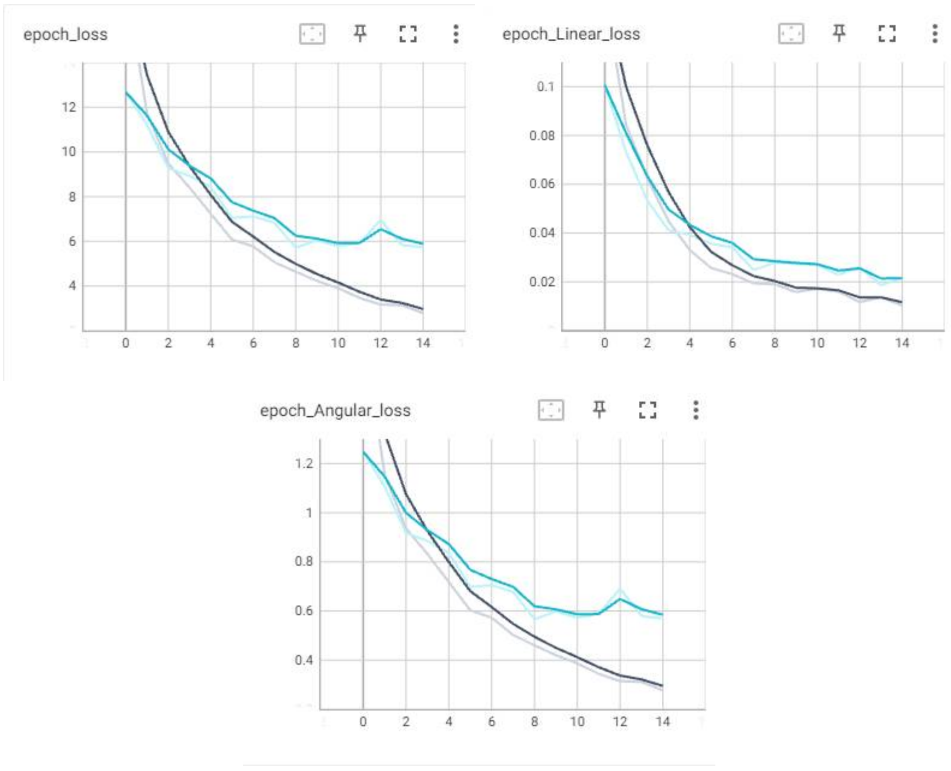


Figure 5: Training result for the convolution only model.

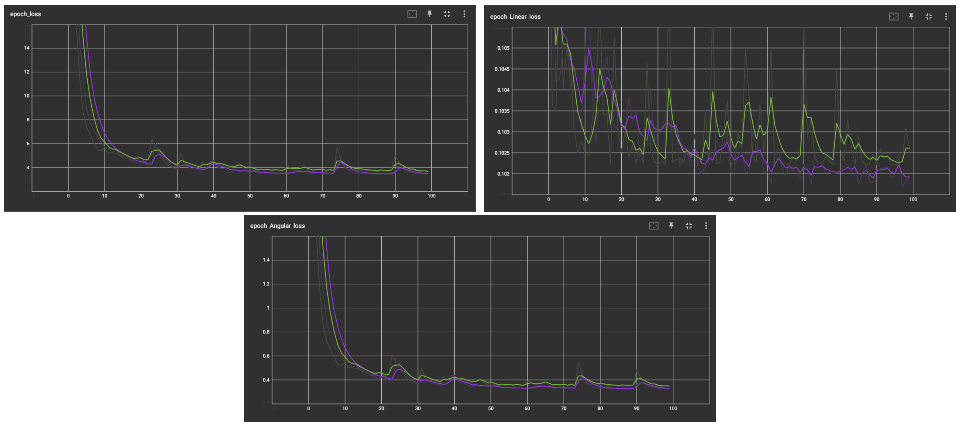
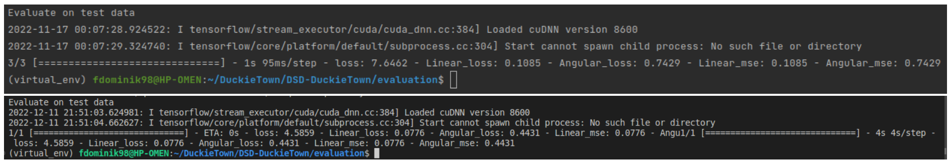


Figure 6: Training result for the LSTM only model.



```
Evaluate on test data
2022-11-17 00:07:28.924522: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8600
2022-11-17 00:07:29.324740: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
3/3 [==============================] - 1s 95ms/step - loss: 7.6462 - Linear_loss: 0.1085 - Angular_loss: 0.7429 - Linear_mse: 0.1085 - Angular_mse: 0.7429
(virtual_env) fdominik98@HP-OMEN:~/DuckieTown/DSD-DuckieTown/evaluation$
```

```
Evaluate on test data
2022-12-11 21:51:03.624981: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8600
2022-12-11 21:51:04.662627: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
1/1 [==============================] - ETA: 0s - loss: 4.5859 - Linear_loss: 0.0776 - Angular_loss: 0.4431 - Linear_mse: 0.0776 - Angu1/1 [==============================] - 4s 4s/step -
 loss: 4.5859 - Linear_loss: 0.0776 - Angular_loss: 0.4431 - Linear_mse: 0.0776 - Angular_mse: 0.4431
(virtual_env) fdominik98@HP-OMEN:~/DuckieTown/DSD-DuckieTown/evaluation$
```

Figure 7: Evaluation result comparison of the two models.

4.4 Evaluating the result

The training result of the two model was compared in TensorBoard. The convolution model evaluation can be seen in Figure 5 while the LSTM model evaluation can be seen in Figure 6. The comparison shows that the overall loss, the training loos and the validation loss are better for the LSTM model in all cases.

The graphs below show the results of each epoch. In case of our CNN approach, the result on the validation dataset is colored as light blue, and the result on the training dataset is colored as dark blue. In case of the hybrid model, which combines CNN with LSTM, the results on the validation set are colored as green and the results on the training dataset are colored as purple. As you could expect, the validation loss is usually bigger, but you can notice, in case of the hybrid network, the the values are closer to each other. We have discussed this, and concluded that the hybrid network is more general, which means it has more predictive power than the CNN.

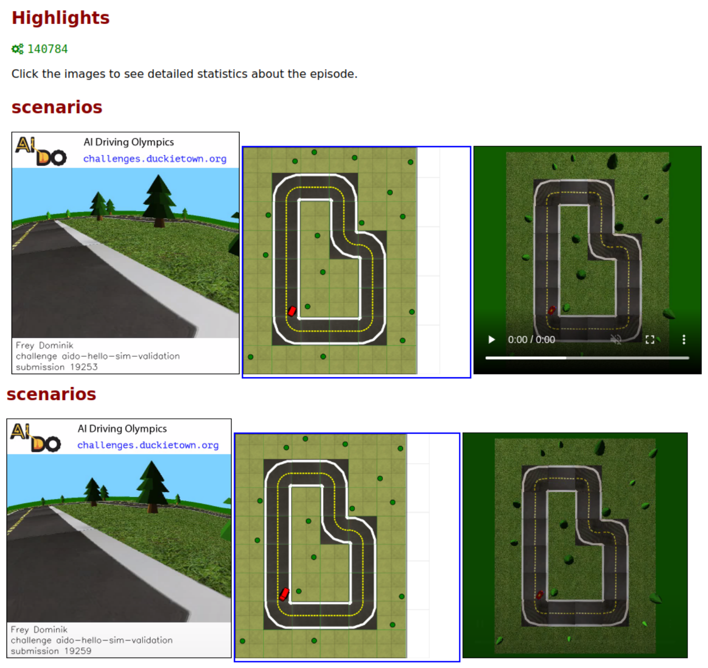
In the graphs below dark blue is the train dataset and light blue is the validation dataset plotted by TensorBoard. In all images we can see that validation loss is bigger than train loss than a few epochs later the other way around, also we used MSE (mean squared error) function in the model.

4.5 Testing on test dataset

The preparations for testing of the convolutional model is available in the README.md file inside the git repository. He we only compare the results. On Figure 7 the first row shows the result of the convolutional network and the second row of the LSTM network. We can see that the LSTM is almost twice as good as the basic convolutional. See Table 4.5

Model	Loss	Linear loss	Angular loss
Convolutional	7.6462	0.1085	0.7429
LSTM	4.5859	0.0776	0.4431

Table 1: Comparing losses of the two models


Highlights

140784

Click the images to see detailed statistics about the episode.

scenarios

AI Driving Olympics
challenges.duckietown.org

Frey Dominik
challenge aido-hello-sim-validation
submission 19253

scenarios

AI Driving Olympics
challenges.duckietown.org

Frey Dominik
challenge aido-hello-sim-validation
submission 19259

Figure 8: Simulation comparison of the two models.

4.6 Testing in simulation

We uploaded the two solution to https://challenges.duckietown.org. The simulation results can be seen in Figure 8, where the first figure is the result based on the convolutional model and the second is based on the LSTM model. In the first case the ego car failed immediately because it went out of bounds of the map. In the second case it survived longer, but it couldn't correct the angle and went out of bounds again. The models need some further improvement.

5 Conclusion and future work

Throughout our work we gathered relatively large training, validation and test datasets by driving the ego car in simulation and applying pure pursuit . Then we designed two models, one that contains only convolutional and dense layers combined with MaxPooling and one that has additional LSTM layers. We trained the models, applied hyper parameter optimization and evaluated the results both on the validation and the test datasets. Lastly we updated our solutions to https://challenges.duckietown.org to get feedback. We expected better result, but it is an open task in the future to improve the model and the dataset. We also want to try out reinforcement learning algorithms to see the difference.

References

[1] Róbert Moni András Kalapos, Csaba Gór and István Harmati. Sim-to-real reinforcement learning applied to end-to-end vehicle control. *Symposium on Measurement and Control in Robotics (ISMCR)*, 2020. https://doi.org/10.48550/arXiv.2012.07461.

[2] Róbert Moni András Kalapos, Csaba Gór and István Harmati. Vision-based reinforcement learning for lane-tracking control. *Symposium on Measurement and Control in Robotics (ISMCR)*, 2021. https://doi.org/10.21014/acta_imeko.v10i3.1020.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[4] Felipe Codevilla, Eder Santana, Antonio M. Lopez, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9931–9932, October 2019.

[5] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[6] Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Jixiang Li, Ji Liu, Ce Zhang, and Bin Cui. Hyper-tune: Towards efficient hyper-parameter tuning at scale. *CoRR*, abs/2201.06834, 2022.

[7] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2022.

[8] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuyama, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, Christopher Carr, Maria Zuber, Sertac Karaman, Emilio Frazzoli, Domitilla Del Vecchio, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. Duckietown: An open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504, 2017.

[9] Jin Wang, Liang-Chih Yu, K Robert Lai, and Xuejie Zhang. Dimensional sentiment analysis using a regional cnn-lstm model. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, pages 225–230, 2016.

[10] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. *Neural Computation*, 31(7):1235–1270, 07 2019.