Modified NVIDIA-based end-to-end self-driving model

Balazs Agoston Nyiro
School of Physics and Astronomy, University of Nottingham, Nottingham, NG7 2RD, UK

In this paper, I present the implementation of a modified NVIDIA behavioral cloning model[4] on a SunFounder PiCar-V toy car. In order to enable the vehicle to navigate safely in a total of 12 different situations during the live test, the neural network was trained using pre-recorded data. Thanks to this, a relatively low validation loss of 0.0126 was achieved. Furthermore, the car was able to successfully perform most of the tasks, except for the red traffic light, the consideration of arrows indicating direction and the execution of turning.

I. Introduction

Over the past decade, with the advent of ever more powerful processors and increasingly efficient machine learning technologies, long-standing desire to develop self-driving cars has become a reality again. Such self-driving systems must not only be able to recognise their surroundings and navigate through space according to their destination, but also be able to handle unexpected changes. The Society of Automotive Engineers (SAE), has classified this capability into a 5-degree scale[6] according to their level of sophistication. At the top (level 0) are systems that, although are not able to control the car, can warn the driver of potential hazards (e.g. lane departure). Level 2 systems can control the steering wheel and change the speed of the car, but the driver still has to intervene from time to time. While at the other end of the scale, self-driving systems no longer require human supervision because they can drive safely in all conditions.

The technologies of these self-driving systems can be divided into two main groups. The robotics-based approach [3] and the learning-based approach [1]. The first breaks the task into perception, planning and control subtasks, which are subdivided into further subtasks. Although this allows separate development of each function and easier implementation of new rules, the algorithmic coordination of the subtasks can be a human challenge for developers. Nevertheless, this is the architecture of the majority of self-driving systems currently on the market. In contrast, end-to-end learning defines autonomous driving as a machine learning problem. This allows the neural network to translate input data directly into commands via non-linear functions. The aim of this paper is to present such an end-to-end level 2 self-driving system, using a so-called imitation learning technique. The application of imitation learning, or Behavior Cloning (BC), in self-driving systems was first presented in 1989 by Pomerleau using a fully connected neural network[5].Later, Muller et.al applied BC to control a real-size RC car using 6 convolutional neural networks (CNN) layers. In 2016, Bojarski et al.[4] extended this to 9 CNN layers and trained the architecture on front faced camera images captured in real environment. This model was able to predict the next required speed and steering angle at 98% efficiency at a given time instant [4]. The accuracy of prediction can be further improved if transfer learning is applied. In the research of Huatao Jiang et.al., the mean squared error loss of the NVIDIA architecture was nearly halved during validation[4].

Although the advantage of these solutions is that in simpler traffic situations (such as holding a lane or stopping for a pedestrian), a satisfactory driving pattern can be achieved relatively easily. However, it has the drawback that based on cameras alone, it requires a lot of data and training, and because of the generalisation, it cannot handle unexpected situations which are not well covered in the training data. In this paper, I will take the NVIDIA model outlined above and compare it with a fine tuned transfer learning model.

II. Background

The project was based on a challenge organised as part of the University of Nottingham, Machine Learning in Science II lecture. This involved adapting self-driving systems to a SunFounder PiCar-V. This platform included, among other things, a Raspberry Pi 4 microcomputer and a camera with a 120-degree field of view. Among other things, the vehicle had to be able to keep in lane during the live test, stop in case of obstacles and respond to road signals (arrows, traffic lights). These abilities were tested on 3 tracks(Figure 1), through a total of 12 different tasks. These tasks are explained in detail in the results section.

III. Implementation

A. Data

A total of 31,673 images were used to teach the network. Out of these, 13,797 were taken from the public database provided for the challenge [7] and another 17,896 images were collected by us. These are RGB images with a resolution of 320x240 pixels, and the file name of each image is used to record the steering position and speed at the time of capture. They were

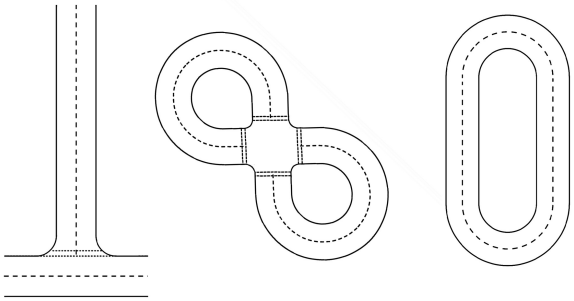

FIG. 1. The three differently shaped tracks

recorded by the vehicle at 1 FPS. For the data collection, the car was manually driven on the track so that 90% of the time it was as perfectly in the lane as possible, but another 10% of the time data was collected where the car was driven from the edge of the track back to the correct position. This was necessary because if the self-driving function fails and the car finds itself off the track, the database would not normally cover the correction of this error. This way, there is a chance that the model can correct itself. During data capture, the available elements (cubes, human figures, traffic lights and arrows) are alternated to best cover the positions and scenarios that can be rotated during testing. Furthermore, the data was split 80-20% between training and test datasets.

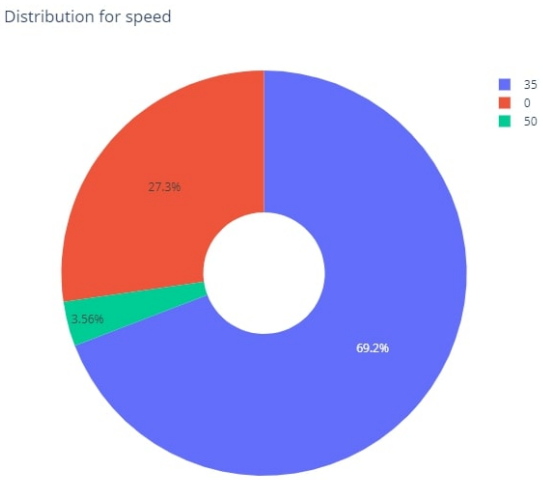

FIG. 2. Distribution of the 3 types of speeds [0,35,50] in the entire recorded data set.

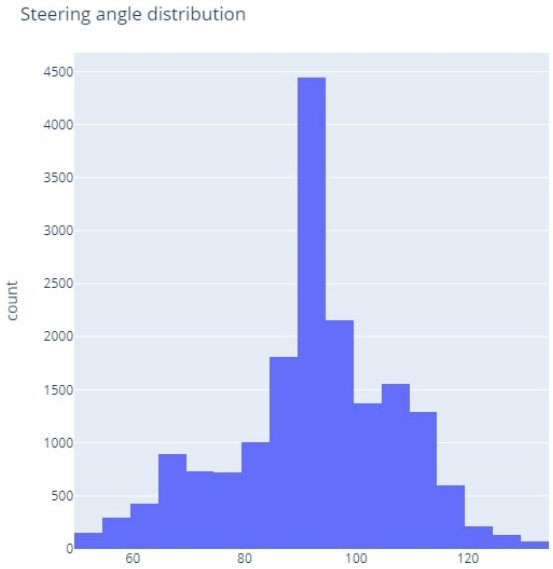

FIG. 3. Distribution of the different steering angles [0,180] in the entire recorded data set.

B. Data augmentation

In supervised learning, the training data should ideally cover all possible pairs of operation-value pairs within the operational design domain (ODD) of the system to be trained. However, it is not always possible to collect such an ideal data set. Furthermore, the methods outlined below can be used to avoid overfitting. In total, six methods were investigated in the project: zoom, brightness adjustment, added blur, flip, cut upper 1/3, undistortion, of which only the first three were used in the final model as the others degraded the quality of learning.

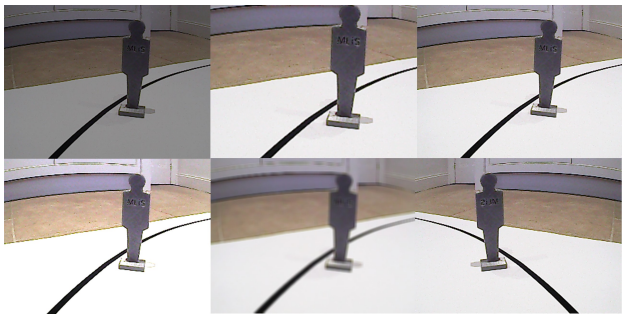

FIG. 4. Different data adjustment methods. In the left column, brightness adjustment in positive and negative direction. In the middle, added blur. On the right, flip the image.

1. *Brightness, Blur and Zoom*

Since the model should be able to perform under any lighting conditions and based on learning data

from any time of day, the brightness of all pixels in the images was adjusted uniformly between 70 and 130 percent during the augmentation. This rate was adjusted randomly from image to image. Another effective way to avoid overfitting is to add blur to the images. This was done in our project with a random kernel size between 1 and 5. Finally, to better capture the details of the images, a zoom of up to 25% was added to the input data. The probability of occurrence of the processes outlined above was the same for each image ($\rho_{zoom} = \rho_{brightness} = \rho_{blur} = 0.5$).

2. Flip

In order to generate more training data, we tested flip images by subtracting the normalized value of the steering position from 1. This methodology, however, resulted in an image of the vehicle's deck, which had previously been driving in the left lane, simulating driving in the right lane. Thus, although the ability to maintain the lane was enhanced, the vehicle did not maintain the left lane. In equation 1, the steering angle θ at discrete time t is represented by θ_t:

$$\theta_t = -\theta_t \tag{1}$$

C. Data preprocessing

Preprocessing is a two-step process of rescaling and normalisation. This allows for faster learning of the model.

1. Resizing

For the NVIDIA model, resizing was not applied to preserve as much information as possible. Thus, for this solution the input size remained 320x240 pixels. However, for EfficientNet B0, the image size was reduced to 224x224 pixels. This was necessary partly because of the predefined network size and partly because the 1:1 layout allowed the best use of square kernels.

2. Normalisation

Normalisation usually allows faster learning due to faster convergence of data. The pixels in the input images take values between [0,255] due to the RGB colour scale. For these, it is common to use normalization to convert the pixel data to values between [0,1]. In our case, we observed that the normalization caused the model to learn the prediction of steering position and speed less efficiently. This may be a consequence of a decrease in the distance between the values of the black and white pixels that give strong contrasts in the image. However, normalisation was applied to the speed and steering angle data as shown in the equations 2 and 3. While the steering angle before the conversion was set to [0.90] and the velocity to [0.50], after normalization both values fell into the range [0.1].

$$angle_{norm} = \frac{(angle - 50)}{80} \tag{2}$$

$$speed_{norm} = \frac{(speed - 0)}{35} \tag{3}$$

IV. Training

In preparation for the training process, images that may have been corrupted or contained incorrect manuscripts during data collection were manually sorted out. However, both augmentation and preprocessing were then performed on the fly during training. After each pass through the entire dataset, the data was shuffled, thus avoiding the network learning any patterns that might exist in the database. The same process was followed for the validation dataset.

A. Neural Network Architectures

1. NVIDIA

The front of the NVIDIA architecture mentioned earlier is made up of five units, each with a convolution section and a max pooling layer, followed by a dropout. These blocks allow the network to extract features from the images. This is finally followed by three fully connected layers that allow the extracted information to be mapped to steering angles and velocities. Finally, in the final architecture, a batch normalization layer is added after each block, as a regularization method to normalize the data and thus allow more accurate prediction.

At the very end of the model, there were two separate output layers (one for velocity and one for steering angle), with sigmoid activation due to the regression property of the problem. Also as a consequence, the loss value was calculated using Mean Square Error (MSE). The total model contained a total of 1,186,494 trainable parameters.

As mentioned in the layers, an exponential linear unit (ELU) activation function was used except for the output layer. As shown in Figure 5, this function returns exponential values for negative input values and linear values for positive values. The advantage of this function over the ReLu function is that since it is able to give negative results as output, it does not lead to

dead neurons. Furthermore, it allows faster convergence. The output layer activation functions in this model were chosen as sigmoid because it allows finer transitions in speed and steering position compared to linear activation.

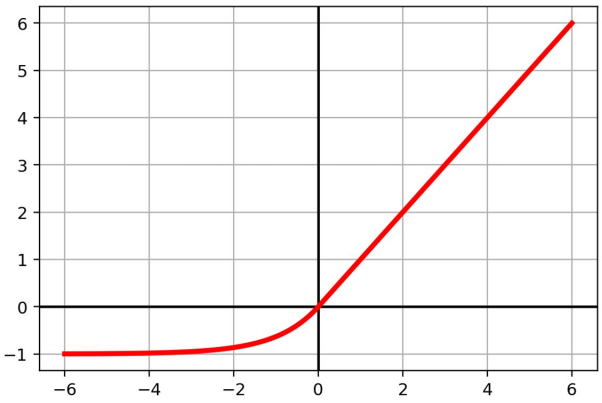

FIG. 5. The graph of the ELU activation function.

2. *EfficientNet B0*

EfficientNet is a widely deployed convolutional neural network architecture that uses complex coefficients to uniformly scale the depth and width dimensions. Unlike conventional methods, the EfficientNet scaling method uses fixed scaling coefficients [8]. It has the advantage that while it is more than 6x faster and 8x smaller than ConvNet, its prediction accuracy is not significantly reduced [8].

In addition to the NVIDIA model, this architecture was also tested, using the pre-trained values of imagenet for the weights. These were frozen so that they did not change during the learning process. This method allowed us to leverage the feature extraction capabilities of the model, which had previously been acquired by training on more images than ours, and to achieve better accuracy. The last layer before the output layers of the EfficientNet B0 network was fed through a batch normalization and a dropout layer into two fully connected output layers. For this architecture, we specified a classification problem for rates and used a softmax activation. While we continued to use the steering position as a regression problem and used sigmoid activation in the output layer. Accordingly, the loss value was calculated as categorical cross entropy for velocity and mean average error (MAE) for steering angle.

B. Hyperparameters and training

As mentioned earlier, the available data was split 80:20 between training and testing groups. Since the available server power allowed, the training was performed over 200 epochs with a learning rate of 0.001. For regularization, we used the built-in early stopping feature of TensorFlow which allowed us to obtain the best possible loss version of our model, thus avoiding the problem of overfitting. When choosing the learning rate, care should be taken that while a too low learning rate would increase the training time too much, a relative value would imply too high gradient update steps, which would risk exceeding the optimal minimum value. Finally, the optimal value was found by guessing. The dropout probability was set uniformly for both models to 0.5. For the optimizer, we chose the Adam optimizer, which is now an industry standard and allows to handle sparse gradients, even on noisy problems.

In terms of batch size, while the best accuracy and the lowest loss was achieved with the NVIDIA model with a value of 128, interestingly, EfficientNet showed optimal results with a value around 50. As no generally accepted method for this was found, this was also chosen by trial and error. At the end of learning, the trained neural network and its weight values were saved in .h5 format using the save function provided by TensorFlow.

V. Deployment

During the deployment phase, after a preprocessing with the trained model, we were able to predict the steering angle and speed required at the next time instant. For this task, we used only the NVIDIA model for both the online and the live test. The last image captured by the car's camera, after a resize and conversion to the appropriate numpy array format, was fed to the pre-loaded network, which allowed TensorFlow's built-in predictor function to return the two outputs (angle and speed) we expected. Since these values fall between zero and one due to the normalization done before loading, they need to be converted back to a meaningful range for the car's API using the equations 4 and 5 below. Because of the sigmoid activation feature, the model rarely gave a value of zero for the speed value for roadblocks and stopped, but predicted a noticeably low value. Therefore, a threshold value was introduced and the output velocity value was set to zero for a lower predicted velocity.

$$angle_{driving} = (angle_{norm} * 80) + 50 \tag{4}$$

$$speed_{driving} = angle_{norm} * 35 \tag{5}$$

VI. Results

During training, the NVIDIA model was able to achieve validation MSE values of 0.0126 and training

MSE values of 0.003. The same values for EfficientNet B0 were 0.0015 validation MSE and 0.0023 training MSE(Figure 6).

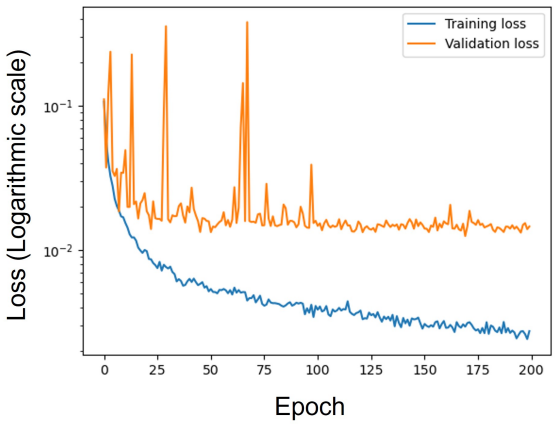

FIG. 6. Evolution of training and validation loss over 200 epochs of learning.

A. Live testing

During the live testing, the car had to be able to solve different traffic situations on an oval, an octagonal and a T-shaped track.

1. *Driving with lane following, without other conditions*

The model was able to follow the lanes smoothly and evenly, although its wheels sometimes touched the continuous line on the outside of the track. It was also able to do this at speeds higher than the initially planned maximum speed.The car was not disturbed by human or other shapes next to the road.

2. *Stopping for obstacles in the way*

During the test, the car was able to stop in the event of road obstacles of all types and locations. In addition, it also stopped in cases not intentionally reported in the training dataset when it reached the edge of the table, had a laptop in front of it, or was lifted out of the plane of the table.

3. *Reacting to traffic lights and arrows*

Unfortunately, the model did not learn to take into account either the red signal from the traffic light or the desired turning direction indicated by the arrows. In addition, the right turn could only be executed with substantial lane departure.

In addition, we have encountered serious problems during the deployment of the self-driving function. One of them was finding the right camera angle. One of the reasons for this was that the vertical and longitudinal angles at which the camera position was set varied from car to car, making it difficult to determine the settings used to capture the data.

VII. Discussion

By modifying the NVIDIA model and adding batch normalization layers, very low loss values were achieved for the architecture. And although the transfer learning model based on EfficientNet could not be tested in the live test, in general, although a more complex neural network, and with pre-trained weights, can achieve higher accuracy, but with a limited hardware environment such as the Raspberry Pi 4 in this case, a significant reduction in interference time would occur. The model we used in the live test typically performed poorly in situations that were not adequately covered by the training dataset. In total, the situations that occurred in the red light tasks were only 1% of the images we captured. When we showed our model based on transfer learning the images of our test dataset containing red traffic lights, the model hit the correct rate 100% of the time. Running this model in real time, however, resulted in an interference time on the car of over 1 second, which made it impossible to use it real time.

In terms of turning, however, it can be said that the training database containing unequal amounts of left and right turns is to blame for the poor performance. In the future, therefore, the self-driving performance of the car could be improved by converting the EfficientNet model for speed to a TensorFlow lite model, which would result in much lower interference times. On the other hand, for cornering, more and better balanced cornering data would be needed. Based on other studies, further improvements would probably be achievable using a CNN+LSTM based model[2].

VIII. Conclusion

In this project, we have presented an implementation of a self-driving function based on a modified end-to-end NVIDIA model, which achieves very low loss values and good driving results for the architecture. The car was not only able to keep itself within the lane and ignore distractions outside the lane, but it could also stop in front of pedestrians or other objects in the lane. At the same time, the model showed failures with regard to dynamic cues (such as traffic lights or directional arrows) that indicate turning or

expected driving behaviour. In a future work, further improvements could be achieved by combining the model with other types of architectures.

References

[1] *[1803.01164] The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches*. URL: https://arxiv.org/abs/1803.01164 (visited on 05/20/2022).

[2] *[2103.08116] Improving Generalization of Transfer Learning Across Domains Using Spatio-Temporal Features in Autonomous Driving*. URL: https://arxiv.org/abs/2103.08116 (visited on 05/20/2022).

[3] *A review of mobile robots: Concepts, methods, theoretical framework, and applications - Francisco Rubio, Francisco Valero, Carlos Llopis-Albert, 2019*. URL: https://journals-sagepub-com.nottingham.idm.oclc.org/doi/full/10.1177/1729881419839596 (visited on 05/20/2022).

[4] *Deep Transfer Learning Enable End-to-End Steering Angles Prediction for Self-driving Car | IEEE Conference Publication | IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/9304611 (visited on 05/20/2022).

[5] *Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios - Tingxiang Fan, Pinxin Long, Wenxi Liu, Jia Pan, 2020*. URL: https://journals.sagepub.com/doi/full/10.1177/0278364920916531 (visited on 05/20/2022).

[6] *J3016_201806: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles - SAE International*. URL: https://www.sae.org/standards/content/j3016_201806/ (visited on 05/20/2022).

[7] *Machine Learning in Science II*. en. URL: https://kaggle.com/competitions/machine-learning-in-science-2022 (visited on 05/20/2022).

[8] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. Tech. rep. arXiv:1905.11946. arXiv:1905.11946 [cs, stat] type: article. arXiv, Sept. 2020. DOI: 10.48550/arXiv.1905.11946. URL: http://arxiv.org/abs/1905.11946 (visited on 05/20/2022).