A machine learning approach to predict a vehicles velocity using dashcam video footage

Florian Wolf
Department of Mathematics and Statistics, University Konstanz
Konstanz, Germany
florian.2.wolf@uni-konstanz.de

Franz Herbst
Department of Physics, University Konstanz
Konstanz, Germany
franz.herbst@uni-konstanz.de

***Abstract*—Calculating the velocity of a moving camera relative to its surrounding, the so-called visual odometry problem, is a very challenging task and heavily studied in the area of computer vision. Especially in the field of self-driving cars, a fast and dependable velocity calculation is a high priority. In this report we will give a machine learning approach to solve the visual odometry problem, using optical flow fields combined with convolutional neuronal networks, as well as siamese neuronal networks. We use a data set with real world dashcam footage and even extend the data by gathering our own driving videos.**

***Index Terms*—deep learning, computer vision, visual odometry, convolutional neuronal network, siamese network, optical flow**

I. Introduction

Autonomous driving is considered to have a key role in the future of human mobility. Recently the topic has therefore gained great attention in research, economy and politics [1]. In this project, we approach this topic in a related, but more simplified setup and present several machine learning techniques and network architectures.

A. Aim of the project

The goal of this project is to predict the velocity of a car based on a dashcam video, inspired by the *comma*[1] speed challenge[2] published in 2018. To achieve this goal, we use optical flow analysis to preprocess the data and neuronal networks with different structures (classical and siamese) to predict the velocity. In order to measure the quality of the predictions, we use the mean squared error (MSE) of the measured velocity.

II. Data collection, analysis and preprocessing

To start the project we used the database given in the *comma* speed challenge project, which consists of a 17 minute training video (20400 frames) and the corresponding car velocity, as well as a 9 minute test video without labels, to validate the model on an unknown data set.

A. Data collection

Due to our limited computational power, we mainly used this data set to study different training techniques and model architectures. Still we could not expect good generalization for this rather small pool of training data. We therefore developed a technique to acquire more data that needed minimal resources. We used the open source smartphone apps *open camera* and *open street maps* to cast a video while driving and map the velocity using GPS at the same time. So we were able to train our final model with 1 hour 40 minutes of video data.

B. Data analysis

In order to evaluate our raw data with the model, we analysed our dataset concerning the recorded images and driving scenarios. The original frames have a size of $(640, 380, 3)$ pixels (RGB). To reduce the computation time, we cut the borders of the frames to exclude parts not needed for detection and sampled them down to half of their pixel size.

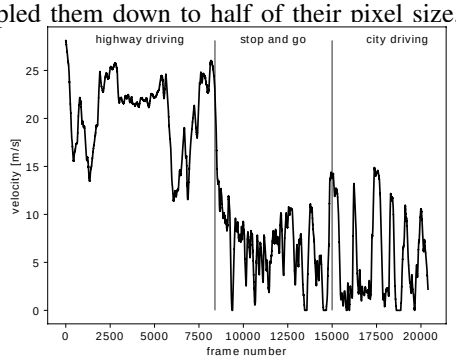

Fig. 1: Distribution of the velocities of all frames in the training video, including the three categories.

To visualize the training process in different driving scenarios, we plotted the velocity in dependence of the frame number (Fig. 1). After additional evaluation of the video we were able to classify three different driving scenarios: *highway driving* with high and relatively steady velocities, *stop and go* with low and fluctuating velocities, as well as *city driving* with very abrupt speed changes between medium and very low velocities. Every driving scenario is represented with around one third of the training data.

We would then evaluate the predictions of each model $p(x_i)$ to the input data x_i using the mean squared error with respect to the real value of the velocity y_i.

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(p(x_i) - y_i)^2$$

[1] https://comma.ai/
[2] https://github.com/commaai/speedchallenge

In order to classify the resulting value, we used the velocity plot and the following rough classification[3]:

$\sqrt{\mathcal{L}} \gtrsim 16$:	no fitting
$16 \gtrsim \sqrt{\mathcal{L}} \gtrsim 10$:	average velocity fitted
$10 \gtrsim \sqrt{\mathcal{L}} \gtrsim 5$:	scenarios detected
$5 \gtrsim \sqrt{\mathcal{L}} \gtrsim 1$:	variances within scenarios detected
$1 \gtrsim \sqrt{\mathcal{L}}$:	perfect fitting

TABLE I: Classification of the result based on the square root of the MSE.

For the training process we used a splitting of 80% training data and 20% validation data. Initially, we choose a hard splitting of the entire data, but this neglects the distribution of different driving scenarios in the video. Therefore we went on first splitting the data into the driving scenarios with then assigning training data and validation data on each.

C. Preprocessing

After the previously explained preparation steps, we needed to find a method to extract information about the motion of the vehicle, to be able to predict its velocity. There are two ways of doing this: either fit two following frames into a network, which will be discussed later, or calculate the relative motion between two frames, the so-called *optical flow*, and fit the results into a model.

To calculate the optical flow, we used the *Farneback pyramid method* [2]. The idea of this method can be split up into two main steps: First, use a polynomial expansion of the image f and second, solve the optical flow equation

$$\partial_x f \cdot V_x + \partial_y \cdot V_y + \partial_t f = 0$$

for different resolutions of the image (pyramid structure). This methods yields a dense optical flow field V with $\mathrm{im}(V) \subset \mathbb{R}^2$ (a two-dimensional vector field).

We used the following parameters for our calculations:

$$\begin{aligned} \text{pyramid levels} &:= 3 \\ \text{pyramid scaling} &:= 0.5 \\ \text{window size} &:= 6 \\ \text{pixel neighborhood size} &:= 5 \\ \text{SD of the gaussian filter} &:= 1.1. \end{aligned}$$

We choose three pyramid levels, because we wanted the calculations to be more accurate. To decrease the training duration, we halved the size of the optical flow frames again, resulting in a resolution of $(160, 105, 3)$ pixels per frame. As we used a window size of six pixels, a comparison between the original optical flow and the down sampled one lead to the result, that we do not loose a lot of information.

The optical flow calculation returns the magnitude and the angle of the flow vectors, which we transformed into polar coordinates.

[3]These rules do only apply on datasets with an equal distribution of all driving scenarios

To get an RGB image representing the optical flow between two consecutive frames, we normalized the magnitudes and put them into the third channel of the frame. The values of the second channel were all set to the value 255. We then multiplied the angle with the factor $180/(2\pi)$ and set this value for the first channel.

III. Method selection and architecture

The prediction of the vehicles speed is a non-linear regression task, so the choice of a neural network is reasonable. Recent architectures (ResNet, GoogLeNet, etc.) have shown that using multiple stacked convolution layers combined with stacked dense layers, perform well on image classification tasks. Therefore the choice of a convolutional neural network is justified.

A. Initial Network

As an initial architecture we used the model presented by the *NVIDIA* work group in [3]. This network was designed for self-driving cars, so the model has enough complexity to handle a task like ours and allowed various possibilities to fine-tune and improve the architecture. The raw structure is shown in Fig. 2.

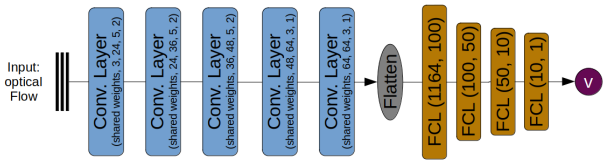

Fig. 2: Initial Network using a series of convolutional layers. The resulting feature vector is then flattened and mapped into several fully connected layers.

Using the initial model on the training data with a hard splitting, we achieved a MSE between 18 and 20 in the validation set, while having a MSE of less than 3 on the training set. The initial network therefore has some clear overfitting problems, which we tried to address with our fine tuning.

B. Siamese Network

In order to improve our results, we also tried to extend the optimized initial network (Section IV) to a siamese network, which would allow us to use also the original images as training data. Therefore we used shared weights for the convolutional layers, concatenated the two resulting feature vectors and mapped them into the fully connected layers as shown in Fig. 3. This method was inspired by the architecture used in [4]. We used this architecture in two different ways: feeding either two consecutive frames into the model, or one image and one optical flow field frame. In order to keep the structure correspondent, we sampled down the regarding images to the size of the optical flow field.

IV. Fine tuning

We experimented with various different options to fine tune the original model which will be discussed and compared:

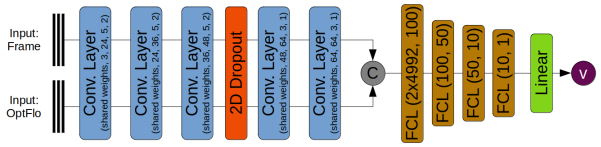

Fig. 3: Siamese architecture based on the fined tuned original network. The two inputs pass several convolutional layers with shared weights and a drop out layer. The resulting feature vectors are concatenated and then mapped into several fully connected layers.

A. Batch normalisation and activation functions

Similar to the lecture, we included batch normalization layers [5], to speed up the training and improve the networks performance.

We also tested different activation functions. As proposed in the lecture, we initially used the ReLu function

$$\mathrm{ReLu} : \mathbb{R} \to \mathbb{R}_0^+, x \mapsto \max\{0, x\}.$$

Using the ReLU function and 15 epochs for training, we achieved a MSE of around 15 on the testing set. We ran the code multiple times, to ensure this result holds. This result was not really promising, so we wanted to decrease the error by modifying the model even more.

To solve the problem of dead neurons[4] of the ReLU function, we tried the leakyReLU function

$$\mathrm{leakyReLU} : \mathbb{R} \to \mathbb{R}, x \mapsto \begin{cases} x, x \geq 0 \\ c \cdot x, x < 0 \end{cases}$$

with a hyperparameter c=0.01. Using the leakyReLU function, we achieved a MSE of around 12 on the testing set and under 3 on the training set. All of the models were trained with only eight epochs, as we still had problems with overfitting the data.

B. Dropout Layer

As we still had a lot of overfitting issues, we decided to include a dropout layer, according to [6]. We tried different positions and different amounts of dropout layers, but using one layer with a probability of p=0.5 after the third convolutional layer seemed to work best.

C. Pooling layers with initial splitting

In order to force the network to compress the data even further, we added two generic pooling layers to the network, to reduce the number of parameters[5] of the model. One after the second convolutional layer and the second one right before the fully connected layers start. We tested maximum and average pooling with the following parameters

$$\text{kernel size} := 2 \times 2 \qquad \text{stride} := 2$$
$$\text{padding} := 1 \qquad \text{dilatation} := \text{None}$$

The implementation of pooling layers helped a lot, as now the loss on the train and test data seemed to decrease nearly proportionally. Our results with the initial splitting are shown Table II. We also tried the network with maximum pooling with 15 epochs, as the loss on the train and test set was decreasing in a pretty stable manner. Using this network, we achieved for the first time a MSE of under 10 on the test set, which is, according to our assumptions in Section I-A, a pretty good result — especially, as we trained the model mostly on highway scenes and tested it only in city driving scenarios.

Initial splitting, 8 epochs	ReLU		leakyReLU	
	Train	Test	Train	Test
No pooling	2.85	12.08	2.45	10.75
Max pooling	5.62	11.82	5.52	10.29
Max pooling (15 epochs)	-	-	**3.22**	**9.63**
Average pooling	7.70	11.40	6.08	13.09

TABLE II: MSE results of the network using different pooling strategies, one dropout layers, two different activation functions and the initial splitting. We trained each of the models for eight epochs.

D. Optimizer and scheduler

For all of our networks we used the *ADAM* [7] algorithm as a (stochastic) optimizer. For scheduling, we tried the *ReduceLROnPlateau*-scheduler of the *PyTorch*[6] module and later switched to the *MultiStepLR*-scheduler.

V. Error analysis and results

In order to compare the potential and accuracy of the different predictions we focused on two major points: Firstly comparing the training and validation loss and how they would converge as well as secondly using the velocity-frame chart of the predictions to classify the results as described in Section II-B. These results are shown in Fig. 4. Finally we also compared the performance on the test video to get an indication how the models would perform on a completely unknown data set (Fig. 5).

We can identify that the two siamese approaches perform very similarly on the training set while producing only a very raw fitting of the driving scenarios on the validation set. The validation loss of both models converges at about 30 epochs while the training data is almost perfectly fitted. So both models, especially the model using only two frames, still produce a large overfitting. This also results in a very bad performance on the test video, as depicted in Fig. 5. The

[4]One can clearly see in the definition of the ReLU function, that neurons with a value below zero cannot participate in the learning process.

[5]Indeed, the number of parameters decreased from a total of 636.225 trainable parameters to 156.225 — a decrease by a factor of 4. We calculated these numbers using the tool *PyTorch summary* (https://github.com/sksq96/pytorch-summary).

[6]https://pytorch.org/docs/stable/index.html

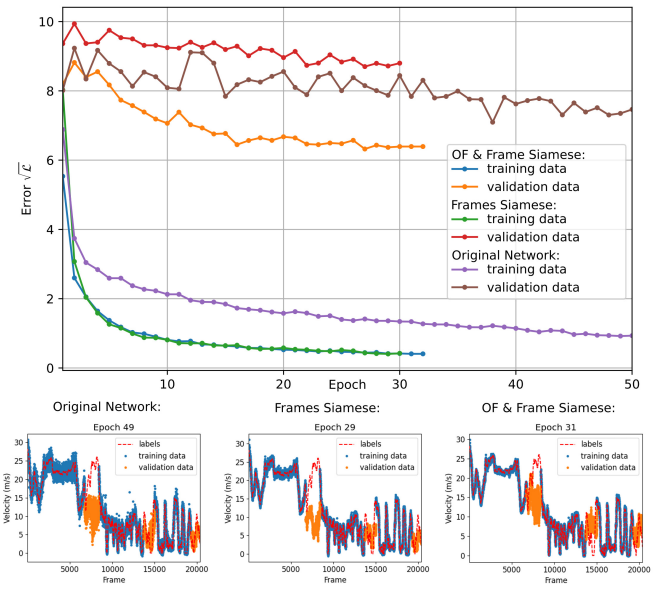

Fig. 4: Comparison of the results using the models: tuned original network (after 50 epochs: $\sqrt{\mathcal{L}} = 7.46$) and siamese model with frames (after 30 epochs: $\sqrt{\mathcal{L}} = 8.82$) and optical flow (after 50 epochs: $\sqrt{\mathcal{L}} = 6.39$); *top*: training and validation error per epoch; *bottom*: predictions after the last epoch with correct values in red

model was not even able to decide between different driving scenarios there.

Our optimized original model using only the optical flow showed a better performance. As shown in 4 the results did converge much slower but at about the same speed for both training and validation set. We stopped our calculations after 50 epochs but the model did not converge finally yet. We can also determine that the predictions are nearly perfect for city driving and stop and go traffic but only very rough for highway driving. Looking at the predictions on the test set the network was able to distinguish between all three different driving scenarios and did even detect all of the stops during the city driving. Still the absolute velocity may be not entirely correct especially for highway driving.

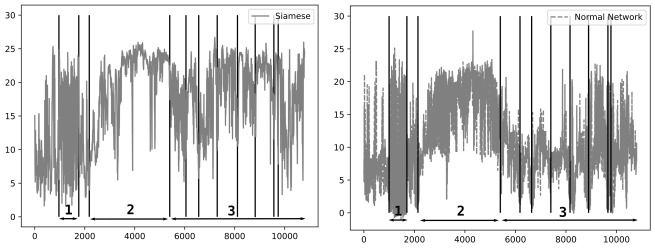

Fig. 5: the networks performance on the test video with the different driving scenarios labeled: 1) longer halt at crossroad, 2) highway driving, 3) city driving with several stops indicated

VI. Discussion and possible optimizations

Especially the performance of the optimized original model was very satisfying, considering the small amount of training data. On the other hand, the siamese network did not show the expected optimization of the predictions.

Additionally, we identified several problems, that need to be solved, to achieve a better performance:

(i) **Predictions are limited to specific video parameters**, especially to camera perspective and changes in brightness. This issue occurred, when we tried testing our models using our own acquired data. As a solution, rescaling and moving frame cutouts for training, might be a possible approach as well as brightness augmentation.

(ii) **Lack of generalization**: As we also experimented with different splittings of training and validation data, we figured out that using randomly shuffled validation data results in a very good fitting which becomes a qualitative fitting for block splitting and a very rough fitting for different videos. This might be solved by using more data from various different sources.

(iii) **Very complex model trained with very little data**: this issue is rather connected with the second and might as well be solved with a larger data set.

References

[1] Markus Maurer et al. *Autonomous Driving: Technical, Legal and Social Aspects*. Berlin, Heidelberg: Springer, 2016.

[2] Gunnar Farnebäck. "Two-Frame Motion Estimation Based on Polynomial Expansion". In: *Scandinavian Conference on Image Analysis* (2003), pp. 363–370.

[3] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars". In: (Apr. 2016). URL: https://arxiv.org/pdf/1604.07316v1.pdf.

[4] Sen Wang et al. "DeepVO: Towards end-to-end visual odometry with deep Recurrent Convolutional Neural Networks". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)* (May 2017). DOI: 10.1109/icra.2017.7989236. URL: http://dx.doi.org/10.1109/ICRA.2017.7989236.

[5] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (Feb. 2015). URL: https://arxiv.org/pdf/1502.03167.pdf.

[6] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[7] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).