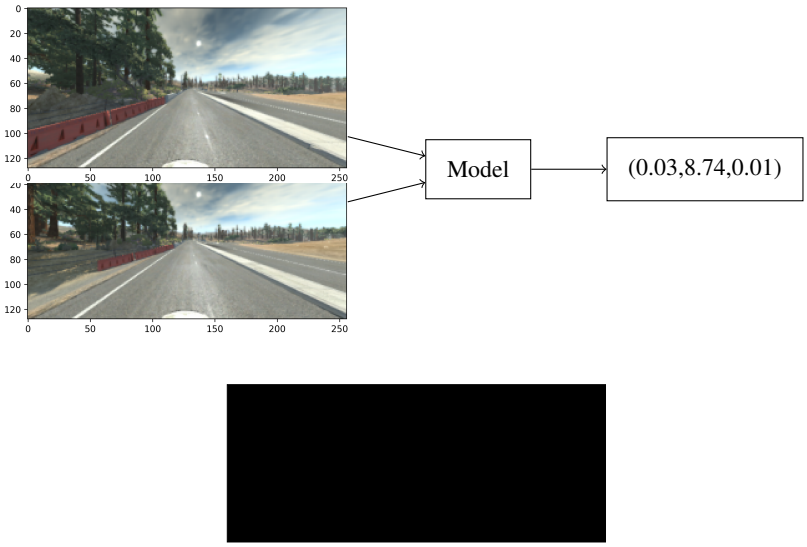
Predicting the speed and movement direction of a vehicle



***Abstract*—Infering the movement of a camera relative to its surrounding is a challenging task that has been researched extensively throughout the computer vision community. Especially for driver-assistance systems, fast and reliable estimation of the movement of a vehicle are of great importance. We propose an end to end deep learning approach to this problem, trained on images from a simulation.**

***Index Terms*—Machine learning, deep learning, computer vision, estimation, speed, movement, driving, simulation**

I. Introduction

Many of the sensors available on a road vehicle have certain cases where they may yield insufficient results for movement estimation. E.g. under heavy braking relying on the wheel speed sensors may lead to underestimation of the actual movement. Therefore adding another sensor, in this case a camera, may increase the performance of driver assistance systems. For this reason, we try to add another movement estimation system based on a single camera. Based on the simulation data of BeamNG.research [1] we train a deep neural network to output movement predictions relative to the first of two subsequent camera frames. The system takes two subsequent camera frames as input and outputs a 3-dimensional vector representing the movement of the vehicle.

II. Data collection

For collecting training data the only viable option for this project scope was to use a simulation. As vehicles are common use cases for the proposed system, a vehicle simulation was a reasonable choice.

Therefore, we used BeamNG.research [1] for simulating a car journey throug a virtual world. BeamNG.research [1] is a driving simulator mainly used for research in the area of driver assistance systems and autonomous driving and offers a comprehensive set of interaction possibilities.

A. Interfacing with the simulation

Using python, it is possible to interface with the simulation engine and control the following and more properties:

- Start a level in the simulation
- Pause the simulation
- Resume the simulation
- Create a vehicle
- Control a vehicle
  - Throttle
  - Breaking
  - Steering
  - Toggle AI driver
- Place virtual cameras
- Get images from virtual cameras
  - RGB images
  - Depth information
  - Pixel-wise annotation data

Especially the posibility to easily fetch images from virtual cameras was usefull for the project.

B. Collection pipeline

For collecting the required image pairs the following steps were taken:

- Start BeamNG.research [1]
- Fetch frames with position information from the simulation as frequent as possible
- Pair subsequent frames
- Remove pairs with a time difference above 1 second
- Transform the movement to a vehicle-relative space
- Write image and position data to disk

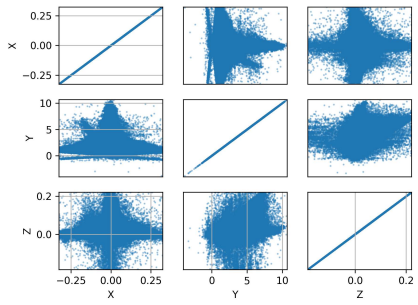


Fig. 1. Movement in X, Y and Z direction off the whole dataset

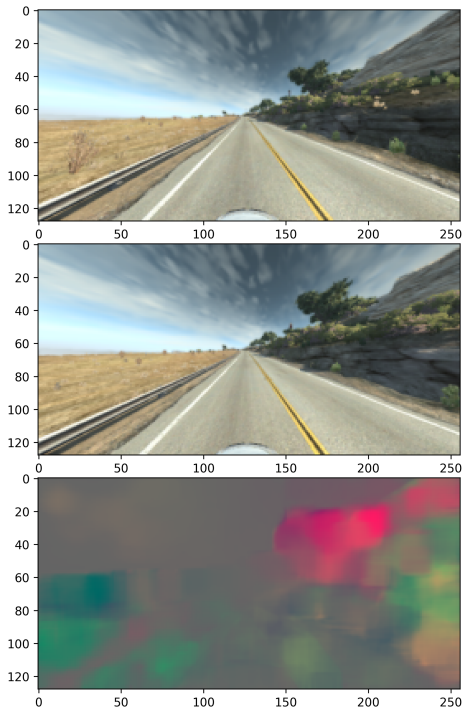


Fig. 2. Downscaled image from the dataset (top), subsequent image (middle), correspondig dense flow image(bottom) with X movement in the red color channel and Y movement in the green color channel

C. Data inspection

The data in Fig. 1 shows, that the vehicle mainly moves into Y direction and the movement is roughly limited to 12 meters inbetween frames. This limited movement distance may give the model a sufficient set of features to track and infer movement from.

D. Dataset size

Those images were saved in the PNG format to save disk space and avoid lossy compression, which may have induced worse results. In total, the dataset size was around 130 GB.

III. Preprocessing

For preprocessing the data, we scaled the collected images down to 256×128 pixels, to decrease the training duration. Furthermore, we also infered the dense optical flow field [9] from each image pair to give the model a clue about the movement. The two channels of the optical flow field are used instead of the raw images, as our experiments have shown that adding the images does not yield a performance increase. Fig. 2 shows that the flow field of an image pair may contain usefull information; especially the relative movement on the tree on the right hand side of the image may be usefull to determine movement. Although other methods [5] have shown that neural networks are able to learn optical flow calculation, this preprocessing step improved the training performance in our case.

IV. Method selection

As the proposed problem requires to solve a non-linear regression task and a nearly infinite amount of data is available, a neural network is the model of choice. Given that the dataset consists of images as input data, a convolutional neural network may take advantage of the structuredness of the input data.

Recent architectures [2], [3], [4] have shown that using multiple stacked convolution layers with appended dense layers yield good results on image classification data. There is also a method [5] for approximating optical flow using deep learning, which makes extensive use of stacked convolutional layers.

As optical flow may be related to the problem introduced in I, the choice of a deep convolutional neural network is further justified. To be able to achieve the best possible results we benchmarked different common model architecture types.

A. VGG [4]

The oldest architecture style discussed here uses blocks of two subsequent convolution operations with a relu activation after each convolutions. After each block, the feature image size is reduced by a factor of 2 using max pooling. After getting a feature image size of 7×7 pixels, a couple of dense layers are used to get the final classification.

B. ResNet [2]

The ResNet architecture makes extensive use of the batch normalization operation and features residual connections around each convolution operation. Residual connections are additions of the input of an operation with its outputs. This is said to preserve information infered in previous layers. The ResNet architecture has proven to be scalable to an extremly high number of layers, namely 1001 [2].

C. DenseNet [3]

DenseNet is a minor modification to the ResNet architecture. Instead of using addition to partially preserve the identity of a layer input, it uses concatenation along the features axis. This yields a high input feature count for the subsequent layer and therefore increases the computational demand but has shown to be able to improve performance [3].

D. Loss function

As our training objective is a regression task, we use the mean squared error as our loss function as it yields smooth gradients compared to mean average error.

E. Training enhancements

To avoid daunting tasks such as manual learning rate tuning and manually determining the stopping point for the training, we use heuristics to automate them.

1) Early stopping: When training for too long, the model performance may worsen again. Therefore we stop the training if the performance has not been increasing for 4 epochs.

2) Reducing the learning rate: In later training epochs a too high learning rate may decrease model performance. Therefore we divide the learning rate by 2 if the model performance was not improving for 2 epochs.

V. Model tuning

To get the best model for this problem, the previously discussed model architectures are used as a starting point for finding a model with decent performance. The model consists of blocks with N convolution layers using 3×3 kernels with an activation function f after each convolution layer and a normalization layer n before each convolution. For each block a residual connection of type r was used. After each block a downsampling operation of type d was applied. Those blocks were chained until the feature image size was no larger than 2×2. The first convolution block starts with a depth of W and after each block the depth is increased by D. After those convolution blocks, 2 dense layers with the activation function f are used with the same depth as the last convolution block. After those dense layers a single dense layer without activation and 3 outputs is added to predict the movement vector. To get decent results, a hyperparameter optimizer was used to find good values for those parameters.

A. Block size

The block size N experimented with was limited to the range

$$N \in [1, 8] \tag{1}$$

B. Activation functions

For the activation function f, all of the following functions were included in the search space

1) Relu: The relu (rectified linear unit) has been successfully used throughout the computer vision community [4], [2], [3], [5] and is one of the most commonly used activation functions.

$$\text{Relu} = x \to \max(x, 0) \tag{2}$$

2) LeakyRelu: The leaky relu functions deals with one of the biggest problems of the relu activation function. So called dead neurons occur when activation can never pass the threshold of 0 to yield any activation and cannot participate in the learning process. By using a very flat slope on the negative side, the leaky relu function allows slow learning for those neurons while still acting as a differentiable nonlinearity.

$$\text{LeakyRelu} = x \to \max(x, x \cdot 0.01) \tag{3}$$

3) Elu: The elu (exponential linear unit) activation function is based on the idea of the leaky relu activation function but provides smoother gradient.

$$\text{Elu} = x \mapsto \begin{cases} x & \text{if } x > 0 \\ \exp(x) - 1 & \text{else} \end{cases} \tag{4}$$

4) Sigmoid: The sigmoid activation function is closely related to the functional behaviour of a neuron in a human brain. The output value is limited and converges to 1 for high input values and -1 for low input values. This behaviour introduces very low gradients in the extreme regions, which may lead to very slow training.

$$\text{Sigmoid} = x \mapsto \frac{1}{1 + \exp(-x)} \tag{5}$$

C. Normalization

To achieve faster training, multiple normalization types have been proposed throughout the years [6], [7]. The more recent and reliable variant of layer normalization is called batch normalization. Batch normalization, as the name proposes, does a intra-batch normalization of activations. To get any results during inference time, the operation keeps a moving average of the normalization parameters during training time and uses those to normalize the activations during inference time. For our hyperparameter search, we tried using batch normalization or the identity function - therefore no normalization at all.

D. Residual connection

Residual connections may improve fitting performance by partially preserving the identity of a previous layer. As explained in section IV, there exist various types. We compared residual connections through addition and concatenation and no residual connection at all.

E. Downsampling

Reducing the size of convoluted images is important in order to get a small number of output neurons. By gradually reducing the size, the network is forced to further compress the gained information to yield a final prediction. The most common downsampling operations are maximum-pooling [4], average-pooling [4] and strided convolution [8]. We try all three of those.

VI. Results and Evaluation

A. Manual parameter search

Despite using a hyperparameter tuner to find a good model, we also did benchmarks on certain aspects of our model, to get an insight about the different options we used. The most signifficant part in model structure may be the type of residual connection used. To our surprise, the results in Table I show that no residual connections yield the best results in our case. One reason for this may be the nature of the optimization problem, which is a regression task. Residual connections are widely tested on classification tasks but may behave differently on regression tasks.

$N = 4, f =$ LeakyRelu, $n =$ BatchNorm, $d =$ conv, $W = 16, D = 16$	
Residual type	validation mean squared error
None	0.0825
Add	0.0856
Concat	0.0857

TABLE I
RESULTS OF THE RESIDUAL TYPE COMPARISON

$r =$ none, $N = 4, f =$ LeakyRelu, $n =$ BatchNorm, $W = 16, D = 16$	
Downsampling type	validation mean squared error
Strided convolution	0.0825
Maximum pooling	0.0757
Average pooling	0.0756

TABLE II
RESULTS OF THE DOWNSAMPLING TYPE COMPARISON

Another part we manually evaluated, was the type of downsampling layer to use. We compared all three ones that were tested and got the results in Table II, showing that pooling outperforms strided convolutions in our case. However, the performance between pooling types did not differ by a significant amount.

Furthermore, we also drew the comparison between the different activation functions we introduced. The data in Table III shows that the best performing function in this case is the Elu activation function. The next best activation is the LeakyRelu followed by the Relu activation. The small margin between them may indicate that there is not a big problem with dying neurons. The Sigmoid activation yields horrible performance, which may have been introduced by the vanishing gradient problem this function with a derivative close to 0 at the extremes has.

1) Hyperparmeter tuning results: After more than 100 trained networks, were some trials were aborted due to insufficient memory, the best model we had found with our hyperparameter search has the parameters listed in equation 6.

$$\begin{aligned} r &= \text{add} \\ N &= 8 \\ f &= \text{LeakyRelu} \\ n &= \text{BatchNorm} \\ d &= \text{avgpool} \\ W &= 11 \\ D &= 17 \end{aligned} \tag{6}$$

The model has an evaluation MSE of 0.0764, which compared with our other tests, was already the lowest result, showing that the hyperparameter search was successfull and gave a good result.

$r =$ none, $N = 4, n =$ BatchNorm, $d =$ conv, $W = 16, D = 16$	
Activation	validation mean squared error
LeakyRelu	0.0825
Elu	0.0807
Relu	0.0823
Sigmoid	1.1304

TABLE III
RESULTS OF THE ACTIVATION FUNCTION COMPARISON

B. Evaluation

To get an intuition for the scale of the validation mean squared error, we first take a look at the possible value range. For a set of model parameters Θ and a model f the perfect fit would be

$$MSE(f_\Theta(x_{\text{val}}), y_{\text{val}}) = 0$$

. A fit with even the most simple model would be to always predict the mean. This would yield

$$MSE(f_\Theta(x_{\text{val}}), y_{\text{val}}) = \text{Var}(y_{\text{val}}) = 1.126$$

. Therefore, any reasonable result should have a MSE lower than 1.126.

The results with an MSE of 0.0764 are therefore by a magnitude better than just using the mean. In combination with the data plotted in Fig. 1 we can see that this is already a very accurate estimation.

ACKNOWLEDGMENT

We thank the creators of BeamNG.research [1] for their excellent and easy to use framework for generating the data for our project. We would also like to thank the Visual Computing group at the University of Konstanz for providing computational ressources for this project.

REFERENCES

[1] BeamNG GmbH, "BeamNG.research." [Online]. Available: https://www.beamng.gmbh/research

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[3] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016. [Online]. Available: http://arxiv.org/abs/1608.06993

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[5] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, "Flownet 2.0: Evolution of optical flow estimation with deep networks," *CoRR*, vol. abs/1612.01925, 2016. [Online]. Available: http://arxiv.org/abs/1612.01925

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[7] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[8] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," in *ICLR (workshop track)*, 2015. [Online]. Available: http://lmb.informatik.uni-freiburg.de/Publications/2015/DB15a

[9] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," in *Proceedings of the 13th Scandinavian Conference on Image Analysis*, ser. SCIA'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 363–370.