Optical Flow with Convolutional Neural Networks for Vision-Based Guidance of UAS

Wilco Schoneveld

August 11, 2017

Abstract

Modern research in the area of unmanned aerial systems (UAS) is pushing the level of autonomy to new limits. Scenarios without GPS, such as indoor environments, are especially challenging for autonomous navigation. These days, most aerial vehicles are equipped with camera sensors which make vision-based guidance an appealing solution. Velocity state can be estimated from optical flow, which is typically implemented with feature based methods. These methods rely on detectable features and lack in robustness. Current advancements in deep learning and convolutional neural networks (CNNs) have led to many achievements in the area of computer vision. This report proposes two CNN architectures which can estimate optical flow from two input frames. The models are trained and evaluated on a custom generated dataset and are shown to outperform the Lucas-Kanade method. The networks are sized such that they can run in real time on a Parrot Bebop 2 quadrotor. No in-the-loop validation has been performed.

Keywords: optical flow, deep learning, convolutional neural network, UAV, CNN

Contents

Preface

This document is the result of a 3-month internship at the Royal Melbourne Institute of Technology for my Masters programme in Aerospace Engineering at the Delft University of Technology. For the specialization in Control & Simulation I did a research project in collaboration with the RMIT UAS Research Team and the TU Delft Micro Air Vehicle Laboratory.

It has been a fantastic opportunity to delve deep into the emerging world of machine learning. Many thanks to my supervisor Dr. Alex Fisher for sharing the same level of enthusiasm and giving me the freedom to explore my personal interests.

Wilco Schoneveld

Overview

This report gives details to the design process of two convolutional neural network (CNN) architectures that can estimate global optical flow. The aim of the project was to apply machine learning in a vision-based guidance control loop of an unmanned aerial vehicle. The network is sized in such a way that it can run in real-time on a Parrot Bebop 2 quadrotor.

It is expected that the reader has reasonable knowledge about the structure and workings of artificial neural networks. The CS231n course by Stanford University [5] is an excellent resource to get a grasp on the overwhelming amount of information that is associated with CNNs and deep learning.

The outline of this document is as follows. Chapter 1 gives an introduction to the background of the problem and a description of the method used. Chapter 2 gives a chronological rundown of the work performed in designing the models. The verification of this model is presented in Chapter 3. Chapter 4 states the results of the project and Chapter 5 lists recommendations for future work.

Chapter 1

Introduction

In this chapter the need for a reliable optical flow estimation method is discussed. A description of the methodology is given next, followed by a brief overview of implementation details.

1.1 Problem

The research and development of small unmanned aerial vehicles (UAVs) has led to significant breakthroughs in the last couple of years. Practical applications for small UAVs include parcel delivery, surveillance and reconnaissance, emergency response, humanitarian assistance and disaster relief, and environmental monitoring, among others. Tasks like these come with a number of complicated challenges, especially in urban environments; the terrain is complex and dynamic, the air can be turbulent on a small or large scale, possibility of radio interference, broken line-of-sight, and absence of a reliable GPS signal [3].

The deficiency of a position fix as typically provided by GPS makes it especially difficult for UAVs to navigate through these environments. Instead, other methods must be utilized to estimate position and velocity state. The availability of camera sensors makes a vision-based approach viable. Marker detection or snapshot comparison techniques can be used to localize the UAV. With a camera facing the ground plane it is possible to determine optical flow and estimate velocity from the result.

This report proposes an optical flow implementation based on artificial neural networks. The heart of the architecture is a tunnel of convolutional layers, transforming the raw pixel data into smaller volumes with each convolution and thus expressing the image in more abstract terms. The implementation can be run onboard without the use of external tools or systems in order to facilitate full autonomy.

The scope of this project is focused towards getting a convolutional neural network (CNN) to run on the Parrot Bebop 2, which is depicted in Figure 1.1. The drone has a dual-core ARM CPU Cortex A9 and a down-facing camera capable of producing images at 90 frames per second.

1.2 Methodology

Global optical flow can be calculated by means of correlation. Two images can be taken in a short time interval and the offset can be determined. A naive correlation algorithm is computationally expensive and does not allow for real-time evaluation.

Optical flow is traditionally implemented with a feature-based approach such as the Lucas-Kanade method. Unfortunately, these implementations typically have a lot of parameters to tune and very much rely on the existence of features in the camera footage. The algorithm performance and execution speed

Figure 1.1: Parrot Bebop 2

depend heavily on the amount of features present.

CNNs have proven to be successful in many computer vision areas, including classification and localization. Since a typical network architecture does not have any conditional statements it's execution time is near identical for each pass. Unfortunately, convolutional operations are computationally expensive and large-sized networks take days to train.

This report explores the existence of a smaller sized CNN which is capable of solving the optical flow problem. The design strategy is derived from the CS231n course provided by Stanford University [5]. The proposed architecture is trained with an infinitely generated dataset and sized such that it can run aboard the Bebop platform in real-time.

1.3 Implementation

The CNN architecture is implemented using Python 3.6 and TensorFlow 1.2.1 [1], an open-source software library for Machine Intelligence. The main benefit of using a library is that low-level routines such as back-propagation are readily available for a wide variety of operations and this allows for rapid evaluation of different model architectures. TensorFlow is chosen because it can run arbitrary graphs and allows for a data set generator instead of a collection of files.

All training is done on an Intel Core i5-7200U CPU and it takes about 10 hours for the network to converge. TensorFlow was compiled from source to include all processor optimizations. Training can also be performed on the GPU, but it does not result in a significant speedup. The data generator is written outside of TensorFlow and is always run on the CPU while training, essentially a bottleneck in a GPU-assisted environment.

[1] `https://www.tensorflow.org/`

Chapter 2

Design

The steps taken in designing the CNN architectures are presented in this chapter. First, a prototype is created based on an existing architecture. A discussion follows with details on how to run a deep neural network on the Bebop platform. The dataset generator is explained next, as a basis for optimizing the model. Varied changes are made to the network and a new architecture is presented. Finally, an effort is made to have the model predict its own accuracy.

2.1 Related Work

Deep neural networks for estimation of optical flow have been successfully applied before, most notable FlowNet [2], and others using a similar architecture [1]. FlowNet is a deep neural network architecture which allows estimation of dense optical flow fields, trained with supervised learning. The architecture uses 9 convolutional layers and a refinement to obtain a high resolution prediction, see Figure 2.1.

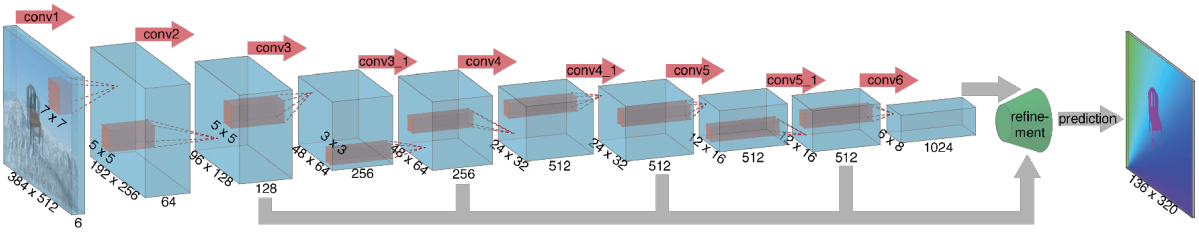


Figure 2.1: The FlowNetSimple convolutional neural network architecture [2]

The paper shows that the CNN can predict optical flow at competitive accuracy at frame rates of 5 to 10 fps on a NVIDIA GTX Titan GPU. Additionally, the paper proposes a synthetic Flying Chairs dataset, since existing ground truth datasets are not sufficiently large to train a CNN. The CNN is trained on the unrealistic data and is shown to still generalize very well to existing datasets such as Sintel and KITTI.

2.2 Prototype

An initial prototype of a CNN capable of predicting global optical flow was created with an architecture primarily based on FlowNet [2]. Since this network has been proven to have the capability of predicting localized optical flow on large images, a scaled down version should be quite successful in predicting global optical flow.

The input for the prototype CNN was chosen to be two images of 64x64 pixels. There is no need for

a high resolution input since only global optical flow is required. Using a small resolution allows for a higher update rate in the control loop. The images are chosen to be grayscale. It is assumed that optical flow is mainly derived from texture instead of color information, and having only 1 input channel per image reduces the amount of parameters in the network. Moreover, optical flow is traditionally defined with a brightness constancy constraint, which is well defined in grayscale.

The output of the architecture was chosen to be a 2-float vector prediction of global optical flow. In concrete words, the vector represents the pixel-offset between the two input images. It is not possible to directly output velocity since this would require a lot of additional inputs to account for scale and rotation (e.g. altitude, angular rate). The network error was defined to be the mean squared error (MSE) between the predicted flow and the ground truth.

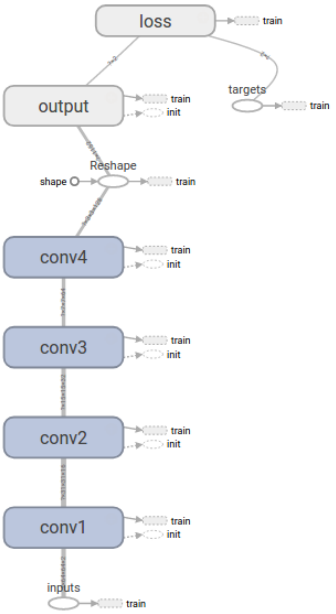


Figure 2.2: Architecture of the prototype CNN

What follows is a brief overview of the prototype architecture (Figure 2.2). The input layer is a 2-channel 64x64 volume, where each channel represents an input image. There are 4 convolutional layers each with a 3x3 kernel. Each has a stride of 2 except for the third layer which has a stride of 1. The filter size starts at 32 in the first convolution and doubles with each layer. Finally, the last convolution layer is flattened and a fully connected layer is added as output. The configuration was chosen as such after some preliminary tinkering indicated convergence.

The convolutional layers have a ReLU (rectified linear unit) activation function and the output layer has linear activation (i.e. no activation function). The optimizer algorithm is Adam [7] with a learning rate of 1e-4. Weights are initialized according to Xavier initialization [4]. No regularization is added to the graph.

The data used for training the CNN prototype is generated while training. The basic principle is to find two images which are offset by a random vector, in order to provide optical flow data in all directions and of varying magnitude. For this prototype, a single image of a garden area was used, which features sharp edges, grass and concrete. A 64x64 window is randomly placed on the image, and a second window is placed with a maximum x- and y-offset of 5 pixels. A single image was used to prevent underfitting of the network. See Figure 2.3a for a schematic overview of this process.

The optical flow ground truth f is a uniform random variable. The expected error for a zero prediction,

(a) Training source (b) Validation source

Figure 2.3: Schematic overview of dataset sampling, the blue arrow indicates the optical flow ground truth and the red rectangles are the sampled images

i.e. predicting no optical flow, is given in equation 2.1.

$$\mathbf{E}[(f-0)^2] = \mathbf{E}[f^2] = \int_{-5}^{5} \frac{1}{10} f^2 df = \frac{5^3}{15} = 8.333 \tag{2.1}$$

Training the prototype was done in a loop where each training step is performed with a batch of image pairs and associated flow vectors. The batch size was set to be 100 to allow the ability of generalization [6]. Since the dataset is randomly generated, and because of the use of batches, there is a lot of noise present in the loss curve. The reported loss was therefore filtered with a first-order infinite impulse response filter (IIR), also called the exponential moving average (EMA), with a weight coefficient of 0.8. See equation 2.2, where l_i is the loss at time step i, s the filtered and c the smoothing coefficient.

$$s_i = s_{i-1} \cdot c + l_i \cdot (1-c) \tag{2.2}$$

Figure 2.4 shows the loss curve when training the network for about 12k steps (≈1 hour). Validation was done on a separate image (Figure 2.3b) representing a floor carpet, where the data was generated with the same method as described above.

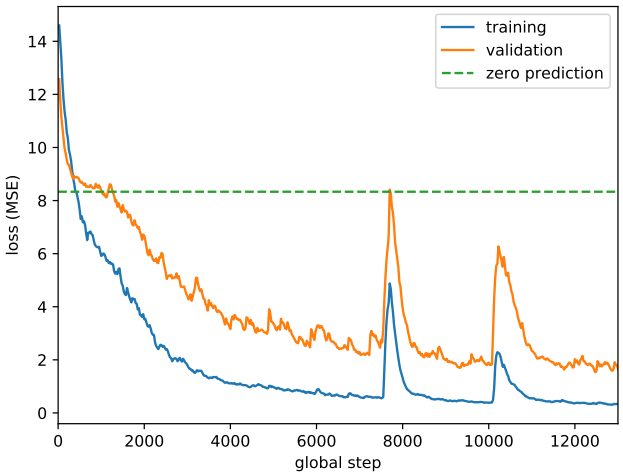


Figure 2.4: Training and validation curve of the prototype CNN

It is clear that the CNN prototype learns the ability to predict global optical flow given two images,

and there is not yet a visible plateau in the loss. The minimum reported MSE is 0.31 (0.54^2), meaning an error of less than 1 pixel in x and y direction. The validation error is also clearly decreasing, and while not nearly as good as the training loss (which is an indication of overfitting), it does surpass the expected zero prediction error and has no visible asymptote. The carpet image does not have any visually distinct features the garden has (in the traditional sense) and this is a good indication of generalization of the network.

There are two spikes in both the training and validation curve. These spikes could be an indication of a too high learning rate, the overshooting of a local optimum, the lack of regularization, or simply the result of numerical inaccuracies. Further investigation is required. In any case, running real-time inference with the trained prototype on a webcam seems to give good predictions, judged visually.

2.3 Running Onboard

In order for optical flow estimations to be useful in the control loop of a drone, such as the Bebop, a high update rate is required. No backpropagation is needed as it is assumed that the network can be adequately trained offboard. Unfortunately, the computations required to do a forward pass through a CNN scale exponentially with respect to the kernel size and amount of filters.

The kernel convolution operator is simple in concept but a naive implementation quickly turns into a multi-nested loop structure which can not be executed efficiently on an embedded platform. One benefit of such an implementation, however, is that it would not require additional libraries. Two open-source projects, simple_cnn[1] and keras2cpp[2] were compiled for the Bebop. Benchmarks indicate that a forward pass through a CNN of required size would take more than a second. Severe optimizations are needed to reduce this number significantly.

Tiny-dnn[3] is a dependency-free framework suitable for deep learning on embedded systems. It supports threading and vectorization for Intel processors and is considered to perform reasonably well. The framework is built on top of the C++14 standard which is not supported by the open-source toolchain released by Parrot. It uses a custom serialization format but does have the ability to import Caffe models.

OpenCV 3.x has a dnn module[4] which allows for forward pass computations. The module features a TensorFlow importer but at the time of writing this was not compatible with the latest release. A dnn_modern module can be found in the contrib repository which is essentially a wrapper to tiny-dnn.

It is possible to build the core of TensorFlow into a library with a corresponding C API. Besides it having optimized operations, the main benefit of using TensorFlow itself for inference is that any trained model can be directly uploaded to the platform. It also allows for execution of arbitrary graphs instead of the traditional layer-wise approach. Raspberry Pi and Android builds are available but they rely on system libraries not found in the Bebop linux environment. ARM cross-compilation is possible but complicated; 32-bit is not officially supported and the official recommendation is to compile natively on the embedded system itself [13]. An attempt was made to compile a TensorFlow library for the Bebop platform to no avail.

Recently made publicly available is the ARM Compute Library[5], a collection of low-level software functions optimized for ARM Cortex CPU and ARM Mali GPU architectures. It includes a full set of convolutional neural network building blocks implemented using NEON intrinsics.

A benchmark was performed to estimate the run time of the prototype CNN. The ARM Compute Library was compiled for maximum performance (no debug and assertions). Table 2.1 shows the execution time of different convolution layers averaged over 1000 passes. Each layer represents a layer of the

[1] https://github.com/can1357/simple_cnn/
[2] https://github.com/pplonski/keras2cpp/
[3] https://github.com/tiny-dnn/tiny-dnn
[4] https://github.com/opencv/opencv/tree/master/modules/dnn
[5] https://github.com/ARM-software/ComputeLibrary

Table 2.1: Benchmark of prototype convolution layers

input size	input channels	num filters	kernel size	stride	execution time (ms)
64x64	1	32	3x3	2x2	5
31x31	32	64	3x3	2x2	15
15x15	64	128	3x3	1x1	41
13x13	128	256	3x3	2x2	34

prototype CNN with uninitialized weights. Added up, the total execution time of these four layers is just under 100 milliseconds.

Several other benchmarks were done to estimate the scaling performance of convolutional layers computed by the ARM Compute Library. As expected, the amount of input channels and filters increases computational cost linearly. The execution time scales quadratically with the kernel size and inverse quadratically with the stride. The latter has an additional impact on the total execution time, since it shrinks the tensor volume for the next layer(s).

This project is not conclusive on how to implement a CNN on the Parrot Bebop 2 platform. Using the ARM Compute Library will most likely yield the best performance due to the instruction set used. However, there is no out-of-the-box solution to load a pre-trained graph and a custom runtime is required.

2.4 Dataset

Section 2.2 introduced a basic data generator, where image pairs are generated with a pixel offset sampled from a uniform distribution. This section talks about improvements on top of this principle, which then result in the dataset generator used in the rest of this project.

Six photographs are used as the basis for training instead of a single image, see Figure 2.5. The images have been captured in 12 MP but were downsized to 1 MP and saved in a lossless format to minimize any compression artifacts. Grayscale versions of these images are used in the dataset.

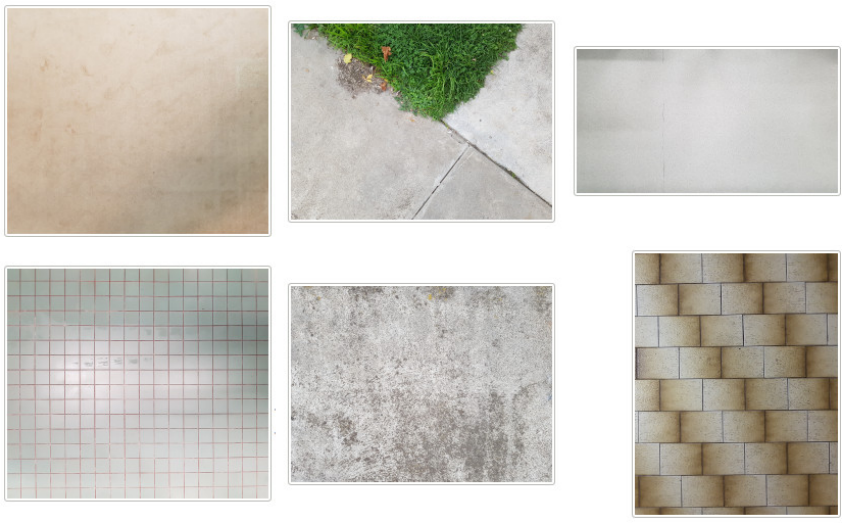


Figure 2.5: Source images used for generating training data

The images represent different ground planes with a varied amount of texture. A random source image is chosen as well as a random scale factor for each data point. A scaled square window is then placed randomly on the image and a second window is placed with a random offset. This offset is the ground truth for the optical flow. The views projected by these windows are then downscaled to the desired size. Finally, the generated image pair are augmented with Gaussian noise. Pseudo code for the data generator is shown in Figure 2.6.

```
def generate_image_pair():
  image = random.choice(images)
  scale = random.float(min_scale, max_scale)

  if sub_pixel_flow:
    offset = random.int(-max_flow*scale, max_flow*scale, size=2)
  else:
    offset = random.int(-max_flow, max_flow, size=2) * scale

  y0, x0 = random.int(margin, image.size - margin)
  y1, x1 = (y0, x0) + offset

  view0 = image[y0, x0, desired_size*scale, desired_size*scale]
  view1 = image[y1, x1, desired_size*scale, desired_size*scale]

  image0 = resize(view0, desired_size, desired_size, interpolation)
  image1 = resize(view1, desired_size, desired_size, interpolation)

  noisy0 = image0 + random.gaussian(0, noise_level)
  noisy1 = image1 + random.gaussian(0, noise_level)

  flow = offset / scale

  return (noisy0, noisy1, flow)
```

Figure 2.6: Pseudo code for the data generator routine

Uniform sampling is chosen because it ensures an even spread of available data points. The range of the distribution is also proportional to the horizontal velocity range on a UAV for which the algorithm can be used. Uniform sampling in both x and y does mean that the magnitude in the diagonals will be bigger. This is not an issue, the CNN will theoretically already have better prediction in the diagonals because the convolution kernels are square.

The main benefit of generating random data on the fly is that the data is inherently 'shuffled' and that the dataset can be made as big as desired. This can be disputed by pointing out the fact that the source of the generated data is a finite set of images, but even the textures themselves could be generated by, for example, a generative adversarial network (GAN). This is outside the scope of this project, however, but doing so could increase the compatibility of the data with real-life scenarios. More on this in Chapter 5.

The main characteristics that set image pairs apart are the position and scaling. Every image is generated at a different location in the source images and will therefore contain different texture and/or have a phase difference in texture. Scaling allows for a greater bandwidth of frequencies and is analogous to changing altitude in UAVs. Subsampling the views to downscale is fast but skipping pixels will result in aliasing. Therefore, bicubic downsampling is used to preserve image quality. Sub pixel flow is generated by choosing an (optical flow) offset that is not integer multiples of the scale factor.

Figure 2.7: Samples from the training data generator. The image pairs are overlapped according to the generated offset. The right-most image is zoomed to show sub pixel flow.

Gaussian noise has been added to the images with an amount that matches recorded footage from the Bebop bottom camera. Other commonly found data augmentation methods include horizontal reflections [8], rotation, and jitter [10], however due to increased complexity these have not been included in this

project. Figure 2.7 displays some samples generated for training.

A kid's car carpet vector illustration [9] is also included as a validation data set, see Figure 2.8. This image is flat and has a lot of sharp edges and corners, easy on feature detectors, which allows for a comparison between different optical flow estimation techniques. Additionally, it is used to determine the generalization capabilities of the trained CNN since the image is fundamentally different from the training set.

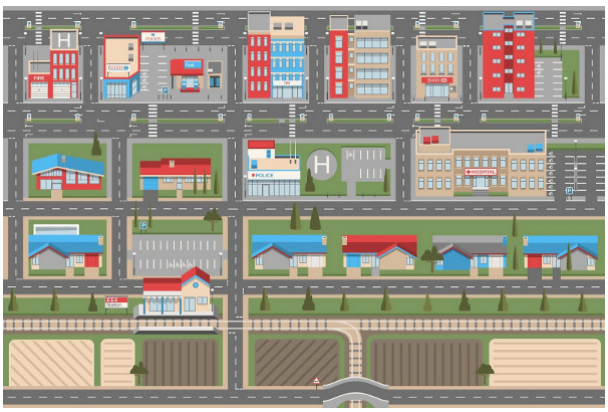


Figure 2.8: Image used for validation, separate from the training data

2.5 Tuning the CNN

In this section, different hyperparameters of the network architecture are optimized. Input preprocessing is discussed, followed by learning rates, resizing of the network and learning rate decay.

2.5.1 Input preprocessing

There are two common forms of data processing; mean subtraction and normalization. Not all scenes have the same overall brightness levels, so subtracting the average brightness of each image individually will center the visible texture around zero. Normalization is not typically required for images because pixels are already in the range of 0 to 255 [5]. However, in this purpose it can be beneficial to bring texture variance to the same level for each input. A downsize is that it could amplify the relative noise in case of low texture variance. Preprocessing was implemented as expressed in Equation 2.3, where I is an input image. Figure 2.9 shows the effect of preprocessing on the loss curve during training.

$$I_p = \frac{I - \text{mean}(I)}{\text{var}(I)} \tag{2.3}$$

From the figure it can be seen that input preprocessing has a positive effect on both the training and validation loss. It is clear that the loss slope starts off steeper and the resulting MSE is significantly lower. The training and validation error might settle to similar values when training for a couple of thousand steps more, but this could also lead to overfitting. In fact, the local minimum visible in the validation curve (without input preprocessing) in the early stages of training might already indicate overfitting of some sort, if only temporarily.

2.5.2 Learning rate

In order to determine the appropriate learning rate for training the network a coarse search is performed. Different learning rates in the range of 1e-5 to 1e-2 have been selected for training and the resulting

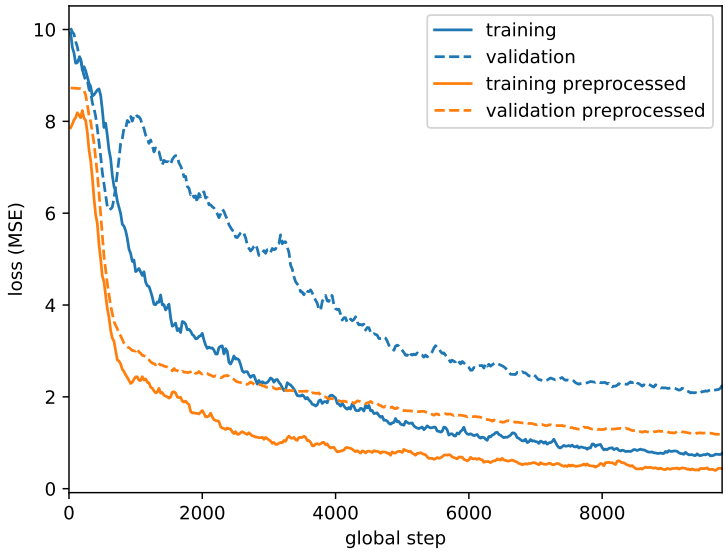
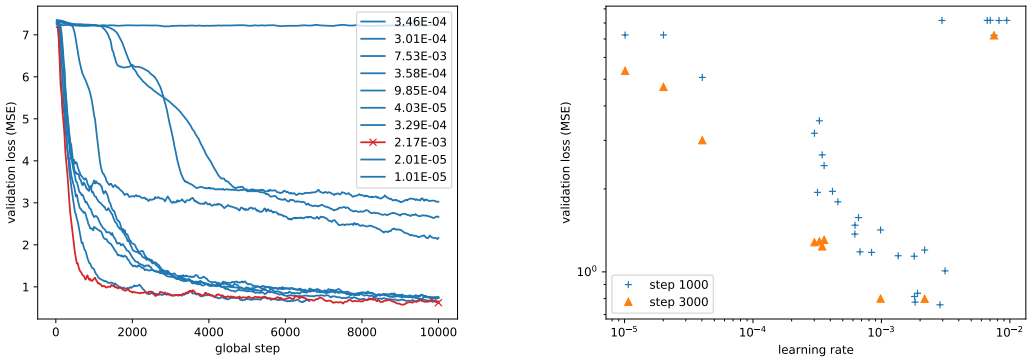


Figure 2.9: Training of a CNN with and without input preprocessing

validation loss is shown in Figure 2.10a. The results show that learning rates around 1e-3 result in decent loss curves. A more fine-grained search was then performed as presented in Figure 2.10b.



(a) (b)

Figure 2.10: Validation loss for different learning rates used in training the CNN

From the scatter plot it can be deduced that the optimal learning rate is indeed 1e-3. Higher learning rates could perform even better but come with a risk of not converging at all.

2.5.3 Sizing

In Section 2.3 an estimation of the inference time of the CNN prototype was presented. In order to run the network on the UAV in real-time it is necessary to drastically reduce the execution time, while maintaining (or possibly even improving) the performance and accuracy. The greater part of computations are spent in the convolution layers, so it is helpful to look at the receptive field of each neuron, see Figure 2.11. The visualization shows how the activation maps of each layer are connected to each other.

There is little overlap in the connections to the input layer, this can be enhanced by increasing the kernel size in the first layer from 3 to 4, which comes with a small time penalty. Another observation

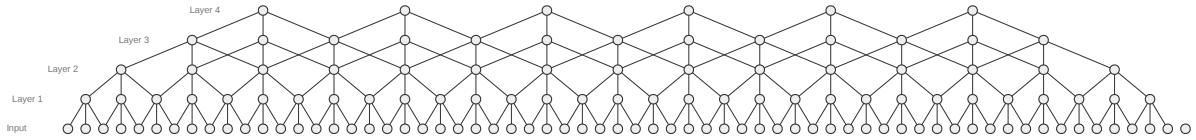


Figure 2.11: Connectivity diagram of prototype CNN

one could make is that the neurons in the outer layer can only reach one-third of the input. This can improved with a stride of 2 in the third layer, which comes with a significant drop in execution time. Additionally, the amount of filters for each convolutional layer has been reduced by half, i.e. from [32, 64, 128, 256] to [16, 32, 64, 128] respectively. The resulting connectivity diagram is presented in Figure 2.12.

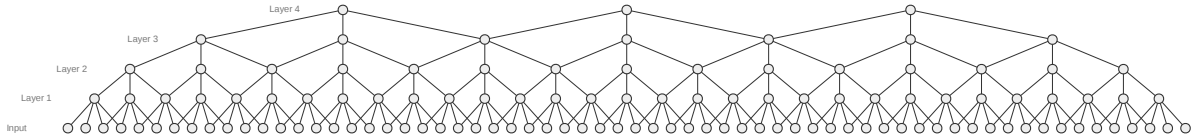


Figure 2.12: Improved connectivity of CNN

The sized-down network is trained and the results are compared, shown in Figure 2.13. It is clear that sizing down the network did not lead to a significant penalty in performance and accuracy. A benchmark with a CNN of this size on the Bebop is shown in Table 2.2. In total the forward pass takes just 13 ms, which allows for real time execution.

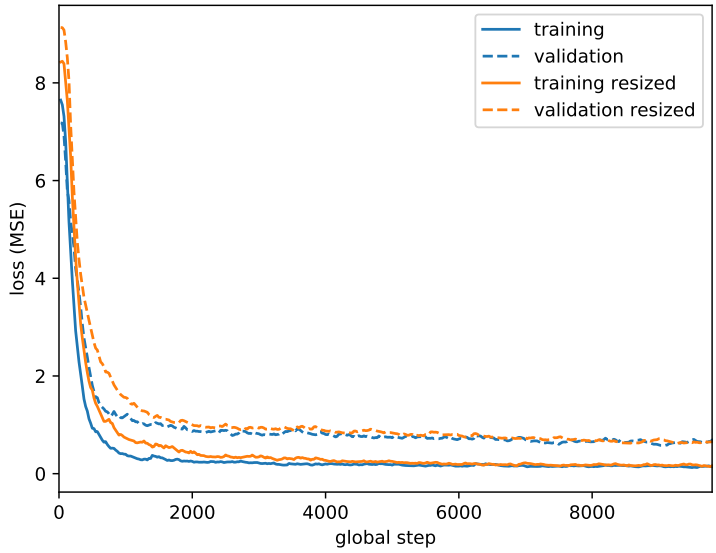


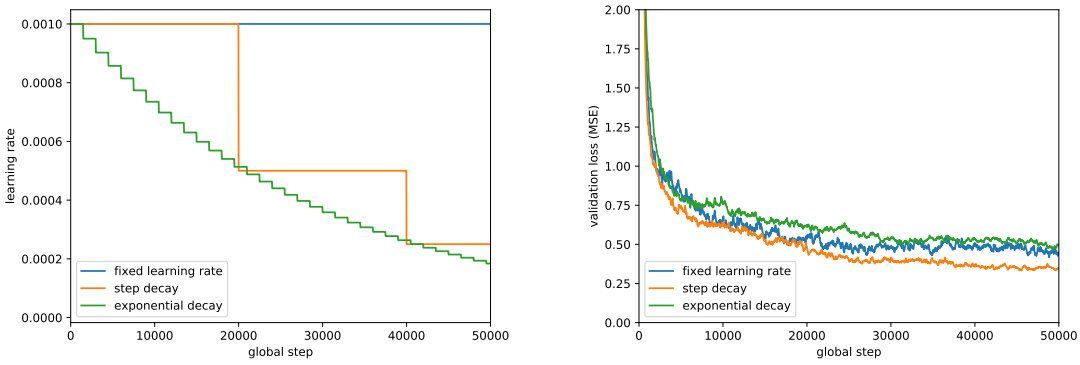
Figure 2.13: Training and validation curve of prototype CNN versus resized CNN

2.5.4 Learning decay

In section 2.5.2, the optimal learning rate was found to be 1e-3. Learning rate annealing was performed to slow down the weight updates over time and to allow the network to settle in a deeper optimum. Two types of step decay were implemented; a slow staircase decay with a rate of 0.5 at 20k steps and a faster exponential variation with a decay rate of 0.95 at 1.5k steps, see Figure 2.14a. The validation curve for both cases including the fixed learning rate case is shown in Figure 2.14b. The step decay shows a slight benefit in validation loss over the fixed learning rate.

Table 2.2: Benchmark of resized convolution layers

input size	input channels	num filters	kernel size	stride	execution time (ms)
64x64	1	16	4x4	2x2	3
31x31	16	32	3x3	2x2	4
15x15	32	64	3x3	2x2	3
7x7	64	128	3x3	2x2	3



(a) (b)

Figure 2.14: Validation loss for step and exponential learning rate decay

2.6 Splitting the CNN

One major downside of the CNN setup so far is that each forward pass requires the input of two images. This means that each capture from the camera would have to be processed twice, once as 'current' frame and once as a 'previous' frame. It would be computational more beneficial if each image is only (partially) processed once, which is the principle behind the split-CNN variant.

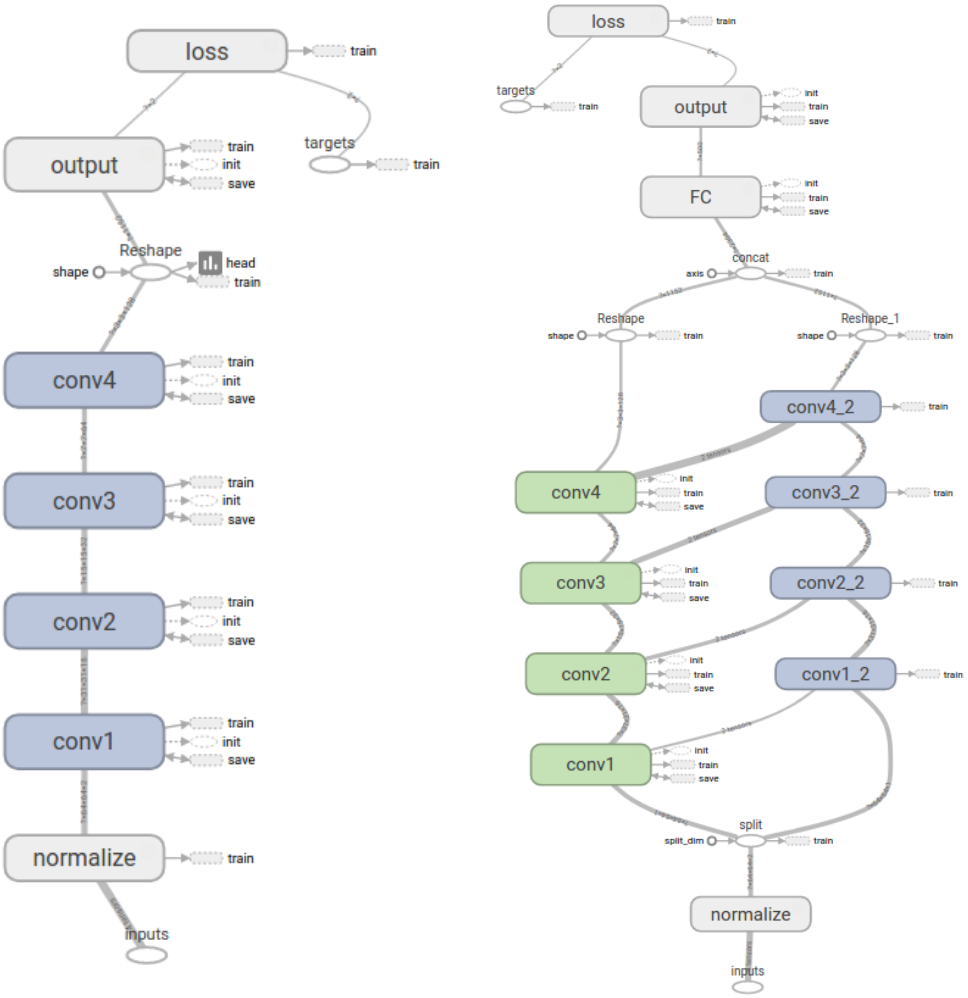
2.6.1 Architecture

The split-CNN features two tunnels of convolutional layers, where the batched images pairs are split and processed individually by two pipelines sharing the same weights. The output of each pipeline can then be concatenated and processed to produce the output. There is no opportunity for interaction and non-linearities between the convolutional activations as is possible in the default CNN, which makes the split-CNN more difficult to train.

One ideal to strive for is having the convolutional tunnels end with a vector having a length significantly smaller than the number of pixels. Different configurations were tested, built upon the already existing sized network. The first was adding another convolutional layer to reduce the output volume into a 1x1 column. The second was using an average pooling layer with the same purpose, similar to GoogLeNet [12]. Neither seems to converge, unfortunately. What does tends to converge is simply concatenating the output volumes together and adding a hidden layer before feeding the result into the prediction layer.

The resulting architecture is shown in Figure 2.15b, alongside the previous network (2.15a). At first glance, this might not seem to be more efficient. However, the inference setup would be different from the training architecture. Instead, a single convolutional tunnel can be used for each captured image and the output can be stored as an embedding in 'optical flow space'. The embeddings can then be fed in the remaining network, which represents just a simple multilayer perceptron (MLP).

In this way, each image would only need to be processed once by the convolutional filters, potentially saving computational cost. However, this can only be true if the architecture is downscaled and refined



(a) CNN (b) Split CNN

Figure 2.15: Network architectures displayed as graphs

to outweigh the small additional overhead imposed by the split-CNN setup.

2.6.2 Training

It turns out that the split-CNN architecture is sensitive to the initial learning rate. Moreover, even with a decent learning rate it takes a couple thousand steps before the network shows any signs of convergence. As can be seen in Figure 2.16, a learning rate of 2e-4 seems to converge well and takes into account sufficient margin.

Figure 2.17 displays the distribution of kernel weights for two convolution layers, in an attempt to understand why it takes a couple thousand steps for the network to start converging. There is no significant difference in the distribution between the initialized weights and the weights at step 2000. This suggests that the reason for delayed convergence is not attributable to the initialization of the convolution layers.

When comparing the split-CNN loss curves with those of the normal CNN (Figure 2.18), two observations stand out immediately. First, the training loss of the split-CNN seems to be significantly higher. This can be explained by the fact that the split-CNN only allows for comparison between the two images in the last dense layers, and this comparison is precisely what is required for predicting the optical flow. However, the validation curve follows very closely compared to that of the normal CNN. This is probably

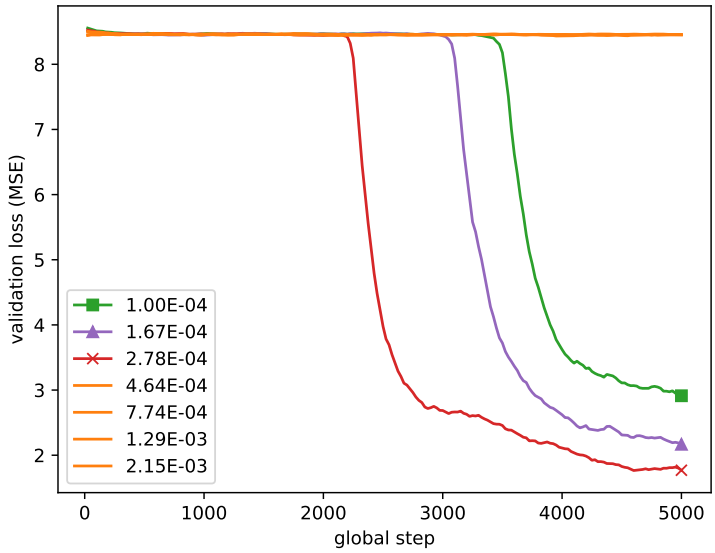
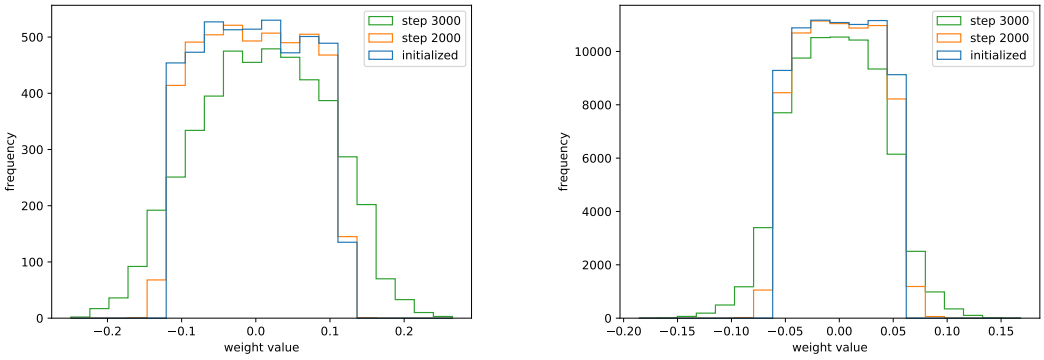


Figure 2.16: Validation loss for different learning rates used in training the split-CNN



(a) Convolution layer 2

(b) Convolution layer 4

Figure 2.17: Kernel weights in a histogram

an indication that the latter is actually overfitting the training data.

It can be concluded that the split-CNN architecture is generalizing better and, with the benefit of faster inference, is the preferable implementation. One downside however, is that overall it takes approximately twice the amount of time to train.

2.6.3 Pruning

The next step in the design of the split-CNN is to investigate whether the network has redundancy and if it can be pruned. A good place to start is looking at the weight distribution. Figure 2.20a displays an image map with the filters of the first convolution layer. It can be appreciated that not all filters are equal in strength, e.g. filter 0 and 14 have strong weights attached while filter 7 and 8 are barely visible. In convolution layer 2, depicted in Figure 2.20b, it can be seen that the second to last filter is almost inactive. Layer 3 (Figure 2.20c) shows even more low valued columns, which could indicate 'dead' filters. Additionally, it seems that input layer 30 is not of importance for this convolution layer.

Unfortunately, simply removing these filters from the convolution layers also requires removal of the

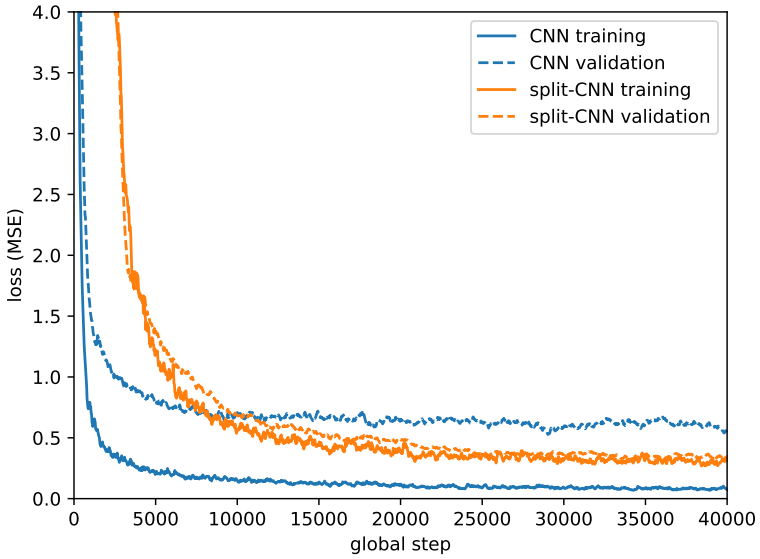


Figure 2.18: Training and validation loss of split-CNN compared with CNN

associated rows in the connected layers. There is however, a clear redundancy present in the architecture. An attempt was made to identify 'dead' weights and removing their influence by setting the weight value to 0. This was done for all weights with a value lower than 20 percent of the standard deviation. This amounts to a total of 15 percent of all weights and setting them to zero increases the network loss by less than 1 percent.

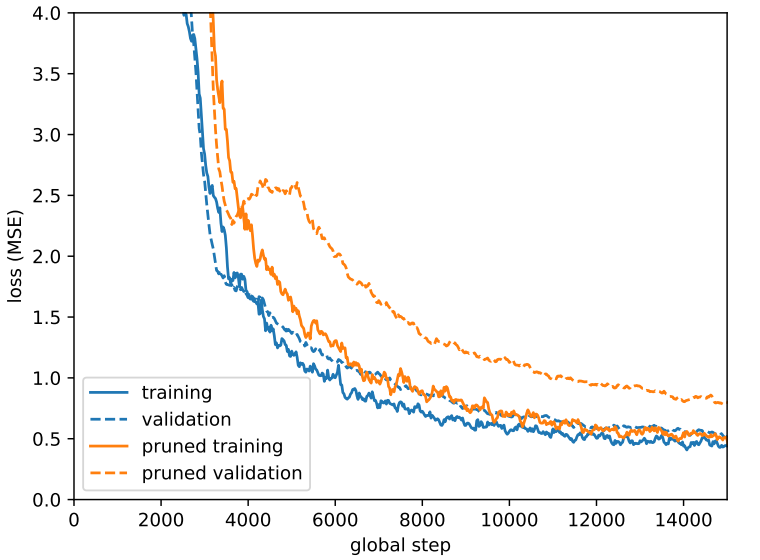
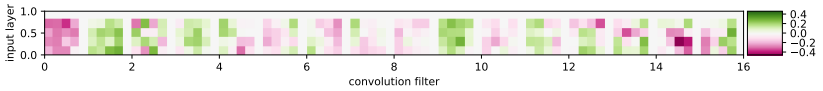
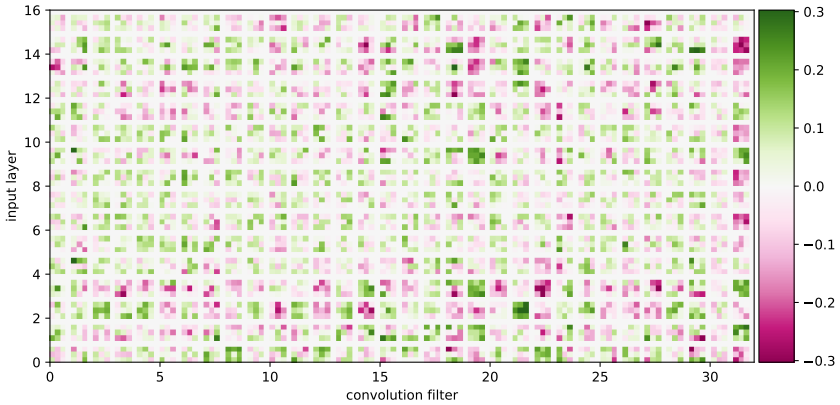


Figure 2.19: Training and validation loss of split-CNN with and without pruning

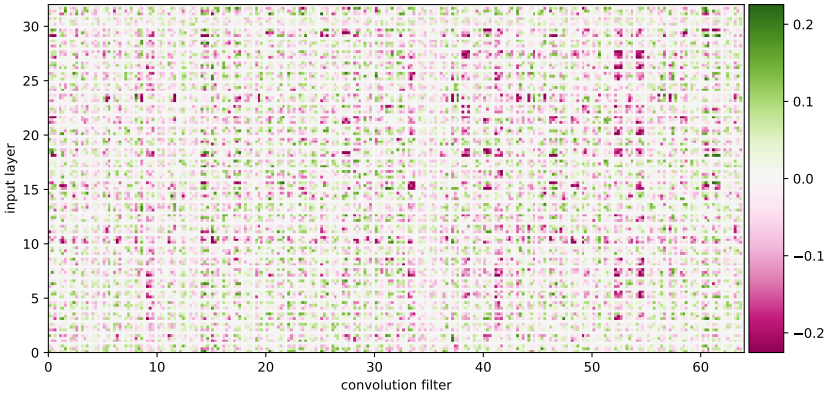
The direct relation between the amount of dead weights and the amount of filters was tested, where 15 percent of the filters were removed for all layers and the network was trained from scratch. The resulting loss curve is presented in Figure 2.19. Although the training loss remains intact, the validation loss is significantly increased. This suggests that the generalization ability of the network is lost.



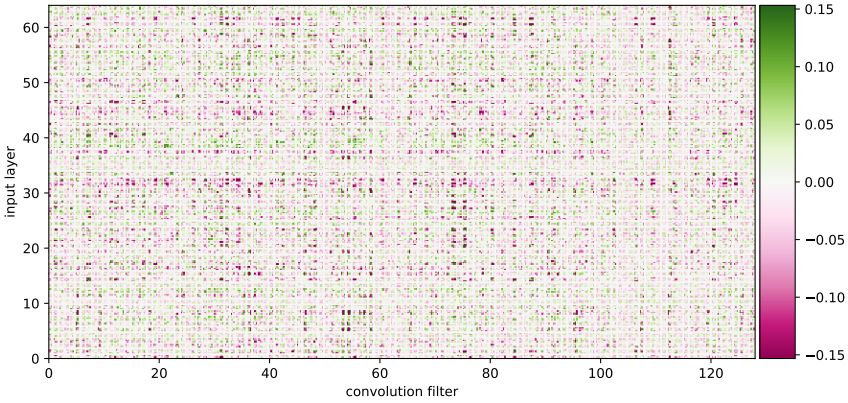
(a) Convolution layer 1



(b) Convolution layer 2



(c) Convolution layer 3



(d) Convolution layer 4

Figure 2.20: Kernel weights visualized in 2D

2.6.4 Sequences

The two CNN architectures discussed both accept two input images and output a global optical flow prediction. Theoretically speaking, the input images are position data and the output velocity. In other words, the CNN can be seen as a black-box which somehow finds the 'derivative' between the two images. Mathematically this is expressed as a backwards finite difference of order 1, see Equation 2.4.

$$f_p(t_i) = \frac{I(t_i) - I(t_{i-1})}{\Delta t} + \mathcal{O}(\Delta t) \tag{2.4}$$

The predicted flow at timestep i is expressed as f_p, calculated from the images I. Higher orders of accuracy can also be calculated, e.g. second order accuracy (Equation 2.5). One limitation is that the timestep Δt between two images is fixed, which is generally not always guaranteed when capturing frames on the Bebop camera.

$$f_p(t_i) = \frac{\frac{3}{2}I(t_i) - 2I(t_{i-1}) + \frac{1}{2}I(t_{i-2})}{\Delta t^2} + \mathcal{O}(\Delta t^2) \tag{2.5}$$

However, the point to be made here is that there is a mathematical basis behind arguing that having more than two input images could be a good thing, and could potentially increase the accuracy achieved by the CNN.

Having an input with more than two images would be possible with a architecture similar to the split-CNN. Another possibility is to attach a recurrent neural network (RNN) to the convolution head, and view the input as a sequence of image frames. An attempt was made to train a LSTM in this manner and even though convergence was achieved the increased complexity in both training and evaluation made it difficult to explore this any further.

2.7 Accuracy Estimation

Humans are especially good in recognizing optical flow, but even for us there are situations where we could not determine exactly how the ground plane is moving. Based on the characteristics of the image, the optical flow can be ambiguous in a single direction or all together. Figure 2.21 shows examples of images that possess these characteristics.

Figure 2.21: Images with ambiguity in optical flow

Humans can easily assess if a texture is suitable for optical flow based on the presence of edges, blobs, symmetry and other visual indicators. The CNN could perhaps learn to additionally estimate the accuracy of its optical flow prediction. This value could be of interest for implementations in the control loop, where a lower accuracy means that less trust should be placed on the optical flow for state estimation, and vice versa.

An additional output was connected to the fully connected layer and another loss was defined. The updated architecture is presented in Figure 2.22. It is desirable to bound the output accuracy value between 0 and 1, to facilitate interpretation. Because of this, an exponential conversation was used for

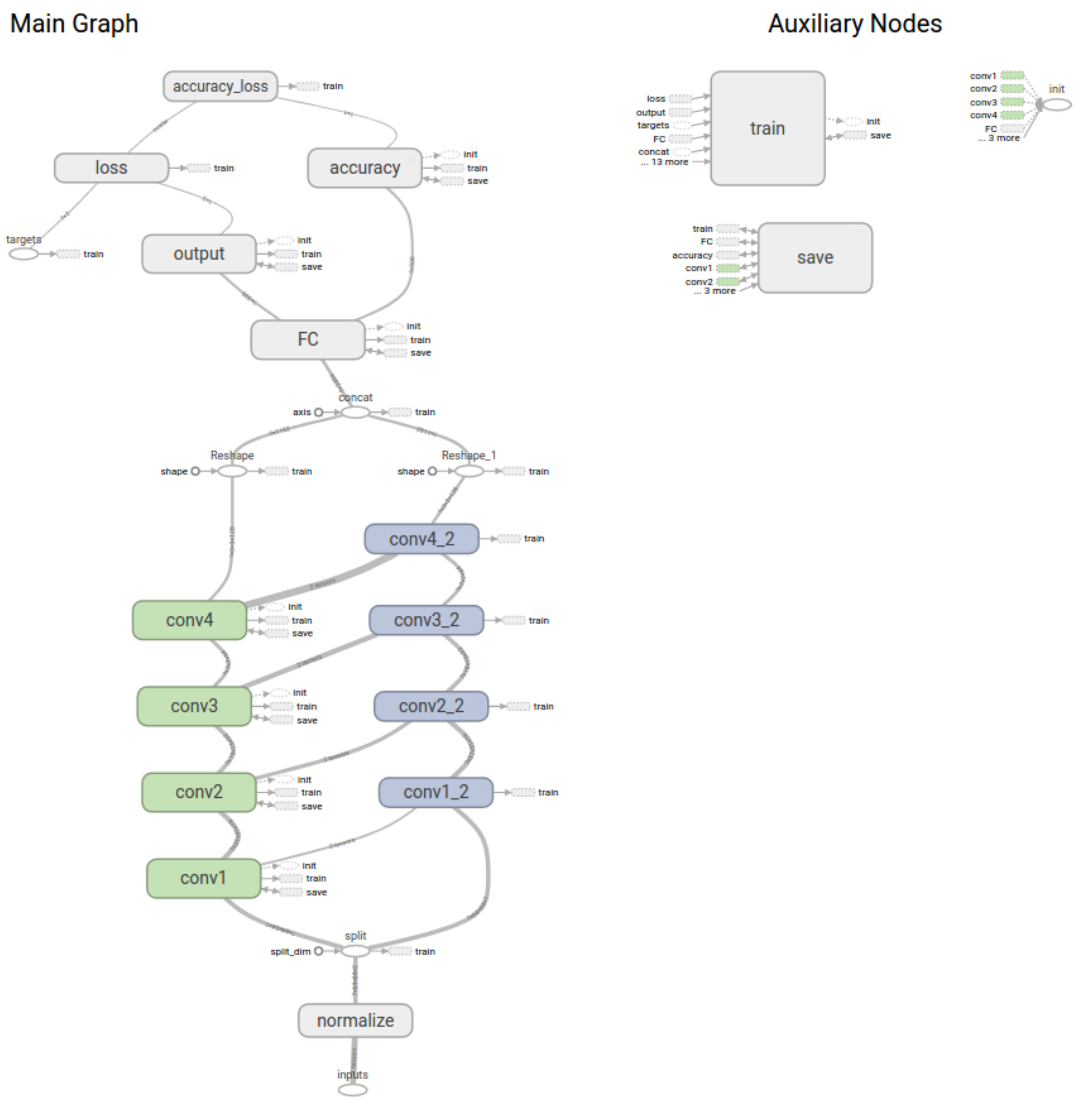


Figure 2.22: Full network architecture including accuracy prediction

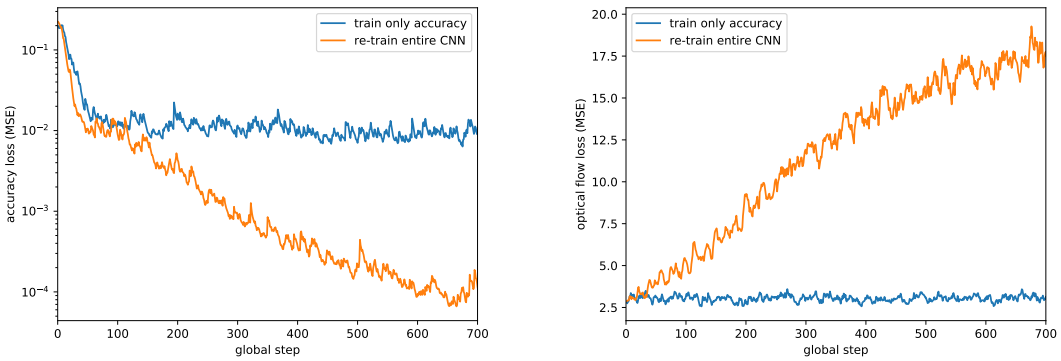
the loss and as activation function, see Equation 2.6. In the formula, Γ represents the estimated accuracy and MSE_f the optical flow loss.

$$\Gamma = e^{-\frac{1}{4}\text{MSE}_f} \tag{2.6}$$

The dataset used for training the CNN has a bias towards 'high' accuracy, since the architecture is optimized for predicting optical flow from these samples. Therefore, it was opted to include the 'low' accuracy images from Figure 2.21 into the dataset when used to train for accuracy prediction. Naturally, the optical flow prediction error is expected to go up when evaluated on this new dataset.

Training the accuracy loss can be complicated when combined with the existing optimization routine, as modifications made by either optimizer could increase the loss of the other. Two training runs were performed on a pre-trained CNN, the first only modifying the weights of the accuracy output layer and the second updating the entire graph, Figure 2.23 displays the losses.

From the plot it can be seen that by purely training the accuracy prediction a minimum mean squared error of 0.01 can be achieved, which converts to an accuracy prediction error of ± 0.1. A near-zero accuracy loss can be obtained by re-training the entire graph, but this completely wipes out any optical flow prediction capabilities, suggesting a constant accuracy of 0.



(a) (b)

Figure 2.23: Accuracy loss of the network in two training strategies

Chapter 3

Results

Chapter 2 explained in fair detail the process of designing a CNN capable of predicting optical flow. In order to evaluate the performance of the deep architecture it is helpful to compare the predictions with that of feature-based solutions. Additionally, a comparison is performed with the zero-prediction, i.e. predicting an optical flow of exactly 0 at all times.

FAST is used as the feature detector and the optical flow is estimated with the Lucas-Kanade method (LK). The type 9 implementation of FAST is used with a variable threshold between 5 and 200, aiming for 40 features per frame. The LK algorithm is applied without pyramids and a window size of 10x10. All other algorithm parameters are taken as the defaults in OpenCV 3.2.

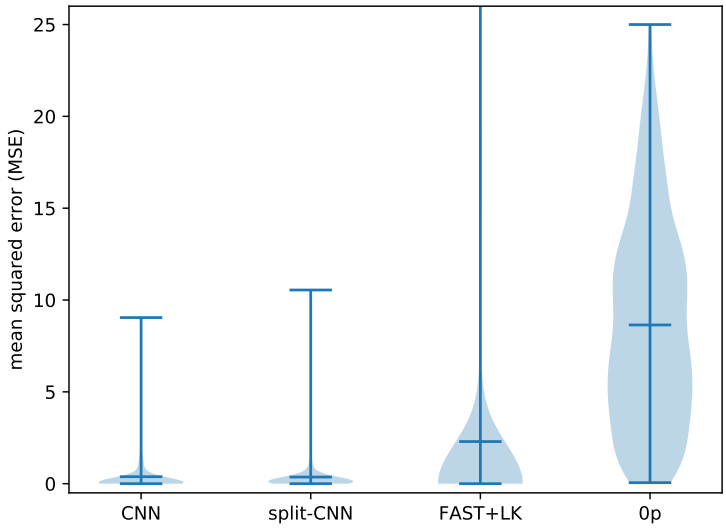


Figure 3.1: Loss distribution for the optical flow prediction with a simplified dataset

A first comparison is performed on a 1000-size batch of the validation dataset with a fixed scale of 3 and no augmented noise. The results are shown in Figure 3.1. As expected, the zero prediction (0p) shows a mean MSE of 8.6 and a maximum of 25. Both the CNN and split-CNN score very good with a mean MSE of 0.4. The Lucas-Kanade method scores noticeably worse with a mean loss of 2.3.

When performing the same comparison but instead using a maximum ground truth optical flow of 2, the FAST+LK algorithm seems to perform a lot better, as seen in Figure 3.2. The mean squared error is significantly reduced to 1e-4 and the spread is virtually zero. This corresponds with the small displacement assumption from which the algorithm is derived. The CNN and split-CNN performance is

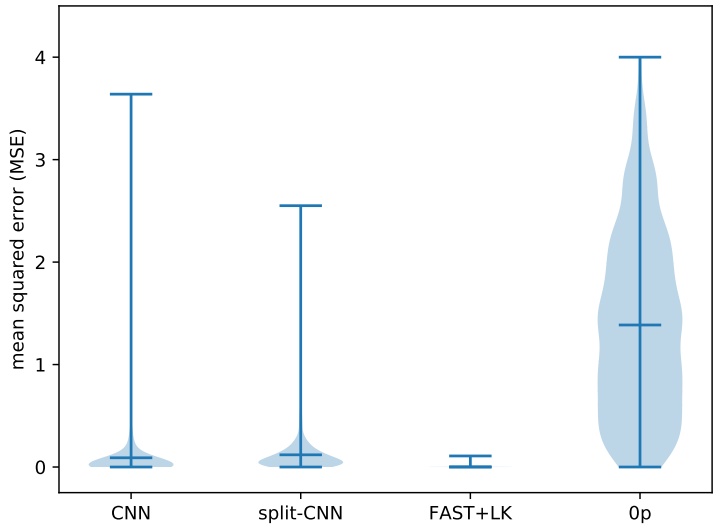


Figure 3.2: Loss distribution for the optical flow prediction with a smaller ground truth

also better, with the mean MSE around 0.1, which is expected because of the lower maximum optical flow.

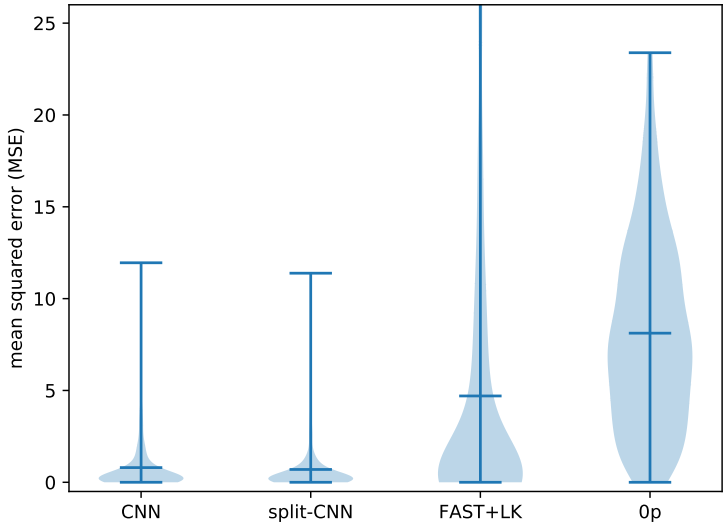


Figure 3.3: Loss distribution for the optical flow prediction with amplified noise

The CNN architectures are tested for robustness by tremendously increasing the noise level to 50, see Figure 3.3. The MSE distribution is nearly double for both, with a mean value of 0.8 and 0.7 for the CNN and split-CNN respectively. The same is found to be true for the FAST+LK predictions, having a mean MSE of 4.7.

Finally, another comparison is done on the training dataset. This includes all the images, scaling and noise, with a maximum optical flow of 5. Figure 3.4 display the loss distributions. The Lukas-Kanade algorithm has a mean MSE of 6.47 with a distribution just marginally better than the zero-prediction. With a mean loss of less than 0.1 the CNN performs significantly better than the split-CNN, the mean value of which is 0.3.

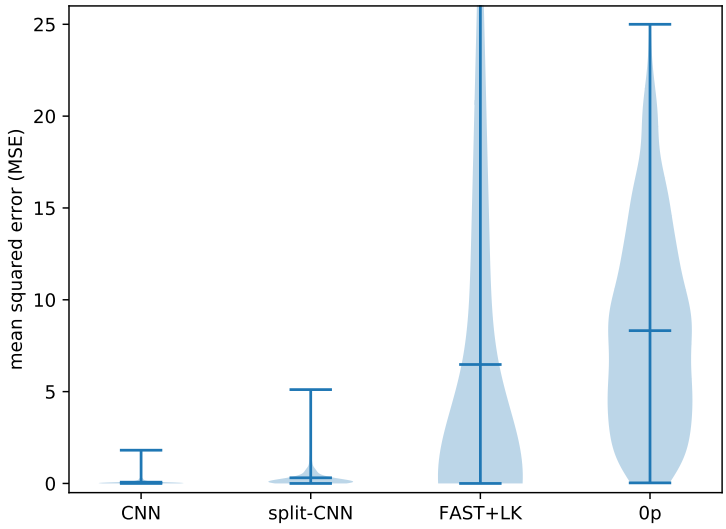


Figure 3.4: Loss distribution for the optical flow prediction with the training dataset

The added accuracy estimator is evaluated on the full dataset, i.e. it contains the training, validation and accuracy source images. A batch of 1000 samples was generated and the true accuracy is compared with the predicted accuracy, see Figure 3.5.

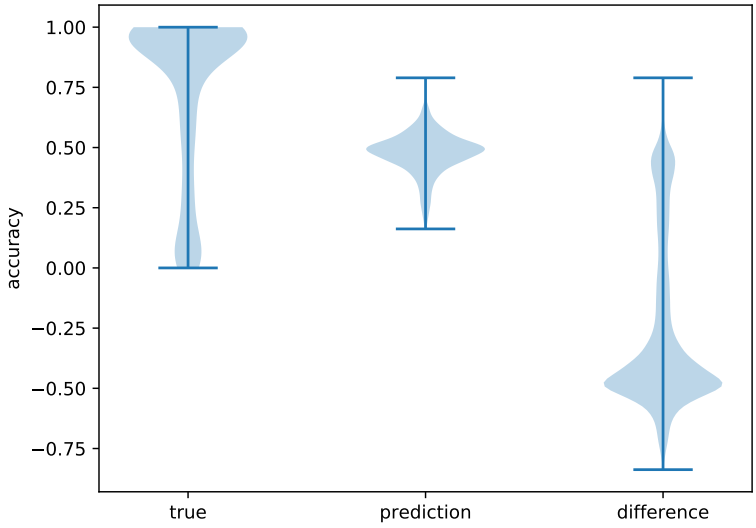


Figure 3.5: Accuracy distribution for the optical flow prediction with the full dataset

From the violin plot it can be seen that the split-CNN predicts the optical flow for *this* dataset with a wide range spread of accuracy, which is desired for proper evaluation of the accuracy estimator. The predicted accuracy however has a much smaller range and a huge bias towards values of 0.5, which can be compared to a lazy man predicting the expected accuracy.

The split-CNN does not seem to have the ability to correctly estimate its own optical flow prediction accuracy, which is especially apparent in the difference. There does not seem to be any correlation between true accuracy and predicted accuracy, as visible in Figure 3.6.

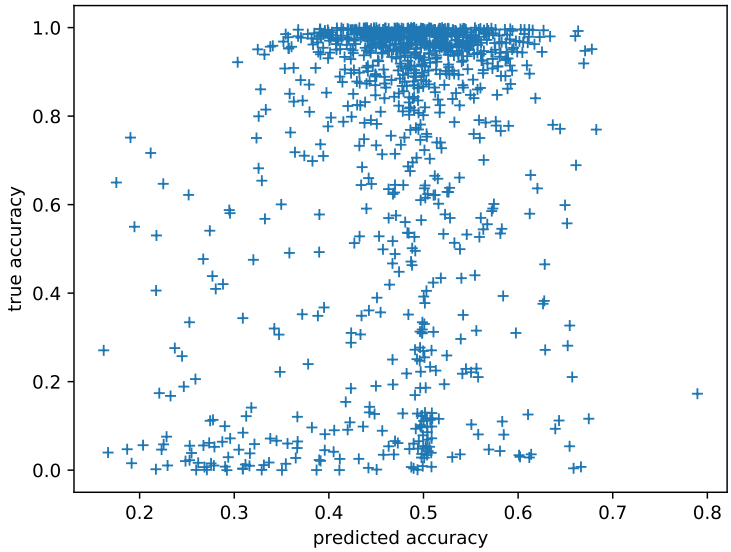


Figure 3.6: Scatter plot of the predicted accuracy versus the true accuracy

Chapter 4

Conclusions

Without GPS it is difficult for autonomous unmanned aerial vehicles to navigate. Vision-based guidance can make up for the lack of a 3D position fix. This report presented the design of a convolutional neural network (CNN) architecture capable of prediction optical flow.

A prototype was created based on the FlowNet architecture as a proof of concept. It was demonstrated that the architecture can learn from a simple dataset and has the capability to generalize to other input.

Convolutional layers are computationally expensive and it was shown that a naive implementation on an embedded system like the Parrot Bebop 2 can not be run in real-time. Depending on the usage of processor optimizations, such as NEON instructions provided by the ARM Compute Library, it is possible to run a CNN at desired frame rates.

Optical flow ground truth was provided by an 'infinite' dataset generated while training. The dataset contains a mixture between smooth surfaces and feature-friendly textures. These characteristics, together with artificial noise, revealed where the Lucas Kanade method fails and a CNN based implementation can still manage to get a decent prediction.

It was shown that straight-forward preprocessing of the data is useful and significantly improves the behavior of the loss curves during training. Additionally, a 50 percent reduction in size of the prototype architecture did not lead to any validation loss.

A second CNN was designed which can be used to further optimize the implementation onboard, by splitting the architecture and removing the need for re-processing frames. The split model is less predictable in training, but the network does seem to generalize better.

There is still plenty of opportunity for reducing the architecture size, while maintaining the same accuracy. It has been demonstrated that 15 percent of the weights can be eliminated with no significant loss in accuracy. This is a good indicator for opportunities to prune the network.

It is possible to use a recurrent network like an LSTM to process sequences of images, but there is no apparent benefit to do so. Moreover, it quickly adds up in complexity making it more difficult to get a decent implementation running on the Bebop.

There was no success in the attempt to estimate optical flow prediction accuracy. With the current architecture there does not seem to be enough information in the final fully connected layer to classify the textures accordingly.

Two deep CNN architectures have been presented which are capable of learning optical flow from a generated dataset. The CNN model is prone to overfitting the training data while the split-CNN shows more generalization capabilities. The architectures perform better than feature-based implementations in the case of large optical flow fields or when there are no clear features present.

Chapter 5

Recommendations

This report proposes two different convolutional neural network (CNN) architectures for predicting a global optical flow field from two input images. While the results are promising, there is still a lot of room for further research. What follows is an overview of topics that could be explored in order to close the gap between theory and implementation.

Preliminary testing with true camera footage shows that the CNN does not necessarily perform as good in real-life conditions, which comes with visual disturbances such as shadows, reflections and specular highlights. Humans can intuitively deal with this. A dataset could be created from simulation, which can additionally include depth parallax effects due to low-height obstacles. During training, this could give the deep neural network an opportunity to recognize and dismiss these distractions.

There is still plenty of optimization to be performed in the setup of the architecture. The number of convolutional layers is not proved to be optimal. The stride, kernel size, activation functions and amount of filters for each layer can be experimented with more. For the split-CNN, a hidden layer could be added to act as a low-dimensional embedding for the input images. An energy-aware pruning algorithm [14] can be used to try to reduce the model size or amount of computation.

One of the main bottlenecks in tweaking the model architecture is that the training takes a long time. Possible solutions to improve the training experience could be to extensively try out different weight initializations and evaluate the effects of regularization, dropout and potentially batch normalization. However, programming a full asynchronous GPU implementation of the model would be the most promising investment, since it would allow for an order of magnitude faster training.

The split-CNN introduced the possibility of processing a single input frame into an embedding which can then efficiently be compared to the next frame. Additionally, it was discussed that comparing more than two frames could lead to higher orders of accuracy, with backwards finite difference as a mathematical ground for this theory. By using a simulator to produce these images as well as a ground truth for optical flow, training a CNN like this could become viable.

Another suggestion is to additionally compute rotation and scale from the input, which could be converted into angular rates and vertical velocity of the UAS. Going one step further is to additionally feed inertial data into the network such that these quantities can be determined directly. This however does require a very efficient evaluation setup such that rapid prototyping and optimization of different architectures can be performed.

Having the ability to estimate its own prediction accuracy would be a very helpful addition to the CNN. This report showed that a straightforward extension does not find the information required to do so. Evolutionary algorithms such as NEAT [11] could be used to find a topology that is capable of estimating accuracy.

Finally, evaluation of the proposed CNN architectures in the loop in a simulator or with real-life test flights will determine its real efficacy.

Bibliography

[1] Aria Ahmadi and Ioannis Patras. Unsupervised convolutional neural networks for motion estimation. *CoRR*, 1601.06087, 2016.

[2] Alexey Dosovitskiy, Philipp Fischery, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. FlowNet: Learning optical flow with convolutional networks. *Proceedings of the IEEE International Conference on Computer Vision*, 11-18-Dece:2758–2766, 2016.

[3] Alex Fisher. Towards autonomous operation of small UAS in cluttered urban environments. Technical report, RMIT, 2017.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proc. AISTATS*, 9:249–256, 2010.

[5] Andrej Karpathy. CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/, 2017.

[6] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR*, 1609.04836, 9 2016.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, 1412.6980, 12 2014.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *NIPS*, pages 1106–1114, 2012.

[9] Daniel Long. An illustration of a simple city for use as a kid's play carpet. http://daniellongillustration.blogspot.com.au/2015/06/, 2015.

[10] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, 1409.1556, 9 2014.

[11] Kenneth O Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *The MIT Press Journals*, 10(2):99–127, 2002.

[12] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. *CoRR*, 1409.4842, 2014.

[13] Pete Warden. Mobile and Embedded TensorFlow. https://www.youtube.com/watch?v=0r9w3V923rk, 2017.

[14] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. *CoRR*, 1611.05128, 11 2016.

List of Figures

List of Tables