Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Improved Descriptor learning for Correspondence Problems

Azin Jahedi

Course of Study:	Computer Science
Examiner:	Prof. Dr.-Ing. Andrés Bruhn
Supervisor:	M. Sc. Daniel Maurer
Commenced:	May 7, 2018
Completed:	December 21, 2018

Abstract

Solving correspondence problems is a fundamental task in computer vision. In the past decades, many approaches tried to find the matches between images. One way to solve this task is to use feature-based methods, such as SIFT, SURF and DAISY. The mentioned methods are based on engineered features.

Learned features are another type of descriptors that are basically learned and computed via a convolutional neural network (CNN). This type of descriptors has gained a lot of attention in the past years. In this thesis, we design and train many CNN to find an architecture and a set of parameters, so that it computes suitable descriptors for the optical flow estimation task.

We implement a fast way of computing the descriptors from images, based on a strategy suggested by Bailer et al. After computing the non-dense set of matches by the Coarse-to-Fine PatchMatch (CPM) algorithm, we use the interpolation technique which is introduced and used in Edge-Preserving Interpolation of Correspondences for Optical Flow (Epicflow), to compute the dense flow field.

We use the approach of CPM, which is a popular method to estimate optical flow for large displacements. We embed our trained CNN model into CPM in such a way that the algorithm uses the learned descriptors to find the matches in a coarse-to-fine manner.

We use the recent benchmarks of KITTI 2015 and MPI-Sintel to train and evaluate our CNN. To this end, we compare the estimated optical flow to the ground truth optical flow provided in the mentioned benchmarks. By computing the Average End-Point Error (AEE) of the obtained optical flow, we thus have a measure to assess the learned descriptors and we can use it as a feedback to change the network so that it leads to more suitable descriptors. In this way, we were able to design and train a network that computes descriptors which perform better than DAISY and SIFT for the MPI-Sintel dataset.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

AEE Average End-Point Error. 22

CNN convolutional neural network. 17

CPM Coarse-to-Fine Patch Match. 17

Epicflow Edge-Preserving Interpolation of Correspondences for Optical Flow. 17

HOG Histogram of Oriented Gradients. 17

NCC Normalized Cross Correlation. 17

NN neural network. 19

NNF Nearest Neighbor Field. 18

SIFT Scale-Invariant Feature Transform. 17

SURF Speeded-Up Robust Features. 17

1 Introduction and Related Works

Identifying matches between two images is a key task in computer vision. It plays a fundamental role in many important tasks, such as optical flow estimation, stereo matching and scene flow estimation. Over the past decades, many approaches have been introduced to solve correspondence problems. They typically rely on a property that stays unchanged in both images for corresponding regions.

Early matching approaches assume brightness constancy for pixels (i.e. corresponding pixels have the same brightness or pixel value). The problem with only considering the pixel values is that the value of each pixel is not always a unique feature between corresponding pixels. The outcome of these early approaches are often too noisy. Some other approaches (e.g. block matching algorithm) assumed that brightness should be similar in corresponding regions and it considers neighborhoods instead of pixels. Of course pixel values of a region contain more unique information compared to the value of only one pixel. Block matching typically yields less noisy results. It is not realistic to assume brightness constancy because of illumination changes between two frames, occlusions, etc. Correlation techniques aim to indicate if two regions are similar. Normalized Cross Correlation (NCC) [Mus+85] is a variation of correlation techniques, which has the advantage of being invariant under global linear illumination changes.

Finding correspondences can also be done using feature-based methods. In feature-based methods, the points of interest (usually corners) of objects in both images are detected. Then, a feature-vector for each point of interest is computed via different methods, such as the Scale-Invariant Feature Transform (SIFT) [Low04]. The point of interest in the second image having the most similar feature-vector to the feature-vector of a reference point in the first image is regarded as the corresponding point of that reference point. Feature-based methods typically lead to sparse but accurate results.

Feature-based methods such as SIFT, Histogram of Oriented Gradients (HOG) [DT05], Speeded-Up Robust Features (SURF) [Bay+06] and DAISY [Tol+10] are all engineered methods, unlike learned descriptors, which gained a lot of attention in recent years. Learned descriptors are computed via convolutional neural networks. convolutional neural networks (CNNs) became very popular recently, as they could contribute to solve complicated problems, such as segmentation, object detection and recognition as well as optical flow estimation. According to the task that CNNs should solve, they must be trained with suitable training samples, a proper loss function and other factors that play a role in the training process. CNNs consist of trainable variables that are initialized at the beginning of the training process and are updated in each iteration of the training process. These trainable variables are updated in such a way that they lead to a minimization of the defined loss function.

In this thesis we want to estimate the optical flow using learned descriptors via the Coarse-to-Fine Patch Match (CPM) [Hu+16] algorithm, which finds the matches between two images in a coarse-to-fine manner. CPM uses feature-based methods to do the matching task in such a way that among some pixel candidates, the one with the most similar feature-vector is chosen as the match for a reference pixel. The CPM algorithm is explained in more detail in Chapter 2. To find the matches using our learned descriptors, a trained CNN should be embedded in the CPM method. As the matching result computed by CPM is not dense, we use the interpolation technique used in the Edge-Preserving Interpolation of Correspondences for Optical Flow (Epicflow) [Rev+15] algorithm to finally obtain dense flow fields.

We used patches of size (16×16) to train the networks, in our experiments. We used KITTI 2015 [MG15] and MPI-Sintel [But+12] datasets to extract the patches from, as they both contain color-images.

Another goal of this thesis is to find architectures and a set of parameters, by which a CNN can be designed and trained to compute suitable descriptors. These descriptors should yield an optical flow estimation which is similar to the ground truth optical flow. If the optical flow obtained by our learned descriptor is more accurate than the optical flow obtained by known descriptors, such as SIFT, we claim that our learned descriptors work reasonably.

Related Work

In the past decades, optical flow estimation has been a fundamental and challenging task in computer vision. Using variational approaches, which goes back to the method of Horn and Schunk [HS81], gained a lot of popularity and there is a large amount of literature that uses variational approaches to solve the optical flow task. Approaches such as PatchMatch [Bar+09] tried to estimate the Nearest Neighbor Field (NNF) and other improved approaches [HS12; KA16] led to improvements in estimation of the NNF. Unlike some NNF-based approaches that are often too noisy for estimating the optical flow, the mentioned method of CPM benefits from a global regularization by using a constrained random search strategy in a coarse-to-fine scheme.

In recent years, CNN-based approaches gained popularity. There are some approaches, such as FlowNet [Dos+15], FlowNet2 [Ilg+17] and PWC-Net [Sun+18], that compute the optical flow directly from image-pairs. There are also some publications on descriptors learning. Our approach is similar to a work from Bailer et al. [Bai+17] in many aspects: the way that the negative (non-matching) patches in the training samples are extracted, the thresholded hinge loss function used for some of our networks and multi-scale feature creation.

A clear difference of our approach to Bailer et al. is that the networks that we design for our experiments are mostly simpler networks having 7 convolutional layers with less than 50 filters per convolutional layer, while in the CNN use by Bailer et al. the number of filters per layer is between 64 and 512. Another difference is that we train all the networks, that we present in this thesis, with patches of size (16×16), but Bailer et al. used patches of size (56×56) for training.

Gadot et al. [GW16] and Schuster et al. [Sch+17] also train and use CNNs to compute descriptors to find the correspondences between two images. Gadot et al. use relatively large patches as well. They made patches of size (51×51) and (71×71). The loss function that they use is a modified version of DrLIM [Had+06] and they claim that using this loss function is the reason that their work was successful. In the work of Schuster et al. the training samples have the same size as in the mentioned approach of Gadot et al. They select the non-matching patches based on the nature of the true match. Unlike in our approach, in the mentioned approaches on descriptor learning, the training samples are gray-scale patches.

Thesis Organization

In the next chapter, we explain foundations of the topics that are required in this thesis, such as the basics of computer vision and the algorithms that we directly use in this thesis. These include the CPM and Epicflow algorithms. Then, we discuss the foundations of neural networks (NNs) and CNNs.

In Chapter 3, we explain how we generate training samples. We also discuss details about designing and training a CNN. After that, in Chapter 4, we explain how we make a portable model of trained CNNs and how to use them in the CPM framework to estimate the optical flow. In that chapter we also discuss a fast way of computing the descriptors, which has been suggested by Bailer et al. [Bai+18]. Then, in Chapter 5, we present our experiments and show their results and finally in Chapter 6 we conclude this thesis with a summary and an outlook.

2 Foundations

In this chapter, we discuss the basics of the topics in computer vision and machine learning that are required in this thesis. First, we explain the basics of computer vision. Then, we discuss two approaches of CPM and Epicflow that we directly use in this thesis. After that, we explain the basics of neural networks and convolutional neural networks.

2.1 Images

A discrete gray-scale image I is a function that maps a rectangular domain $\Omega = \{0, 1, \ldots, N-1\} \times \{0, 1, \ldots, M-1\}$, where N represents the number of columns and M denotes the number of rows, to a co-domain, as follows:

$$\{I(i, j) \mid (i, j) \in \Omega\} . \tag{2.1}$$

A color image with three color channels is a similar function, with domain $\Omega' = \{0, 1, \ldots, N-1\} \times \{0, 1, \ldots, M-1\} \times \{0, 1, 2\}$:

$$\{I(i, j, c) \mid (i, j, c) \in \Omega'\} . \tag{2.2}$$

The picture element at position (i, j) in a gray-scale image or the element at position (i, j, c) in a color image is called a pixel and if its value is encoded by one byte, the co-domain is given by $\{0, 1, \ldots, 255\}$.

2.2 Correspondence Problems

Given two sets of entities $A = \{a_1, a_2, \ldots\}$ and $B = \{b_1, b_2, \ldots\}$, matches between these sets are wanted. There are four main scenarios regarding matching, which are one-to-one, one-to-many, many-to-one and non-dense.

In the one-to-one scenario, every entity of A has a matching entity in B. In one-to-many, there is more than one entity in B that match an entity in A. In many-to-one, there is more than one entity in A that correspond to one entity in B. In the non-dense scenario, there is at least one entity in A that has no match in the entities of B. Correspondence problems can be defined in many areas. In the next section, we specifically discuss the correspondence problems in computer vision.

2.3 Correspondence Problems in Computer Vision

Given $A = (A(0,0), \dots, A(N-1, M-1))^\top$ and $B = (B(0,0), \dots, B(N-1, M-1))^\top$, where A and B are two discrete images and each entity represents a pixel value, the displacement field $df = (u(0,0)^\top, \dots, u(N-1, M-1)^\top)^\top$ is wanted between the matching pixels in images A and B, where $u(i, j) = (u_{i,j}, v_{i,j})^\top$.

In this thesis, to measure the quality of the estimated displacement field (df^e) given a ground truth displacement field (df^t), we measure the Average End-Point Error (AEE), which is:

$$\mathrm{AEE}(df^t, df^e) = \frac{1}{N \cdot M} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \sqrt{(u^t_{i,j} - u^e_{i,j})^2 + (v^t_{i,j} - v^e_{i,j})^2} \, . \tag{2.3}$$

One way to solve correspondence problems in computer vision is by using feature-based methods. In the next section, we describe the concept of feature-based methods.

2.3.1 Feature-Based Methods

In feature-based approaches, precise but typically sparse correspondences between two images are found. Firstly, points of interest (usually corners) of the objects in both images are detected. Then, a set of features for each point of interest is computed via methods such as SIFT, SURF and DAISY.

For a reference pixel in the first image, the pixel in the second image that has the most similar feature vector to the feature vector of the reference pixel is considered as the corresponding pixel for that reference pixel. The distance of two feature vectors is computed via:

$$dist(a, b) = \sum_{i=1}^{N} (a_i - b_i)^2 \, , \tag{2.4}$$

where a and b are feature vectors of size N. The mentioned methods compute features that are engineered. For example, in SIFT, for each point of interest, a unit feature vector of size 128 is made that represents the neighborhood around that point as unique as possible. The feature vectors are based on accumulated histogram of gradients. In this thesis, we want to compute a set of features via CNNs. In Section 2.7, we explain the basics of CNNs and in the next chapter we discuss how to build and train a CNN for our goal.

After training a network that computes the descriptors, we use the CPM algorithm to find the matches between two images, using the computed learned descriptors. In the next section, the CPM algorithm will be explained. After that, we discuss the interpolation technique used in the Epicflow method, that we use to get dense flow results from the non-dense matches that is resulted from the CPM method.

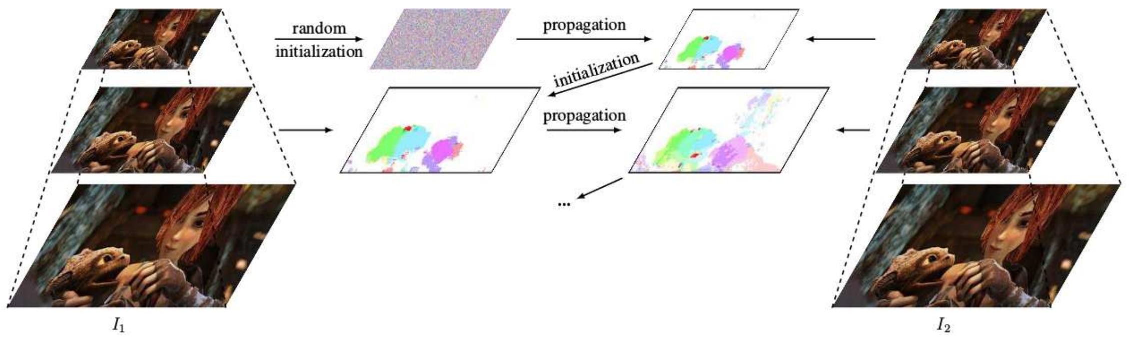


Figure 2.1: In the approach of CPM, matches are found first between the images in the highest pyramid level. Then, they are used as an initialization for the matching task of the images in lower pyramid levels. This figure is taken from Hu et al. [Hu+16].

2.4 Efficient Coarse-to-Fine Patch Match

The CPM method is an iterative algorithm that finds the correspondences between two images, in a coarse-to-fine scheme. The image pyramids of two images at the time t and $t+1$ are constructed and the correspondences between both images at each pyramid level are computed. Each level of a low-pass pyramid is made by low-pass filtering the image of the lower level, by applying a low-pass filter, such as a Gaussian filter, to reduce aliasing artifacts, and then the image should be down-sampled with the factor of 0.5. Using the mentioned down-sampling factor the spatial size of the image is reduced by half in each dimension. The lowest level of the pyramid consists of the original image.

As it is shown in Figure 2.1, matches are found first between the images in the highest pyramid level and used as an initialization for the matching task in its lower level. By this strategy, the matching results are propagated between adjacent pyramid levels from the top to bottom. After the initialization, in each level a random search method like in PatchMatch [Bar+09] is performed to further improve the matching results. After finding the correspondences between the raw images in the lowest pyramid level, a forward-backward check is done to remove the outliers.

In the following section, we describe how the matching process is done in every level and in Section 2.4.2, the coarse-to-fine structure as well as the propagation step between adjacent levels will be explained.

2.4.1 The Matching Strategy via Propagation and Random Search

In this section we explain the matching task, which is done via a propagation step and a random search step. Each iteration in CPM consists of these two steps. The goal of the matching task is to find the correspondences in a set of defined seeds, instead of all pixels. This choice helps speeding up the approach. Given two images I_1, I_2, for which the correspondences must be found, the set of seeds in I_1 is defined as $S = \{s_m | \, m \in \{1, 2, \ldots, n_s\}\}$ with positions $\{p(s_m) \in \Omega_1 | \, s_m \in S\}$, where

Ω_1 denotes the domain of the first image and n_s represents the number of seeds. The flow of each seed is calculated via:

$$f(s_m) = M(p(s_m)) - p(s_m) \in \mathbb{R}^2, \tag{2.5}$$

where $M(p(s_m))$ is the position of the matching seed in I_2 for the reference seed s_m in I_1. In a regular image grid with a d-pixel spacing, seeds are the cross points of the grid.

The flow of the seeds are calculated in scan-order (from the top to bottom and from left to right) in odd iterations and in reverse scan-order (from the bottom to top and from right to left) in even iterations. For each seed s_m, a set of adjacent neighbors that are already examined in the current iteration is defined and denoted by $\mathcal{N}_m$. Importantly, flow values are propagated from the neighboring seeds by the following criterion:

$$f(s_m) = \underset{f(s_i)}{\operatorname{argmin}}\, C(f(s_i)), \quad s_i \in \{s_m\} \cup \mathcal{N}_m, \tag{2.6}$$

where $C(f(s_i))$ represents the match cost of the patch centered at $p(s_m) + f(s_i)$ in I_2 being the match for the patch centered at $p(s_m)$ in I_1. In fact, to improve the flow of the seed s_m, the calculated flow values of its neighbors (e.g. the left, top-left, top and the top-right seeds) are also taken into account, as it is probable that the neighboring seeds belong to the same object in the image, therefore, they might have the same flow values. A seed can be examined in odd iterations via:

$$f(s_m) = \underset{f(s_i)}{\operatorname{argmin}}\, C(f(s_i)), \quad s_i \in \left\{s_m, s_{left}, s_{top\_left}, s_{top}, s_{top\_right}\right\}. \tag{2.7}$$

After the propagation of the flow values from the examined neighboring seeds, a random search is done to further improve the matching process. This is done by examining a list of flow candidates around the current best flow. With the current best flow $v_0 = f(s_m)$, the candidate offsets lie at an exponentially decreasing distance from v_0 by:

$$u_i = v_0 + w\alpha^i r_i, \tag{2.8}$$

where r_i is a random vector uniformly distributed in $[-1, 1] \times [-1, 1]$, the ratio α is set to 0.5 and w is the maximum search radius. For $i = \{0, 1, 2, \ldots\}$ the patches centered at $p(s_m) + u_i$ are examined until the current search radius $w\alpha^i$ is smaller than one pixel.

Obviously, the cost function in this approach highly affects the accuracy and robustness of the matching results. The original work of CPM uses the dense SIFT flow feature descriptor [Liu+11]. The match cost of each patch of size 8×8 is calculated by summing up the absolute differences over all the 128 dimensions of SIFT features. Our goal is to use learned descriptors to compute the cost function. In Chapter 4 we explain how to integrate a trained network to contribute to the computation of the cost function.

2.4.2 The Coarse-to-Fine Structure

CPM uses a coarse-to-fine scheme to improve the matching process. First a pyramid with k levels is constructed for both images I_1 and I_2 with down-sampling factor of $\eta = 0.5$. The ith image of the lth level is denoted by I_i^l, $i \in \{1, 2\}$, $l \in \{0, 1, \ldots, k-1\}$. Images I_1^0, I_2^0 are the raw images at the bottom level of the pyramid. For every seed in I_1^0, a match in I_2^0 is wanted. On each level of the pyramid, $\left\{s_m^l\right\}$ represents the set of seeds on the lth pyramid level at the positions $\left\{p(s_m^l)\right\}$. The position of the down-sampled version of the seeds on the lower level is computed by:

$$p(s_m^l) = \eta \cdot p(s_m^{l-1}), \quad l \geq 1, \; \forall s_m^l. \tag{2.9}$$

Note that from the bottom to top of the pyramid, the neighboring relation as well as the number of the seeds do not change and the positions of the seeds do not have sub-pixel precision, which means that the position of every seed on each level is rounded to the nearest integer. As a consequence, there will be some seeds at the same position in the coarser images at higher pyramid levels, if $d \cdot \eta^{k-1} < 1$.

After constructing the pyramids and generating the seeds, the flow values associated to the seeds on the coarsest level$\left\{s_m^{k-1}\right\}$ are set randomly. After that, the propagation step and the random search is done, as explained in the last section. The resulting flow values of the seeds from the coarsest level $\left\{f(s_m^{k-1})\right\}$ propagate to the seeds of the lower level $\left\{s_m^{k-2}\right\}$ as an initialization and similarly, in each level the resulting flow is used as an initialization to the seeds of the lower pyramid levels.

The matching process, which was explained in the last section, is done after the flow values of the seeds on level $l < k - 1$ are initialized by:

$$f(s_m^l) = \frac{1}{\eta} \cdot f(s_m^{l+1}), \quad \forall s_m^l. \tag{2.10}$$

In the matching step, the search radius in the coarsest level is set to the maximum image dimension and on the lower levels, the search radius is much smaller. By using a large search window on the coarsest levels, a rough global optimization is found. On the other hand, a small search radius around the initialized match, ensures the smoothness of the final flow, as it avoids finding a match far from the initial match.

As explained before, after the correspondences between the images in the lowest pyramid level are found, a forward-backward check is done. To do so, first, we look for the corresponding pixel in the second image for the reference pixel in the first image, and then we look for the match in the first image for the reference pixel in the second image. If the reference pixel in the first / second image is (sufficiently close to) the corresponding pixel for its matching pixel in the second / first image, this match consisting of these reference and corresponding pixels is a valid match and is not regarded as an outlier.

The output of the CPM algorithm is a set of correspondences. As the matches are not found for all pixels, CPM does not yield a dense flow field. To get a dense flow field, we need to benefit from interpolation techniques. In the next section an interpolation method which is used in this thesis will be explained.

2.5 Interpolation of a Non-Dense Set of Matches

After computing a non-dense set of correspondences by CPM, we use the interpolation technique which is used by Epicflow [Rev+15] to compute the pixels' flow values that are not estimated by CPM. The values of such pixels are computed by interpolating the existing flow values of other pixels. In this technique, a sparse-to-dense interpolation is performed while respecting the motion boundaries.

An important observation, which this method is based on, is that motion boundaries often appear near edges in images. Taking this observation into account, an edge-aware distance measure is used in this interpolation method.

A dense flow field df between two images I_1 and I_2 is wanted. This is done by interpolating a set of given correspondences $\mathcal{M} = \{(p_m, M(p_m))\}$, where $(p_m, M(p_m))$ represents a match between two pixels at positions p_m and $M(p_m)$ in I_1 and I_2, respectively.

Epicflow considers two approaches for interpolation: the Nadaraya-Watson (NW) estimation [Was13] and the Locally-Weighted affine estimation (LA) [HZ03]. We use the latter approach, which is explained next.

2.5.1 Locally-Weighted Affine estimation

This method does the interpolation task by fitting a local affine transformation. The dense correspondence field df is interpolated by estimating the locally-weighted affine transformation parameters for pixels position $p \in \Omega_1$ where interpolation is needed (i.e. the pixels whose matches are not found) via:

$$u_{LA}(p) = A_p \cdot p + t_p^\top, \tag{2.11}$$

where A_p and t_p are the estimated parameters for an affine transformation. Note that $u_{LA}(p)$ is the interpolated flow value of the pixel at position p and it is a member of the dense displacement field df. The values of A_p and t_p are computed by solving an overdetermined system of equations which is obtained by considering two equations for each match $(p_m, M(p_m)) \in \mathcal{M}$:

$$K_D(p_m, p)(A_p \cdot p_m + t_p^\top - M(p_m)) = 0, \tag{2.12}$$

where $K_D(p_m, p) = \exp(-a \cdot D(p_m, p))$ is a Gaussian-like kernel with parameter a and distance D.

Obviously, remote matches should not contribute to the interpolation as much as close matches, as close pixels often belong to the same object and pixels that belong to the same object usually move similarly. Likewise, remote matches probably belong to other objects, therefore their effect should be negligible. With this reasoning, the set of correspondences that are used in interpolation at the pixel at position p is limited to its k nearest neighbors, based on distance D.

2.5.2 Distance Function

The interpolation used in Epicflow uses an edge-aware distance. It is assumed that the edges in the image I_1 are the motion boundaries. Based on this assumption, a geodesic distance between two pixels at positions p and q is represented as the shortest distance with regard to the cost map C, which is computed via:

$$D_{\mathcal{G}(p,q)} = \inf_{\Gamma \in \mathcal{P}_{p,q}} \int_\Gamma C(p_s)\, \mathrm{d}p_s, \tag{2.13}$$

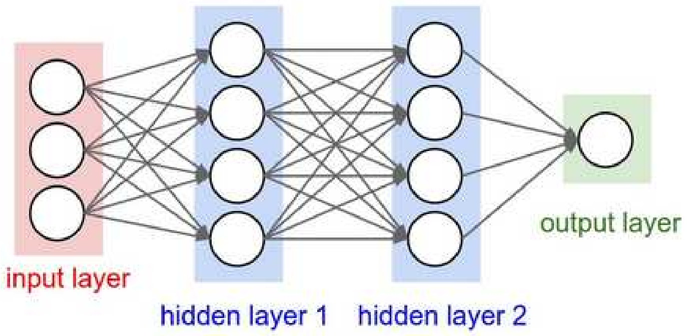


Figure 2.2: This figure shows a simple NN architecture with two hidden layers. Every neuron in each hidden layer and the output layer is connected to all the neurons of its previous layer. This figure is taken from the notes that accompany the Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition [KJ18].

where $\mathcal{P}_{p,q}$ shows the set of all possible paths between the pixels at positions p and q and $C(p_s)$ denotes the cost of crossing the pixel at position p_s. Essentially, the cost of crossing a pixel that belongs to the motion boundaries is higher than a pixel that does not lie on the motion boundaries. The cost map C corresponds to the motion boundaries, which are assumed to be the edges in image I_1. These edges are detected by the SED edge detector [DZ13], which is a real time state-of-the-art edge detector that predicts local edge masks, by applying structured forests. It is important to mention that the computational complexity of calculating D is very high, thus the value of this geodesic distance is approximated [Rev+15].

So far, we discussed the basics of computer vision topics that are required in this thesis, as well as the algorithms that are directly used. Next, we discuss the foundations of NNs and CNNs.

2.6 Neural Networks

Neural Networks (NNs) are popular tools in machine learning. In this section, we explain the basics of NNs.

As shown in Figure 2.2, there three types of layers, namely input, hidden and output layers, in NNs. The input is fed to the network by the input layer. NNs usually contain several hidden layers, and each layer (in all types) may contain several neurons. The output layer is the last layer in the network. An important feature in the architecture of NNs is that each neuron in each hidden layer and in the output layer is connected to all the neurons of its previous layer and the output of each neuron in one layer is the input of neurons in the next layer. Every neuron in each hidden layer and in the output layer does the following operation:

$$neuron\_output = \phi\left(\sum_{i=1}^{N} x_i w_i + bias\right), \tag{2.14}$$

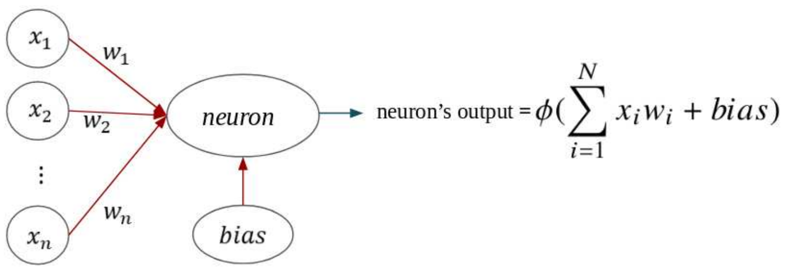


Figure 2.3: This figure shows the operation that each neuron in each hidden layer and in the output layer does. It multiplies every input by a weight, sums them up and adds a bias. Then a non-linearity is applied to the result of the mentioned operations. This output is used as an input for the neurons in the next layer.

where x_i is the ith inputs of the neuron, w_i is the weight that is multiplied by the input x_i and N is the number of inputs of the neuron. A $bias$ is a scalar defined for every neuron. ϕ is called an activation function or a non-linearity. This function should be non-linear and monotone increasing. The relation of the mentioned components is more clear in Figure 2.3.

Basically, the activation of each neuron is defined as the input of the activation function, which is $\sum_{i=1}^{N} x_i w_i + bias$.

Importantly, the weights and biases in a network are the trainable variables of that network. These variables are initialized at the beginning of the training process and their value is updated in each iteration of the training process.

Activation Functions

As explained before, activation functions used in NNs should be non-linear and monotone increasing. There are variety of non-linearities implemented in Tensorflow [Ten18], which is the software library that we used to implement our networks, in this thesis. We discuss the activation functions used in the networks that we built in our experiments. A set of experiments with networks that are built and trained using the following non-linearities is represented in Table 5.19.

1. **Sigmoid** (see Figure 2.4a):

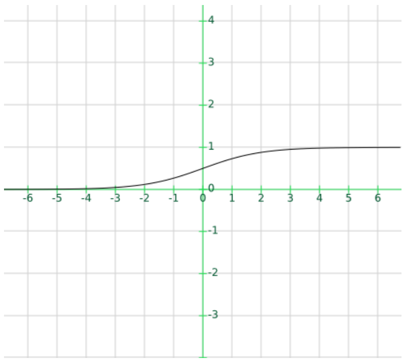
$$\text{sigmoid:}\mathbb{R} \to (0,1),\ x \mapsto \frac{1}{1+e^{-x}}. \tag{2.15}$$

2. **Hyperbolic tangent** (tanh) (see Figure 2.4b):

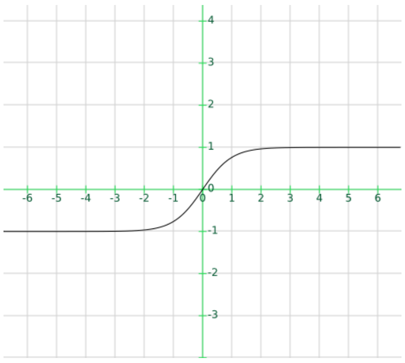
$$\text{tanh:}\mathbb{R} \to (-1,1),\ x \mapsto 2\text{sigmoid}(2x) - 1. \tag{2.16}$$

3. **Soft sign** (see Figure 2.4c):

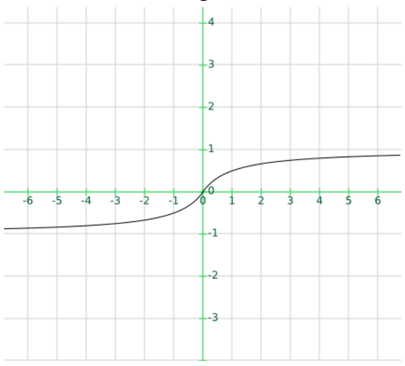
$$\text{soft sign:}\mathbb{R} \to (-1,1),\ x \mapsto \frac{x}{|x|+1}. \tag{2.17}$$



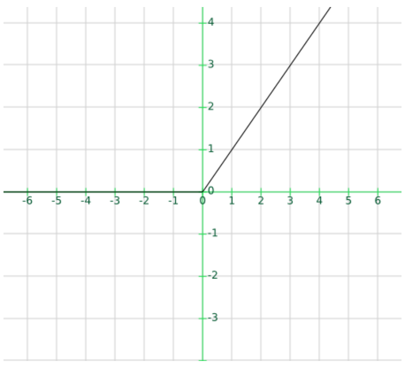
(a) Sigmoid



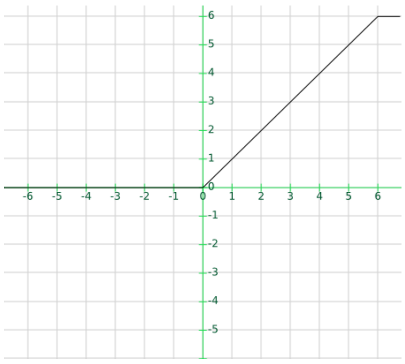
(b) tanh



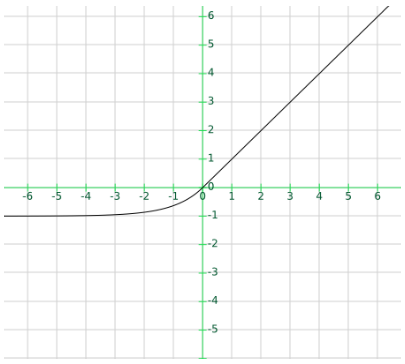
(c) soft sign



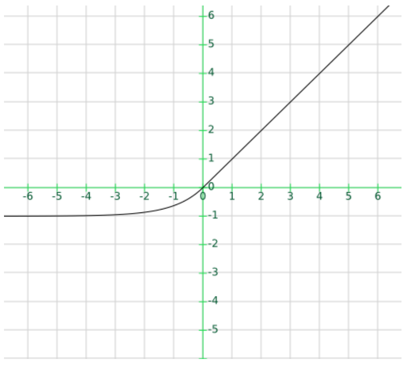
(d) relu



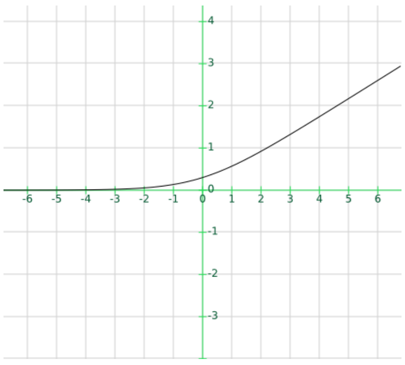
(e) relu6



(f) selu



(g) elu



(h) soft plus

Figure 2.4: Diagram of different activation functions. In Figure 2.4f, the value of λ and α in the selu function are set to 1.

4. **Rectified linear unit** (relu) (see Figure 2.4d):

$$\text{relu:}\mathbb{R} \rightarrow \mathbb{R}_0^+,\ x \mapsto max(0, x). \tag{2.18}$$

This activation function is very popular and in most of the networks we built for our experiments, this non-linearity is used.

5. **Rectified linear unit 6** (relu6) [KH10] (see Figure 2.4e):

$$\text{relu6:}\mathbb{R} \rightarrow [0, 6],\ x \mapsto min(max(0, x), 6). \tag{2.19}$$

6. **Scaled exponential linear unit** (selu) [Kla+17] (see Figure 2.4f):

$$selu(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda(\alpha e^x - \alpha) & \text{if } x \leq 0 \end{cases}, \tag{2.20}$$

where $x \in \mathbb{R}$ and parameters λ and α are derived in such a way that the mean and standard deviation of the activation of a neuron of one layer (x_i) is mapped to mean and standard deviation of the activation of a neuron the next layer $(\phi(\sum_{i=1}^{N} x_i w_i + bias))$. Note that unlike in other cases, the range of this function is not specified, as it depends on the value of the parameters λ and α.

7. **Exponential linear unit** (elu) [Cle+15] (see Figure 2.4g):

$$\text{elu:}\mathbb{R} \rightarrow (-1, +\infty),\ x \mapsto \begin{cases} x & \text{if } x \geq 0 \\ e^x - 1 & \text{if } x < 0 \end{cases}. \tag{2.21}$$

8. **Soft plus** (see Figure 2.4h):

$$\text{soft plus:}\mathbb{R} \rightarrow \mathbb{R}^+,\ x \mapsto \log(e^x + 1). \tag{2.22}$$

So far, we discussed the general structure of NNs. As mentioned before, the weights and biases are trainable variables. These trainable variables are learned by solving:

$$\min_{\theta} J(\theta) = \sum_{i=1}^{N} (t_i - f(x_i; \theta))^2 , \tag{2.23}$$

where N is the number of training samples, x_i is the ith training sample, t_i is the correct label of the ith sample and $f(x_i; \theta)$ is the output of the network parameterized by the set of trainable variables θ. In fact $f(x_i; \theta)$ is the label that the network assigned to that training sample. Minimization of this loss function $J(\theta)$, simply implies that the label decided by the network should be the same as the correct label of that sample. Importantly, in this case, we considered a classification problem.

Note that, the loss function $J(\theta)$ can be used for tasks other than classification tasks as well, but with a different interpretation. For example if the task that the network has to solve is estimating house prices, t_i is the correct price of the house in the ith training sample and $f(x_i; \theta)$ is the price that the network estimated given that sample. Of course the network should be trained in such a way that the correct and estimated prices become similar, so their difference should be compensated.

Essentially, the loss function should be defined based on the task that should be solved by the network. In the next chapter, we discuss suitable loss functions for descriptor learning. In the training process, the loss function should be minimized by updating the values of the trainable variables using an optimizer. In the next part, we describe the optimizers used in the training process of the networks we used in our experiments, shown in Chapter 5.

Optimization Methods

Explanations of this part are based on: An overview of gradient descent optimization algorithms by Sebastian Ruder [Rud16].

1. **Mini-batch gradient-descent**: This method is an iterative method to minimize the loss function $J(\theta)$. The parameters θ are updated in each step via:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} J(\theta_t)\,, \tag{2.24}$$

   where $\nabla_{\theta_t} J(\theta_t)$ is gradient of the loss function with regard to the parameters θ at time t, and η represents the learning rate, which is the size of each step to reach a minimum. Importantly, in mini-batch gradient-descent, in each iteration the loss function is computed given a batch of training samples. Obviously from Equation (2.23), to compute gradient of $J(\theta_t)$, gradient of $f(x;\theta_t)$ should be computed. As we explained before $f(x;\theta)$ is the output of the network given input x, parametrized by θ, and as it is a nested function, its gradient is computed by back-propagation.

2. **Momentum**: The gradient-descent algorithm struggles going towards the minimum when facing surface curves of the loss landscape that are steeper in one dimension than another. In these cases, gradient-descent makes hesitant progress towards local minima by oscillating across the slope of the curves. In such cases, the momentum algorithm can accelerate gradient-descent to find the right direction and decrease the oscillations. This improvement is done by adding a fraction γ of the update vector from the previous time step to the current vector:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_{\theta_t} J(\theta_t)\,, \\ \theta_{t+1} &= \theta_t - v_t\,, \end{aligned} \tag{2.25}$$

   where γ is the momentum parameter and its value is usually set to 0.9.

3. **Adaptive moment estimation** (Adam) [KB14]: This method computes adaptive step-size values for all parameters. It stores an exponentially decaying average of the previous squared gradients (v_t) and exponentially decaying average of the previous gradients (m_t):

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t\,, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2\,, \end{aligned} \tag{2.26}$$

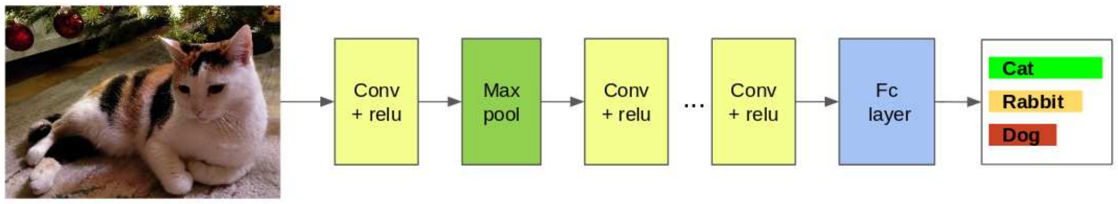


Figure 2.5: This figure shows a simple architecture of a CNN. The image of the cat is the input of the network. In this example, the task of the network is to specify which animal is shown in the input image. All three different types of layer is used in this architecture, which are convolutional layers (yellow rectangles), a pooling layer (the green rectangle) and a fully-connected layer (blue rectangle). Note that, the output of each layer is the input of the next layer. The specifications of each of these layers are further discussed in Section 2.7.1.

where $g_t = \nabla_{\theta_t} J(\theta_t)$ is the gradient of the loss function with regard to the parameters θ at time t. m_t and v_t are the first and second moments of the gradients at time t, respectively. β_1 and β_2 represent the decay rates. It is observed that these parameters are biased towards zero [KB14]. The unbiased first and second moments are estimated by:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}, \end{aligned} \tag{2.27}$$

where the default values of β_1 and β_2 are 0.9 and 0.999, respectively. The aforementioned bias-corrected moments are used to update the parameters via the Adam update rule, as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t, \tag{2.28}$$

where the default value of ϵ is set to 10^{-8}.

Up to this point, we explained the basics of NNs. In the next section, we discuss the basics of CNNs.

2.7 Convolutional Neural Networks

2.7.1 Structure

In CNNs, there are different types of layers, as shown in Figure 2.5, such as convolutional layers, pooling layers and fully connected layers. We discuss the features of each layer, separately.

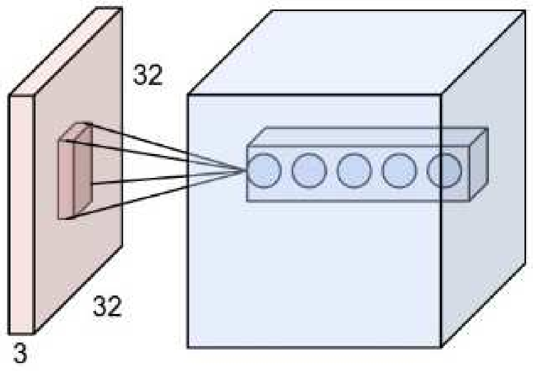


Figure 2.6: Each neuron in convolutional layers is spatially connected to a small region of its input volume, but with full connectivity in depth. This figure is taken from the notes that accompany the Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition [KJ18].

Convolutional Layers

These layers are similar to hidden layers in NNs. In both layers, there are some neurons and each neuron does an operation and has an output. There is an important difference between convolutional layers in CNNs and hidden layers in NNs. As explained before, each neuron in hidden layers is connected to all the neurons from its previous layer; however, this is not the case in convolutional layers. The input of each convolutional layer is a volume with some spatial size. Each neuron in CNNs is connected spatially to a small region of its input, but has full connectivity in depth. This is shown better in Figure 2.6. The mentioned small spatial window is called the receptive field. Each neuron in the convolutional layers is also called a filter. As it convolves with its input, adds a bias, performs a non-linearity (to the result of summation of the bias and the output of the convolution) and returns an activation map (see Equation (2.30)). The spatial size of the activation map is computed via:

$$\text{Output spatial size} = \frac{(\mathrm{sp} - \mathrm{rf} + 2\mathrm{p})}{\mathrm{s}} + 1\,, \tag{2.29}$$

where sp and rf are the spatial size of the input and the size of the receptive field, respectively. p represents the size of the zero padding which is done at the borders of the input. Zero padding is particularly useful in cases that no changes between the spatial size of the input volume and the activation map is wanted and this is achieved if p is set to $(\mathrm{rf} - 1)/2$. s denotes the stride parameter, which shows how many pixels the filter is moved while sliding over the input volume during convolution, in every step. In this thesis, for all convolution layers s is set to 1.

As explained before and shown in Figure 2.6, the neuron or filter has full connectivity with the input volume in depth. Every neuron can be interpreted as a filter with three dimensions ($\mathrm{rf} \times \mathrm{rf} \times c$) with $\mathrm{rf} \cdot \mathrm{rf} \cdot c$ weights, where c is the depth of the input volume. Each neuron, denoted by g, performs the following operation:

$$\text{Neuron's output} = \phi(g * I + bias)\,, \tag{2.30}$$

where $g * I$ is convolution of filter g and input volume I. $bias$ is a scalar as discussed in Section 2.6, and ϕ is an activation function. Note that the depth of the filter should be always equal to the depth of its input volume. The convolution operation between filter g and input volume I with depth c is done by:

$$(g * I)_{n,m} = \sum_{k=1}^{c} \sum_{i=-\mathrm{rp}/2}^{\mathrm{rp}/2} \sum_{j=-\mathrm{rp}/2}^{\mathrm{rp}/2} g_{i,j,k} I_{n-i,m-j,k} \ , \tag{2.31}$$

where rp is assumed to be an odd number.

So far, we discussed the operation done by a single neuron in a convolutional layer. The output of each convolutional layer is simply a volume of all activation maps that are resulted from each neuron. The spatial dimension of the output volume is explained before and its depth is equal to the number of neurons in that layer, as it is resulted by stacking up the individual output of each neuron in that layer.

Also in CNNs, the weights and biases are the trainable variables that are trained in the training process. As the distribution of the input of each convolutional layer changes in each training iteration (because the trainable variables of the previous layers change), the process of training CNNs becomes more complicated, as a more careful initialization and smaller learning step-size values are needed. This phenomenon is called *internal covariate shift*. It is suggested to use batch normalization [IS15] to normalize the activations of each layer to speed up the convergence and also to reduce the sensitivity of the results to initialization.

To be more specific, for a layer with d-dimensional input $x = (x^{(1)}, \ldots, x^{(d)})$, every dimension of each activation $x^{(k)}$ is normalized as:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}} \ , \tag{2.32}$$

where $\mathrm{E}[x^{(k)}]$ and $\mathrm{Var}[x^{(k)}]$ are the expectation and variance of the activation $x^{(k)}$, respectively, and their value is computed over the training samples in each batch. At this point, for each activation $x^{(k)}$, two extra parameters $\gamma^{(k)}$ and $\beta^{(k)}$ is introduced to scale and shift the normalized value as:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}. \tag{2.33}$$

Importantly, parameters $\gamma^{(k)}$ and $\beta^{(k)}$ are also trainable variables and their value is learned in the training process. Considering a batch of size m, as the each activation is normalized independently, we take only an activation $x^{(k)}$ into account, for clarity we omit k. Let $\mathcal{B} = \{x_1, \ldots, x_m\}$ be the m values of the activations of the batch $\mathcal{B}$ and let the linear transformation of the normalized activation be $y_1, \ldots, y_m$, the transform

$$\mathbf{bn}_{\gamma,\beta} : x_1, \ldots, x_m \rightarrow y_1, \ldots, y_m \tag{2.34}$$

is the batch normalization transform. Note that this transformation is done on the activations before the non-linearity is applied.

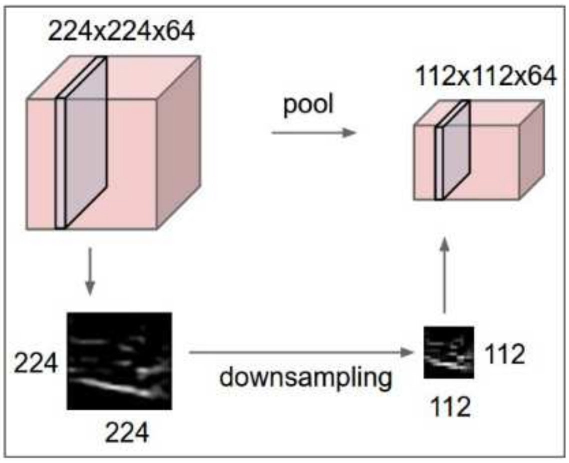


Figure 2.7: This figure shows the input and output of a pooling layer. The spatial size of the volume is reduced by half the size in each direction but the depth is preserved. At the bottom an activation map which is one slice of the input volume is visualized. It shows that the pooling operation is done in each depth slice independently. This figure is taken from the notes that accompany the Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition [KJ18].

Pooling Layers

Using pooling layers in the CNNs is common. Unlike convolutional layers and fully connected layers, this layer has no trainable variables. Pooling layer is used to reduce the spatial size of its input volume (see Figure 2.7). This helps to speed up the network by reducing the size of the volume and therefore decreasing the computations. There are a few types of pooling layers, such as average-pooling and max-pooling. In max-pooling, a window of size (2×2) is considered. This window moves over the whole volume spatially with the stride 2, and computes the maximum value of the elements in each window of size (2×2) (windows do not overlap). In average-pooling, the the average of the values in each window is computed. For each pooling type, the elements in that window are replaced by the element having the computed value in that window and the spatial size of the volume is decreased because a region of size (2×2) is reduced to (1×1).

Importantly, pooling is done for each depth slice independently, therefore it does not affect the depth of its input volume. Note that the stride parameter and the window size have the mentioned values in this thesis and of course, one could choose other values for them. Indeed, the spatial size of the output of this layer depends on the window size and the value of the stride parameter. Note that in the architecture shown in Figure 2.5, max-pooling layer is used.

Fully-connected Layers

Fully-connected layers are similar to hidden layers in NNs and unlike in convolutional layers, in fully-connected layers, each neuron is connected to all the neurons of its previous layer (the whole input volume). Using this layer in CNNs, like pooling layers, is optional. Fully-connected layers are usually used in the last layer of CNNs that are used to solve classification tasks. In that context, it computes the class scores (see Figure 2.5).

So far, we talked about the foundations of the topics used in this thesis. In the next chapter, we describe how to build and train CNNs.

3 Descriptor Learning via Convolutional Neural Networks

Estimating the optical flow is a fundamental task in Computer Vision. Many approaches have been proposed to calculate the optical flow between two images. One way of finding the correspondences is by using feature-based methods. One goal of this thesis is to compute descriptors via CNNs and use them to estimate the optical flow. In this chapter, we explain how to train a network in order to calculate suitable descriptors, as well as how to create the training samples.

To find the matches between two images I_1 and I_2, for each image-patch in I_1, the corresponding patch in I_2 should be found. We refer to the patches in I_1 as the reference patches. Evidently, the descriptors computed from image-patches must be similar (e.g. having small $L2$ distance) for corresponding patches, and dissimilar for non-matching patches. Hence, the descriptors computed by a trained CNN should have the following property:

$$||d_r - d_c|| < ||d_r - d_w||, \tag{3.1}$$

where d_r is the descriptor computed for the reference patch in I_1 and d_c and d_w represent the descriptors computed the patch in I_2 that corresponds to the reference patch and a patch that does not correspond to the reference patch, respectively. Here, we refer to non-matching patches as wrong patches.

The CNN should be trained in such a way that the above-mentioned property of its resulting descriptors is taken into account. Considering a CNN with an input layer and a few convolutional layers, we assume that the output of the network is a vector, resulted from flattening the output of the last convolutional layer. A simple choice of a loss function that is used for training a CNN, such that its resulting descriptors have the mentioned desired property is:

$$loss_{simple} = \frac{1}{N} \sum_{n=1}^{N} ||d_{r,n} - d_{c,n}|| - ||d_{r,n} - d_{w,n}||, \tag{3.2}$$

where N represents the number of training samples and $d_{r,n}$, $d_{c,n}$ and $d_{w,n}$ denote the output of the network (computed descriptors) for the nth reference, correct and wrong non-corresponding patch, respectively.

This loss simply encourages the output of the network to be similar (with a small $L2$ distance) given two corresponding patches and dissimilar for two non-matching patches, therefore it reflects the expected characteristic of the resulting descriptors in Equation (3.1).

There are other popular loss functions. In our experiments in the next chapter, we use two different loss functions which are the mentioned simple loss, and thresholded hinge loss, which is suggested by Bailer et al. [Bai+17]. To better describe the thresholded hinge loss, first we explain the hinge loss, which is defined as:

$$loss_h = \frac{1}{N}\sum_{n=1}^{N} \|d_{r,n} - d_{c,n}\| + \max(0, m - \|d_{r,n} - d_{w,n}\|)\ . \tag{3.3}$$

This loss function, decreases the distance of the descriptors of the matching patches and increases the distance of the descriptors of the non-matching patches until it becomes larger than m. The problem with the simple loss or the hinge loss is that they push the descriptors distance of the matching patches towards zero unlimitedly and training until we achieve a very small descriptor distance for patches that differ due to motion blur or rotation is costly. A solution is to add a threshold t to prevent the network to decrease the descriptor distance of the matching patches too much. The thresholded hinge loss is defined as:

$$loss_t = \frac{1}{N}\sum_{n=1}^{N} \max(0, \|d_{r,n} - d_{c,n}\| - t) + \max(0, m - (\|d_{r,n} - d_{w,n}\| - t))\ . \tag{3.4}$$

Note that there is no need to use the threshold to the second term, but it is claimed that it keeps the virtual decision boundary at $m/2$ [Bai+17].

Clearly, to train such a network, the value of the loss function should be computed and the trainable variables should update, in each training step in such a way that the loss value is minimized. Obviously, to compute the value of the above-mentioned simple loss, d_r, d_c and d_w should be computed by the network given the training samples, which should be a relatively large set of reference, corresponding and non-matching patches.

Essentially, d_r, d_c and d_w, for each training sample should be computed using a siamese network, which means that the descriptors of each patch in the training sample are resulted from three network branches that have the exact same variables. The siamese architecture will be discussed in more detail later in this chapter. In the next section, we explain how to prepare the training samples.

3.1 Creating the Training Samples

Like in a previous research [Jah17], we create the training samples from KITTI 2015 and MPI-Sintel datasets. We want to benefit from the color information of the patches for finding correspondences, therefore we did not use KITTI 2012, as it consists of gray-scale images. For both KITTI 2015 and MPI-Sintel we consider the training images, for which the ground truth optical flow information is available.

We divide the original training sets of both datasets into three sets: a training, an evaluation and a test set. The mentioned training, evaluation and test set are all subsets of the training folder of both datasets. In other words, we divided the training set (the images in the folder '`training`' of both datasets) into a training, evaluation and a test subset. In this thesis, we do not consider the original test set provided in the datasets, as it does not contain ground truth information. We need this ground truth optical flow values of the training subset, that we defined, to create the training samples. Also the ground truth values are needed for our evaluation and test set, so we can compare our evaluation and test results to have an assessment for the learned descriptors.

We use the evaluation set to assess the descriptors, while we do experiments. The optical flow values estimated for the evaluation set images are compared to the ground truth values. The result of this comparison is used as a feedback for us to change the network in such a way that it yields better results. The architecture and the parameters that play a role in the training process of the network should be modified in such a way that the modified trained network leads to better optical flow estimation. After finding a network which leads to reasonable results, finally we can test it on the images in the defined test set.

We considered the image pairs 0 – 159 from the KITTI 2015 dataset and all image pairs in MPI-Sintel dataset except the images in the folders `ambush_2`, `bamboo_2`, `sleeping_2`, and `temple_3` as the training set. Around 80% of the images from both datasets are in our training set. The image pairs 160 – 179 in the KITTI 2015 dataset and the image pairs in the folders `ambush_7` and `bamboo_2` are used for evaluation. The image pairs 180 – 199 in KITTI 2015 and folders `sleeping_2` and `temple_3` in the MPI-Sintel dataset are used for the final test. Each of these evaluation and test set contain around 10% of the images in both datasets. Note that in this thesis, we use the folder `final` of the MPI-Sintel dataset.

Next we will discuss how to extract the reference, corresponding and non-matching patches.

Reference Patches

For two images I_1 at time t and I_2 at time $t + 1$, we look for a corresponding patch in I_2 for each reference patch in I_1, therefore we extract the reference patches in I_1. We only consider a patch around pixels whose ground truth flow is valid (was measured). Of course, we prefer if the whole patch is inside I_1, thus we only take the pixels into account that their surrounding patch is completely inside I_1. Therefore, in x and y directions, we go through pixels from `half_patch_size` to $w-$ `half_patch_size` and from `half_patch_size` to $h-$ `half_patch_size` , respectively, where `half_patch_size`, w and h denote half of the patch-size, the width and the height of image I_1, respectively. Notably, we consider patches to have square shape. In the mentioned area, the patch around every pixel whose valid ground truth flow value exists can be extracted and the surrounding patch is entirely in I_1. In fact, not for all pixels a flow value exists. For instance, in KITTI 2015, there are pixels for which there is no valid flow value. Of course, we do not extract a patch around such pixels, because for each reference patch, we ought to extract its corresponding patch by knowing its valid flow value.

For both KITTI 2015 and MPI-Sintel datasets there are some occluded pixels that are visible in I_1, but not in I_2. We also do not consider the patches around those pixels, as we do not want the reference and the corresponding patch to look completely different and importantly, the network should not learn such cases, where the reference and the corresponding patches look completely different, because the object (or the center of the patch) is occluded by another object. Of course, the patches involved with partial occlusion, where the center of the reference patch is visible in both images, are still considered, as occlusions often happen in image sequences and we expect to distinguish the matches, even if a small part of a patch is occluded by another object, in I_2.

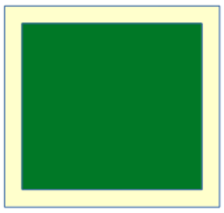


Figure 3.1: This figure shows the image region (the green region), from which we extract patches to generate our training samples. The width of the yellow area is equal to half of the patch size. A patch around every pixel in the green area lies entirely in the image.

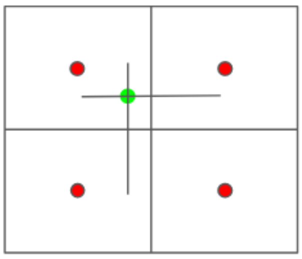


Figure 3.2: The green dot shows the center pixel of the corresponding patch. It lies in between the center of its neighboring pixels, that are shown as red dots. The position of the center of the corresponding patch is resulted from adding non-integer flow values to the integer pixel coordinates of the center of the reference patch. The gray lines from the green dot show the distance of that position in x- and y-coordinates to the center of its neighboring pixels. Based on this distances, the value for that position should be computed by interpolating the values of its neighboring pixels.

As we explained earlier for all the pixels in I_1 the shown green area in Figure 3.1 we can extract the patches and consider them as reference patches if their center has a valid flow value and if its corresponding pixel is not occluded in I_2. This means we can create a very large number of training samples. Basically, based on the complexity of the network and how many training samples it needs to be trained properly, one can decide about creating the suitable number of training samples.

For images in KITTI 2015, which have roughly the spatial size of (1242×375), we extracted the patch around every eighth pixel in both x and y directions and for images in MPI-Sintel dataset, the patch around every 30th pixels in both directions was extracted. With this setting, the number of the patches made from KITTI 2015 and MPI-Sintel image will be almost the same and the number of all the patches will be roughly 500,000.

We created patches with the spatial sizes (64×64) and (16×16) pixels. Larger patch sizes are typically used in the literature [GW16] for similar tasks, however, we did not use the patches of size (64×64) for training our networks, because we observed that training a network with such patches was too time-consuming.

Corresponding Patches

For every reference patch created in the above section, there should be its corresponding patch in each training sample. Obviously, the corresponding patches are extracted from I_2. For each pixel around which we extracted the patch as the reference patch, the flow value is added to the coordinates of that pixel. The resulting coordinates will be the position of the center of the corresponding patch to that reference patch. We only consider the training samples in which the corresponding patch lies entirely in the image.

Essentially, the flow values are floating point numbers and by adding them to the coordinates of the center pixel of the reference patch, the resulting pixel might lie in between the center of pixels (see Figure 3.2). To get the values of non-integer positions in the patch, interpolation among some neighboring pixels is needed. Taking the simple bilinear interpolation algorithm into account, the image value in the position (x, y) is calculated via:

$$\begin{aligned} I(x, y) = {} & (1-\epsilon_i)\cdot(1-\epsilon_j)\cdot I(i,j) \\ & + \epsilon_i\cdot(1-\epsilon_j)\cdot I(i+1,j) \\ & + (1-\epsilon_i)\cdot\epsilon_j\cdot I(i,j+1) \\ & + \epsilon_i\cdot\epsilon_j\cdot I(i+1,j+1) \end{aligned} \tag{3.5}$$

with the separation of x and y in integer and fractional parts

$$\begin{array}{llllll} x = i + \epsilon_i, & \text{with} & i = \lfloor x \rfloor & \text{and} & \epsilon_i = x - i \\ y = j + \epsilon_j, & \text{with} & j = \lfloor y \rfloor & \text{and} & \epsilon_j = y - j \end{array}$$

Note that the values of all the pixels in corresponding patches have to be computed via interpolation, not only for the center pixel.

Non-matching Patches

As we explained before, each training sample should consist of a reference patch, the corresponding patch to the reference patches and a patch that does not correspond to the reference patches. By this definition, we have some freedom to choose non-matching patches and depending on the task that has to be done by the network, the training samples should be chosen carefully. For example, if the network has to learn that the reference and non-matching patches look very different, we can even extract the non-matching patch from an irrelevant image (i.e. neither I_1 nor I_2). However, if among some neighboring patches that look similar, the network has to distinguish which patch is the corresponding patch of a reference patch and which patches are non-matching patches, one should consider spatially close patches to the corresponding patch as the non-matching patches, in training samples.

As later on we will use the learned descriptors in CPM, which was discussed in Chapter 2, and in CPM, there are steps in the matching algorithm in which the pixels close to a possible match candidate are examined to find the best match for a reference pixel, we decided to extract the non-matching patches from I_2 with some small spatial distance from the corresponding patch. This distance is a degree of freedom in preparing the training samples. In our previous work [Jah17], we randomly chose a number between 3 to 8 pixels distance to the integer part of the coordinates of the reference patch center in both x- and y-coordinates and extracted the patch around that pixel, as a non-matching patch.

Other than the discussed strategy, we also tried another way of creating non-matching patches, as Bailer et al. [Bai+17] suggested. We consider the center of non-matching patches with a distance chosen from a normal distribution, that prefers patches closer to the corresponding patch. We set the minimum distance of two pixels from the integer part of the coordinates of center of matching patches. In Chapter 5 we will investigate which of the aforementioned strategies to extract the non-matching patch leads to better results.

In each row of Figure 3.3 a training sample is shown. The non-matching patches are extracted with the first strategy. As you can see, in homogeneous regions, all the reference, the corresponding and the non-matches look the same.

Valid Training Samples

So far, we explained how to create the training samples. In this part, we discuss if all training samples can nicely contribute to the training process.

In our previous research [Jah17], we observed a special situation while training CNNs: By keeping the order of the training samples fixed, the values of all trainable variables, as well as the value of the simple loss, that we discussed at the beginning of this chapter, jumped to zero on a certain iteration and stayed unchanged for all iterations after that. We tried different learning rates to make sure that the value of the step-size is not causing that issue. After checking the training samples that were fed to the network on that specific iteration, we found out that in that specific batch, there were a few training samples where the reference, corresponding and non-matching patches looked the same.

With the strategies we discussed about generating the training samples, most probably this situation could happen where there is a large homogeneous area in the image, such as an overexposed sky. As we could not further train the networks, we suggested a criteria to discard the problematic training samples that were extracted from uniform areas. In $loss_{simple}$ we compare the descriptors of the reference and corresponding patches and the descriptors of the reference and the non-matching patches. Inspired by this formula, we decided that if the reference and corresponding patches or the reference and the wrong patches are too similar (or dissimilar less than a threshold), we discard the training sample. In other words, if $\sigma_{\Delta_{\text{corresponding}}}$ is less than a threshold or $\sigma_{\Delta_{\text{non-matching}}}$ is less than that threshold, the training sample will be discarded, where $\Delta_{\text{corresponding}} = (\text{patch}_{\text{reference}} - \text{patch}_{\text{corresponding}})$ and $\Delta_{\text{non-matching}} = (\text{patch}_{\text{reference}} - \text{patch}_{\text{non-matching}})$ and σ denotes the standard deviation operation.

In our previous research the measure of dissimilarity used in that criterion, was the standard deviation of difference of two patches, and we set the threshold to 1.3 in that project. With that criterion, around 10% of the training samples were discarded. The value of the threshold was chosen by experiments, as it could discard uniform or mostly uniform patches. Using that solution, the mentioned problem in the training process disappeared.

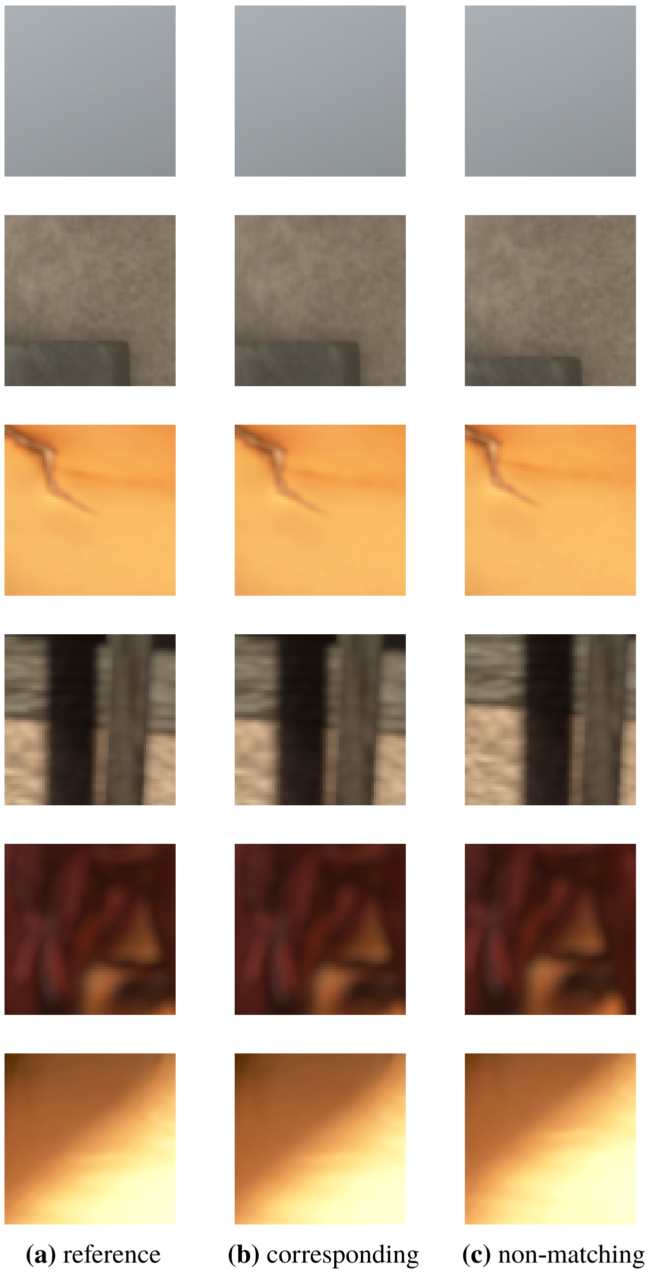


Figure 3.3: With the discussed method for generating the training samples, in the first glance, all patches look similar. With a closer look, we see that non-matching patches are a shifted version of corresponding patches. In homogeneous areas all three patches look very similar (the first row in this figure) and this might cause troubles during training, which is discussed later on in this chapter. These patches are made from the MPI-Sintel dataset.

In this thesis, we did a few experiments regarding this issue and we found out that if we use batch normalization in the convolutional layers, that problem does not show up unless all the training samples in a batch are too similar. Therefore, to avoid the mentioned issue, in this case there is no need to discard such training samples using the aforementioned criterion, however, we thought how those similar patches could contribute to the training process.

Obviously, we do not want the network to learn that reference and non-matching patches or corresponding and the non-matching patches are similar and unfortunately, this is what the training samples, consisting of very similar patches, could teach the network. Therefore, once more, we tried to find a way to discard the training samples, in which reference and non-matching patches and corresponding and non-matching patches were less dissimilar than a threshold. This time, we took a more careful look at the above-mentioned discarding criterion and we found it faulty, as it discards more training samples then it should. Firstly, that threshold does not have to be as large, so we reduced it to 1.0. Secondly, in the aforementioned solution, we also discarded the samples where reference and corresponding patches were too similar, as this was the first condition in our if-clause and between the conditions there is an `or`. This means if a pixel does not move or it moves in such a way that its reference and corresponding patch are less than a threshold dissimilar, the training sample is discarded.

Instead of the mentioned discarding criterion, we consider this observation: If the extracted non-matching patch is too similar to the reference patch, typically it is also too similar to the corresponding patch and this case most probably happens when the patches are extracted from a homogeneous area. Thus, if the corresponding and non-matching patches are not too similar, typically the reference and non-matching patches are not too similar, as well. Therefore, it is enough to check if the corresponding and non-matching patches are not dissimilar less than a threshold, to discard the training samples that consist of patches that look alike. Thus as a new criterion, if in a training sample $\sigma_{\Delta_{\text{non-matching}}}$ is less than a threshold, the training sample is discarded.

Of course, one could also train a network with all the training samples without discarding any training samples. In Chapter 5 we show that using the new criteria leads to a slight improvement of the learned descriptors.

After generating the training samples, we can build a CNN and then train it. Earlier in this chapter, we mentioned some basics about training a network and what characteristics we expect the descriptors to have. In the next section, we explain how to build a network and in the section after that we discuss the training process in more detail.

3.2 Building a CNN

In this section, we describe how to construct a CNN, given the essential parameters.

There are several parameters that have to be specified to design a network such as the number of convolutional layers and pooling layers, the order of pooling layers, the number of filters in each convolutional layer and the size of the receptive field. In this section, we explain how a CNN is implemented and built with the Python API of Tensorflow [Ten18] and in the next section, we describe the process of training the network.

To build and train CNNs, there are other software libraries available, as well, such as Caffe [Ber18], MatConvNet [Ved+18], Theano [The18], the Microsoft Cognitive Toolkit [Mic18], deeplearning-hs [Tul15] and neon [Int17]. We used Tensorflow to implement our networks, as it is a powerful open source library, which has one of the largest number of contributors on GitHub [Git18].

Listing 3.1 presents how a network is built given these parameters. In our previous research [Jah17], there was a similar function to construct a CNN, which is slightly changed here so that it is more comprehensible. Also a wider range of architectures can be constructed by the new version of this function.

The input arguments `filter_size` and `input_depth` are the size of the receptive field and the number of channels in the input image, respectively. The argument `image` is the input image, which is an image-patch, for which the descriptors should be computed via the network during training. Obviously at this point, by mentioning descriptors of a patch, we just mean the output of the network given the patch. After a proper training, we expect the output of the network to be the descriptors with the expected characteristics. `num_filters` is a list of integers that represent the number of filters (i.e. neurons) in each convolutional layer. Basically, the length of this list shows how many convolutional layers we use in the network.

The last input argument of this function is `pooling_order`, that is a list of binary digits, showing after which convolutional layer a pooling layer should be inserted. Obviously, the length of `num_filters` and `pooling_order` should be the same. If we assign `num_filters` to [10, 15, 20] and `pooling_order` to [0, 1, 1], the function makes a network with three convolutional layers with 10, 15 and 20 filters for the first, second and third convolutional layers, respectively. And after each of the second and third convolutional layer, a max-pooling layer is built. As a result, the first layer is a convolutional layer with 10 filters, the second layer is also a convolutional layer with 15 filters followed by a max-pooling layer and then the third convolutional layer with 20 neurons again followed by the second max-pooling layer. The output of the last layer, which is the second max-pooling layer in this case, is flattened and considered as the output of the network or the descriptors of the given input image `image`.

The similar function `model_network` in our previous work [Jah17] handled the insertion of pooling layers differently. There was an input argument `pooling_rate`, that described how often a max-pooling layer should be inserted in between convolutional layers. For instance, the zero value for `pooling_rate` expressed that after each convolutional layer there should be a pooling layer inserted. And with the value one, pooling layers would be inserted after every other convolutional layer. The way that `polling_rate` was able to handle the insertion of pooling layers limited the possible choices of the order of pooling layers. For example, using `pooling_rate`, building a network, that was described above, with three convolutional layer and two pooling layers after the second and third convolutional layer is not possible, as with that strategy the insertion of pooling layers only with regular frequencies is possible.

Let us now discuss some important code details. In Listing 3.1, at Lines 16, 17, two lists to keep the output of each convolutional layer and pooling layer are defined. Then the layers with suitable inputs and dimensions are built and their output is appended to the mentioned lists. Obviously, the input of the first convolutional layer is the given image-patch, `image`. The necessary dimensions to define the weights and biases are the size of the receptive field (`filter_size`), depth of the input

```
def conv_relu(input, kernel_shape, bias_shape):
    weights = tf.get_variable("weights", kernel_shape,
                              initializer=tf.random_normal_initializer())
    biases = tf.get_variable("biases", bias_shape,
                             initializer=tf.constant_initializer(0.0))
    conv = tf.nn.conv2d(i, weights,
                        strides=[1, 1, 1, 1], padding='VALID')
    return tf.nn.sigmoid(tf.contrib.layers.batch_norm(conv + biases))

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='VALID')

def model_network(filter_size, input_depth, image, num_filters, pooling_order):
    num_hidden_layers = len(num_filters)
    hidden_conv = []
    hidden_pool = []
    poolCounter = 0
    for i in range(num_hidden_layers):
        if (i == 0):
                with tf.variable_scope("conv0"):
                    hidden_conv.append(
                    conv_relu(image, [filter_size, filter_size,
                    input_depth, num_filters[i]], [num_filters[i]]))
                    if pooling_order[i]==1:
                        hidden_pool.append(max_pool_2x2(hidden_conv[i]))
                        poolCounter += 1
        else:

                with tf.variable_scope("conv" + str(i)):
                    if pooling_order[i-1] == 1:
                        hidden_conv.append(conv_relu(hidden_pool[-1],
                        [filter_size, filter_size, num_filters[i - 1],
                        num_filters[i]], [num_filters[i]]))
                    elif pooling_order[i-1] == 0:
                        hidden_conv.append(conv_relu(hidden_conv[-1],
                        [filter_size, filter_size, num_filters[i - 1],
                        num_filters[i]],[num_filters[i]]))
                    else:
                        print("hmm :( wrong pooling argument. The arguments should be
                        either 0 or 1.")
                    if pooling_order[i]== 1:
                        hidden_pool.append(max_pool_2x2(hidden_conv[i]))
                    poolCounter += 1

    if pooling_order[-1]== 1:
        lasthidden_flat = tf.layers.Flatten()(hidden_pool[-1])
    else:
        lasthidden_flat = tf.layers.Flatten()(hidden_conv[i])
    return lasthidden_flat
```

Listing 3.1: The function `model_network` builds a convolutional neural network with the given parameters. Functions `conv_relu` and `max_pool_2x2` are supplementary functions that are used in `model_network`.

volume, which is the number of channels of the input image-patch, and the number of filters for that layer. Other parameters such as zero padding options and the stride parameter are defined in the function `conv_relu`.

After the first layer is built, its output is appended to the list `hidden_conv`. At this point, if `pooling_order[0]` equals one, a pooling layer will be inserted after the first convolutional layer. Surely, in this case the output of the first convolutional layer is the input of the first pooling layer. The parameters, such as zero padding options and kernel size and the stride parameter are specified in the function `max_pool_2x2`. In Line 26, the output of the pooling layer is appended to the list `hidden_pool`.

To build the other layers of the network, one has to check where its input is resulted from, therefore to specify the input of the ith convolutional layer the value of `pooling_order[i-1]` should be checked, because If `pooling_order[i-1]` equals one, the input of the ith convolutional layer is resulted from the pooling layer before this convolutional layer,and therefore it is the last element of the list `hidden_pool` up to that point. If `pooling_order[i-1]` equals zero, the input of the ith convolutional layer is resulted from the previous convolutional layer, and it is the last member of `hidden_conv`, up to that point. Obviously, after making the ith convolutional layer, the value of `pooling_order[i]` should be checked. If the value is one, a pooling layer should be inserted and otherwise not, and of course, the input of pooling layers is always the last member of `hidden_conv`, as pooling layers' input is always resulted from their previous convolutional layer.

After all layers are built properly, the output of the last layer is flattened and returned as the output of the network. After the training process, we expect the output of the network to have the mentioned features of desired descriptors. In this chapter, we also refer to the output of the network as descriptors, for convenience.

So far, we explained how to build a CNN. Next we explain how the training process works and how we implemented it.

3.3 Training a CNN for Descriptor Learning

Listing 3.2 shows the implementation of the training process in Python. Lines 9 to 11 show an example of some values assigned to some parameters to build the CNN. The network is constructed given these parameters by the function `model_network` which was shown in Listing 3.1.

In Lines 15 to 22, the function `model_network` is called three times and the first time it is called a name is assigned to the variables of that scope. The second and third time that the function `model_network` is called, the variables of the previous scope that were defined by calling the function for the first time are reused.

To be more specific, when the function is called for the first time, the structure of the network is built and the variables of the network are generated based on the given parameters and the name `NN_model` is assigned to the scope of the defined variables. The input image-patch (i.e. here a reference patch) which should be fed to the network is kept in the placeholder `x1` and the output of the network is assigned to `reference_descriptors`. When the function is called for the second and third time, the same structure of the network is built with the exact same parameters and the variables of the scope

```
def train():
    patch_size = 16
    x1 = tf.placeholder(tf.float32, shape=(None, patch_size, patch_size, 3), name="input_node")
    x2 = tf.placeholder(tf.float32, shape=(None, patch_size, patch_size, 3))
    x3 = tf.placeholder(tf.float32, shape=(None, patch_size, patch_size, 3))
    epoch_per_batch = 5
    network_details_path = "some path"
    channels_num = 3
    num_neurons = [8, 16, 32, 64, 128, 256, 128]
    pooling_order = [0, 0, 0, 0, 0, 0, 1]
    receptive_field_size = 3
    pooling_str = ""
    for i in pooling_order:
        pooling_str += str(i)
    with tf.variable_scope("NN_model") as scope:
        reference_descriptors = model_network(receptive_field_size,
        channels_num, x1, num_neurons, pooling_order)
    with tf.variable_scope("NN_model", reuse=True):
        corresponding_descriptors = model_network(receptive_field_size,
        channels_num, x2, num_neurons, pooling_order)
        nonMatching_descriptors = model_network(receptive_field_size,
        channels_num, x3, num_neurons, pooling_order)
    with tf.name_scope('loss'):
        matching_dist = tf.norm(reference_descriptors - corresponding_descriptors, axis=1, ord=2)
        nonmatching_dist = tf.norm(reference_descriptors - nonMatching_descriptors, axis=1, ord=2)
        loss = tf.reduce_mean(matching_dist - nonmatching_dist, axis=0)
    stepSize = 5e-3
    with tf.name_scope("train_step"):
        train_step = tf.train.AdamOptimizer(stepSize).minimize(loss)
    queue = Queue(100)
    read_process = Process(target=fill_queue, args=(queue, epoch_per_batch, path))
    read_process.start()
    description = "some_description_if_necessary"
    for ind in range(len(num_neurons)):
        description = description + "_" + str(num_neurons[ind])
    saver = tf.train.Saver()
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        j = 0
        while True:
            queue_result = queue.get(block=True)
            if queue_result == "done":
                break
            batch_reference, batch_corresponding, batch_nonmatching = queue_result
            if len(batch_original) == 0:
                break
            _ = sess.run(train_step, feed_dict={x1: batch_reference,
            x2: batch_corresponding, x3: batch_nonmatching})
            j += 1
        save_path = saver.save(sess,network_details_path + "trained_networks/" + description + "_" +
        pooling_str + "_" + str(stepSize) + "_" + str(epoch_per_batch) + "/trained_variables.ckpt")
```

Listing 3.2: This listing shows how we train a CNN for descriptor learning.

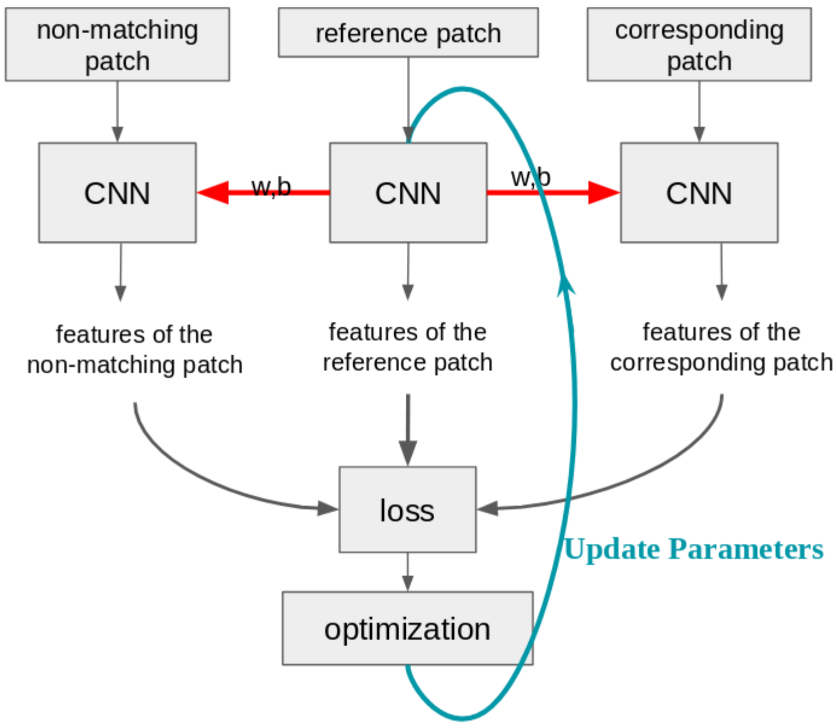

non-matching patch
reference patch
corresponding patch
CNN
w,b
CNN
w,b
CNN
features of the non-matching patch
features of the reference patch
features of the corresponding patch
loss
Update Parameters
optimization

Figure 3.4: This figure shows how the trainable variables (weights w and biases b) are shared among siamese branches. When a network is built and its trainable variables are initialized, two other networks (in this case) are made with shared trainable variables as the first one. After each training iteration, the values of the trainable variables of the first network are updated and of course, these updated values trainable variables are also shared with those two other branches (networks). In other words, the trainable variables of all different branches always have the same values, in our case.

`NN_model` are reused in the structure. The only differences are the input image-patches which are the corresponding and a non-matching patches and the outputs of the networks are assigned to `corresponding_descriptors` and `nonMatching_descriptors`.

The reason of sharing the variables is that for training the network, we need to compute the descriptors of each patch (i.e. a reference, the corresponding and a non-matching patch) in one training sample with the exact same network and use these values to compute the value of the loss function that we explained earlier. This sharing of variables in some parts or complete structure of the network is called a siamese architecture. Figure 3.4 visualizes the siamese architecture that we use in our training process.

It is important to mention that the aforementioned siamese architecture is only used for the training process, as the descriptors for all three reference, corresponding and non-matching patches have to be computed with the same network. After training the network, we compute the descriptors with one single-branch network for all the patches in an image.

In Lines 23 to 26, the simple loss is implemented and in Line 29 the training-step is defined by specifying an optimizer and a learning rate for minimization of the loss function. In Tensorflow the operations are executed via a session, therefore in Line 37 a session is defined and all the existing trainable variables that have been defined up to that point are initialized. In the loop inside the session, the prepared training samples are fed to the network to compute the value of the loss function. Basically, every training sample can contribute to the training process many times. This number is specified by the variable `epoch_per_batch`. For example, if `epoch_per_batch = 5`, all the training samples are fed to the network five times.

The way that the training samples are read will be discussed in detail, a bit later in this section. As a short clarification, a batch of the size 100 training samples will be fetched at Line 44, then the training-step is run via the session at Lines 47 and 48. The goal of the training-step is to minimize the loss function. And for computing the value of the loss function, the output of the network given reference, corresponding and non-matching patches should be computed. Therefore the training samples are fed to the network in Line 44 in order to run the training-step.

In every iteration, a batch of training samples is fed to the network, and then by running the training-step the trainable variables that have been initialized in Line 38 will be updated in such a way that the loss-function is minimized, however this optimization problem is non-convex, thus there is no guarantee to reach the global minimum.

After the training process is finished, we save the trained variables to be able to load the trained network to use it later (e.g. for optical flow estimation). Lines 50 and 51 shows how the trained variables are saved.

3.4 Loading the Training Samples

Regarding loading the training samples for the training process, there are different strategies. One option is to read the whole training samples first and then start the training process. Of course if the training samples are placed in RAM, they are accessed much faster while training, however, if the size of all the training samples is larger than some amount, it does not fit in RAM. Another drawback about this strategy is that if we load the training samples first and then start the training process, a lot of time is wasted.

Another way is to read one batch of training samples and train the network with that batch and again read another batch, train and so on. This strategy needs a small space in RAM, also the training process starts much earlier compared to the last option. On the other hand, the training process is interrupted in every iteration waiting for the training samples to be loaded from disc.

We decided to choose another option, which is a compromise between the above-mentioned strategies, i.e. we load the training samples and start the training process as soon as a batch of training samples is available. When a batch is being processed for training, next training samples are loaded and placed in a queue, in parallel. As we chose the batch-size to be 100, as soon as 100 training samples are loaded, the batch is placed in the queue and it is used in the training process. If less than 100 samples are read, the training process waits until a complete batch is loaded and placed in the queue.

```
def fill_queue(queue, epoch_per_batch, path):
    global filenames
    imgs_ref = []
    imgs_correct = []
    imgs_wrong = []
    for i in range(epoch_per_batch):
        for name in filenames:
            try:
                img_orig = imglib.read_image(path + "/original/" + name)
                img_correct = imglib.read_image(path + "/correct/" + name)
                img_wrong = imglib.read_image(path + "/wrong/" + name)
                imgs_ref.append(img_orig)
                imgs_correct.append(img_correct)
                imgs_wrong.append(img_wrong)
            except:
                continue
            if len(imgs_wrong) == 100:
                imgs_ref = np.asarray(imgs_ref, dtype=np.float32)
                imgs_correct = np.asarray(imgs_correct, dtype=np.float32)
                imgs_wrong = np.asarray(imgs_wrong, dtype=np.float32)

                imgs = (imgs_ref, imgs_correct, imgs_wrong)
                queue.put(imgs, block=True)
                imgs_ref = []
                imgs_correct = []
                imgs_wrong = []
    queue.put("done", block=True)
```

Listing 3.3: This listing shows how we read the training samples to train a defined CNN.

In Line 30 in Listing 3.2, a queue of size 100 is made. This queue is filled by another function `fill_queue` shown in Listing 3.3, which reads a batch of 100 training samples and puts it as a tuple of three Numpy arrays in the given queue. Notably, every batch of 100 training samples is one member of the queue. When the method `fill_queue` is called, the whole queue is filled with 100 batches of training samples, i.e. 100 batches of 100 training samples. As the queue is filled completely, the process of filling the queue gets blocked until there is at least one free slot in the queue.

As `fill_queue` is called before the training process begins, the queue is filled with 100 batches of training samples when the training process starts. As soon as an element is fetched from the queue at Line 44, a slot will be freed in the queue and the process of filling in the queue resumes and continues filling in the queue while the training process is done. Importantly, if during training the queue becomes empty, the training process waits until there is at least one batch in the queue. Using this strategy, training a network that took two days by reading all the training samples first and then training, takes about a day.

So far, we explained how we generate the training samples with two different strategies. We also discussed that in homogeneous areas, the training samples are very similar and we suggested a way to discard them. Then we discussed how we built a network using the function `model_network`, represented in Listing 3.1. We also explained how the training process works and how we tried to save time by reading the training samples and training them in parallel. In the next chapter, we explain how the complete model of a trained CNN is exported to be used in other programs. Then we explain how to compute the descriptors for the images to do the matching task via the CPM algorithm. Importantly, we also discuss an efficient way of computing the descriptors for each image, by a single forward pass through the trained CNN.

4 Embedding the Learned Descriptors in the Optical Flow Estimation Pipeline

Our previous research on descriptor learning [Jah17] evaluated the learned descriptors by considering it as a binary classification problem. If the L2-distance of the calculated descriptors of corresponding test patches was smaller than the L2-distance of the descriptors of non-matching test patches, this situation was considered as a right decision and otherwise, as a wrong decision. This way of evaluation can not directly show how successful descriptors are in the optical flow estimation task, as the corresponding patch should be identified not only compared to one non-matching patch, but all the candidates in a region. In this thesis, the evaluation of descriptors is done differently. We compute the descriptors via a properly trained CNN and use the descriptors to compute the so-called cost function of the CPM algorithm.

As we explained in Chapter 2, CPM is a popular method for the optical flow estimation task for large displacements. It computes the correspondences based on a cost function, in a coarse-to-fine scheme. After estimating the optical flow, we can compare our results to the ground truth optical flow values. Using the result of this comparison, we can judge the effectiveness of the learned descriptors in combination with CPM. Figure 4.1 shows the pipeline of our approach.

Next, we discuss how we embed a trained network to the mentioned pipeline to contribute to the matching task for optical flow estimation.

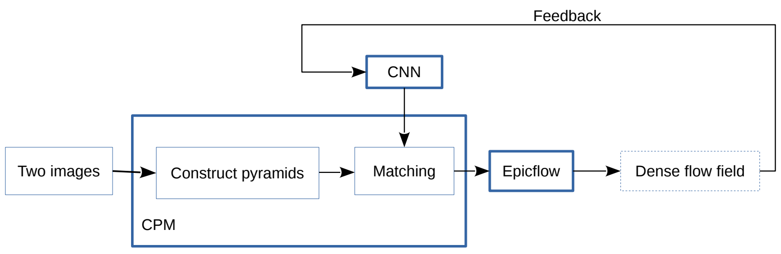


Figure 4.1: The pipeline of our approach.
We use a trained CNN to compute descriptors for the matching process of the CPM algorithm. The resulting non-dense matches are given to Epicflow to compute the dense flow field. This result should be compared to the ground truth and the CNN should be modified to improve the results.

As we described before, the goal of this chapter is to embed a trained CNN in the CPM method to find the matches between two images. The code of CPM algorithm is released [Hu17] in C++ language. After training a network, we should make a portable model from it so that we can import the trained network in other programs. In this section, we describe how to export the model of a trained network to be able to run it from another program.

In the previous chapter, we showed how to train a network and save the trained variables to a checkpoint file to be able to restore and use the values of the trained variables again. To export the complete trained model, one has to save the graph of the model, as well. As explained in Chapter 2, a Tensorflow graph represents a sequence of operations and it shows how individual operations are related with one another. The complete model of the trained network can be exported from the graph of the network along with the value of the trained variables. Importantly, before saving the graph, the input and output of the graph should be labeled to specify what part of the graph should be executed.

Listing 4.1 shows how the complete trained model is exported. In Lines 2 and 10, a label is assigned to the input and output nodes. The input node is a placeholder for the input image (or the image-patch) and the output is the set of descriptors computed via the trained CNN, given a patch.

Importantly, there is no siamese architecture here for creating the portable model, as we just need a single branch network to compute the descriptors for a patch. At Lines 16 and 17, the graph of the single branch network is saved.

The module `freeze_graph()` in the file `freeze_graph.py` exports the complete trained model by combining the graph structure and the values of the trained variables that have been saved in a file after the training process. It converts the variables to constant values and it generates the complete trained model, which can be run independently, and saves it to a given path as a `.pb` file.

To use the C++ API of Tensorflow, one should build Tensorflow from source. After building Tensorflow, the necessary files and libraries should be added to the C++ project (i.e. the CPM project, in our case).

Listing 4.2 shows how a trained model is loaded and run in the C++ environment. At Line 14, the trained model, which was saved in the path specified at Line 6, is loaded. To run the network, we need to specify the label of the input and output node in the graph, as well as the input tensor and a vector of tensors, as a placeholder for the output of the network. Essentially, the input image-patch, for which we want to compute the descriptors, should first be converted to a tensor. For the sake of conciseness, we assume that function `fill_tensor`, which is called at Line 13, copies the input image into the tensor `ten_patch`, defined at Line 12.

In the available code of CPM, the matching cost is computed as the sum of absolute differences over SIFT descriptors, which is computed for the match candidates. The available code also has the option to use DAISY descriptors to compute the cost function, instead of SIFT descriptors. Considering w and h as the width and the height of the input image and d as the number of descriptors computed for each pixel, CPM needs a descriptor matrix with $w \cdot h$ columns and d rows to do the matching task using this descriptor matrix. Therefore we need to compute such a descriptor matrix via our trained CNN.

```python
def save_model():
    x = tf.placeholder(tf.float32, shape=(1, None, None, None), name="input_node")
    channels_num = 3
    num_neurons = [10, 15, 20, 25, 30, 35, 40]
    pooling_order = [0, 0, 0, 0, 0, 0, 1]
    receptive_field_size = 3
    with tf.variable_scope("NN_model") as scope:
        descriptors = model_network(receptive_field_size,
        channels_num, x, num_neurons, pooling_order)
        descriptors = tf.identity(descriptors, name= "output_node")

    path_prefix = "/some_directory/"
    trained_model_name = "model_name"
    trained_vars_path = path_prefix + "trained_networks/" + trained_model_name
    with tf.Session() as sess:
        tf.train.write_graph(sess.graph.as_graph_def(), path_prefix +
        'graph_def/'+ trained_model_name, "train.pbtxt")
        output_graph_name = "output_graph.pb"
        input_graph_path = path_prefix + 'graph_def/'+ trained_model_name + '/train.pbtxt'
        input_saver_def_path = ""
        input_binary = False
        input_checkpoint_path = trained_vars_path + "/trained_variables.ckpt"
        output_node_names = "NN_model/output_node"
        restore_op_name = trained_vars_path + "/trained_variables.ckpt"
        filename_tensor_name = path_prefix + "output_models/" + trained_model_name + "/Const:0"
        output_graph_path = path_prefix + "output_models/"+
        trained_model_name +"/" + output_graph_name
        clear_devices = False
        freeze_graph.freeze_graph(input_graph_path, input_saver_def_path,
                                  input_binary, input_checkpoint_path,
                                  output_node_names, restore_op_name,
                                  filename_tensor_name, output_graph_path,
                                  clear_devices, initializer_nodes= "")
```

Listing 4.1: This listing shows how the graph of a built network is created and combined with the values of the trained variables to obtain a complete and portable model of the trained network.

To compute the mentioned descriptor matrix, there are different strategies. In the next section we discuss a simple solution and in Section 4.2, we explain a strategy which is more complicated but much faster than the simple solution.

4.1 A Simple Way of Computing the Descriptor Matrix

As we discussed in the previous chapter, we trained a CNN with patches. Therefore to compute the descriptors for each pixel in the image, the common solution is to consider a patch around each pixel and feed each patch to the network to compute the descriptors for the center pixel of that patch. Clearly, the images for which we want to compute the matches should be padded, so extracting a patch around the pixels that are close to image boundaries is possible and thus, the descriptors for each pixel in the image can be computed.

```cpp
using tensorflow::Status;
using tensorflow::string;

std::vector<Tensor> run_net(cv:Mat inputImage)
{
  string model_path = "path of the exported model";
  std::unique_ptr<tensorflow::Session> session;
  string input_layer = "input_node:0";
  string output_layer = "NN_model/output_node:0";
  //the input image/patch will be copied into the following tensor,
  //with the following dimensions via the function fill_tensor().
  Tensor ten_patch(DT_FLOAT, TensorShape({1 , input_height, input_width, num_channels}));
  fill_tensor(ten_patch, inputImage);
  Status load_graph_status = LoadGraph(model_path, &session);
  //the output will be stored here.
  std::vector<Tensor> features;
  Status run_status = session->Run({{input_layer, ten_patch}}, {output_layer}, {}, &features);
  return features;
}
```

Listing 4.2: This listing shows how the trained model is loaded in a C++ environment. For the sake of conciseness, we did not present the complete code about filling the tensor with the image-patch data.

This way of computing the descriptors for every pixel, by extracting a patch around each pixel and feed them to the network is a simple solution, however, it is a very slow approach. Finding the matches for an image-pair of size (436 × 1024), by computing descriptors via a network with seven layers consisting of a number of filters smaller than 50 for each layer, takes more than 30 minutes using our system (Intel Core i7 2.60 GHz ×8, 16 GB RAM, Nvidia GTX 860M).

In this thesis, we want to use learned descriptors to find the matches and estimate the optical flow and another goal is to improve the learned descriptors. To have a comparison among the descriptors calculated by different networks, we can compare the AEE of the resulted flow field on the evaluation set we defined in the previous chapter. The evaluation set consists of around 120 image pairs, thus finding the matches for this set takes about 60 hours. This is because descriptors have to be computed for the patches extracted around all pixels in both images in all image-pairs.

Obviously, as we should compare the outcome (i.e. the optical flow estimates via descriptors using CPM) of each network, and as the training each CNN itself is a time consuming process, the above-mentioned execution time is unacceptable. The time-consuming task of extracting patches around each pixel and computing the descriptors for each patch is not feasible in this thesis, therefore a faster way has to be taken into account.

In the next section, we explain another strategy, which speeds up the process of computing the descriptors for images.

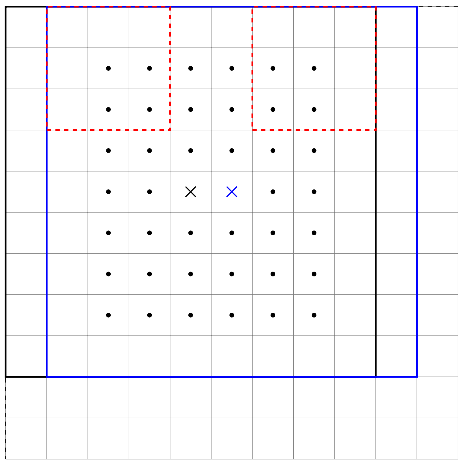


Figure 4.2: This figure shows that doing convolution for adjacent patches has many redundant operations. The black and blue adjacent patches of size (9×9) are shown. Their center pixel is shown by black and blue crosses, respectively. If we perform convolution for each patch separately and we consider the receptive field of size (3×3), the black dots show the positions for which the convolution operation is redundant, as the receptive field center passes each of the specified positions, when doing convolution for both patches. The (3×3) red squares show two areas that the receptive field covers when performing convolution. These areas are completely contained in both patches, therefore the resulting convolution operations for them yield the same results.

4.2 A Faster Strategy to Compute the Descriptor Matrix

In the previous section, we discussed the necessity of finding a fast way to compute the descriptors for every pixel in the images. This can be done by skipping the time-consuming step of extracting a patch around each pixel. Also running a CNN for patches around adjacent pixels, has many redundant operations. Figure 4.2 shows this redundancy.

To speed up the process of computing descriptors for every pixel, one way is to pass the whole image through the network. For architectures with no pooling layers, this is done easily. Consider a CNN with three layers consisting of 10, 20 and 30 neurons for the first, second and third convolutional layer, respectively, and receptive field of size (3×3). We assume the patch of spatial size (7×7) and the image with the spatial size (100×100) pixels. Without a zero padding in the convolutional layers, if a patch is passed to the network, the dimension of the resulting descriptors will be $(1 \times 1 \times 30)$, as in each convolutional layer, the spatial dimensions reduce by 2 pixels. The input of the first layer is the patch with dimensions $(7 \times 7 \times 3)$ and the dimension of the output of the first layer is $(5 \times 5 \times 10)$. The outputs of the second and third layers have the dimensions $(3 \times 3 \times 20)$ and $(1 \times 1 \times 30)$, respectively.

If the color image with the spatial size (100×100) is fed to the network, the output will have the dimensions $(94 \times 94 \times 30)$ and if we pad the image before passing it through the network by 6 pixels (so that for every pixel, there is patch of size (7×7) around), the output will be of the size $(100 \times 100 \times 30)$.

Using pooling layers is common in the architecture of CNNs, as it speeds up the the network by reducing the spatial size of the volume and helps to avoid over-fitting. However, using pooling layers in a CNN, makes the mentioned scenario much more complicated. Assume the same aforementioned architecture, but with a max-pooling layer after each convolutional layer. We also use a proper zero padding for both convolutional and pooling layers so that the spatial dimension of the input and output of convolutional layers stays unchanged and the spatial size of the output of pooling layers reduces by half of the spatial input size, for even spatial input sizes, and half of the spatial input size plus one, for odd spatial input sizes. With this setup, by passing a patch to the network, dimensions of the output of the first, second and third pooling layer will equal $(4 \times 4 \times 10)$, $(2 \times 2 \times 20)$, and $(1 \times 1 \times 30)$, respectively. If we feed the image to the network, without padding it, the outputs of the first, second and third pooling layer will have the dimensions $(50 \times 50 \times 10)$, $(25 \times 25 \times 20)$ and $(13 \times 13 \times 30)$, respectively.

As we explained earlier in this chapter, we need the descriptor matrix to be of the size $(w \cdot h \times d)$, therefore the descriptor matrix for the image with the spatial size of (100×100), should have the dimensions $(100 \cdot 100 \times 30)$, for the mentioned network, which means for each pixel there should be a descriptor vector of size d. The output of the network without pooling layers, given the padded image has the expected dimensions (i.e. for each pixel there is a descriptor vector of size d and of course, the output can be reshaped to have the expected dimensions: $(w \cdot h \times d)$), however if there are pooling layers in the network, the spatial dimension of the output of the network is much smaller than $(w \cdot h \times d)$.

Fortunately, the situation (i.e. not getting the same spatial output size), that pooling layer cause by reducing the spatial size, is addressed by a solution that Bailer et al. suggested [Bai+18]. In the next section, we will discuss how to get the expected descriptor matrix from a network that also uses pooling layers.

4.2.1 Fast Dense Calculation of Descriptors with CNNs that have Pooling Layers

Considering image I with width w and height h, we define a patch $p(x, y)$ with width p_w and height p_h centered at each pixel in I with positions $(x, y), x \in \{0, \dots, w - 1\}$, $y \in \{0, \dots, h - 1\}$. We assume that we pad the image I, so we can extract a patch around every pixel in the original image. The goal is to efficiently run a CNN C_p, which was trained by patches, on all patches $p(x, y)$ in I, at once by passing I once through the network.

The output matrix $O(h \times w \times d)$, which is the descriptors matrix O with dimensions $(h \times w \times d)$, contains $O(x, y) = C_p(p(x, y))$, which is a vector of size d, where d is the number of descriptors computed for each patch. Passing the image and calculation of the output matrix O is done by using the new CNN C_I that computes the desired output $O(h \times w \times d)$ from the input image I without performing the redundant operation that C_p does on the patches around adjacent pixels, as it runs for every patch independently. Figure 4.3 shows different architectures of C_I and C_p. Note that the desired output matrix $O(h \times w \times d)$ can be easily converted to the descriptor matrix with shape $(w \cdot h \times d)$, that CPM expects.

As we discussed earlier in this section, one can feed an image to the convolutional layers and get the output matrix with the desired shape, as the output of convolutional layers only depend on their input values and not the spatial position of the values, unlike in the case of pooling layers. Figure 4.4 shows the complication of pooling layers. The patches at different positions require

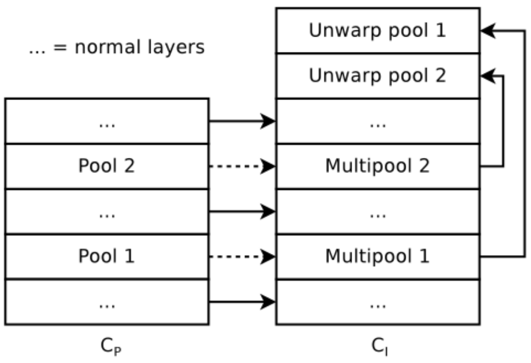


Figure 4.3: This figure shows the different architecture of networks C_p and C_I. C_p is like the networks we discussed in the last chapter, which has normal convolutional and pooling layers. C_I, on the other hand, has multipooling layers instead of pooling layers, and after the input goes trough all the convolutional and multipooling layer, there is an unwarping process. This figure is taken from Bailer et al. [Bai+18].

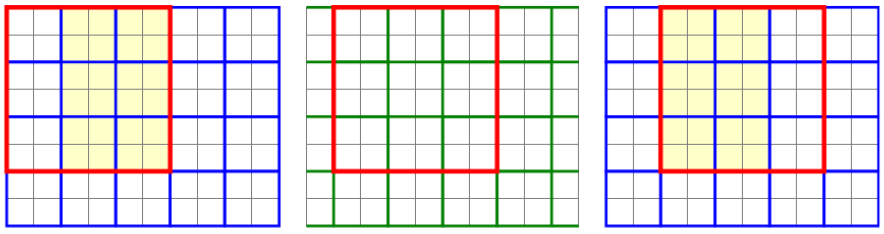
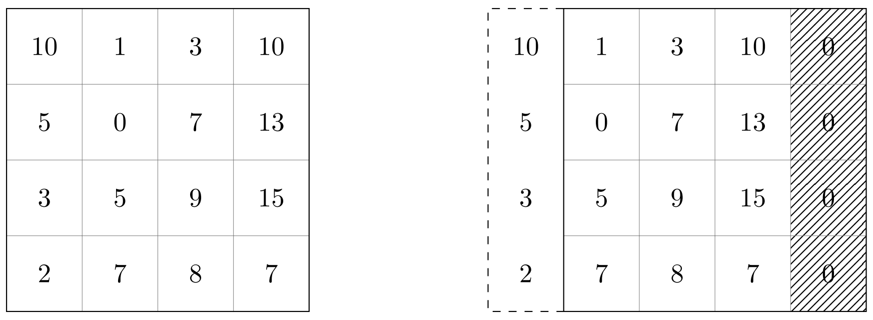


Figure 4.4: This figure shows that different patches in an image, may need different pooling operations, as the pooling operation depends on the position of the patch. Red patches in each figure show adjacent patches in an image. The red patches on the left and right figures have the same pooling situation and share the resulted values from pooling, computed from the yellow area. The red patch in the middle figure does not have the same pooling situation as its neighboring patches, therefore the output of the pooling layer for this patch is not shared with its neighbors. This figure is taken from Bailer et al. [Bai+18].

different pooling operation. Clearly, if we use max-pooling layer, for all patches the operation of computing the maximum value is the same, but as the windows, in which the maximum values are computed from, change based on the position of each patch, we consider it as a different pooling operation.

As shown in Figure 4.4, the left and middle red squares are neighbors, but with different pooling windows, however, not all the patches require different pooling operations, as there is a limited set of pooling windows.

Considering s as the stride parameter of a pooling layer, there are $s \cdot s$ different pooling situations that need to be computed independently. The patches $p(x, y)$ and $p(s \cdot a + x, s \cdot b + y)$, where a and b are integer values, share the same pooling windows, as in Figure 4.4, the left and right red squares have the same pooling windows and share the highlighted yellow parts.



(a) An image **(b)** The shifted image

Figure 4.5: The left figure shows an image. The right figure shows the same image which is shifted one pixel to the right. Note that the pixel-values of the shifted image are different than the image shown in the left figure, due to the shift operation.

Obviously, by a pooling layer with stride s in both x- and y-directions, the spatial dimension of the output will reduce to $(\frac{h}{s} \times \frac{w}{s})$ and to have the output matrix O with spatial dimensions of $(h \times w)$, we need to combine all $s \cdot s$ situations. To produce all $s \cdot s$ pooling situations, we should define a shift operation $shift_{y,x}(I)$ that shifts its input I by x pixels to the right and y pixels downwards. Figure 4.5 shows an example of shifting an image one pixel rightwards. Then a shifted pooling operation can be defined as follows:

$$pool^{x,y}_{s\times s}(I) = pool_{s\times s}(shift_{y,x}(I)), \tag{4.1}$$

where $pool_{s\times s}$ denotes a pooling operation with stride s in both x- and y-directions.

As shown before in Figure 4.3, in C_I instead of normal pooling layers, there are multipooling layers. A multipooling layer is defined as a set of shifted pooling operations with shift distances from 0 to $s-1$ in both x- and y-directions, as shown below:

$$L_I^{multipool} = \{pool^{0,0}_{s\times s}, pool^{0,1}_{s\times s}, \ldots, pool^{0,s-1}_{s\times s}, \ldots, pool^{s-1,0}_{s\times s}, \ldots, pool^{s-1,s-1}_{s\times s}\}, \tag{4.2}$$

where index I in $L_I^{multipooling}$ shows that this layer belongs to C_I.

After performing these pooling operations in the set, shown in the above equation, we should stack their outputs together. As we used only a 2×2 stride in pooling layers of our trained CNNs, let $s = 2$ for the rest of the section.

Considering a color image with shape $(h \times w \times 3)$ as the input of a network with one convolutional layer (with proper zero padding so that the spatial dimension of its input and output volumes stays unchanged) followed by a multipooling layer, we perform the shifted pooling operations $pool^{0,0}_{2\times 2}$, $pool^{0,1}_{2\times 2}$, $pool^{1,0}_{2\times 2}$ and $pool^{1,1}_{2\times 2}$ on the output of the convolutional layer. After stacking the results of those four shifted pooling operations, we get an output of the shape $(4 \times \frac{h}{2} \times \frac{w}{2} \times d)$, where d is the number of filters in the convolutional layer. If we add another convolutional layer followed by a multipooling layer, each of the four outputs of the shifted pooling operations (of the first multipooling layer) will be treated independently. We denote the first dimension of the output of the

```
1  translate_10 = [0, -1]
2  translate_01 = [-1,0]
3  translate_11 = [-1 ,-1]
4  def do_multipooling(ten):
5      pool_00 = max_pool_2x2(ten)
6      conv_01 = tf.contrib.image.translate(ten, translate_01)
7      pool_01 = max_pool_2x2(conv_01)
8  
9      conv_10 = tf.contrib.image.translate(ten, translate_10)
10     pool_10 = max_pool_2x2(conv_10)
11 
12     conv_11 = tf.contrib.image.translate(ten, translate_11)
13     pool_11 = max_pool_2x2(conv_11)
14 
15     output = tf.stack([pool_00, pool_01, pool_10, pool_11])
16     output = tf.reshape(output, (4 * ten.shape[0], tf.shape(output)[2], tf.shape(output)[3],
   tf.shape(output)[4]))
17     return output
```

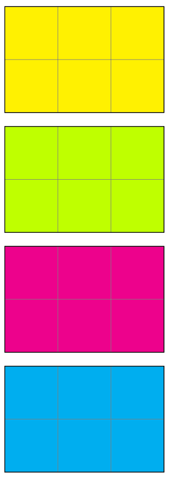
Listing 4.3: This listing shows how the multipooling layer is implemented using Tensorflow.

first multipooling layer by M, which is equal to 4, as the stride parameter $s = 2$. In fact, M is made up of two dimensions and can be interpreted as $M = (y \times x)$, due to the shifted pooling operations and we assume that the output of each shifted pooling layer is stacked with the order shown above (i.e. the value of y in the shift operation increases after pooling is performed for all shifts in x).

In the case of n multipooling layers, M changes to $(y_n \times x_n \times \cdots \times y_1 \times x_1)$, where x_1 and y_1 belong to the first multipooling layer and y_n and x_n belong to the nth multipooling layer. Importantly, what we discussed about different dimensions of M is just an interpretation of the outputs of multipooling layers, which will be discussed in the unwarping step. The output of multipooling layers have the mentioned four dimensions, in which M is equal to 4 in the output of the first multipooling layer with the shape $(4 \times \frac{h}{2} \times \frac{w}{2} \times d)$ and $M = 16$ in the output of the second multipooling layer, having the shape $(16 \times \frac{h}{4} \times \frac{w}{4} \times d')$, where d is the number of filters in the first convolutional layer and d' the number of filters in the second convolutional layer.

Listing 4.3 shows how multipooling is done using Tensorflow. First, max-pooling operation is done on the input tensor `ten` without any translations, at Line 5. Then the required shifts are performed on the input tensor followed by a max-pooling operation (in Line 6 to 13). Shifting the input tensor is done using the method `tf.contrib.image.translate`, which translates the input tensor by the translation parameter. The translation in x-direction is shown by the first element of the translation parameter and the second element represents the shift in y-direction. After performing the necessary translations, results of all max-pooling operations are stacked and the output is reshaped so that it has four dimensions, so we can feed the output of this multipooling layer to another convolutional layer. Because the input of the convolutional layers should be a four dimensional tensor.

Assume that there is an input color image of shape $(h \times w \times 3)$ and a CNN with two convolutional layers with 10 and 20 neurons for the first and second layer, respectively. Each convolutional layer is followed by a multipooling layer, and once again we assume that zero padding is used in the convolutional layers in such a way that they do not affect the spatial dimension of their input volume.


(a) Input of the unwarping process

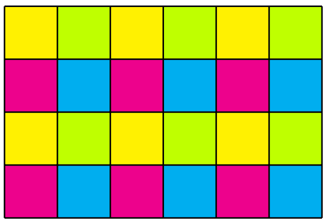

(b) The desired results

Figure 4.6: This figure shows how the values of the output of the last layer should be repositioned in the unwarping process, to yield the desired output matrix $O(h \times w \times d)$. This figure is inspired by a figure in Bailer et al. [Bai+18].

Firstly, the image should be reshaped to $(1 \times h \times w \times 3)$. After feeding the reshaped input image to the first convolutional layer, we get the output of shape $(1 \times h \times w \times 10)$. The output of the first multipooling layer will have the shape $(4 \times \frac{h}{2} \times \frac{w}{2} \times 10)$, as stride $s = 2$. Outputs of the second convolutional and multipooling layers are of the shapes $(4 \times \frac{h}{2} \times \frac{w}{2} \times 20)$ and $(16 \times \frac{h}{4} \times \frac{w}{4} \times 20)$. Finally, the results that we get from the last layer has to be unwarped. The unwarped output is the output matrix O that we expected to compute. In the next part, we explain the unwarping step.

The Unwarping Process

Assume the previously mentioned input color image of shape $(h \times w \times 3)$ and the lastly discussed CNN, but only with the first convolutional and multipooling layers, for simplicity. The output of this network has the shape $(4 \times \frac{h}{2} \times \frac{w}{2} \times 10)$, which we wish to unwarp to a volume with the shape $(h \times w \times 10)$. Figure 4.6 shows how the elements must be repositioned in such a way that the desired output is achieved.

The goal of unwarping is to get the desired output $O(h \times w \times d)$ (see Figure 4.6b) from a current output $W((M = s \cdot s) \times \frac{h}{s} \times \frac{w}{s} \times d)$ (see Figure 4.6a). Let $y^* = \frac{h}{s}$ and $x^* = \frac{w}{s}$. As we discussed before, M can be interpreted as two dimensions $(y_1 \times x_1)$, due to the shifts in multipooling layers, in both x- and y-directions. Note that first we explain how to unwarp the results of one multipooling layer, based on the new notation. The current output of the network can be represented as $W(s \cdot s \times y^* \times x^* \times d)$. By using a transpose operation we get $W(y^* \times x^* \times s \cdot s \times d)$ and it can be reshaped to $W(y^* \times x^* \times y_1 \times x_1 \times d)$. With another transpose operation between the second and third dimensions, we will get $W(y^* \times y_1 \times x^* \times x_1 \times d)$ and by fusing the first and second dimensions and the third and fourth dimensions by a reshape operation we get $W(h \times w \times d) = O(h \times w \times d)$.

Before generalizing this solution to several multipooling layers, we explain why we needed to do the first transposing step. At the beginning, by performing a reshape operation on $W(s \cdot s \times y^* \times x^* \times d)$ we get $W(y_1 \times x_1 \times y^* \times x^* \times d)$ and by transposing the second and third dimensions we get $W(y_1 \times y^* \times x_1 \times x^* \times d)$. If we fuse the first and second dimensions (see Figure 4.7b) and third and fourth dimensions, the result will be $W(h \times w \times d)$ which is not equal to $O(h \times w \times d)$, even though it has the same dimensions. Figure 4.7 shows why the first transpose operation, explained in the previous paragraph, was necessary to get the right results.

To generalize the upwarping process to n multipooling layers, we define $y^* = \frac{h}{s_1 \cdot \ldots \cdot s_n}$ and $x^* = \frac{w}{s_1 \cdot \ldots \cdot s_n}$, the current output of the network W can be written as

$$W(\underbrace{(s_1 \cdot s_1 \cdot \ \ldots \ \cdot s_n \cdot s_n)}_{M} \times y^* \times x^* \times d),$$

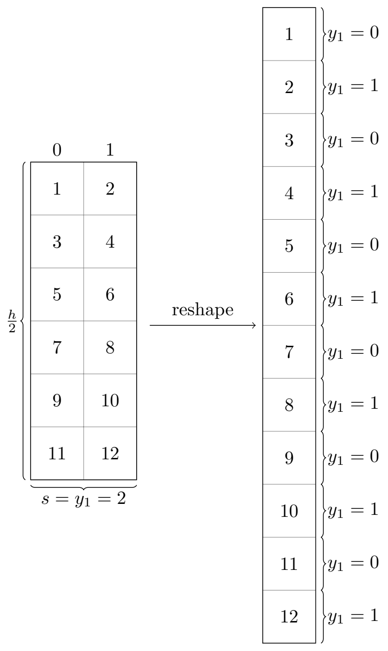
where s_1 and s_n represent the stride parameter in the first and nth multipooling layers, respectively. Then, by a transpose operation, we can get $W(y^* \times x^* \times M \times d)$ and, as for n multipooling layers M can be interpreted as $(y_n \times x_n \times \cdots \times y_1 \times x_1)$, by a reshape operation we can have $W(y^* \times x^* \times y_n \times x_n \times \cdots \times y_1 \times x_1 \times d)$. Then we should perform the following instructions n times:

1. Transpose the second and third dimensions.
2. Fuse the first and second dimensions and the third and fourth dimensions by the reshape operation.

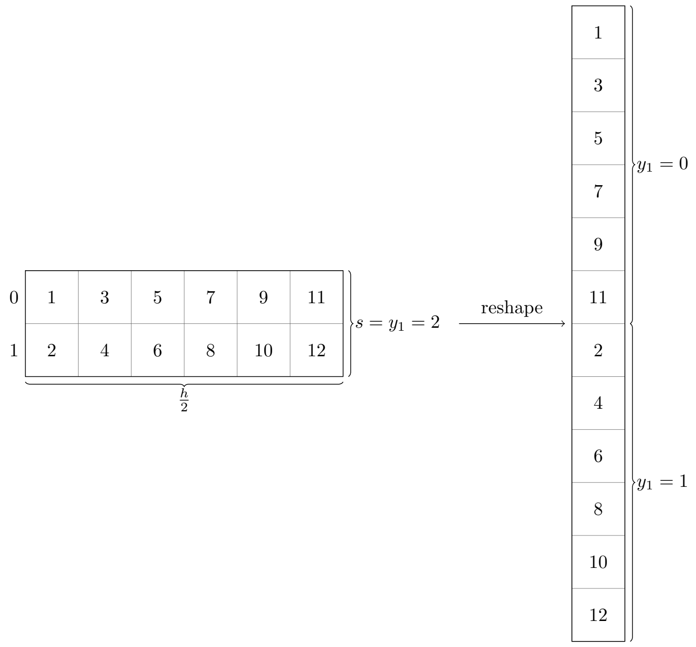
After performing these operations, the resulted output will be $W(h \times w \times d) = O(h \times w \times d)$. Listing 4.4 shows how the unwarping process is implemented using Tensorflow. Firstly at Line 2, the input of the function $W(M \times y^* \times x^* \times d)$, which is the output of the CNN, is transposed in such a way that the result becomes $W(y^* \times x^* \times M \times d)$. Then the number of multipooling layers is computed. The stride parameter in our setting was set to 2 for both x- and y-directions, for all pooling operations. The number of multipooling layers can be calculated by $\log_2(M)/2$. For example, in the case of two multipooling layers, $M = 16$ and n, which is computed at Line 4, is equal to 4. First, we should build the shape, to which we want to reshape our input tensor. The elements of this shape list are y^*, x^*, Then we append 4 times the value 2 and finally the value d. The resulted shape looks like $(y^* \times x^* \times 2 \times 2 \times 2 \times 2 \times d)$. Now we can reshape the input tensor to this shape.

At this point, we update the value of n by dividing it by 2, at Line 12. The new value of n shows the number of multipooling layers. In the loop over n at Line 14, a list, which shows how the transpose operation should be done, is created. For instance, for two multipooling layers, length of the dimensions of the input tensor, which is already reshaped to $(y^* \times x^* \times 2 \times 2 \times 2 \times 2 \times d)$ is equal to 7. In the first iterations of the loop, `list_shape_trans` $= [0, 2, 1, 3, 4, 5, 6]$ and the input is transposed according to the dimension orders in this list. As a result, the new shape of the tensor will be $(y^* \times 2 \times x^* \times 2 \times 2 \times 2 \times d)$.

After that, a tuple that explains how the reshape operation should change the shape of the input, is created. A list should be defined and the result of the multiplication of the first and second dimensions, and the third and fourth dimensions are appended to the list, at Lines 19 and 20. Then all other dimensions after the fourth dimension are appended to that list. In our example, `shape_for_reshape` is equal to $((y^* \cdot 2), (x^* \cdot 2), 2, 2, d)$ and the input is reshaped to this generated



(a)



(b)

Figure 4.7: This figure shows the right and wrong volume that has to be reshaped in the unwarping process. In both figures, the numbers in the left rectangles show the right order of components in the reshaped volume. In figure a, the volume, shown in the left rectangle, has the dimensions ($\frac{h}{2} \times 2$) and fusing its dimensions yields the volume with the right order of the elements, shown in the right rectangle. In figure b, the left volume has the shape ($2 \times \frac{h}{2}$) and fusing its dimensions does not lead to the elements with the right order.

```
def unwarp(last_layer_output):
    last_layer_output = tf.transpose(last_layer_output, perm=[1, 2, 0, 3])
    shape = last_layer_output.get_shape().as_list()
    n = int(np.log2(shape[2]))
    shape_prepare_unwarp = []
    shape_prepare_unwarp.append(tf.shape(last_layer_output)[0])
    shape_prepare_unwarp.append(tf.shape(last_layer_output)[1])
    for i in range(n):
        shape_prepare_unwarp.append(2)
    shape_prepare_unwarp.append(tf.shape(last_layer_output)[3])
    last_layer_output = tf.reshape(last_layer_output, tuple(shape_prepare_unwarp))
    n = int(n / 2)
    for i in range(n):
        list_shape_reshape = []
        n_dim = len(last_layer_output.shape)
        list_shape_trans = list(range(n_dim))
        list_shape_trans[1], list_shape_trans[2] = list_shape_trans[2], list_shape_trans[1]
        last_layer_output = tf.transpose(last_layer_output, perm=list_shape_trans)
        list_shape_reshape.append(tf.shape(last_layer_output)[0] * tf.shape(last_layer_output)[1])
        list_shape_reshape.append(tf.shape(last_layer_output)[2] * tf.shape(last_layer_output)[3])
        for j in range(4, n_dim):
            list_shape_reshape.append(tf.shape(last_layer_output)[j])
        shape_for_reshape = tuple(list_shape_reshape)
        last_layer_output = tf.reshape(last_layer_output, shape_for_reshape)
    return last_layer_output
```

Listing 4.4: This listing shows how the unwarping process is implemented using Tensorflow.

new shape. In the second iteration of the loop, again a list for the order of transposing dimensions is made : $[0,2,1,3,4]$ and the input is transposed accordingly and the resulting shape becomes $(y^* \cdot 2 \times 2 \times x^* \cdot 2 \times 2 \times d)$ and finally the tensor will be reshaped to the shape $(y^* \cdot 2 \cdot 2 \times x^* \cdot 2 \cdot 2 \times d)$ and the resulted output matrix with shape $(h \times w \times d)$ is achieved.

In the next section, we discuss how to integrate this fast approach to the network model, to be able to use it in the matching process in CPM.

Integration of the Fast Dense Descriptor Calculation

In the last section, we discussed how building blocks of the fast approach are implemented using Tensorflow. Importantly, using multipooling layers and also the unwarping process does not affect the training process. We trained the CNNs exactly as we explained in the last chapter and there was no multipooling layer or unwarping step in the training process. The approach that we explained in the last section is useful when we want to run the trained network to compute the descriptors.

To benefit from the fast approach, we need to make the mentioned changes, such as using multipooling layers and performing the unwarping step, in the graph that we save and freeze to export the complete model. The function `model_network` that we used before, to make the graph in Listing 4.1, can be updated by changing pooling layers to multipooling layers and adding an unwarping step before returning the output. To be more specific, in a program, a network with a certain architecture is trained and its trained variables are saved in a file. There is another program that prepares the model of the network with the same structure of the trained network in such a way that the number of

```
def model_network_fast(filter_size, input_depth, image, num_filters, pooling_order):
    num_hidden_layers = len(num_filters)
    hidden_conv = []
    hidden_pool = []
    for i in range(num_hidden_layers):
        if (i == 0):
                with tf.variable_scope("conv0"):
                    hidden_conv.append(
                    conv_relu(image, [filter_size, filter_size,
                    input_depth, num_filters[i]], [num_filters[i]]))
                    if pooling_order[i]==1:
                        pooling_output, num_imgs = do_multipooling(hidden_conv[i])
                        hidden_pool.append(pooling_output)
        else:

                with tf.variable_scope("conv" + str(i)):
                    if pooling_order[i-1] == 1:
                        hidden_conv.append(conv_relu(hidden_pool[-1], [filter_size, filter_size,
                        num_filters[i - 1], num_filters[i]], [num_filters[i]]))
                    elif pooling_order[i-1] == 0:
                        hidden_conv.append(conv_relu(hidden_conv[-1], [filter_size, filter_size,
                        num_filters[i - 1], num_filters[i]],[num_filters[i]]))
                    else:
                        print("hmm :( wrong pooling argument. The arguments should be
                        either 0 or 1.")
                    if pooling_order[i]== 1:
                        pooling_output, num_imgs = do_multipooling(hidden_conv[i])
                        hidden_pool.append(pooling_output)
    if pooling_order[-1]== 1:
        output = unwarp(hidden_pool[-1])
    else:
        output = unwarp(hidden_conv[i])
    return output
```

Listing 4.5: This listing shows how the function `model_network`, presented in Listing 3.1, is modified to replace the pooling layers by multipooling layers. The unwarping process is called from this network so that the out put of the graph has the desired shape.

convolutional layers, number of neurons per convolutional layer and all trainable variables and the number of pooling layers are the same, but pooling layers are replaced with multipooling layers and the output of the network is unwarped to yield the desired output matrix.

Listing 4.5 shows how the `model_network` is updated to support fast calculation of descriptors by using functions `do_multipooling` and `unwarp` that we explained earlier in Listing 4.3 and Listing 4.4, respectively. Importantly, now in Listing 4.1, the new function `model_network_fast` should be called, instead of the function `model_network`. So that the new graph structure is coupled with values of the trained variables and the complete portable model is made, as we showed in Listing 4.1. As we claimed before, finding the matches between two images of size (436×1024) via learned descriptors for a network with seven layers consisting of number of filters smaller than 50 for each layer, with the common strategy,which was discussed in Section 4.1, it took more than 30 minutes. With the mentioned fast strategy mentioned in this section, using the same 7-layer network, the matching task for the two images of the same size as before takes only about 8 seconds. By using parallel loops implemented using OpenMP [Ope18] in converting the image to a tensor and also filling the final descriptor matrix, that time reduces to about 3.5 seconds, using the same system as before.

Empirical Observations

So far, we discussed only CNNs that had zero padding in their convolutional layers so that the input and output of the convolutional layers have the same spatial dimensions. Also for the discussed cases, we considered doing zero padding for pooling layers, as well. As the networks we trained used such zero padding, in order to use the network to compute the descriptors properly, their inputs should be also patches that can be padded in the network.

In Section 4.1, we explained how a network was loaded and used to compute descriptors for the image patches. In that section, we also explained that patches around each pixel should be extracted, in the image and they should be passed through the network. The trained network computed the descriptors for each patch.

If the network is trained by doing zero padding for each patch in the training samples, there is no surprise if it does not give proper results when we pass a whole image to it. When a network is trained with layers that use zero padding, the values of its trainable variables get affected by the zeros that were padded to the patches of the training samples, and the network is expected to work properly when it is fed by inputs that can be padded by the network in the same way as its training samples. Considering a network getting trained with patches that were zero padded by the network, if we feed a large image to the network, the large image gets padded and there are no zeros around each patch in the large image.

In other words, if we consider a patch separately and the same patch in a large image, and the network does zero padding in its layers, the result of the network for the same region (patch) becomes different, in those two cases. Because if the patch itself is padded the network has a different output, compared to the case that the large image is padded and the network is run for the large image. The output of the network for a specific patch should be the same, if the patch is fed to the network separately or if it is a part of a larger image, however, if the the layers in the network use zero padding, this will not be the case (i.e. output of the networks C_p and C_I for the patch centered at (x, y), which are $C_p(p(x, y))$ and $O(x, y)$, respectively, will not be equal). Therefore, to benefit from the fast approach correctly, we must not train the networks that use zero padding in their layers.

The descriptors that were made with the fast approach by networks that performed zero padding in their layers were not as helpful (i.e. did not lead to high optical flow accuracy), compared to the case that the descriptors were computed via the same networks, but with the common approach explained in Section 4.1. Obviously, the reason is that in the common approach, patches are fed to the networks, which expect patches as input so that they can use zero padding in their layers and perform the operations, as networks were trained having layers that used zero padding on the patches in their training process.

Undoubtedly, doing zero padding, especially in convolutional layers causes a lot of convenience, such as more freedom in designing a network, as the spatial size of the input and output of the network can remain unchanged. If we do not use zero padding in convolutional layers, as the input goes through each layer, the spatial size decreases. For instance, if we have patches with spatial size of (16×16), by a receptive field size of (3×3), we can only have up to 7 convolutional layers in our network, as in every layer, the spatial size is reduced by 2 pixels, in each direction.

Clearly, by insertion of pooling layers the number of possible convolutional layers in the network decreases, in most cases. For example, if we insert a pooling layer after the first convolutional layer, we can have only 4 convolutional layers, as:

$$16 \times 16 \xrightarrow{C1} 14 \times 14 \xrightarrow{P1} 7 \times 7 \xrightarrow{C2} 5 \times 5 \xrightarrow{C3} 3 \times 3 \xrightarrow{C4} 1 \times 1,$$

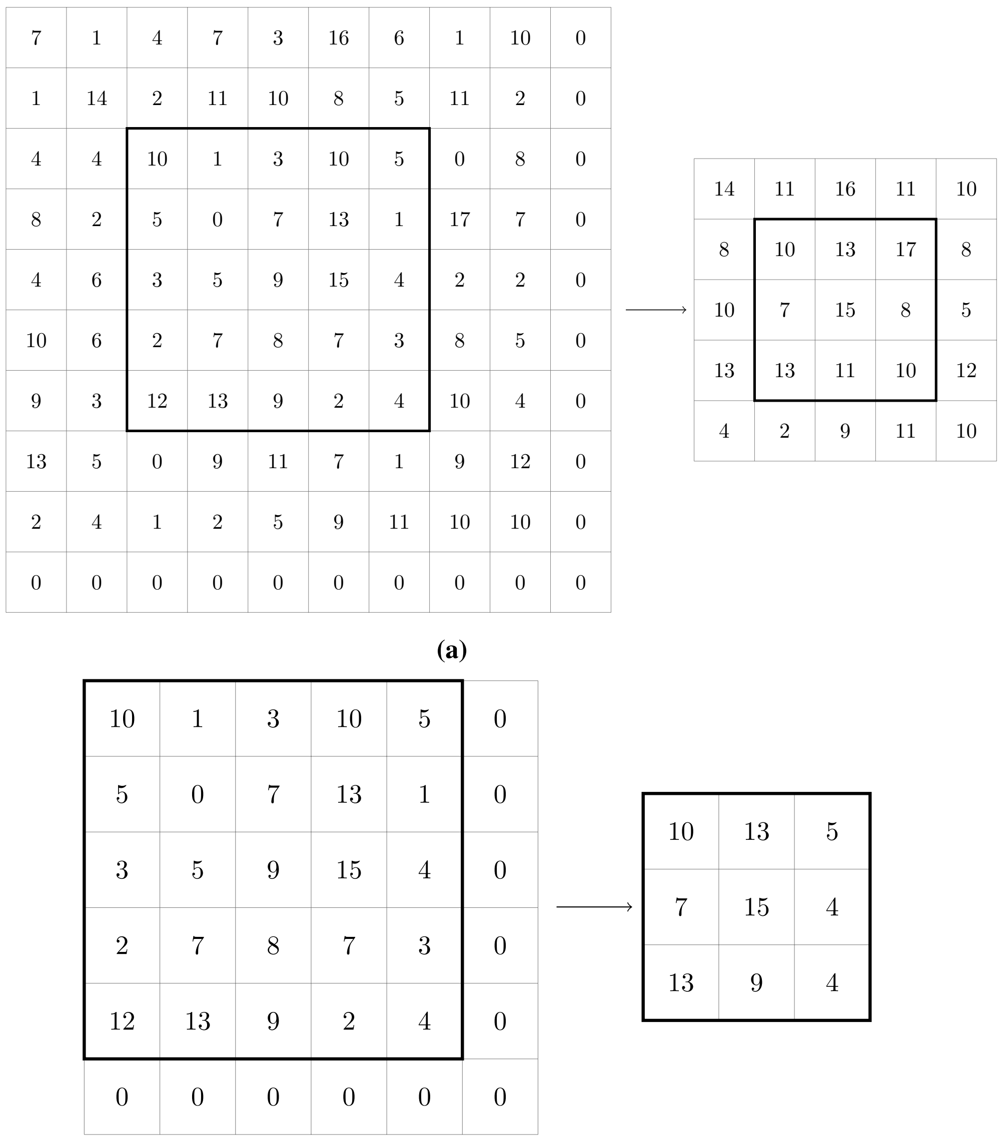
where $C2$ to $C4$ represent the first and 4th convolutional layers, respectively and $P1$ shows the first pooling layer, and the numbers between arrows show the spatial size of the volume, at that point. At the beginning, the spatial size of the patch is equal to (16×16), and as we chose the receptive field size to be (3×3), based on equation Equation (2.29) (with parameters p = 0, s = 1), the spatial size of the volume is reduced by 2 pixels, after passing each convolutional layer.

Clearly, in the above-mentioned case and most other cases, using pooling layers reduces the possible number of convolutional layers, as the spatial size gets smaller by each layer, but there can be cases, in which the possible number of convolutional layers stay unchanged. As an example, it is possible to have a CNN with 7 convolutional layers and a pooling layer after the 7th convolutional layer, as:

$$16 \times 16 \xrightarrow{C1} 14 \times 14 \xrightarrow{C2} 12 \times 12 \xrightarrow{C3} 10 \times 10 \xrightarrow{C4} 8 \times 8 \xrightarrow{C5} 6 \times 6 \xrightarrow{C6} 4 \times 4 \xrightarrow{C7} 2 \times 2 \xrightarrow{P1} 1 \times 1.$$

Of course, without the pooling layer the spatial size of the output would be equal to (2×2), but adding a pooling layer in this case did not limit the possible number of convolutional layers.

Despite the limitations of not using zero padding in convolutional layers, we have to train our networks without zero padding. Also in pooling layers, doing zero padding may cause problems, however, we normally perform max-pooling and the padded zeros themselves do not directly affect the pooling results. Now let us discuss why using zero padding in pooling layers is not the right thing to do. Assume that we use patches of size (5×5) for training, and the network only consists of one convolutional layer and one pooling layer. For simplicity, we assume that the convolutional layer does zero padding, so it does not affect the spatial dimension of the input image, but the pooling layer uses zero padding. The spatial size of the output of the pooling layer will be equal to (3×3). If we perform zero padding while training the network, we use the information from a larger region than the patch region (see Figure 4.8b). This becomes problematic if we train the the network with the zeros padded to the volume, and use the network without the zeros padded. Essentially, if we have a single patch or the same patch in a larger image, the computed descriptors for them must be identical. Using the fast approach, we pass a large image to the network and if we perform pooling, there are no zeros padded per patch in that large image and instead of zeros that were padded during the training process, the pixel values of the neighboring regions are taken into account (see Figure 4.8a). Consequently, if we perform zero padding while training, with the notation defined in Section 4.2.1, the output of the networks C_p and C_I for one patch, which are $C_p(p(x, y))$ and $O(x, y)$ centered at (x, y), respectively, will not be equal. Therefore, we do not use zero-padding in our networks, neither in convolutional layers, nor in pooling layers.



7	1	4	7	3	16	6	1	10	0
1	14	2	11	10	8	5	11	2	0
4	4	10	1	3	10	5	0	8	0
8	2	5	0	7	13	1	17	7	0
4	6	3	5	9	15	4	2	2	0
10	6	2	7	8	7	3	8	5	0
9	3	12	13	9	2	4	10	4	0
13	5	0	9	11	7	1	9	12	0
2	4	1	2	5	9	11	10	10	0
0	0	0	0	0	0	0	0	0	0

$\longrightarrow$

14	11	16	11	10
8	10	13	17	8
10	7	15	8	5
13	13	11	10	12
4	2	9	11	10

(a)

10	1	3	10	5	0
5	0	7	13	1	0
3	5	9	15	4	0
2	7	8	7	3	0
12	13	9	2	4	0
0	0	0	0	0	0

$\longrightarrow$

10	13	5
7	15	4
13	9	4

(b)

Figure 4.8: This figure shows a patch in a larger image (a-left) and separately (b-left). The result of max-pooling with zero padding is different for identical patches, considered in a large image or separately. The results of max-pooling operation using zero padding is shown in for each case. If we perform max-pooling using zero padding on the larger image, the larger image will be padded, not that patch in the larger image. The output of max-pooling for the patch in the image will have the spatial size (3×3) (as shown with black border in a-right) and to perform max-pooling, some pixels outside the patch itself, which are the pixels of the neighboring patches, are taken into account. If we consider the patch separately, the patch itself is padded and only extra zeros are considered in the max-pooling operation, which do not affect the output values of the max-pooling operation.

4.2.2 Extension of the Fast Approach

In Section 4.2.1, we discussed the approach of Bailer et al. [Bai+18] to calculate descriptors of an image by a single pass through a trained network. Clearly, the descriptors calculated for each patch were assumed to be a d-element vector. This assumption limits the possible set of architectures. For instance, assuming the case discussed in the last section, given a (16×16) patch, If we use a network with 7 convolutional layers with receptive field of size (3×3), the spatial dimension of the output descriptor will be equal to (2×2), as:

$$16 \times 16 \xrightarrow{C1} 14 \times 14 \xrightarrow{C2} 12 \times 12 \xrightarrow{C3} 10 \times 10 \xrightarrow{C4} 8 \times 8 \xrightarrow{C5} 6 \times 6 \xrightarrow{C6} 4 \times 4 \xrightarrow{C7} 2 \times 2.$$

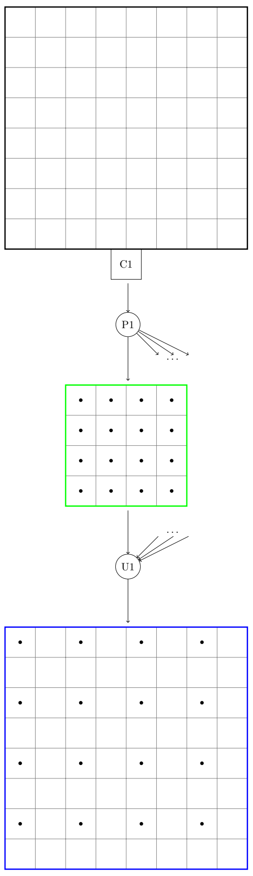
In the fast approach discussed in Section 4.2.1, for every pixel (or the patch around it) a set of descriptors of shape $(1 \times 1 \times d)$ was considered. We want to know if we can use the fast approach to calculate the descriptor matrix from a network with an architecture similar to the above-mentioned network, where the descriptors of each patch have spatial dimensions larger than (1×1). We will show that the descriptors with any spatial dimensions can be extracted from the descriptor matrix $O(h \times w \times d)$, that is produced by the fast approach. Note that in this case, the final descriptor matrix that CPM expects should have the shape $(w \cdot h \times d \cdot s_d \cdot s_d)$, where $(s_d \times s_d)$ is the spatial dimension of the descriptors computed for one patch.

Figure 4.9 shows a patch of size (8×8) that goes through two different CNNs. For simplicity, we assume that the input and output of convolutional layers have identical spatial dimensions. For each network, two cases of C_p and C_I are taken into account, which means: first we consider the input image as input patch to a normal CNN (C_p) with convolutional and pooling layers and in another case, we take the patch into account as input of C_I with convolutional and multipooling layers followed by an unwarping step, at the end. In the first network (see Figure 4.9a), dots in the (4×4) volume with the green border show the values of the output of the network C_p, given the image-patch. Notably, considering the network C_p there is no multipooling layer, but a simple pooling layer, however the output of C_p is equal to the output of $pool(shift_{0,0}(I))$, where I is the input of the multipooling layer which is the output of the first convolutional layer. If we pass the input patch through the network C_I, the output will be the unwarped (8×8) volume with the blue borders. Interestingly, the black dots in the unwarped volume show the values of the elements of the output of C_p. Clearly, one can extract the values of the output of C_p from the output of C_I. For the second CNN (see Figure 4.9b), there are two convolutional and multipooling layers. From Figure 4.9, by observing both networks, we can see that the more pooling layers in C_p, the smaller will be the spatial size of the output and the further are the elements of the output of C_p from each other, in the output of C_I. Note the positions of the elements of the output of C_p in the output of C_I.

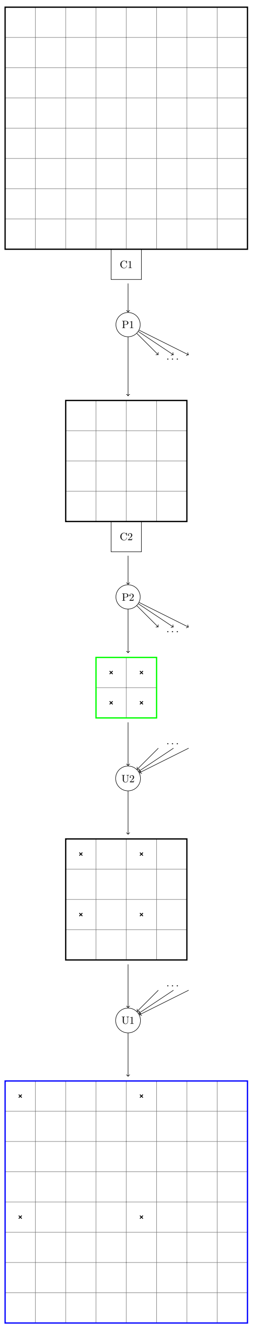
At this point, an important task is to specify the positions of the elements of the computed descriptors of a patch from the unwarped descriptor matrix of a larger image.

Figure 4.10 shows an example of this scenario. In this case, the goal is to extract the descriptors of the (8×8) patch shown in the larger image. If we consider a network with two pooling layers, the descriptors computed for the patch will have the spatial size of (2×2), as:

$$8 \times 8 \xrightarrow{P1} 4 \times 4 \xrightarrow{P2} 2 \times 2.$$



(a)



(b)

Figure 4.9: This figure shows the position of the computed descriptors in the unwarped volume. The small rectangles, with the letter C inside, show the convolutional layers and the circles having the letter P inside show the multipooling layers. The circle with the letter U shows the unwarping processes. To avoid visual clutter, only one (out of four) output of multipooling layers, which is the result of pooling operation with no shift in x- and y-directions, is shown.

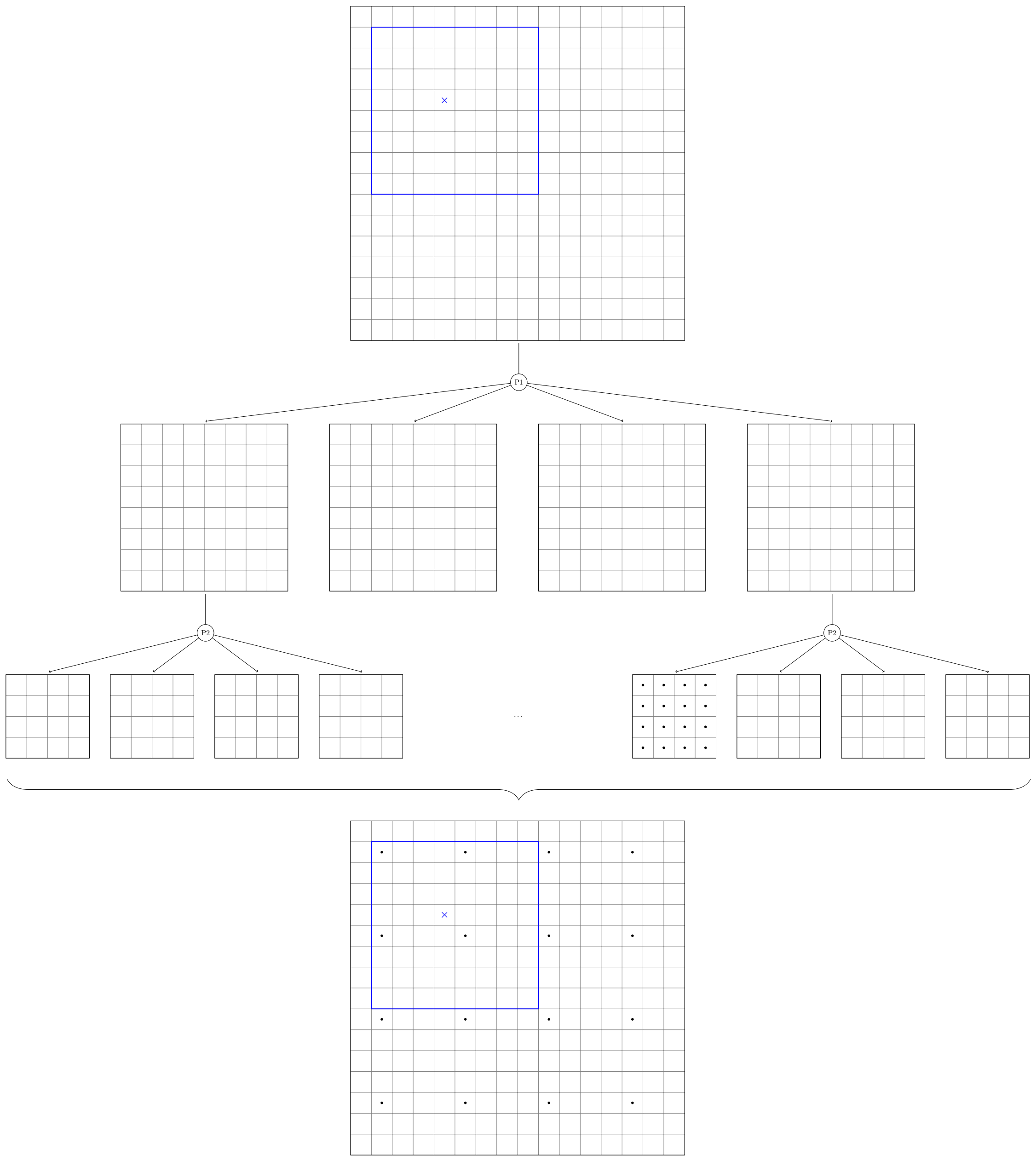


Figure 4.10: This figure shows an image going through a CNN with two convolutional and multipooling layers. To have a simpler figure, convolutional layers are not shown. The elements in output of the second multipooling layer that contributes to computation of the descriptors of that blue patch specified in the input image, is marked with dots. The positions of those elements are marked in the unwarped (16×16) volume, at the bottom. This is further described in Section 4.2.2.

Again for simplicity, we assume that the input and output of the convolutional layers have the same spatial size. Note the position of the patch in the image. Clearly, in the first multipooling layer, output of $pool(shift_{1,1}(I))$ and in the second multipooling layer, output of $(pool(shift_{0,0}(I')))$ contribute to computation of descriptors of that patch, where I is the output of the first convolutional layer and I' is the output of the second convolutional layer, given the input $pool(shift_{1,1}(I))$. Note that the convolutional layers are not shown in the figure, to make the figure simpler. In the unwarped result, elements of the output of $pool(shift_{0,0}(I'))$ (from the second multipooling layer) are marked. The marked elements in the (8×8) window, shown in the unwarped volume, are the (2×2) descriptors of the (8×8) patch specified with blue borders in the input image. Importantly, the center of the patch has the same spatial coordinates as the same window marked in the unwarped volume, in which the wanted descriptors are placed. Easily, by extracting the value of every fourth pixel, in both directions, as the network has two multipooling layers, from the (8×8) area with the blue borders shown in the unwarped volume, we get the descriptors that were computed for the specified (8×8) patch in the input image. Algorithm 4.1 shows how to extract the descriptors from the output matrix $O(h \times w \times d)$ for all the pixels (or the patches around each pixel) in the input image. Note that the output matrix that CPM expects is also filled. Depending on the spatial size of the descriptors computed for each pixel (or the patches around each pixel), the final output matrix is filled.

In this chapter, we explained how the descriptors are computed efficiently by passing a large image through the network, at once. As we discussed before, CPM uses the descriptor matrix to compute its cost function. By using CPM to find the matches between two images, we obtain a non-dense flow field. Then we take advantage of the interpolation technique used in Epicflow to compute a dense flow field. The obtained flow is then compared to the ground truth optical flow which is available by the datasets we used. The result of this comparison is a measure of robustness of the descriptors calculated from the networks that we trained. There are many things to change in a network to improve the resulting descriptors. In the next chapter, we present different networks and their resulting optical flow computed by the learned descriptors.

Algorithm 4.1 Extract features

```
let $w$ and $h$ be the width and height of the input
let $(s_d \times s_d)$ be the spatial size of the descriptors computed for one patch
let $d$ be the number of channels that each descriptor has.
let $n$ be the number of multipooling layers

function ExtractAllFeatures($O$)                    // $O$ is the descriptor matrix $O(h \times w \times d)$
    let $output(w \cdot h \times d \cdot s_d \cdot s_d)$ be the final output matrix that CPM expects.
    for $x = 0$ to $w - 1$ do
        for $y = 0$ to $h - 1$ do
                // Extract the descriptors for the pixel at $(x + hps - 1, y + hps - 1)$ or the patch
              // around that pixel from $O$ and fill it in $output$, where $hps$ is half of the patch size.
            ExtractFeatures($output, O, x, y$)
        end for
    end for
    return $output$
end function

function ExtractFeatures($output, O, x, y$)
                              // $(x, y)$ are the coordinates of the top-left of a patch region in $O$.
    $c = y + x \cdot w$                           // $c$ is the column of the output for $(x, y)$.
    $r = 0$                  // $r$ is the row of the current descriptor element in the output
    for $i = 0$ to $s_d - 1$ do
        for $j = 0$ to $s_d - 1$ do
            for $k = 0$ to $d - 1$ do
                $output(c, r) = O(x + s^n \cdot i, y + s^n \cdot j, k)$
                $r = r + 1$
            end for
        end for
    end for
end function
```

5 Evaluation

Previously, we explained how to define and train a CNN to compute descriptors. We also explained how to compute the descriptors in an efficient way and use them for optical flow estimation between two images using the CPM algorithm. In this chapter, we compare different networks that we designed and trained. These CNNs are different in their architectures or the way they are trained. First, we explain some general facts about these experiments. Then we introduce different networks and we show how well the descriptors, resulted from each network, contribute to optical flow estimation of the pairs, in the evaluation set we defined in Section 3.3. After that, we evaluate the network that worked best in our evaluation set, on the whole MPI-Sintel and KITTI 2015 datasets, for the image-pairs that the ground truth optical flow value is available.

As explained in Section 3.1, we prepared training samples including patches of both (16×16) and (64×64) sizes. Size (64×64) seems to be a reasonable choice for this task, as some literature in this field [Bai+17; GW16] suggested similar sizes. However, training a network with patches of that size is time-consuming. For example, for about 500,000 training samples of (64×64), training a relatively simple CNN with seven layers, where each layer has less than 50 neurons, for five epochs per training sample takes about a day using Intel Core i7 2.60 GHz $\times 8$, 16 GB RAM, Nvidia GTX 860M. Gadot et al. [GW16] claimed that larger patch sizes are more useful for training a CNN; however, as a part of our goal is to compare a lot of different networks, we should not choose a setup with which training each network is too time-consuming.

With the training process discussed in Section 3.3, using training samples of size (16×16) takes less than eight hours for the same number of training samples, the same architecture and the same number of epochs per training sample. In some cases, where the training samples were placed in RAM after all training samples are trained once, the training process with the above-mentioned setup even takes less than an hour. Not all training processes that used (16×16) patches took less than an hour, but we could clearly observe that fast training process many times.

For all the experiments, the CNNs are trained with patches of size (16×16), as the training process with patches of this size is not as time-consuming as the case that the networks are trained with the patches of size (64×64). Note that we did not train any network with the training samples consisting patches of size (64×64) that we created before discussed in Section 3.1. The size of the receptive field is chosen to be (3×3) in all convolutional layers in all the networks, and as we discussed in the previous chapter, the patches are not padded by zeros during the training process by convolutional or pooling layers.

In the literature [Bai+17], usually architecture of a CNN is represented in tabular form. We compare many networks and many of them have the same architecture and are very little different in the way they are trained, therefore to introduce every network in detail, we do not use tables. Instead, we assign a long label which describes important aspects of the networks that we consider in that set of experiments.

Layer	1	2	3	4	5	6	7	8
Type	Conv	Conv	Conv	Conv	Conv	Conv	Conv	Max-pool
Input spatial size	16×16	14×14	12×12	10×10	8×8	6×6	4×4	2×2
Input depth	3	10	15	20	25	30	35	40
Output depth	10	15	20	25	30	35	40	40
Filter size	3×3	3×3	3×3	3×3	3×3	3×3	3×3	2×2
Stride	1	1	1	1	1	1	1	2
Non-linearity	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	-

Table 5.1: Architecture of a CNN with seven convolutional layers and one pooling layer. Convolutional layer is represented by 'Conv' and the max-pooling layer is shown by 'Max-pool'.

For example, for a CNN having seven convolutional layers, which has 10, 15, 20, 25, 30, 35 and 40 filters for the first to the seventh convolutional layer, respectively, with relu non-linearity, the label we assign to this network could be a string which has the substring '10-15-20-25-30-35-40_relu'. If we insert a pooling layer after the last convolutional layer, the substring will change to '10-15-20-25-30-35-40P_relu'. By this label, we could easily see the number of convolutional layers, which is equal to seven, and the number and order of pooling layers. Of course, for very deep CNNs, this kind of labeling is not the best choice, but in our experiments there are no networks with more than seven layers. As we said before, in all the networks we designed and trained, the size of the receptive field for all convolutional layers is set to (3×3) and size of the patches in our training samples is (16×16) and based on the equation Equation (2.29), in every convolutional layer the spatial size decreases by two pixels in each direction, therefore, there is no need to repeat this information by presenting the network by a table such as Table 5.1. Therefore, instead of presenting the aforementioned network with Table 5.1, we just refer to it as '10-15-20-25-30-35-40P_relu.'

Of course, we could make the label more complete, by adding substrings of all important details. For instance, for a trained network, sometimes we like to know how many epochs (iterations) per training sample results in better descriptors (in a sense that the descriptors lead to better optical flow estimation having a smaller difference to the ground truth). For a training process with epoch = 5 per training sample, the label could change to '10-15-20-25-30-35-40P_relu_ep5'. Of course, there are other factors in the training process that affect the resulting network, such as the loss function, optimizer, learning rate and training samples that were used in the training process. Based on all these factors that were used for training, we can concatenate the proper substring to the label, but to avoid names that are too long and confusing, we leave out some of these factors from the labels and specify them separately in each experiment set.

As there is no specific way to design and train a network that works best for all tasks, we just start using the above-mentioned architecture to do a few experiments.

Initial Experiments

In the first set of experiments, we want to consider the effect of the number of epochs per training sample on the learned descriptors. The measure we use to compare the descriptors resulted from the trained network is the average of AEE values of the estimated optical flow on the evaluation

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
10-15-20-25-30-35-40P_**ep3**	2.72	3.11	7.45
10-15-20-25-30-35-40P_**ep5**	**2.40**	**2.71**	**7.21**
10-15-20-25-30-35-40P_**ep7**	2.62	2.96	7.89
10-15-20-25-30-35-40P_**ep10**	2.59	2.97	7.78

Table 5.2: Comparison of different networks trained with different number of epochs per training sample.

set. Note that the optical flow in all experiments is estimated using descriptors, which are used to compute the cost function, used in the CPM algorithm. In other words, CPM computes the matches using the descriptors and then the dense flow field is obtained using Epicflow. This dense estimated flow field should be compared to the ground truth optical flow and the value of AEE is computed for each image-pair. Then we compute the average of all AEEs of all image-pairs in our evaluation set.

In all experiments, the `minimum-width` parameter in CPM to create the image pyramids was set to 70 pixels, which means for images in MPI-Sintel dataset, there are three levels in the image pyramid and for KITTI 2015 images there are four levels in the image pyramid constructed by CPM algorithm.

We represent the average of AEE values of all pixels and non-occluded pixels of MPI-Sintel images and non-occluded pixels of KITTI 2015 images in the evaluations set. Note that for each image pair, the value of AEE is computed. And then, to have a measure of all the pairs in our defined evaluation set, we compute the average of all AEE values.

We do not present the average of AEE values for all pixels in KITTI 2015 images, as the value of AEE did not change notably in our experiments. Therefore those values were not helpful in deciding what parameters to choose to get better descriptors.

Table 5.2 shows the networks trained with different number of epochs per training sample. The AEE of the flow field resulted from using the descriptors computed by the network trained with epoch value of 5 seems to be smaller in all cases, however we can not conclude that by increasing the number of epochs the results will become necessarily worse, as the network trained with 10 epochs led to smaller AEE than the network with 7 epochs per training sample.

Note that in all tables that show a set of experiments, bold values show the smallest average AEE value in that set (column), and to make the comparisons easier, the differences among different networks (e.g. in their structure or the training parameters) are shown with bold fonts. Also for tables that contain more than one group of experiments, in each group the best results have bold fonts and the best results in all groups is underlined.

For the mentioned set of experiments, we used Adam optimizer and learning rate value of 10^{-6}. We chose this learning rate, as it was the largest learning rate that led to stable optimization, in the training process of networks, defined in our previous research [Jah17]. We also used relu non-linearity for all convolutional layers. Using Adam optimizer, the mentioned learning rate value and relu as the activation function are the default optimizer, step-size value and non-linearity,

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
10-15-20-25-30-35-40P_ep5	2.72	3.11	7.45
10-15-20-25-30-35-40P_**bn**_ep5	1.39	1.96	7.02
10-15-20-25-30-35-40P_**norm**_ep5	2.50	2.82	7.57
10-15-20-25-30-35-40P_**bn_norm**_ep5	1.35	1.80	**6.77**
10-15-20-25-30-35-40P_**bn_npi**_ep5	**1.19**	**1.73**	7.88

Table 5.3: Comparison of different networks trained with different normalization strategies.

therefore they do not appear in the label of the networks, unless we change them. Note that earlier in this chapter, we specified the relu non-linearity in the label of the network, but our goal was only to show how we add extra specifications of a network to the label assigned to a network.

Also the training samples that are used in training were extracted with the first strategy, discussed in Section 3.1 with the distance of 3 to 8 pixels between the center of non-matching patches and the center of the corresponding patches. We also do not mention this in the label of networks, unless we use another set of training samples.

Next, we compare networks that use different normalizations. There are different ways of normalizing the training samples while the network is being trained. One possibility is normalizing the patches by mean and standard deviation of all the images in a dataset. Another way is to normalize the patches by mean and standard deviation of each image that the patch is extracted from. Note that if a network is trained with the normalized patches, the input images of such trained networks should be normalized in the same way. In the experiments represented in Table 5.3, we kept the number of epochs fixed and equal to 5, as it led to the smallest value of average AEE. In this set of experiments, the positive effect of batch normalization on resulting AEEs is clear. The substring 'bn' shows that the network uses batch normalization in all of its convolutional layers. The substring 'norm' shows that the network is trained with the patches that were normalized by mean and standard deviation of the whole KITTI 2015 and MPI-Sintel datasets. Clearly, the images in the evaluation set, that were fed to the network to obtain the optical flow, were also normalized with the same mean and standard deviation values.

Another substring is 'npi' (normalization per image), which explains that the training patches were normalized by mean and standard deviation of the image that the patch was extracted from. Again, the images in the evaluation set, of which the network computes the descriptors, were also normalized by the their mean and standard deviation during the evaluation process.

In the first two experiments in Table 5.3 are only different in their architecture, as the first one does not use batch normalization in its convolutional layers and the second one does. As we saw that the second and fourth networks with batch normalization leads to much smaller value of average AEE, we did not do an experiment with normalization per image (npi) without using batch normalization. Of course, to find the best network, one should test all different combination of parameters and architectures, but as our time was limited, we often used a parameter that worked best for one network, also for other networks. Of course, we tried many different combinations, but obviously there is no guarantee that we chose the best architecture and parameters.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
10-15-20-25-30-35-40P_bn_npi_ep5	1.19	1.73	7.88
10-15-20-25-30-35-40P_bn_npi_**copy**_ep5	1.53	1.99	7.50
10-15-20-25-30-35-40P_bn_npi_**noShuf**_ep5	**1.00**	**1.37**	8.46
10-15-20-25-30-35-40P_bn_npi_**noShuf_copy**_ep5	1.11	1.60	**6.58**

Table 5.4: Comparison of different networks trained with the same architectures and parameters, but with different order of training samples and different initializations. All networks were initialized differently and the first two networks were trained with different order of training samples. The order of training samples was kept fixed in the training process of the third and fourth network, as the training samples were not shuffled after each epoch.

In this experiment set, the smallest value of average AEE was observed in the fifth experiment, which is the network using batch normalization and trained with patches that are normalized with mean and standard deviation of the images that the patch was extracted from.

So far, all the networks that we designed and trained were initialized randomly. Also after each epoch, the training samples were shuffled. In Table 5.4 we show that with the same architecture and parameters, but with different order of training samples and a different initializations, the resulting values of average AEE vary notably. Note that we used the network 10-15-20-25-30-35-40P_bn_npi_ep5, as it had the smallest AEE in Table 5.3 for MPI-Sintel image-pairs. Essentially, we consider the parameters that lead to a decrease in average AEE values, but often the parameters that lead to a reduce in the value of average AEE for MPI-Sintel image-pairs are different than the ones that lead to a decrease the value of AEE for KITTI 2015 image-pairs. We decided to choose the parameters that lead to a smaller AEE value for MPI-Sintel image-pairs, for most cases. If we also considered the parameters that lead to smaller value of average AEE for KITTI 2015 images in the evaluation set, the number of experiments would increase dramatically.

The substring 'noShuf' in Table 5.4 shows that in the training process of that network, training samples were not shuffled, after each epoch. This case is important for us, as we wanted to know if the order of training samples changes the resulted average AEE. Moreover, we want to know by using the same architecture and parameters and by keeping the order of the training samples fixed, how the results will change. The substring "copy"means that the network is trained for the second time with the same architecture and parameters.

Table 5.4 shows that if networks are initialized differently and trained with different orders, they can lead to very different AEE values. If we keep the order of training samples fixed, with different initializations, still the results change. Therefore, to have a systematic strategy to find a network that is able to compute useful descriptors, one has to keep the order of training samples, as well as the initialization, fixed. Of course, we cannot keep the initialization fixed for networks with different architectures, as the trainable variables must correspond to each other to be able to restore the same initial values for the same variables.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
init1_10-15-20-25-30-35-40*P*	2.35	2.72	7.02
init2_10-15-20-25-30-35-40*P*	**2.25**	**2.56**	**6.52**
init3_10-15-20-25-30-35-40*P*	2.26	2.58	6.55
init4_10-15-20-25-30-35-40*P*	2.41	2.71	7.03

Table 5.5: Comparison of different networks that are only randomly initialized, without being trained.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
init1_10-15-20-25-30-35-40P_bn	1.33	1.77	7.11
init2_10-15-20-25-30-35-40P_bn	**1.21**	**1.65**	6.87
init3_10-15-20-25-30-35-40P_bn	1.33	1.78	**6.55**
init4_10-15-20-25-30-35-40P_bn	1.25	1.75	6.88

Table 5.6: Comparison of different networks that are only initialized with the values as in Table 5.5 and not trained by any training sample. Batch normalization is used in all networks in this table.

Up to this point, we already presented quite some networks. A way to check if a network is learning properly is to compare its resulting average AEE to a network which is not trained but just initialized. Table 5.5 presents four different networks with the same architecture as before, but they are not trained by any training samples.

Table 5.6 shows the resulting AEE of descriptors computed via networks initialized like the ones in table Table 5.5, but also used batch normalization. Like in the previous case, these networks are not trained with any training sample. When a network uses batch normalization, it will have more trainable variables, however, in these networks they are just initialized and not trained, like other trainable variables.

For MPI-Sintel images, the positive effect of using batch normalization is clear, as all the average AEE values decreased using batch normalization. Another important observation from Table 5.6 is that there are networks that we trained with many training samples (such as the networks shown in Table 5.3) that lead to a larger value of average AEE than the networks that are only initialized. This means that those networks did not learn properly from the training samples for various possible reasons, that we need to find out. One possibility is using improper learning rate values. Another possibility is that the training samples might not be suitable to train the network properly for this task. As a basic criteria, the networks we train must lead to better results than the networks that are only initialized, such as the ones shown in Table 5.6. Another way to check whether the descriptors are working reasonably or not is to compare the value of the average AEE resulted from using descriptors computed via our trained networks to the average AEE values resulted from using DAISY or SIFT descriptors. Table 5.7 shows AEE values of the optical flow obtained by using DAISY and SIFT descriptors on the image-pairs in our evaluation set. Clearly, the optical flow

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
DAISY	0.67	1.03	**3.87**
SIFT	**0.56**	**0.88**	4.07

Table 5.7: Comparison of DAISY and SIFT descriptors. The optical flow is estimated by CPM algorithm using these descriptors and the value of average AEE of the estimated optical flow is computed.

estimated using DAISY and SIFT descriptors also have been achieved using the CPM framework. If the AEE values resulted from using learned descriptors are close to the AEE values resulted from using SIFT or DAISY, we can conclude that the learned descriptors are working, to some extent.

Experiments to Find a Proper Step-size Value

Our next step is to further design and train networks to improve the optical flow results to get smaller average AEE values. To make it as most systematic as possible, we keep the initialization fixed for networks having the same architecture to check the impact of other parameters, such as learning rates, loss functions, etc., on the resulting descriptors. And as the second initialization shown in Table 5.6 led to the smallest average AEE value for both MPI-Sintel and KITTI 2015 image-pairs, we use that initialization for all networks that have the mentioned 7-layer architecture.

As explained before, for networks with different architectures, we cannot use the same initializations, but we can keep the order of training samples fixed, to make our experiments more systematic.

In the next set of experiments we want to investigate the impact of different learning rates on learned descriptors. There are different strategies to set a learning rate for the optimization step in the training process. One way is to use a fixed step-size value through all iterations. In all the trained networks that we have shown so far, we used a fixed learning rate, which was equal to 10^{-6}. There are other ways to assign the step-size value, such as using exponential decay or cosine decay strategies. With these strategies, step-size is assigned by an initial value and it decreases, as the number of iterations increase. The values of step-size are computed in different iterations using exponential decay strategy as:

$$\text{decayed\_step\_size} = \text{step\_size} \cdot \text{decay\_rate}^{(\text{currecnt\_iteration/decay\_steps})}, \tag{5.1}$$

where step_size is the initial learning rate value and decay_rate and decay_steps are hyper-parameters.

Table 5.8 shows a comparison of different CNNs using different learning rates. ’lr’ in the label of the CNNs shows the initial step-size, which decreases exponentially, using exponential decay strategy. The substring ’ed’ followed by a number represents that the networks used exponential decay strategy to update the learning rate value in each iteration. The number which is followed by ’ed’ shows the value of decay_steps. In all the networks that the step-size value is decreased exponentially, the decay_rate is set to 0.96, therefore we do not put this constant information in the label of the network, in our tables.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_ep5_**lr10^{-5}**_ed5000	1.38	1.93	5.96
*_ep5_**lr10^{-4}**_ed5000	0.79	1.17	**5.35**
*_ep5_**lr10^{-3}**_ed5000	0.75	1.13	6.10
*_ep5_**lr5 · 10^{-3}**_ed5000	**0.70**	**1.06**	5.49
*_ep5_**lr5 · 10^{-3}**_ed5000_**copy**	0.71	1.08	5.37

*: 10-15-20-25-30-35-40P_bn

Table 5.8: Comparison of networks trained using different learning rate values. In the last two experiments, two networks are trained with the exact same parameters. Even the initialization and the order of training samples is kept fixed, however the networks lead to slightly different results.

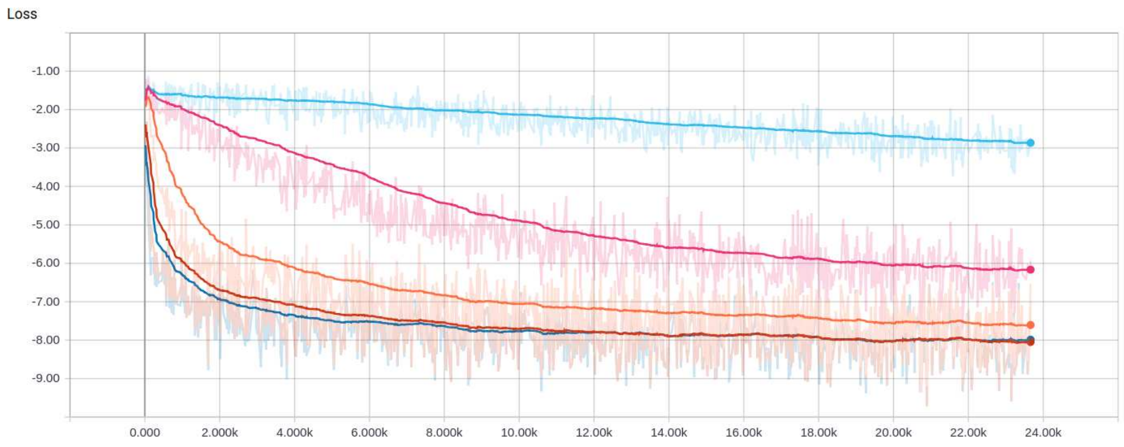


Figure 5.1: These graphs show the training loss values over the training iterations of the networks with the architecture 10-15-20-25-30-35-40P_bn, trained for five epochs. These networks are trained using different learning rate values. The light-blue, magenta, orange, red and dark-blue graphs, show the training loss values for the networks trained with step-size values 10^{-5}, 10^{-4}, 10^{-3}, $5 \cdot 10^{-3}$ and 10^{-2}. Clearly, there is a very slight difference between the red and dark-blue graphs and in the last 12000 iterations, they seem to converge to each other. This figure is generated by Tensorboard with smoothing value of 0.988.

As we said before, in the experiments shown in Table 5.2, Table 5.3 and Table 5.4, the learning rate was constant and equal to 10^{-6} in all iterations. In Table 5.8, we increased the value of the initial step-size gradually in each experiments. We can see obvious improvements in the results, for MPI-Sintel image-pairs.

Of course, we could experiment further with larger learning rates to check of there are further improvements. However, by considering the values of the training loss with different learning rates, we found it unnecessary to increase the value of the step-size further.

Figure 5.1 shows that increasing the learning rate values leads to large decreases in values of the training loss. The graphs in the mentioned figure represent training loss values in different training iterations, for the networks shown in Table 5.8. The red and dark-blue graphs show the training loss values in different training iterations of networks, that were trained with step-size values of $5 \cdot 10^{-3}$ and 10^{-2}, respectively. As these two graphs converge to each other, we considered them as equally suitable for the training process, therefore we did not evaluate the images with the network trained with learning rate value of 10^{-2}. That is the reason that there is no experiment with that learning rate in Table 5.8.

As it is shown in Table 5.8, among the networks trained with different learning rates, the one which was trained with step-size value of $5 \cdot 10^{-3}$ led to the best results for MPI-Sintel image-pairs. Therefore, we considered this step-size value for other experiments[1].

In the last two experiments shown in Table 5.8, two networks are trained with the exact same parameters. Even the initialization and the order of training samples is kept fixed, however the networks lead to slightly different results. This means that there is a random factor in training different networks. Unfortunately, we are not aware what this random factor is. By observing this slight result difference for the networks having the exact same architecture and trained with the exact same way, we conclude that differences below 0.02 in the values of AEE are negligible.

Experiments on Different Normalizations

Table 5.9 shows another set of experiments. In this set, we investigate the effect of normalizing the training samples for networks trained with different learning rate values. The experiments use 'npi' and 'norm', as mentioned in Table 5.3. Again using 'npi' the learning rate value of $5 \cdot 10^{-3}$ led to the best results. We also tested the effect of normalizing with mean and standard deviation of the whole datasets (i.e.'norm'). We observe that the latter leads to better results with the same architecture and training parameters. We only tested 'norm' normalization with step-size value of $5 \cdot 10^{-3}$, as it led to the best results in table Table 5.8 and for the other experiments in Table 5.9. Of course, one could argue that it is possible if this kind of normalization gives better results with another value of learning rate, but as our time was limited to check all different possibilities, we had to use parameters that led to the best results for some new experiments.

[1]It is worth mentioning that after we did all the experiments, we evaluated the network trained with the step-size value of 10^{-2}. We witnessed that using that network led to slightly better results (about 0.02 in the values of AEE of MPI-Sintel image-pairs), compared to the network that was trained with the learning rate value of $5 \cdot 10^{-3}$.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_npi_**lr10^{-5}**	1.26	1.73	6.77
*_npi_**lr10^{-4}**	1.50	1.96	**6.06**
*_npi_**lr5 · 10^{-3}**	**1.18**	**1.63**	6.30
*_npi_**lr5 · 10^{-2}**	3.66	3.95	7.12
*_norm_lr5 · 10^{-3}	**0.75**	**1.14**	**5.30**

*: 10-15-20-25-30-35-40P_bn_ep5_ed5000

Table 5.9: Comparison of networks trained with differently normalized training samples. Like in Table 5.3, 'npi' in the label of the network means that the patches in training samples are normalized with the mean and standard deviation of the image that the patch is extracted from. In the second group, we tested the effect of another normalization strategy. Again like in Table 5.3, 'norm' in the label of the network means that the training samples are normalized by the mean and standard deviation of all images in MPI-Sintel and KITTI 2015 datasets. Obviously, the input images are also normalized with the same values, for evaluation.

Obviously, the last experiment led to the best results in Table 5.9, however the resulted AEE is larger than the same network without normalization for MPI-Sintel images. Therefore, we decide not to use these for other experiments.

Experiments on Different Decay Steps

The parameter that is investigated in Table 5.10 is the decay step parameter. As shown in Equation (5.1), it affects the reduction of the value of the learning rate, in each training iteration. The larger this parameter is, the slower the value of step-size decreases. As we can see, the difference of the resulted AEEs of the optical flow estimated for MPI-Sintel image-pairs are really slight in all the experiments in Table 5.10. Therefore, for the next experiments, we use the decay step value of 1000, as it led to the best results for KITTI 2015 images, with a large difference to others.

Experiments to Find Suitable Parameters for Thresholded Hinge Loss Function, with Different Learning Rate Values

One of the most important tasks in training a CNN is choosing a suitable loss function, based on the task the CNN should solve. We explained a simple option for the loss function in Chapter 3. For all the networks we trained so far, we used the simple loss. Another option is the thresholded hinge loss, that we discussed in Chapter 3. As we showed, that loss function has two parameters of m and t. In Table 5.11, first we try different values for m and t, then with the combination of parameters that led to the best result, we try two other step-size values and then we check two different decay step values.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_**ed1000**	0.710	1.09	**4.27**
*_**ed2000**	0.712	1.08	5.69
*_**ed3000**	0.712	1.09	5.13
*_**ed4000**	0.703	1.13	5.69
*_**ed5000**	0.704	1.065	5.49
*_**ed6000**	0.708	1.08	5.19
*_**ed10000**	**0.692**	**1.03**	5.48

*: 10-15-20-25-30-35-40P_bn_ep5_lr5 $\cdot 10^{-3}$

Table 5.10: Comparison of networks trained with different decay steps. For MPI-Sitel image-pairs, AEE values change very slightly.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:1-0.3**	0.94	1.47	5.22
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:1-0.1**	0.81	1.27	4.79
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:1-0.02**	0.72	1.11	4.79
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:5-0.01**	0.69	1.06	4.59
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:10-0.02**	**0.66**	**1.00**	**4.54**
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:10-0.01**	0.69	1.06	4.58
*_lr5 $\cdot 10^{-3}$_ed1000_**m-t:15-0.02**	0.67	1.00	4.83
*_**lr10^{-2}**_ed1000_m-t:10-0.02	**0.69**	**1.08**	**4.71**
*_**lr10^{-4}**_ed1000_m-t:10-0.02	1.09	1.60	5.42
*_lr5 $\cdot 10^{-3}$_**ed500**_m-t:10-0.02	$\underline{\mathbf{0.650}}$	$\underline{\mathbf{0.95}}$	4.84
*_lr5 $\cdot 10^{-3}$_**ed5000**_m-t:10-0.02	0.653	0.98	$\underline{\mathbf{4.41}}$

*: 10-15-20-25-30-35-40P_bn_ep5

Table 5.11: Comparison of different networks trained with different values of m and t, different learning rates for a constant value for m and t, and different decay steps.

’m-t:value1-value2’ in the label of the network shows that the network is trained with thresholded hinge loss function with the values value1 and value2 for m and t, respectively. For the first group of experiments, we can observe that with m = 10 and t = 0.02, we get the best results. With this combination of m and t, we tested two networks trained with larger and smaller values of learning rates to check if any of them leads to better results. Then two networks trained with smaller and larger decay steps are evaluated. Table 5.11 shows that in these experiments, the mentioned values of m and t, learning rate value of $5 \cdot 10^{-3}$ and decay step of 500 led to the best results.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_**lr10^{-4}**_ed500_m-t-10-0.02	1.04	1.43	5.47
*_**lr5 · 10^{-3}**_ed500_m-t-10-0.02	0.74	1.13	5.38
*_**lr10^{-2}**_ed500_m-t-10-0.02	**0.68**	**1.06**	**5.24**
*_**lr10^{-1}**_ed500_m-t-10-0.02	0.83	1.23	7.24

*: 40-35-30-25-20-15-10P_bn_ep5

Table 5.12: Comparison of networks with the architecture 40-35-30-25-20-15-10P_bn trained with different learning rates.

In the literature [Bai+17; GW16], usually the network architectures have (exponentially) increasing number of filters in their convolutional layers and usually the number of neurons is large (larger than 50 filters) for every convolutional layer. So far, we only considered one architecture with consecutively increasing number of neurons in convolutional layers, but with much smaller number of filters per layer (10-15-20-25-30-35-40).

We started with a simple network to check whether the number of filters per convolutional layer has to be necessary large to lead to reasonable results or not. Of course, it is preferable if the network is simpler, as it has a lower run time. Moreover with lower number of trainable variables, it needs less training samples to be trained properly and also there is a less chance of overfitting to the training samples.

Experiments on Different Networks with Different Architectures

In Table 5.12, Table 5.13 and Table 5.14, we show some experiments with different architectures. In each set of experiments we try to find a reasonable learning rate and we check whether those networks lead to a better result than our best current result or not. In all the experiments, the networks are trained with the thresholded hinge loss. We used m and t values, and decay steps that led to the best results for MPI-Sintel image-pairs in Table 5.11.

In Table 5.12, we compare different learning rates for networks with the architecture 40-35-30-25-20-15-10-5P-bn. In Table 5.13, in the architecture of networks, the number of filters per layer is kept constant and equal to 40, and in Table 5.14, the number of filters per convolutional layer exponentially increases, except in the last layer, that it decreases. The architecture of the networks evaluated in Table 5.14 is 8-16-32-64-128-256-128P. We did not increase the number of neurons in the last convolutional layer, as the descriptor dimension computed for one single patch would be $1 \times 1 \times 512$. And for a large image, computing 512 floating point features for each pixel needs a lot of memory and it also leads to a slow run time. To prevent issues that arise because of not having enough memory, we decreased the number of filters of the last convolutional layer.

As we mentioned before, exponentially increasing number of neurons in convolutional layers is a popular choice, therefore we were interested in investigating that architecture a bit more. Table 5.15 shows a set of experiments, in which we compare different values of m and t, for the thresholded hinge loss, to check whether using a different values can lead to a drastic improvement, or not.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_**lr5 · 10^{-3}**_ed500_m-t-10-0.02	0.67	1.03	4.73
*_**lr10^{-2}**_ed500_m-t-10-0.02	**0.66**	**1.02**	**4.67**
*_**lr10^{-1}**_ed500_m-t-10-0.02	0.68	1.07	4.76

*: 40-40-40-40-40-40-40P_bn_ep5

Table 5.13: Comparison of networks with the architecture 40-40-40-40-40-40-40P_bn trained with different learning rates.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_**lr10^{-4}**_ed500	1.52	2.02	5.67
*_**lr10^{-3}**_ed500	0.94	1.34	4.67
*_**lr5 · 10^{-3}**_ed500	**0.691**	**1.01**	4.57
*_**lr10^{-2}**_ed500	0.692	1.04	**4.50**

*: 8-16-32-64-128-256-128P_bn_ep5

Table 5.14: Comparison of networks with the architecture 8-16-32-64-128-256-128P_bn trained with different learning rates.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_lr5 · 10^{-3}_ed500_**m-t-10-0.02**	0.70	1.08	4.37
*_lr5 · 10^{-3}_ed500_**m-t-10-0.5**	**0.69**	**1.07**	**4.30**
*_lr5 · 10^{-3}_ed500_**m-t-10-1.0**	0.71	1.07	4.41
*_lr5 · 10^{-3}_ed500_**m-t-10-2.0**	0.78	1.16	4.49

*: 8-16-32-64-128-256-128P_bn_ep5

Table 5.15: Comparison of networks with the architecture 8-16-32-64-128-256-128P_bn trained with different m and t parameters of the thresholded hinge loss. The learning rate is kept fixed and equal to $5 \cdot 10^{-3}$, as the network trained with that step-size led to the best results in Table 5.14.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_DataStd2_**ep5**	0.65	1.01	4.94
*_DataStd2_**uniDiscarded_ep5**	0.61	0.95	4.87
*_DataStd2_**uniDiscarded_ep3**	**0.59**	**0.90**	4.83
*_DataStd2_**uniDiscarded_ep2**	0.63	0.97	5.02
*_DataStd2_**uniDiscarded_ep1**	0.61	0.94	4.92
******_DataStd2_**uniDiscarded_ep3**	0.64	1.01	**4.58**

*: 10-15-20-25-30-35-40**P**_bn_lr5 · 10^{-3}_ed500_m-t-10-0.02
**: 10-15-20-25-30-35-40_bn_lr5 · 10^{-3}_ed500_m-t-10-0.02

Table 5.16: Comparison of networks trained with the training samples generated with the second strategy discussed in Section 3.1 with standard deviation value of 2 in the normal distribution. The label 'uniDiscarded' means that the uniform training samples were not used in training the networks.

None of the networks we evaluated in Table 5.12, Table 5.13, Table 5.14 and Table 5.15 led a result better than the network 10-15-320-25-30-35-40P_bn_lr5 · 10^{-3}_ed500_m-t:10-0.02 shown in Table 5.11 for MPI-Sintel image-pairs, in our evaluation set. However they led to very close results.

Experiments on Using Other Sets of Training Samples

One of the most important factors in training a CNN is the training samples that are fed to the network. In Section 3.1, we explained how we generate the training samples. We discussed two different strategies to extract the non-matching patch. So far, all the trained networks we discussed were trained by the first strategy (i.e. to extract the non-matching patch, a random number between 3 to 8 pixels was added to the integer part of the coordinates of center of the corresponding patch and the patch around that pixel was extracted and considered as the non-matching patch).

In Table 5.16 we evaluate the networks trained with training samples that were generated with the second strategy. In the second way, center of the non-matching patch is computed by adding a random number from a normal distribution that prefers closer distances to the coordinates of the center of corresponding patches. It means that closer non-matching patches are preferred. We set the minimum distance of the center of corresponding and non-matching patches to 2 pixels. The substring 'DataStd2' in the label of the network shows that the network is trained with the training samples that were generated with the second strategy and the standard deviation value of 2 in the normal distribution was considered.

The first network evaluated in Table 5.16 is trained with all the training samples generated with the second strategy. All other networks in that table were trained with the same training samples, but not with the homogeneous patches in the training sample. As we discussed in Section 3.1, uniform patches may not contribute to the learning process, suitably. The substring 'uniDiscarded' explains that the uniform training samples in the generated training samples were discarded and not used for the training the networks.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_**DataStd1**_uniDiscarded_ep3	0.69	1.00	5.40
*_**DataStd3**_uniDiscarded_ep3	**0.63**	**0.98**	**4.71**

*: 10-15-20-25-30-35-40P_bn_lr5 $\cdot$ 10^{-3}_ed500_m-t-10-0.02

Table 5.17: Comparison of networks trained with the training samples that were generated with the second strategy, discussed in Section 3.1, using different standard deviation values for the normal distribution.

The network trained with training samples without homogeneous samples for three epochs led to the best results in that table. Note that the last network evaluated in the table has a different architecture, as it has no pooling layer after its last convolutional layer, and the spatial size of the descriptor computed with this network for one pixel has the size 2×2. We trained this architecture with three epochs, as using that number of epochs led to the best result in the second group of experiments, in Table 5.16.

We also checked whether changes in the standard deviation of the mentioned normal distribution leads to some improvements in descriptors or not. Table 5.17 shows evaluation of two networks trained with training samples with the new strategy, but with different standard deviation values for the mentioned normal distribution (i.e.'DataStd1' for value of 1 and 'DataStd3' for 3, in the first and second network, respectively). By considering lower values of standard deviation, non-matching patches are extracted with a smaller distance to corresponding patches. If this distance is too small, the non-matching and corresponding patches will look too similar to have descriptors that are dissimilar enough, also probably the network can not learn to compute suitable descriptors, given such training samples. In Table 5.17 the second network performs better than the first one. Perhaps the standard deviation value of 1 was too small, therefore the network using the training samples generated with that standard deviation performed worse than the second network shown in the table.

So far the third network evaluated in Table 5.16 led to the best results for MPI-Sintel image-pairs in our evaluations set. We evaluated a few networks with different architectures in Table 5.12, Table 5.13 and Table 5.14. All those networks were trained with the first set of training samples. In Table 5.18, we evaluate some networks with different architectures trained with the training samples, 'DataStd2', which was used to train the networks shown in Table 5.16. Clearly the second network (30-45-60-75-90-105-120P_bn_DataStd2_uniDiscarded_ep3_lr5 $\cdot$ 10^{-3}_ed500_m-t-10-0.02) led to the best results for MPI-Sintel image-pairs, in that table. However, the current best result is computed via 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_lr5 $\cdot$ 10^{-3}_ed500_m-t-10-0.02, which was shown in Table 5.16. In Table 5.18, all networks use the same parameters (number of epochs, step-size value, etc.) as the network with the best current results, and they are only different in their architectures.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
10P-15-20-25_*	0.65	1.03	5.02
30-45-60-75-90-105-120P_*	**0.60**	**0.92**	4.80
30-45-60-75-90-105-120_*	0.62	0.97	4.54
8-16-32-64-128-256-128P_*	0.64	0.97	**4.51**
10-10-10-10-10-10-10P_*	0.86	1.25	7.58
40-40-40-40-40-40-40P_*	0.64	0.98	4.74

*: bn_DataStd2_uniDiscarded_ep3_lr5 · 10^{-3}_ed500_m-t-10-0.02

Table 5.18: Comparison of networks with different architectures, trained with the training samples, 'DataStd2'.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_**tanh**	0.92	1.31	8.89
*_**relu6**	0.62	0.96	4.93
*_**elu**	**0.596**	0.90	4.71
*_**selu**	0.598	**0.89**	**4.67**
*_**softplus**	0.66	1.01	4.99
*_**softsign**	0.91	1.28	8.64
*_**sigmoid**	0.74	1.12	7.18

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_lr5 · 10^{-3}_ed500_m-t-10-0.02

Table 5.19: Comparison of some networks using different activation function.

Experiments on Using Different Activation Functions

Up to this point, in all the networks we discussed, used the activation function relu in their structure. This choice can affect the network drastically. In table Table 5.19 shows different networks using different non-linearities. Clearly, the network using the selu non-linearity performed better than others and now, the current best result is computed by this network.

While training the networks evaluated in Table 5.19, we witnessed strange behavior of some networks while training, as the values of the training loss did not decrease notably during training. Non-linearities similar to relu, such as relu6, elu and selu, and also soft plus had a significant decrease, unlike tanh, soft sign and sigmoid. Figure 5.2 shows how the training loss evolves over the iterations of the training process, for networks using the activation functions relu6, selu, elu and soft plus.

Figure 5.3 shows the training loss values over different training iterations for the networks using relu6, tanh, soft sign and sigmoid non-linearities. Obviously, the values of the training loss for the network using relu6 drops more than others. The training loss values of the network using

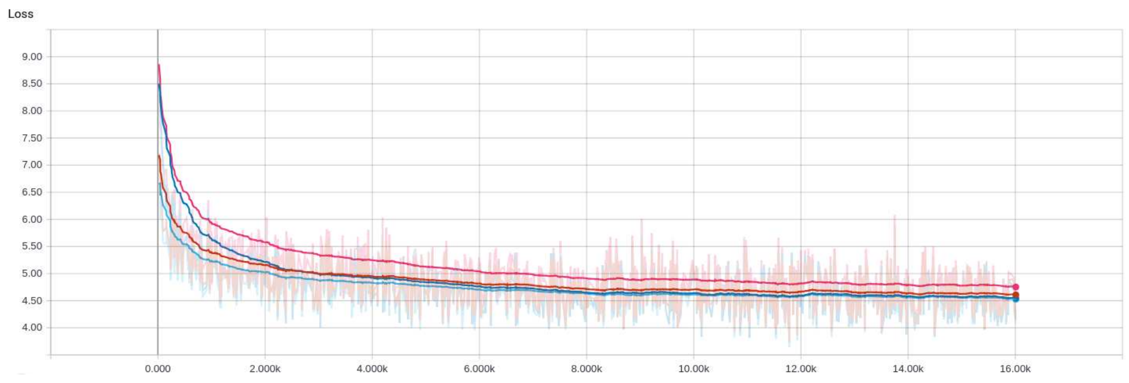


Figure 5.2: The light-blue, red dark-blue and magenta graphs show the values of training loss over the training iterations of networks using selu, elu, relu6 and soft plus non-linearities, respectively. Other details of the architecture and parameters of the networks is shown in Table 5.19. This figure is generated via Tensorboard using smoothing parameter of value 0.988.

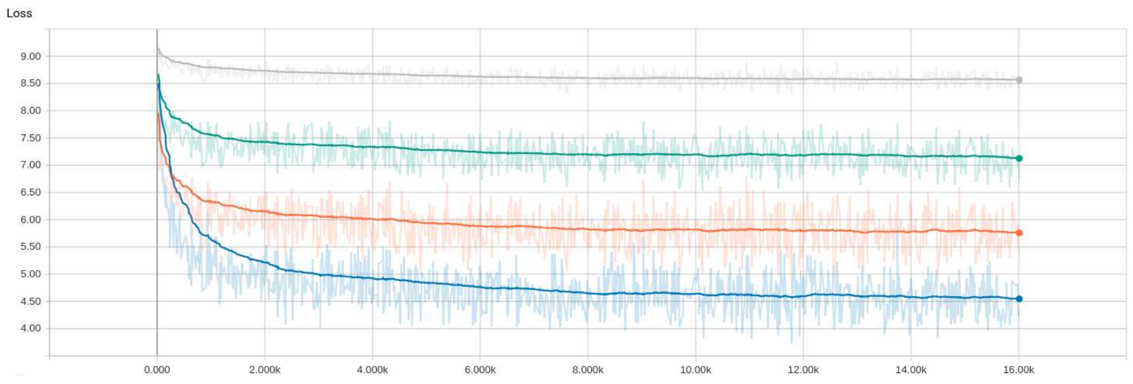


Figure 5.3: The blue, orange, green and gray diagrams show the training loss values for networks that use relu6, tanh, soft sign and sigmoid activation functions. Other details of the architecture and parameters of the networks is shown in Table 5.19. Obviously the the values of the training loss of the network that uses relu6 change much more than the other diagram. The diagram associated to the network that uses sigmoid has the slightest change. This figure is generated via Tensorboard using smoothing parameter of value 0.988.

sigmoid activation function has the least change. As we were surprised by this behavior, we found it reasonable to check other learning rates for training the network that uses the sigmoid non-linearity.

In Table 5.20 we evaluated networks that use sigmoid non-linearity trained with different learning rate values. The network trained with the step-size value of $5 \cdot 10^{-6}$ led to the best results for MPI-Sintel image-pairs, and performed even better than SIFT and DAISY for MPI-Sintel image-pairs in our evaluation set. This was surprising for us, as we expect the training loss values to decrease significantly during the training process. Also in Figure 5.1 the more decrease we witnessed in the training loss values, the better was the results of the network. However, apparently it is not always

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_sigmoid_**lr5 · 10^{-1}**	0.86	1.19	7.98
*_sigmoid_**lr5 · 10^{-4}**	0.69	1.08	6.06
*_sigmoid_**lr5 · 10^{-5}**	0.58	0.88	5.05
*_sigmoid_**lr5 · 10^{-6}**	**0.53**	**0.82**	**5.00**
*_sigmoid_**lr5 · 10^{-7}**	0.55	0.85	5.69

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_ed500_m-t-10-0.02

Table 5.20: Comparison of networks using the sigmoid non-linearity with different learning rates. The network trained using the step-size value of $5 \cdot 10^{-6}$ led to the best results for MPI-Sintel image-pairs in our evaluation set.

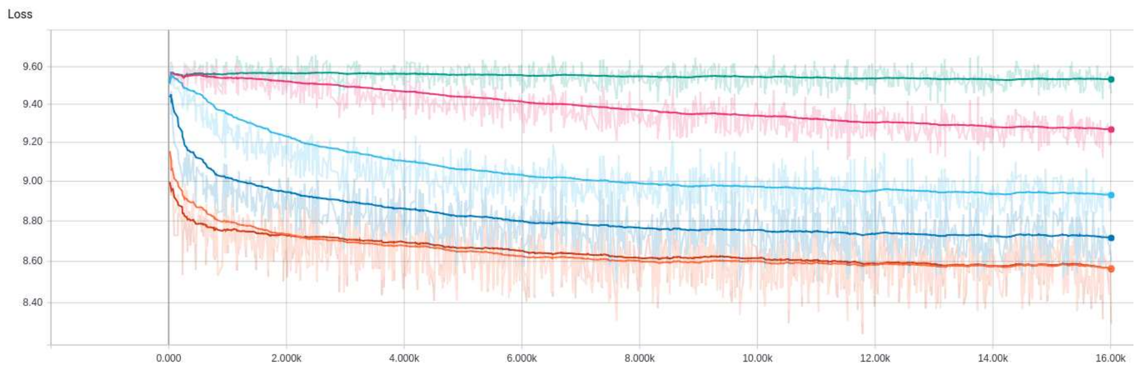


Figure 5.4: Diagrams of the training loss values over the training iterations of networks using sigmoid non-linearity with different step-size values. The red, orange, dark-blue, light-blue, magenta and green diagrams show how training loss values evolve in the training process of networks using step-size values of $5 \cdot 10^{-1}$, $5 \cdot 10^{-3}$, $5 \cdot 10^{-4}$, $5 \cdot 10^{-5}$, $5 \cdot 10^{-6}$ and $5 \cdot 10^{-7}$, respectively. The other details about the architecture and other parameters of the networks is mentioned in Table 5.20. This figure is generated via Tensorboard using smoothing parameter of value 0.988.

the case, as the network using sigmoid as the activation function with the step-size value of $5 \cdot 10^{-6}$ led to the best result, however values of the training loss in that network has a very slight decrease compared to the networks that were trained with larger step-size values and the ones using other non-linearities.

In Figure 5.4 the magenta diagram shows how the values of the training loss evolve in the training process of the network with step-size value of $5 \cdot 10^{-6}$, using sigmoid non-linearity. Clearly the loss values dropped very slightly compared to the orange diagram, which shows how the training loss values evolve in the training process of the network with step-size value of $5 \cdot 10^{-3}$ using sigmoid activation function. Note that the mentioned orange diagram is the same gray diagram shown in Figure 5.3, which has the lowest decrease in the training loss values in that figure, compared to the other networks using other non-linearities.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_tanh_**lr5 · 10^{-5}**	0.68	1.01	5.77
*_tanh_**lr5 · 10^{-6}**	**0.55**	**0.86**	**4.86**
*_tahn_**lr5 · 10^{-7}**	0.57	0.87	5.61
*_softplus_**lr5 · 10^{-4}**	0.62	0.96	4.63
*_softplus_**lr5 · 10^{-5}**	**0.62**	**0.96**	**4.59**
*_softsign_**lr5 · 10^{-5}**	0.62	0.93	5.43
*_softsign_**lr5 · 10^{-6}**	**0.58**	**0.89**	**5.22**
*_softsign_**lr5 · 10^{-7}**	0.60	0.91	5.77

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_ed500_m-t-10-0.02

Table 5.21: Comparison of networks using different non-linearities trained with different step-size values.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_sigmoid_**Adam-optimizer**	**0.53**	**0.82**	**5.00**
*_sigmoid_**gradient-descent-optimizer**	0.58	0.93	5.60
*_sigmoid_**momentum-optimizer**	0.59	0.93	5.90

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_lr5 · 10^{-6}_ed500_m-t-10-0.02

Table 5.22: Comparison of two networks with the same architecture and parameters trained with different optimizers of Adam, gradient-descent and momentum optimizers.

By facing a significant improvement in the resulting optical flow estimated by the network that uses sigmoid non-linearity with a suitable step-size value, we became interested in looking for other values of learning rate for the networks that use other activation functions. In Table 5.21 we tried other step-size values for networks that use soft plus, tanh and soft sign activation functions, as the results computed by the network using these non-linearities were worse than the results computed via the networks using relu, relu6, elu and selu, shown in Table 5.19. From Table 5.21, we can observe that none of the evaluated networks performed better than the network that used sigmoid with step-size value of $5 \cdot 10^{-6}$. Note that we trained those networks shown in Table 5.21 with different step-size values for find a reasonable value. In the case of networks that used the soft plus non-linearity, we did not check smaller learning rate values, as we did not witness clear changes by changing the step-size value from $5 \cdot 10^{-4}$ to $5 \cdot 10^{-5}$.

Experiments on Using Different Optimizers

So far, for all the networks we trained, we used Adam optimizer to minimize the loss function. In Table 5.22 we evaluate networks that are trained with gradient-decent and momentum optimizers to check weather using those optimizers leads to some improvements or not. Of course, all other

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_sigmoid	0.5332	0.8266	5.0039
*_**3Levels**_sigmoid	**0.5311**	**0.8208**	**4.9991**

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_lr5 · 10^{-6}_ed500_m-t-10-0.02

Table 5.23: Comparison of the estimated optical flow with two different strategies. The first strategy is by using the fourth network shown in Table 5.20. The second way is using multiple networks with the same architecture and parameters of the network used in the network used in the first experiment, but trained with different sets of training samples down-sampled with different factors.

parameters of the networks are considered as the network, which led to the best current result. The network that was optimized with gradient-descent led to slightly better results than the one optimized with momentum optimizer, but both performed worse than the network optimized with Adam optimizer, with the same parameters.

As it is explained before, CPM does the matching task in a coarse-to-fine scheme, also we mentioned before that we construct pyramid of three levels for MPI-Sintel images and four for KITTI 2015 images. Based on this, we can consider the idea suggested in the literature [Bai+17] which is training the networks with patches extracted from different levels of the pyramid. To do this, we built and trained three different networks, where each network is trained with the training samples extracted from the images of one level of the pyramid. For example, for the lowest level, we used a network that is trained with training samples that are extracted from the original images, in the datasets. For the next levels, we train a network with the training samples that are extracted from the down-sampled images from the first and second levels of the pyramid.

As each of these network can have a different architecture, and each has a different set of training samples, one has to do many experiments to find a reasonable architecture and parameters. We kept the architecture and other parameters fixed for all three networks, as our time was limited and the only difference between the three networks is the set of training samples they are trained with. We considered the architecture and parameters of the network that led to the best current results. As we explained before, that network was trained with 16×16 patches and lead to descriptors of spatial size 1×1. To train the same network with the down-sampled images we made the training samples with the aforementioned second strategy that were used first in the networks evaluated in Table 5.16, with the original images. We made training samples of size 32×32 and 64×64 and down-sampled them with the factors 0.5 and 0.25, to train the networks used in the matching task of the second and first level of the pyramids, respectively.

In other words, we built and trained three networks with features '10-15-20-25-30-35-40P_bn_sigmoid_ep3_lr5 · 10^{-6}_ed500_m-t-10-0.02' with training samples that are down-sampled with different factors. For the matching task of the lowest pyramid level, the same network '10-15-20-25-30-35-40P_bn_sigmoid_DataStd2_uniDiscarded_ep3_lr5 · 10^{-6}_ed500_m-t-10-0.02' is used with the same training samples. For the matching task of the second and first level, the networks are trained with training samples down-sampled with factors 0.5 and 0.25, respectively, if the pyramid has three levels.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE
*_3Levels_sigmoid	2.08	3.55	10.84
DAISY	1.97	3.77	**6.59**
SIFT	**1.81**	**3.35**	7.04

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_lr5 $\cdot$ 10^{-6}_ed500_m-t-10-0.02

Table 5.24: Comparison of SIFT, DAISY and our approach on the test set. Our approach performed worse than both SIFT and DAISY on this test set.

Network	Sintel-nocc AEE	Sintel-all AEE	KITTI-nocc AEE	KITTI-all AEE	EPE > 3 px KITTI-all
*_3Levels_sigmoid	**2.61**	**3.73**	5.04	10.77	29.20%
DAISY	2.85	4.15	**3.37**	7.86	**21.93**%
SIFT	2.78	4.00	3.54	**7.50**	23.03%

*: 10-15-20-25-30-35-40P_bn_DataStd2_uniDiscarded_ep3_lr5 $\cdot$ 10^{-6}_ed500_m-t-10-0.02

Table 5.25: Comparison of SIFT, DAISY and our approach on the whole MPI-Sintel and KITTI 2015 datasets. The fourth column shows the AEE values for KITTI 2015 images, considering all pixels that have a valid ground truth value. The fifth column shows the bad point error, which is the percentage of the pixels whose computed end point error is larger than 3 pixels, for KITTI 2015 images.

For KITTI 2015 images, the pyramid has four levels. In this situation, we use the same network for the matching task of the first and second level, which uses the training samples down-sampled with factor of 0.25. For the matching task of the third level, the network trained with training samples down-sampled with factor of 0.5 is used, and for matching task of the lowest level, the same mentioned network, which is used in the lowest level for MPI-Sintel images, is used.

Table 5.23 shows the results of the network that led to the best results, in the previous experiments and also an experiment in which three different networks were used to contribute to the matching task, in each pyramid level. Those three networks were trained with training samples down-sampled differently with the down-sampling factors of 1, 0.5 and 0.25 to compute the descriptors for images in the third, second and first pyramid levels. As it is shown in Table 5.23, using the mentioned three networks had a very slight positive effect on the results.

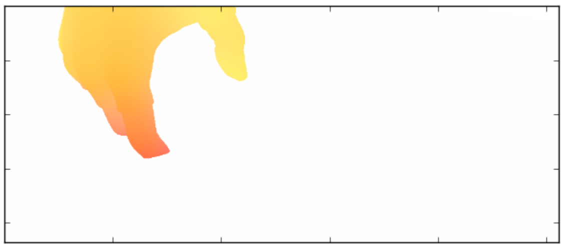
Evaluation of Our Approach on a Test Set and the Whole MPI-Sintel and KITTI 2015 datasets

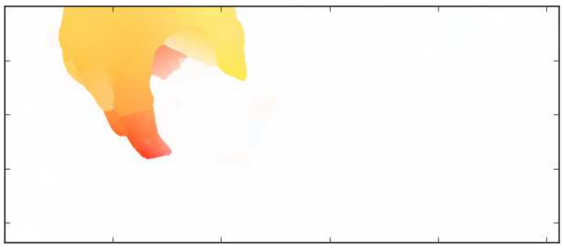
In Table 5.24 we compare optical flow results estimated via our last network, which led to the best results we got for MPI-Sintel image-pairs, SIFT and DAISY. Our network performed worse, compared to SIFT and DAISY on this test set, which is defined in Section 3.1. A possible reason is that the images used in this test set were not used for training the networks used in our approach.

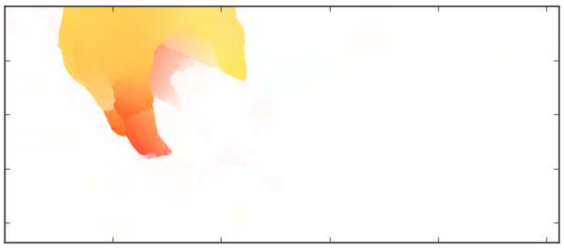
(a) Image at time t

(b) Image at time $t + 1$

(c) Images at time t and $t + 1$ overlaid


(d) Ground truth optical flow


(e) Our approach


(f) DAISY

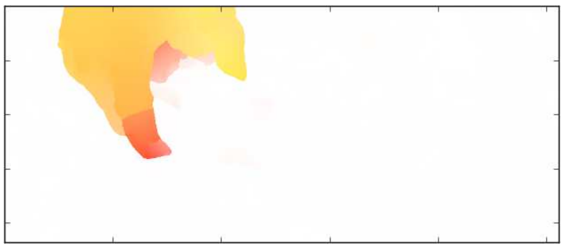

(g) SIFT

Figure 5.5: Comparison of the flow field between images 0011 and 0012 from folder `ambush_7` in the MPI-Sintel dataset.

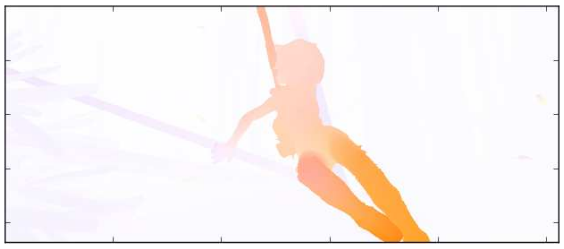
Table 5.25 shows a comparison of our approach, SIFT and DAISY. our approach performs better than both SIFT and DAISY for the whole MPI-Sintel dataset for all and non-occluded pixels, but it performed worse than both other approaches for KITTI 2015 image-pairs.

Lastly, we present some image pairs in Figure 5.5 to Figure 5.13 and visualize the ground truth optical flow between them, flow values computed by our approach, DAISY and SIFT, to compare them visually.

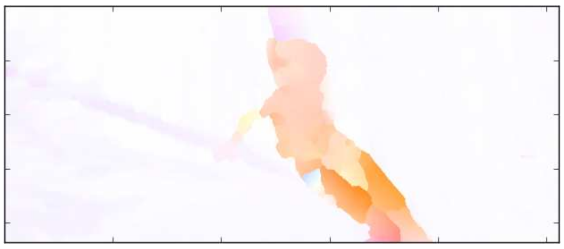
(a) Image at time *t*

(b) Image at time *t* + 1

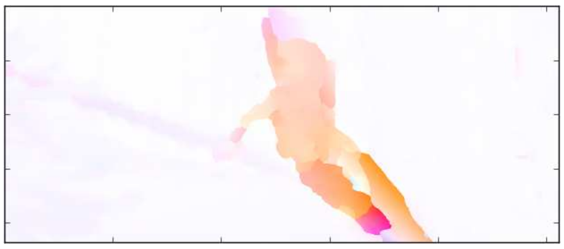
(c) Images at time *t* and *t* + 1 overlaid



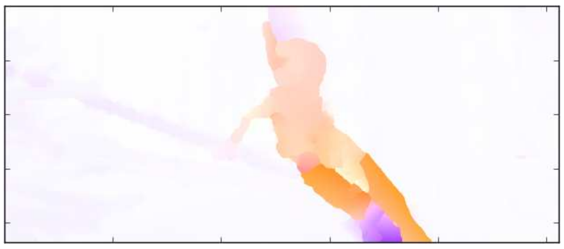
(d) Ground truth optical flow



(e) Our approach

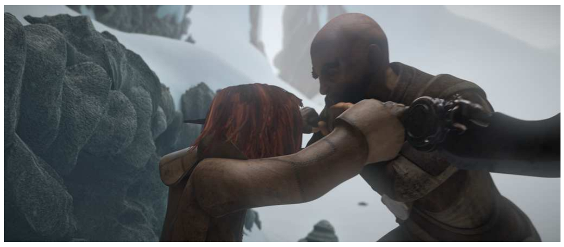


(f) DAISY

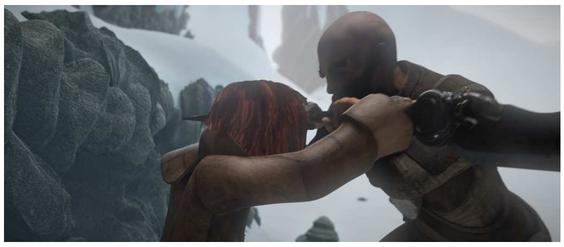


(g) SIFT

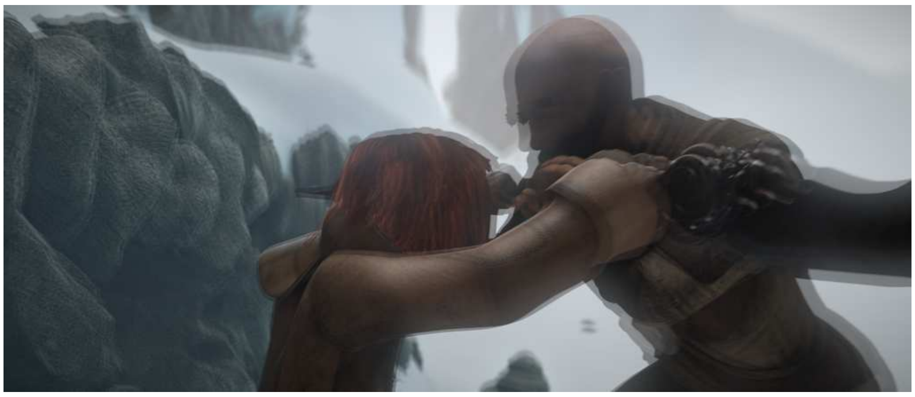
Figure 5.6: Comparison of the flow field between images 0043 and 0044 from folder `bamboo_2` in the MPI-Sintel dataset.



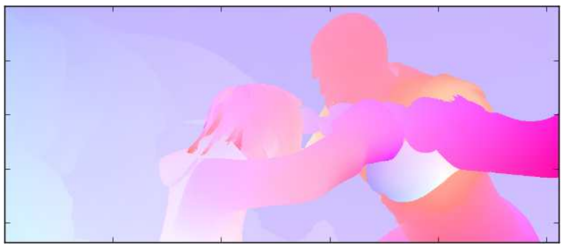
(a) Image at time *t*



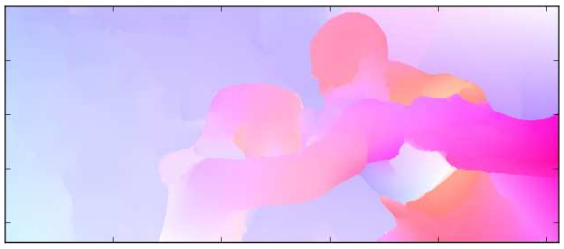
(b) Image at time $t + 1$



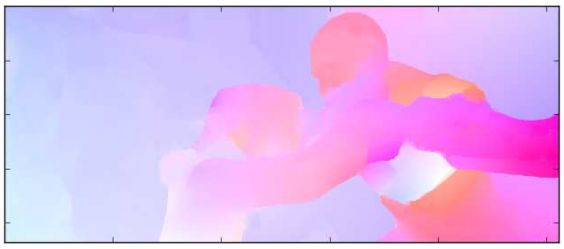
(c) Images at time *t* and $t + 1$ overlaid



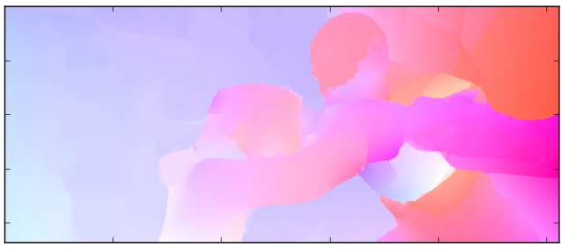
(d) Ground truth optical flow



(e) Our approach



(f) DAISY



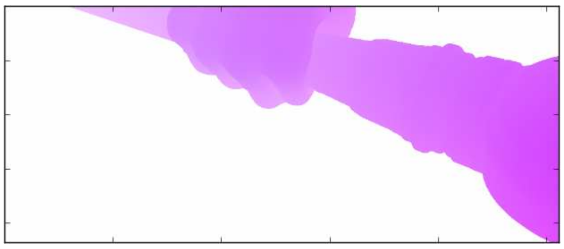
(g) SIFT

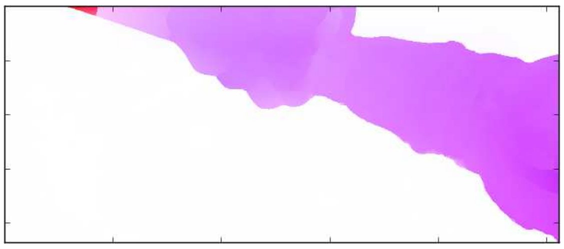
Figure 5.7: Comparison of the flow field between images 0018 and 0019 from folder `ambush_5` in the MPI-Sintel dataset.

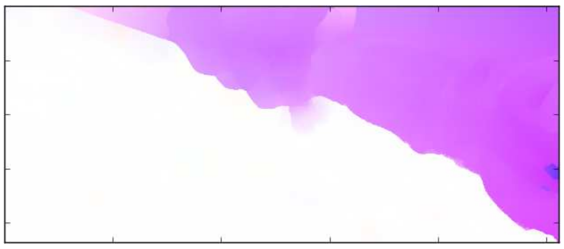
(a) Image at time *t*

(b) Image at time *t* + 1

(c) Images at time *t* and *t* + 1 overlaid


(d) Ground truth optical flow


(e) Our approach


(f) DAISY

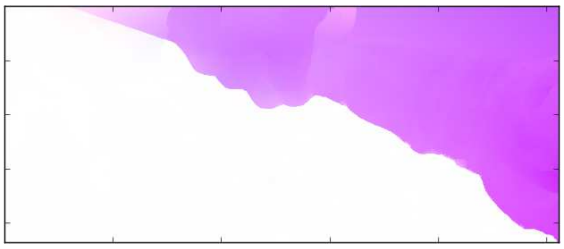
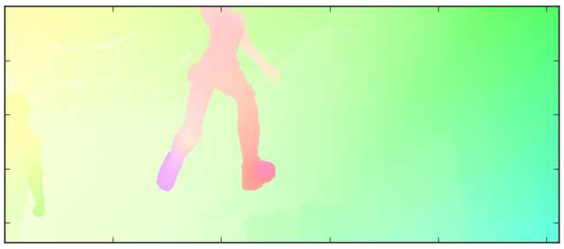

(g) SIFT

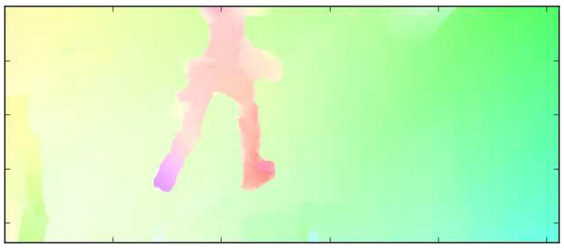
Figure 5.8: Comparison of the flow field between images 0041 and 0042 from folder `ambush_7` in the MPI-Sintel dataset.

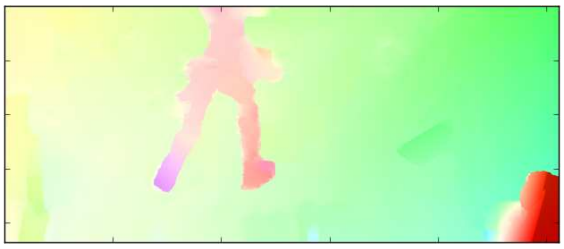
(a) Image at time t

(b) Image at time $t + 1$

(c) Images at time t and $t + 1$ overlaid


(d) Ground truth optical flow


(e) Our approach


(f) DAISY

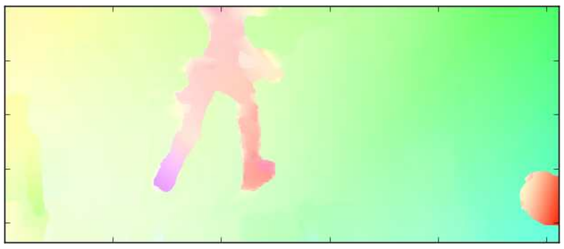
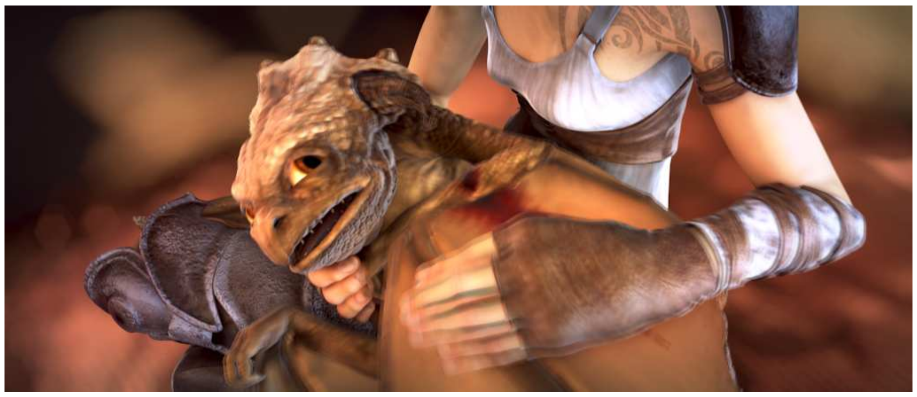
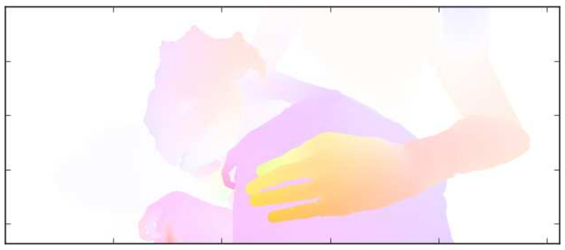

(g) SIFT

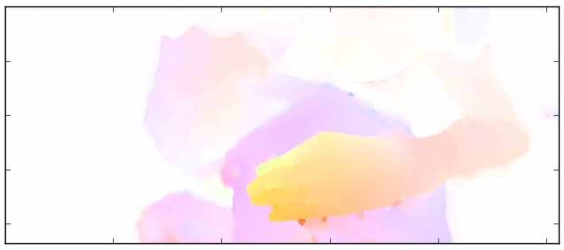
Figure 5.9: Comparison of the flow field between images 0039 and 0040 from folder `market_5` in the MPI-Sintel dataset.

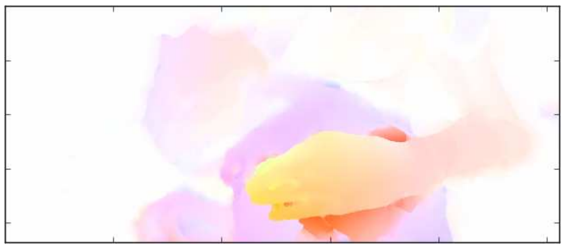
(a) Image at time t

(b) Image at time $t + 1$


(c) Images at time t and $t + 1$ overlaid


(d) Ground truth optical flow


(e) Our approach


(f) DAISY

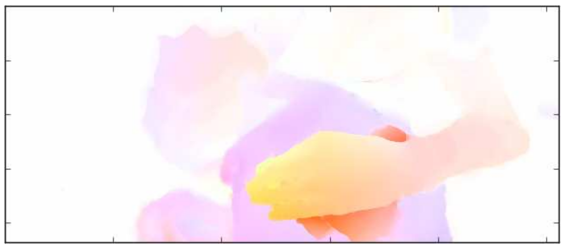
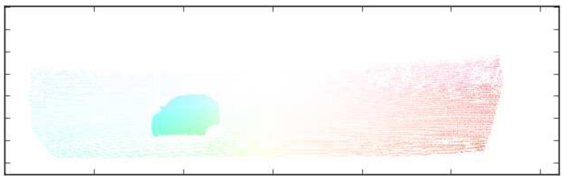

(g) SIFT

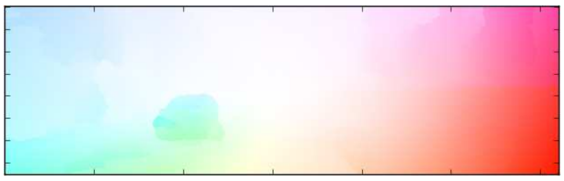
Figure 5.10: Comparison of the flow field between images 0045 and 0046 from folder `bandage_1` in the MPI-Sintel dataset.

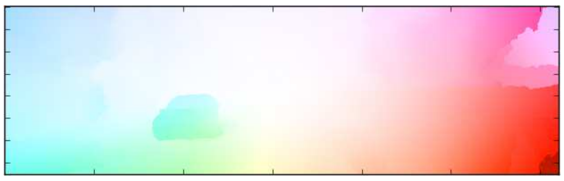
(a) Image at time *t*

(b) Image at time *t* + 1

(c) Images at time *t* and *t* + 1 overlaid


(d) Ground truth optical flow


(e) Our approach


(f) DAISY

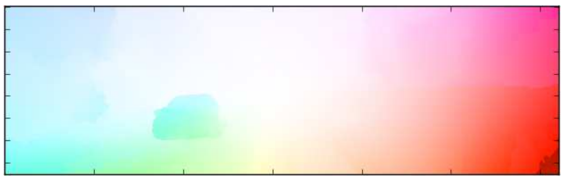
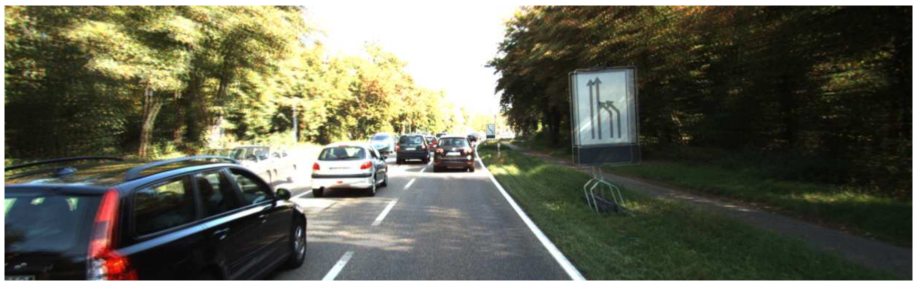

(g) SIFT

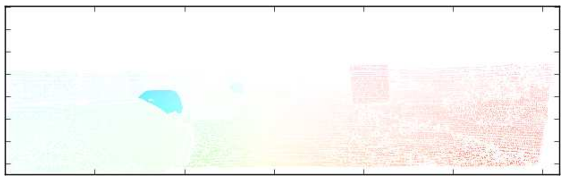
Figure 5.11: Comparison of the flow field between images 000056_10 and 000056_11 from the KITTI 2015 dataset.

(a) Image at time *t*

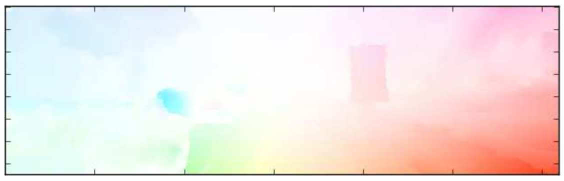
(b) Image at time *t* + 1



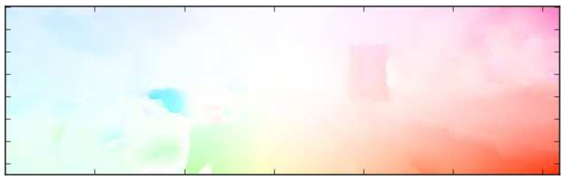
(c) Images at time *t* and *t* + 1 overlaid



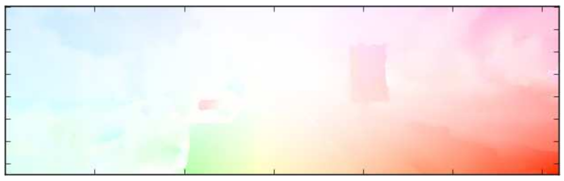
(d) Ground truth optical flow



(e) Our approach



(f) DAISY

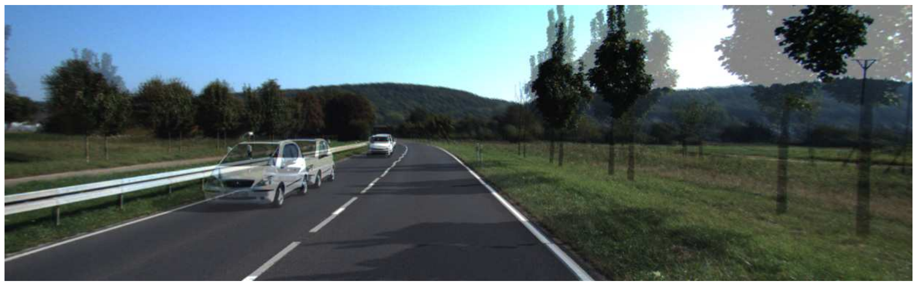


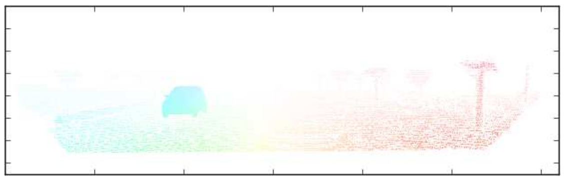
(g) SIFT

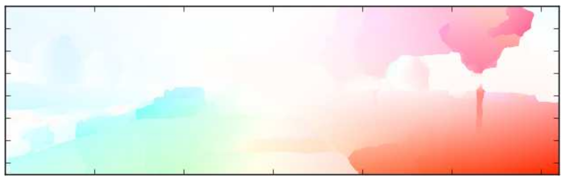
Figure 5.12: Comparison of the flow field between images 000161_10 and 000161_11 from the KITTI 2015 dataset.

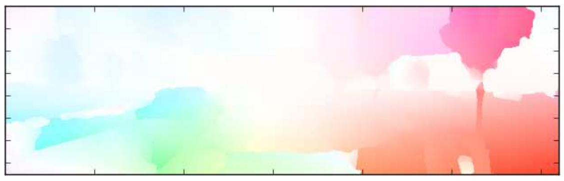
(a) Image at time *t*

(b) Image at time $t + 1$


(c) Images at time *t* and $t + 1$ overlaid


(d) Ground truth optical flow


(e) Our approach


(f) DAISY

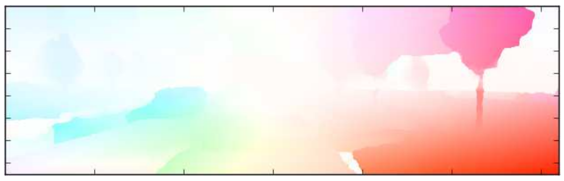

(g) SIFT

Figure 5.13: Comparison of the flow field between images 000180_10 and 000180_11 from the KITTI 2015 dataset.

6 Conclusion and Future Work

In this thesis, we used learned descriptors to compute the optical flow using the CPM method. We designed and trained CNNs and embedded them in the CPM framework, so that the correspondences between two images are found based on the learned descriptors.

We used the KITTI 2015 and MPI-Sintel datasets to generate the training samples to train our networks. We chose these datasets as we wanted to benefit from the color information of the images. We made about 500,000 training samples. Each training sample consists of a reference, a corresponding and a non-matching patch of size (16×16). We made the training samples in such a way that half of them are extracted from the KITTI 2015 dataset and the other half is extracted from the MPI-Sintel dataset. As the training samples consisting of patches that are extracted from homogeneous areas do not contribute to the training process properly, we discussed a criterion to discard them.

We implemented our CNNs using Tensorflow. We implemented a function that produces the network given the necessary parameters. After training a network, we made the complete portable model of the trained network by combining the graph of the network and the value of its trainable variables. The complete model was imported to the CPM framework to compute the cost function via learned descriptors computed by the imported network.

To contribute to the computation of the cost function in CPM, we need a set of descriptors for every pixel in the input images. This can be done by extracting a patch around every pixel in the input images and feed the patches to the network to compute the descriptors for each patch. We discussed that this solution is too time-consuming. Another way to compute the descriptor for all the pixels in the images is by passing the image through the network instead of the patches extracted from the image. We implemented the approach of Bailer et al. [Bai+18] to compute the descriptors by feeding the image through the network. Also for the case that the spatial size of the descriptor is not equal to (1×1), we extended the mentioned approach to extract the descriptors from the output matrix resulted from the approach of Bailer et al. [Bai+18].

We trained different networks to investigate the impact of different parameters and architectures. The networks that we trained used different architectures, step-size values, normalizations, non-linearities, training data, loss functions and optimizers. Our learned descriptors led to more accurate optical flow results compared to SIFT and DAISY for all the image-pairs in the `final` folder in the MPI-Sintel dataset. Of course, as 80% of the dataset was used for training our CNN, there is no surprise to achieve reasonable results. To assess how useful our learned descriptors are in general, one can compare their results to the result of other descriptors for other datasets.

In our experiment, we considered the parameters that led to the best result in terms of AEE (for MPI-Sintel image-pairs in our evaluation set) to use it for training other networks in our next experiments. Importantly, in most cases we assumed that if using a parameter in training a network leads to better results, it will also yield better results in combination with other architectures and parameters, therefore we use that parameter in the next experiment. For instance, if a number of epochs is suitable for training a network with an architecture, for the next experiment, if we want to investigate the impact of a different architecture, we will use the same number of epochs. However, it is probable that with a different architecture, using another number of epochs yields better results. The mentioned assumption is obviously flawed, however to have a clear strategy to do the experiments and also to keep the number of experiments smaller we needed such an assumption.

In future works, we are eager to further investigate the impact of each parameter in combination with other parameters and architectures. Also it would be interesting to investigate other criteria to discard the training samples that consist of homogeneous patches. Using other strategies to create the training samples can be done as well. For example, one could create more than one non-matching patch per training sample and instead of a non-matching patch, a non-matching neighborhood around the corresponding patch can be considered in each training sample.

Dedication

This thesis is dedicated to Mr. Mohammad Ali Taheri, the founder of two Iranian complementory and alternative medicine of Faradarmani and Psymentology that helped thousands of people, teacher of peace and love, who has been kept in jail and tortured for more than eight years and was sentenced to death, because of expressing his opinions. This thesis is dedicated to all people who have been treated unfaily by others.

Moreover, this thesis is dedicated to my parents, Vajiheh and Ahmad, who supported me and gave me the opportunity to study abroad, my sister Azadeh and my brother Nima that have been always supportive and caring to me, my second and third parents Beate Massa and Lutz Marx, and Ursula Rivinius and Martin Rivinius who gave me so much love and support and treated me like their own daughter.

Importantly, this thesis is also dedicated to my Imzadi, Marc Rivinius, who gave me unlimited love and support.

Bibliography

[Bai+17] C. Bailer, K. Varanasi, D. Stricker. “CNN-based patch matching for optical flow with thresholded hinge embedding loss”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 2. 3. 2017, p. 7 (cit. on pp. 18, 38, 42, 75, 86, 94).

[Bai+18] C. Bailer, T. Habtegebrial, D. Stricker, et al. “Fast Feature Extraction with CNNs with Pooling Layers”. In: *arXiv preprint arXiv:1805.03096* (2018) (cit. on pp. 19, 58, 59, 62, 70, 105).

[Bar+09] C. Barnes, E. Shechtman, A. Finkelstein, D. B. Goldman. “PatchMatch: A randomized correspondence algorithm for structural image editing”. In: *ACM Transactions on Graphics* 28.3 (2009), p. 24 (cit. on pp. 18, 23).

[Bay+06] H. Bay, T. Tuytelaars, L. Van Gool. “Surf: Speeded up robust features”. In: *European Conference on Computer Vision*. Springer. 2006, pp. 404–417 (cit. on p. 17).

[But+12] D. J. Butler, J. Wulff, G. B. Stanley, M. J. Black. “A naturalistic open source movie for optical flow evaluation”. In: *European Conference on Computer Vision*. Ed. by A. Fitzgibbon et al. (Eds.) Part IV, LNCS 7577. Springer-Verlag, Oct. 2012, pp. 611–625 (cit. on p. 18).

[Cle+15] D.-A. Clevert, T. Unterthiner, S. Hochreiter. “Fast and accurate deep network learning by exponential linear units (elus)”. In: *arXiv preprint arXiv:1511.07289* (2015) (cit. on p. 30).

[DT05] N. Dalal, B. Triggs. “Histograms of oriented gradients for human detection”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 1. IEEE. 2005, pp. 886–893 (cit. on p. 17).

[DZ13] P. Dollár, C. L. Zitnick. “Structured forests for fast edge detection”. In: *IEEE International Conference on Computer Vision*. 2013, pp. 1841–1848 (cit. on p. 27).

[Dos+15] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. Van Der Smagt, D. Cremers, T. Brox. “Flownet: Learning optical flow with convolutional networks”. In: *IEEE International Conference on Computer Vision*. 2015, pp. 2758–2766 (cit. on p. 18).

[GW16] D. Gadot, L. Wolf. “Patchbatch: a batch augmented loss for optical flow”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4236–4245 (cit. on pp. 18, 40, 75, 86).

[Git18] GitHub. *GitHub Octoverse 2018: Highlights from 2018*. 2018. URL: `https://octoverse.github.com/` (visited on 12/18/2018) (cit. on p. 45).

[HS12] K. He, J. Sun. “Computing nearest-neighbor fields via propagation-assisted kd-trees”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 111–118 (cit. on p. 18).

[HS81] B. K. Horn, B. G. Schunck. “Determining optical flow”. In: *Artificial Intelligence* 17.1-3 (1981), pp. 185–203 (cit. on p. 18).

[HZ03] R. Hartley, A. Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003 (cit. on p. 26).

[Had+06] R. Hadsell, S. Chopra, Y. LeCun. “Dimensionality reduction by learning an invariant mapping”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2006, pp. 1735–1742 (cit. on p. 18).

[Hu+16] Y. Hu, R. Song, Y. Li. “Efficient coarse-to-fine patchmatch for large displacement optical flow”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 5704–5712 (cit. on pp. 17, 23).

[Hu17] Y. Hu. *Efficient Coarse-to-Fine PatchMatch for Large Displacement Optical Flow*. 2017. URL: https://github.com/YinlinHu/CPM (visited on 12/20/2018) (cit. on p. 54).

[IS15] S. Ioffe, C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on p. 34).

[Ilg+17] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, T. Brox. “Flownet 2.0: Evolution of optical flow estimation with deep networks”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 2. 2017, p. 6 (cit. on p. 18).

[Jah17] A. Jahedi. *Descriptor Learning for Correspondence Problems*. 2017 (cit. on pp. 38, 42, 45, 53, 77).

[KA16] S. Korman, S. Avidan. “Coherency sensitive hashing”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.6 (2016), pp. 1099–1112 (cit. on p. 18).

[KB14] D. P. Kingma, J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on pp. 31, 32).

[KH10] A. Krizhevsky, G. Hinton. “Convolutional deep belief networks on cifar-10”. In: *Unpublished manuscript* 40.7 (2010) (cit. on p. 30).

[KJ18] A. Karpathy, J. Johnson. *CS231n Convolutional Neural Networks for Visual Recognition*. 2018. URL: cs231n.github.io/ (visited on 12/15/2018) (cit. on pp. 27, 33, 35).

[Kla+17] G. Klambauer, T. Unterthiner, A. Mayr, S. Hochreiter. “Self-normalizing neural networks”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 971–980 (cit. on p. 30).

[Liu+11] C. Liu, J. Yuen, A. Torralba. “Sift flow: Dense correspondence across scenes and its applications”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.5 (2011), pp. 978–994 (cit. on p. 24).

[Low04] D. G. Lowe. “Distinctive image features from scale-invariant keypoints”. In: *International Journal of Computer Vision* 60.2 (2004), pp. 91–110 (cit. on p. 17).

[MG15] M. Menze, A. Geiger. *Object Scene Flow for Autonomous Vehicles*. 2015 (cit. on p. 18).

[Mic18] Microsoft. *Microsoft Cognitive Toolkit*. 2018. URL: https://www.microsoft.com/en-us/cognitive-toolkit/ (cit. on p. 45).

[Mus+85] H. G. Musmann, P. Pirsch, H.-J. Grallert. “Advances in picture coding”. In: *Proceedings of the IEEE* 73.4 (1985), pp. 523–548 (cit. on p. 17).

[Ope18] OpenMP. *The OpenMP API for parallel programming*. 2018. URL: https://www.openmp.org/about/about-us/ (visited on 11/28/2018) (cit. on p. 67).

[Rev+15] J. Revaud, P. Weinzaepfel, Z. Harchaoui, C. Schmid. "Epicflow: Edge-preserving interpolation of correspondences for optical flow". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1164–1172 (cit. on pp. 18, 25, 27).

[Rud16] S. Ruder. *An overview of gradient descent optimization algorithms*. 2016. URL: http://ruder.io/optimizing-gradient-descent/ (visited on 12/03/2018) (cit. on p. 31).

[Sch+17] T. Schuster, L. Wolf, D. Gadot. "Optical flow requires multiple strategies (but only one network)". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2017 (cit. on p. 18).

[Sun+18] D. Sun, X. Yang, M.-Y. Liu, J. Kautz. "Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8934–8943 (cit. on p. 18).

[Tol+10] E. Tola, V. Lepetit, P. Fua. "Daisy: An efficient dense descriptor applied to wide-baseline stereo". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.5 (2010), pp. 815–830 (cit. on p. 17).

[Tul15] A. Tulloch. *deeplearning-hs*. 2015. URL: http://hackage.haskell.org/package/deeplearning-hs (cit. on p. 45).

[Ved+18] A. Vedaldi, K. Lenc, A. Gupta. *MatConvNet*. 2018. URL: http://www.vlfeat.org/matconvnet/ (cit. on p. 45).

[Was13] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013 (cit. on p. 26).

[Ber18] T. Berkeley Vision and Learning Center. *Caffe*. 2018. URL: http://caffe.berkeleyvision.org/ (cit. on p. 45).

[Int17] Intel Nervana. *neon*. 2017. URL: http://neon.nervanasys.com/ (visited on 11/28/2017) (cit. on p. 45).

[Ten18] T. TensorFlow Authors. *TensorFlow*. 2018. URL: https://www.tensorflow.org/ (visited on 11/28/2018) (cit. on pp. 28, 44).

[The18] Theano Development Team. *Theano*. 2018. URL: http://deeplearning.net/software/theano/ (cit. on p. 45).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature