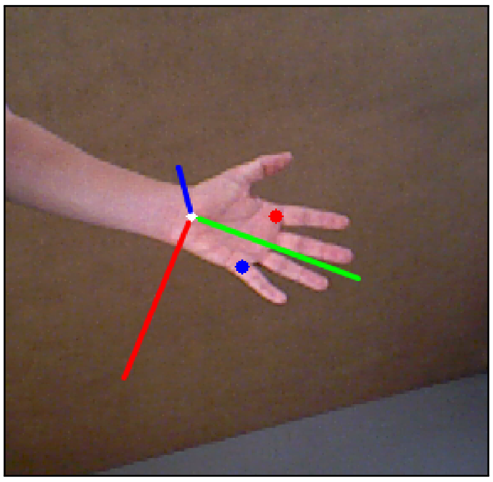
University of Freiburg
Faculty of Engineering
Department of Computer Science
Autonomous Intelligent Systems Group



Hand Orientation Estimation using Deep Neural Networks

Bachelor Thesis

Submitted By:	Lukas Hermann
1st Examiner:	Prof. Dr. Wolfram Burgard
Supervisors:	Andreas Eitel
Date:	September 1, 2015

Declaration

I hereby declare, that I am the sole author and composer of my thesis

Hand Orientation Estimation using Deep Neural Networks

and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Lukas Hermann

Freiburg im Breisgau, September 1, 2015

Zusammenfassung

Körperteilerkennung und die Detektion der menschlichen Pose sind wichtige Aspekte in der Robotik und der Mensch-Roboter Interaktion. Von besonderem Interesse ist die Detektion der Handpose. Da die Hand die Hauptverbindung zwischen Roboter und Mensch in Zusammenarbeitsszenarien darstellt, ist sie Gegenstand aktueller Forschung.
In dieser Arbeit präsentieren wir eine Methode um die Orientierung der Hand aus einzelnen Tiefenbildern in Echtzeit zu schätzen. Unser Ansatz ist so entwickelt, dass er mit existierenden Hand- bzw. Körperteildetektoren kombiniert werden kann, welche eine erste Schätzung der Handposition liefern. Unsere zweiteilige Architektur verfeinert zuerst die Schätzung des externen Handdetektors und berechnet daraufhin die Orientierung. Beide Ebenen unserer Methode benutzen auf anotierten Datensätzen echter Hände trainierte Convolutional Neural Networks.
Wir entwickelten zwei verschiedene Arten um die Orientierung zu berechnen, wobei eine Methode Punkte auf der Hand benutzt um die Pose zu beschreiben und die andere die 3D Rotation mithilfe von Quaternionen darstellt. Wir vergleichen beide Methoden und arbeiten Vorteile und Nachteile heraus.
Auf einem Testdatensatz führen wir eine Reihe von Experimenten durch und evaluieren die Leistung unseres Ansatzes. Die Ergebnisse zeigen, dass die Methode eine hohe Genauigkeit erreicht, selbst wenn die Distanz zwischen der Hand und dem Tiefensensor in der Nähe der maximalen Messweite liegt. Sie funktioniert in Echtzeit auf gewöhnlicher Hardware und kann mit einem beliebigen Körperteildetektor kombiniert werden.

Abstract

Body part detection and human pose estimation are important aspects of robotics and human-robot interaction. Hand pose estimation is particularly interesting and part of current research since the hand is the main link between humans and robots in collaboration scenarios.
In this thesis we propose a method to estimate a hand's orientation from single depth images in real-time. Our approach is designed to be combined with existing hand / body part trackers which give an initial guess for the hand position. The two-stage architecture of our method first refines the initial guess and subsequently estimates the orientation. Both stages use convolutional neural networks trained on realistic labeled hand datasets.
We developed two different methods to estimate the orientation, one using three keypoints on the hand to represent the pose and the other one using quaternions which describe a 3D rotation. We compare both methods and determine advantages and disadvantages.
Using an independent test dataset we conduct a series of experiments and evaluate the performance of our approach. The results show that the method estimates the orientation with high precision, even if the distance between hand and depth sensor is near the maximum range. It runs in real-time on ordinary hardware and can be combined with any body part detector / tracker.

Contents

1 Introduction

In human-robot interaction scenarios perception of human body parts such as hands is an important component in a robot's repertoire of skills. Especially an exact estimate of the pose of a human hand is required in various tasks, such as learning from demonstration and visual tasks, human robot collaboration (e.g. passing objects from robot to human or vice versa) or gesture recognition. Actions and movements of humans, that seem intuitive, are not easy to be reproduced by a robot. In a scenario of grasping an arbitrary object, not only the grasp, i.e. where the object is touched, but also the trajectory of the hand towards the object and its orientation during the whole process has to be determined to allow a humanoid robot to learn from a human instructor and imitate his/her movements.
Tracking and detecting human body parts has been widely studied in the computer vision community, but most methods provide only a rough estimate of the hand's position. Most human body tracking methods require the full body to be visible in order to perform well and don't provide a guess for the hand pose. Other approaches directly focus on hands, but they are mostly designed to work in close range of an RGB-D sensor (such as the *Microsoft Kinect*) and work well when the hand is placed right in front of the camera [25, 26]. Hand pose estimation is very challenging due to the high variability in hand appearance, many degrees of freedom of the hand and self-occlusions [16].

In this work we present a method to get a good real-time estimate of the hand orientation which also works in a wide-range scenario with arbitrary camera positions and perspectives and can thus be applied to ordinary tasks, such as moving the hand towards an item. Due to the existence of good body part trackers the focus of this work lies on the pose estimation and therefore our method is meant to be combined with arbitrary body trackers. More precisely it uses the tracker's estimated hand

position as an initial guess and feeds it into a two stage architecture, which consists of two convolutional neural networks trained with deep learning algorithms. Further, we recorded two datasets of both near-range and far-range depth images with annotated keypoints for training the networks after applying data augmentation techniques, to enlarge the datasets.
The objective of the first stage is to give a very precise estimate of the hand's center point since the initial guess is usually not precise enough. Knowing the exact hand position is of great importance for the second stage because we have experienced that performance of the orientation estimation drops if the region of interest (ROI) that covers the full hand is not well defined.
We developed two different methods to estimate the hand's orientation which differ in the way they represent the pose. The first one predicts three keypoints of the hand (namely wrist, index and pinkie) from which the orientation can be computed subsequently. The other one uses quaternions to directly represent the orientation. We evaluate the performance of both methods and compare the advantages and disadvantages.

Our approach is designed to be used with *Microsoft Kinect* or *Asus Xtion* sensors, which are both RGB-D cameras, although our method only uses depth images. In contrast to RGB images, they are invariant to changes of illumination (if not exposed to direct sunlight which affects the infrared depth sensor) and the detection is therefore more robust. The exact details of the theory behind this method and the implementation will be provided in the following chapters.

2 Related Work

In this chapter we want to give an overview about current body part detection/tracking methods and compare them. There exist a great variety of approaches which all have advantages and disadvantages. These methods can be divided into two groups: generative methods and discriminative methods.
Generative learning is also called model-based learning because a model of the body or body part, e.g. the hand, is fitted to an image. Assumptions about the pose are made based on which model configuration matches the image best. Commonly generative approaches have a high accuracy, but require a lot of computation and hence are mostly non real-time. Additionally, some generative methods like [21] need a good prior state estimate and therefore rely on initialization from a certain pose. Model-based locality assumptions can cause problems because especially hands tend to move fast and have a lot of degrees of freedom, e.g. it can lead to error propagation in subsequent frames. Despite that, generative methods are widely used in tracking and detection.
On the other hand, discriminative approaches learn the direct mapping from input images to the target space, such as joint labels or joint coordinates [26]. These methods do not use a visual model, but a labeled training dataset to train a pose estimator. Datasets can be either synthetic or real. While synthetic datasets are usually easier to obtain and less noise-corrupted, real datasets are more accurate and inherently have the correct joint angle constraints whereas most artificial models are capable of anatomically incorrect poses (e.g. wrong joint angles) [27]. Unfortunately it is very time-consuming to label a real dataset. Discriminative learning methods strongly depend on the the quality of the training dataset. In comparison with generative approaches they are commonly more robust and computationally efficient, but also less accurate.

2.1 Human Pose Estimation

Shotton et al. [24] from *Microsoft Research* developed a state-of-the-art discriminative approach for human pose recognition that is designed to work with *Microsoft Kinect* RGB-D cameras. Their method only requires single depth images for body part segmentation. They created a large, synthetic and highly diversified depth image database of human poses and trained a deep randomized decision forest (a machine learning algorithm). Major emphasis was placed on fast runtime as their approach should be able to be used in real-time on consumer hardware. It is able to detected poses of arbitrary body rotation and works even for multiple people.
Another discriminative human pose estimator was proposed by Tompson et al. [27]. It uses a hybrid architecture of a deep convolutional network and a markov random field in order to use a higher level spatial model to correct poses that are anatomically incorrect and constrain joint interconnectivity and thus overcoming a major problem of discriminative methods. Their technique requires only RGB images and runs at almost real-time frame rates.
Sapp et al. [23] developed a more traditional generative method for pose estimation with RGB images using deformable part models. They proposed a multimodal model approach with a cascaded mode selection and focused mainly on the trade-off between speed and accuracy.

2.2 Hand Pose Estimation

Hand pose estimation is a similar problem to 3D full body pose estimation. For both the key challenge is to recognize the configuration of an articulated object with a high degree of freedom. However, self-occlusions, object-occlusions, viewpoint changes, high variety in poses and noise caused by lower image resolution presents an additional challenge for hand pose estimation.
Tompson et al. [26] presented a real-time hand pose recovery method for single depth images. They used an artificial hand model and applied a random decision forest for image segmentation to create a labeled real hand dataset which was used to train a deep network. Their approach can be extended for arbitrary articulated objects, it only requires a 3D model of the object. The focus lies on hands in close-range and the method can handle self-occlusions but no object occlusions.
Tang et al. [25] published another discriminative method for real-time articulated hand pose estimation with depth images. They created a small, partially labeled real

hand dataset and a big synthetic dataset of more than 300.000 images on which they trained a special semi-supervised learning algorithm called transductive regression forest. This approach is interesting because it shows a way to resolve the general problem of discriminative methods, which are the high costs for labeling real datasets. Their work is designed to work on single images, so no prior information is needed. Like [26] it cannot deal with occlusions through objects and is designed for the close-range domain. We evaluated their synthetic dataset because we considered using it for our training, but found it not suitable for our purposes due to missing hand poses (e.g. dorsal view).
Another discriminative approach was developed by Romera et al. [22] focusing on handling object context and interaction. It is able to deal with all kinds of occlusions and partially corrupted data and runs in real-time using only RGB images. Pose ambiguities are solved through temporal consistency. The tracking method is non-parametric and uses histogram of orientated gradients (HOG) as feature representation to perform a nearest neighbor search in a database of hand poses. However, the hand-background segmentation method is susceptible to errors since it is only based on skin color.
Very Recently, Oberweger et al. [20] improved over state-of-the-art approaches testing several deep network architectures on existing artificial hand datasets such as [25]. They applied a joint-specific refinement stage to improve the joint localization using a different network for each joint.
A model-based articulated hand pose estimation method is presented by Kuznetsova et al [16]. Their method uses single RGB-D images and does not rely on prior information. The model is fitted to the data using a non-rigid iterative closest point (ICP) algorithm. Like other generative approaches, it is not fast enough to be run in real-time on normal hardware and is not able to deal with object occlusions.
Unlike most other hand pose estimation techniques our method focuses only on estimating the hand's 3D orientation rather than the articulated hand pose. It works in real-time using single depth images, aims to be applied both in close and far range scenarios and does not need a pose initialization. Like most other approaches, our method can only resolve self-occlusions.

3 Background

3.1 Introduction

Originally, efforts to develop artificial intelligence (AI) focused primarily on problems that defied human solution but were easily handled by computers. Typically, these problems required massive computational capacity, and since they adhered to a machine's formal rules, they were also able to exploit the machine's inherent advantages over a human's computational ability.
The difficulties began when AI was applied to solve tasks that are easy or even intuitive for people to do, but cannot be easily described by a set of rules or a formal language. The most basic everyday tasks like recognizing speech or faces turned out to be very hard to be learned by a machine because they require a lot of knowledge about the world. People acquire this knowledge automatically in the course of their life, but fail to teach it explicitly to a computer. In the advent of AI, several projects tried to create databases with statements and inference rules, also known as the *knowledge based* approach. But none of these inference engines was very successful as they failed to comprehend easy matters, because it is extremely difficult to find a precise formal description of the complex real-life relations and situations.
Machine learning is a different approach in AI. For example if the task is to design an algorithm for face recognition, knowledge based approaches would try to define hard-coded rules what a face should look like. In contrast, a machine learning approach would use a set of example images of faces and images, that do not show faces, to *learn* what a face is. After the period of learning (training) it is able to discriminate a face from a non-face.
There are different kinds of machine learning algorithms. First of all, we distinguish between supervised and unsupervised algorithms. In supervised learning, the training data are labeled, like in the previous example of face detection. In unsupervised

learning, examples are not labeled. The algorithm tries to find common patterns in big data and clusters them into groups [12]. In this thesis, we will focus on supervised learning methods. There are also different subcategories of supervised learning, mainly classification and regression problems. A classification is a mapping of inputs x to a discrete number of output classes y. For example, suppose we have images as input, a classification algorithm could group the content of the images into learned categories like cars or faces. In regression, by contrast, the output variable is continuous, e.g. it could be used to predict housing prices from the living area [17].
Success or failure of machine learning strongly depends on the representation of data. The different pieces of information that form the representation are called features. Machine learning algorithms learn how to combine these features to obtain a certain result. For some problems it is easy to find the right set of features, but for some tasks it is not intuitive what features should be extracted. For example, if we want to find a feature suitable for detecting cars, we could use the wheels. However, it is very hard to describe what a wheel looks like in pixels, because of the image's perspective, lighting, occlusion, etc.
Deep learning is a way to deal with this problem. High level representations of features are described by a combination of lower level representations. Easy features like edges are combined to corners and contours which finally can represent the concept of an image. As this hierarchy can include various stages and is thereby a *deep* hierarchy, the method is called *deep learning*. We used *deep learning* with *artificial neural networks* for our hand pose estimator and will therefore explain it in detail in this chapter [12].

3.2 Basics

Before we have a look at more advanced machine learning techniques, we will cover basic machine learning principles that help to understand the more sophisticated algorithms of *deep learning*.

3.2.1 Linear Regression

Linear regression is a simple machine learning model for supervised learning and more precise - as the name already implies - regression problems. The relation of input x and output y is approximated by a linear function [12]. We define the linear function as a hypothesis h

$$h(x) = w^\intercal x, \tag{3.1}$$

with input vector x and a vector of parameters w, called weights. It can be also expressed in a non-vectorized form

$$h(x) = \sum_{i=0}^{n} w_i x_i, \tag{3.2}$$

with n being the number of features.

Let's have a look at an easy example. Suppose we want to predict housing prices with the size of the living area, given a dataset of m examples of houses with their area x and price y, as shown in Table 3.1. Linear regression tries to find an optimal

living area (m^2)	price (1000€)
185	120
120	410
80	280
202	610
63	250
115	432
.	.
.	.
.	.

Table 3.1: Housing prices

assignment for weights w, so as to find a linear function h which approximates the dataset best (Figure 3.1).
We define a cost function

$$J(w) = \frac{1}{2} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2, \tag{3.3}$$

which measures the distance between prediction $h(x)$ and training output y. In order to learn the weights w, the algorithm considers the training examples and tries to minimize $J(w)$, i.e. make $h(x)$ close to y [18]. To minimize $J(w)$, we can solve for where its gradient is 0. However, to solve the function analytically one has to compute

$$w = (x^\intercal x)^{-1} x^\intercal y, \tag{3.4}$$

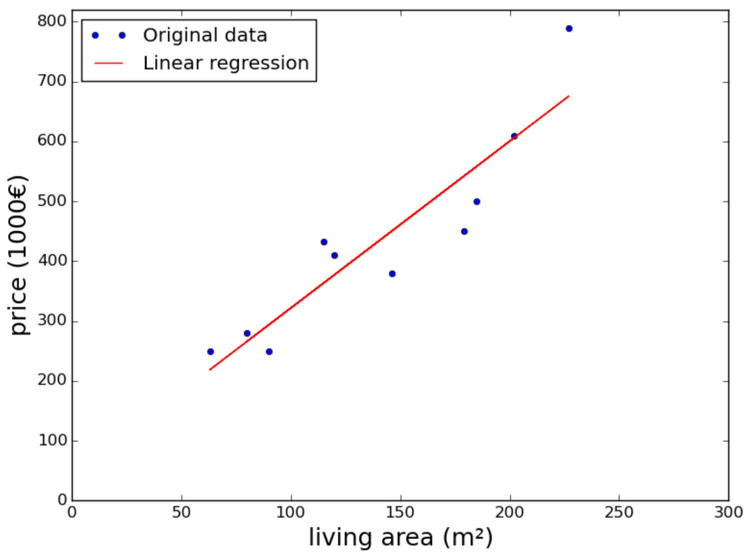


Figure 3.1: Housing prices: The dataset of houses (blue points) is approximated by a linear regression function (red).

which can result to be very inefficient in time, especially for datasets with a lot of features. That's why in practice, an approximative method called *gradient descent* is used to minimize $J(w)$.

3.2.2 Gradient Descent

Gradient descent is a search algorithm which starts with an initial guess for w and tries to minimize $J(w)$. The update step for each w_j of w, $(j = 0, ..., n)$ looks as follows

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(w), \tag{3.5}$$

with α being called the learning rate, a parameter that controls the speed of convergence. Intuitively, this algorithm moves in the direction of the steepest descent of $J(w)$ until convergence or failure (if the learning rate is too big). In general, gradient descent just finds local minima, but in case of linear regression, the cost function J is convex quadratic and the algorithm thus always finds the global minimum as there is no other minimum. To apply the algorithm, we have to solve the partial derivative of $J(w)$ with respect to every w_j. To make it easier, we will first consider having only one training example:

$$\frac{\partial}{\partial w_j}J(w) = \frac{\partial}{\partial w_j}\frac{1}{2}(h(x)-y)^2 \tag{3.6}$$

$$= 2\cdot\frac{1}{2}(h(x)-y)\cdot\frac{\partial}{\partial w_j}(h(x)-y) \tag{3.7}$$

$$= (h(x)-y)\cdot\frac{\partial}{\partial w_j}(\sum_{i=0}^{n} w_i x_i - y) \tag{3.8}$$

$$= (h(x)-y)x_j. \tag{3.9}$$

If we insert this in Equation 3.5, we get the following update rule:

$$w_j := w_j + \alpha(y^{(i)} - h(x^{(i)}))x_j^{(i)}. \tag{3.10}$$

For more than one training example there are two modifications of the algorithm: Batch gradient descent (BGD) and stochastic gradient descent (SGD). BGD always considers every training example before it updates the weights. The algorithm looks as follows:

```
Repeat until convergence {
        $w_j := w_j + \alpha\sum_{i=1}^{m}(y^{(i)} - h(x^{(i)}))x_j^{(i)}$        (for every $j$).        (3.11)
}
```

In contrast, SGD updates the weights for every training example:

```
Loop {
    for i=1 to m, {
        $w_j := w_j + \alpha(y^{(i)} - h(x^{(i)}))x_j^{(i)}$        (for every $j$).        (3.12)
    }
}
```

SGD does not have to scan the whole dataset before moving towards the minimum and thereby often gets good results faster than batch gradient descent, although it may never fully converge [18].

3.3 Deep Learning

Machine learning models like linear regression or logistic regression and other simple machine learning algorithms, that we did not consider in this work, use a fixed set of features to train a predictor. However, there are problems, such as visual recognition tasks, where it is difficult and very time-consuming to manually craft features. How would you design features for let's say a face detector? You would have to find a way to encode which pixel values correspond to eyes, nose or other parts of the face in order the detect it.
By contrast, deep learning solves this problem by learning new features and autonomously creating a new feature space. Originally, research on deep learning was inspired by new findings in neuroscience. E.g. it is believed that the visual cortex first extracts edges, then surfaces, then objects, etc. [17]. Deep learning also uses this kind of layered architecture, with different stages that learn increasingly complex features, and thus being able to represent arbitrarily complex non-linear transformations [12]. The name *artificial neural networks* which is often used instead of deep learning also symbolizes the connection to the brain and the objective of replicating its functionality.
Deep Learning first became popular in the 80s and 90s, but then disappeared from the scene until the resurgence in recent years. The development of faster computers, especially graphics processing units, was crucial since deep networks, i.e. networks with many layers, have enormous hardware requirements and could not realize the same functions as today with hardware from the 80s. Now deep learning is the state of the art in machine learning and its algorithms have outperformed other machine learning techniques in various competitions.

3.3.1 Neural Networks

To understand how neural networks work we will first have a look at a simple model, a network with only one neuron as illustrated in Figure 3.2. This neuron takes 3 inputs x_1, x_2, x_3 and a +1 bias term and computes the output

$$h_{W,b}(x) = f(W^\intercal x) = f(\sum_{i=1}^{3} W_i x_i + b), \tag{3.13}$$

where W denotes the vector of weights, b denotes the bias term and f is the activation function. While the weights change the steepness of the activation function, the

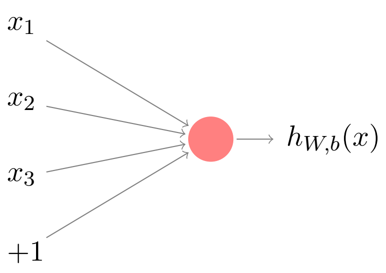


Figure 3.2: Model of a single neuron with 3 inputs x_1, x_2, x_3, a $+1$ bias term and 1 output.

bias allows to shift it to the left or right. There are several possible activation functions such as the sigmoid function, the hyperbolic tangent function or the rectifier function. Although our approach uses Rectified Linear Units (ReLU), i.e. a unit that uses the rectifier function, we will explain neural networks by means of the sigmoid function

$$f(z) = \frac{1}{1 + exp(-z)}. \tag{3.14}$$

Figure 3.3 shows a plot of the sigmoid function.

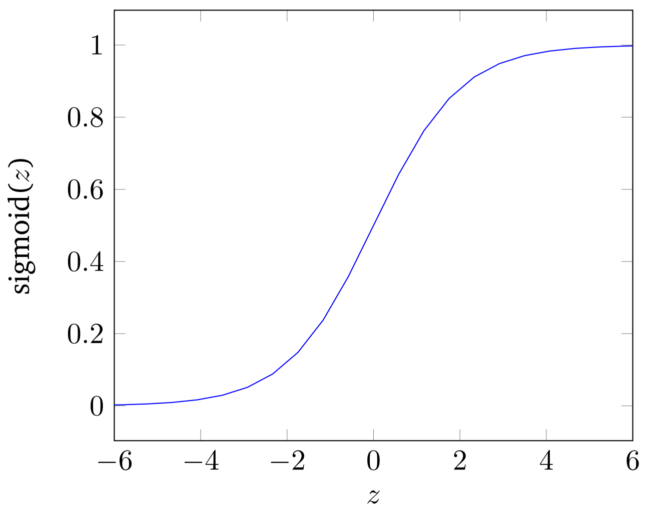


Figure 3.3: The sigmoid function maps the inputs to the range [0,1].

A neural network is the connection of many neurons (or units, as they are also referred to). We will use the simple network shown in Figure 3.4 for explanation. The units are grouped into different layers, the left layer (L_1) is called input layer, the right one (L_3) output layer and the middle layer (L_2) hidden layer. Accordingly our

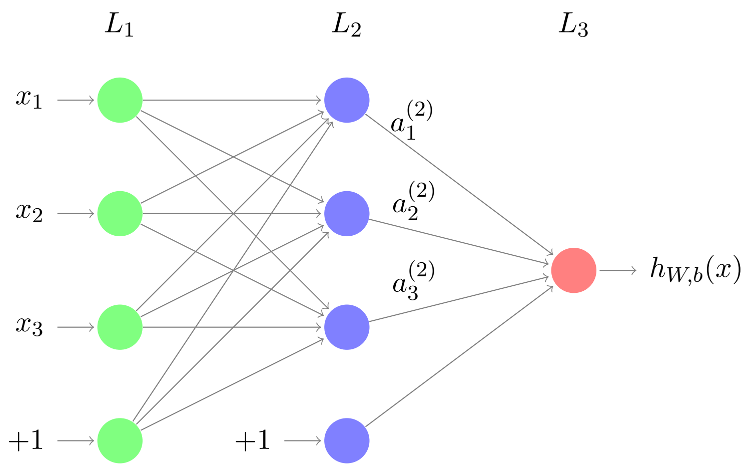


Figure 3.4: Neural network model with 3 layers.

model has 3 input units (without counting the +1 bias unit), 3 hidden units and 1 output unit. Of course a neural network can have multiple hidden layers or output units, but for the sake of simplicity we explain a model with only one hidden layer and one output unit. It is an example of a feedforward neural network because its graph is acyclic [19]. As all units from neighboring layers are connected, the layers are also called fully-connected layers.

3.3.2 Forward Propagation

Let's have a look at how a network computes the output on the basis of the former example. First of all it is necessary to define some denotations. A layer l has a matrix of weights $W^{(l)}$, where $W_{ij}^{(l)}$ denotes the weight of the connections between unit j in layer l and unit i in layer $l+1$. The bias of unit i in layer $l+1$ is written as $b_i^{(l)}$. The output of unit i in layer l is called activation $a_i^{(l)}$, accordingly the input $x_i = a_i^{(1)}$. The final output of the network, hypothesis $h_{W,b}^{(x)}$ is computed as follows:

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \tag{3.15}$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \tag{3.16}$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \tag{3.17}$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1 + W_{12}^{(2)}a_2 + W_{13}^{(2)}a_3 + b_1^{(2)}). \tag{3.18}$$

It is possible to introduce another variable $z_i^{(l)}$ which signifies the weighted sum of inputs to unit i in layer l, so that $a_i^{(l)} = f(z_i^{(l)})$. Now the equations above can be expressed in a vectorized fashion:

$$z^{(2)} = W^{(1)}x + b^{(1)} \tag{3.19}$$

$$a^{(2)} = f(z^{(2)}) \tag{3.20}$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{3.21}$$

$$h_{W,b}(x) = a^{(3)} = f(z^{(3)}) \tag{3.22}$$

Generically this can be rewritten as

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)} \tag{3.23}$$

$$a^{(l+1)} = f(z^{(l+1)}) \tag{3.24}$$

This step is called forward propagation [19].

3.3.3 Backpropagation

Assuming we have m training examples we denote the labeled training set as $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$. Analogous to linear regression, we can define a cost function J that measures the nets accuracy by comparing the hypothesis $h_{W,b}(x)$, after having performing a forward propagation, with the ground truth y (i.e. the correct value associated with x). Considering only a single training example one possible way to define the cost function is:

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2. \tag{3.25}$$

Then the cost function for a whole set with m training examples is defined in the following way:

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2 \tag{3.26}$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2, \tag{3.27}$$

where n_l denotes the number of layers and s_l denotes the number of units in layer l (without the bias). The second term is called weight decay, it is a regularization

method applied to prevent weights from adopting big values by penalizing large weights. Thus the weights converge to smaller absolute values which helps to prevent overfitting. Overfitting means that the function the network computes specifically fits the training set, but does not provide a general solution for the problem. Hence test results are bad even though the training results were good.
In the previous section, we explained how gradient descent is used to minimize the cost function of linear regression. Now the same can be done for neural networks, where the goal is to minimize $J(W, b)$.
First, all weights and biases are initialized randomly with small values and the output of the network is computed with the forward propagation algorithm for every example of the batch. Now gradient descent can be used to update the parameters:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \tag{3.28}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b). \tag{3.29}$$

The partial derivatives of the cost function for the whole training set is composed of the sum of the partial derivatives with respect to a single training example and the derivative of the weight decay term:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \tag{3.30}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)}). \tag{3.31}$$

There is an efficient way to compute the partial derivatives called backpropagation. It first computes the error $\delta_i^{(n_l)}$ at the outputs and then propagates the error term $\delta_i^{(l)}$ backwards into the network to compute how much every unit was responsible for the output error:

1. Compute the outputs performing a feedforward pass given a training example (x, y)

2. Compute output error

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{3.32}$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$
   For each node i in layer l, set

$$\delta_i^{(l)} = (\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_i^{(l+1)}) f'(z_i^{(l)}) \tag{3.33}$$

4. Compute the needed partial derivatives

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)} \tag{3.34}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}. \tag{3.35}$$

Again these equations can be rewritten in a matrix-vectorial fashion with $\bullet$ denoting the element-wise product. Then the algorithm looks as follows:

1. Feedforward

2. Compute output error

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \tag{3.36}$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$
   Set

$$\delta^{(l)} = ((W^{(l)})^\intercal \delta^{(l+1)}) \bullet f'(z^{(l)}) \tag{3.37}$$

4. Compute the needed partial derivatives

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^\intercal \tag{3.38}$$

$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}. \tag{3.39}$$

Now the algorithm can be implemented very efficiently using linear algebra routines [19].

3.3.4 Convolutional Neural Networks

In the former section we explained the basic ideas of neural networks and how their algorithms work. However, for some tasks it is useful to apply special kinds of neural networks, namely convolutional neural networks. They are used, if the data has a grid-like structure, like pixel grids of images in the 2D case. The main difference to conventional networks is that instead of performing matrix multiplications, a convolutional network uses convolution at least once. Mathematically, convolution is an operation that for two functions f and g returns a third function $f * g$. Convolution can be explained by weighing function f (also called input) with g (also called kernel) and thus obtaining something like a weighted mean of f. Convolution can be performed in continuous and discrete space, here we will focus on the discrete case as our inputs are discrete (e.g. images expressed by pixel matrices). Furthermore, in machine learning the term convolution is often used when actually cross-correlation is meant, both are quite similar with the difference that convolution flips the kernel and cross-correlation does not. Below we will refer to convolution as convolution without kernel flipping i.e. cross-correlation, which is defined as

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n]K[m,n], \tag{3.40}$$

where s denotes the output, I the input and K the kernel with dimensions $m \times n$. Figure 3.5 shows an simple example of 2D convolution.

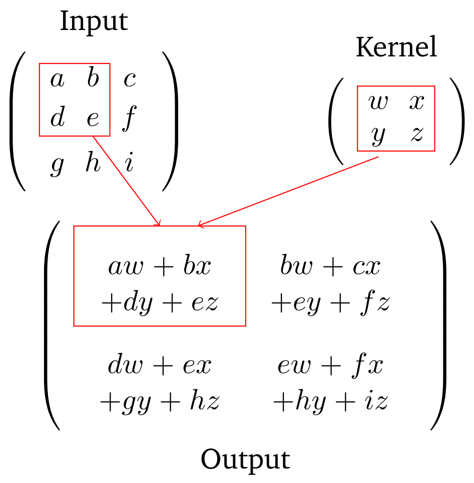


Figure 3.5: Convolution without kernel-flipping: Note that the output dimensions decrease if padding (i.e. extending the input with zeros) isn't applied.

The same operation could also be performed by a matrix multiplication, that means any neural network algorithm could theoretically perform convolution, although it would not make much sense. The matrix would be very large for big inputs (e.g. images with high resolution) with a lot of same entries and most entries equal to zero. In normal neural networks each input unit is connected to each unit in the next layer.
In contrast, convolutional networks use kernels that are much smaller than the input and thus have sparse connectivity, i.e. not every output unit interacts with every input unit (Figure 3.6). To detect features like edges or corners in possibly large images it is not necessary to consider every pixel at once, but it is sufficient to look at small regions with kernels of only tens of pixels. Small kernels mean that fewer parameters have to be stored and fewer operations are needed to compute the output, thus improving memory requirements and efficiency. This way to process images was inspired by findings in neuroscience. Cells in the visual cortex respond to small sectors of the visual field, called receptive field. Convolutional neural networks imitate this concept, with the convolution kernel being the equivalent of the receptive field [12].

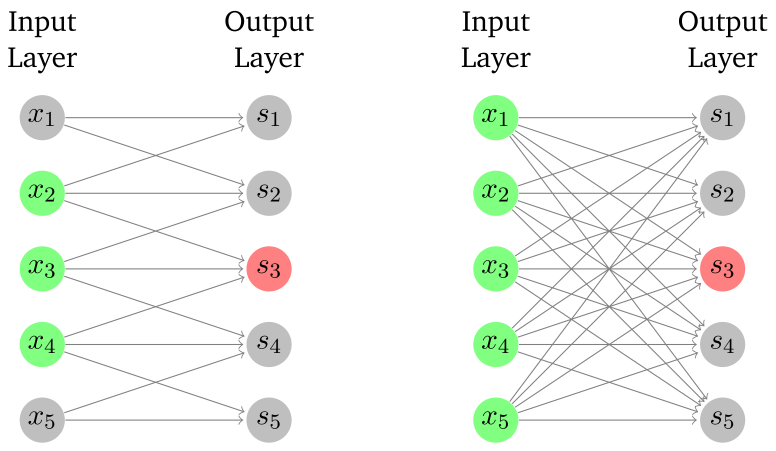


Figure 3.6: Sparse connectivity: (Left) In the convolutional network only the green highlighted units affect output s_3, they are called receptive field of s_3. (Right) In the fully connected network all input units affect output unit s_3.

Another advantage of convolutional networks is parameter sharing. In traditional fully connected neural networks each weight, i.e. each entry of the weight matrix, is only used once. A convolutional network, however, needs significantly less weights because each kernel is moved over the whole input using the same weight at every

position. Thus shared weights require less memory and increase learning efficiency because fewer free parameters have to be learned.
Furthermore, parameter sharing brings the useful property of equivariance to translation. This means if the input is shifted, the output is equally shifted. E.g. if the input is an image and convolution detects the edges, a translation in the input images leads to the same translation in the edge map. Convolution can also deal with changing input sizes whereas conventional neural networks with fixed-shape matrices require constant input sizes [12].
In a typical convolutional network architecture, each layer consists of three stages. In the first stage various kernels perform convolutions. The output is run through a rectified linear unit (ReLU) which is the usual activation function for convolutional networks. The rectifier function $rectifier(x) = max(0, x)$ is more biologically plausible and efficient than sigmoid or tanh activation functions and leads to a sparse representation of the network because real zero values can be adopted. With a random initialization, only half of the units are activated (i.e. having non-zero outputs) [13].
The third stage is the pooling stage, which is another important concept of convolutional neural networks. Pooling is a form of non-linear down-sampling and thus decreases the output space. The most common pooling function is max-pooling. The output of the previous stage is divided into rectangular cells and replaced by the maximum value in each cell. Apart from reducing computation for upper layers, pooling is also beneficial because it makes the network invariant to small local translations. That means that even if the input is slightly shifted, the output response does not vary [12].
Usually a convolutional neural network does not exclusively use convolution layers, but a mixture of convolutional layers at the beginning and fully-connected layers (with regular matrix multiplications) towards the output.

4 Method

In this chapter we want to present our method in detail. As mentioned before, our approach does not aim to improve existing body part tracking methods, but solely focuses on estimating the hand pose. Therefore we make use of existing hand trackers and build our method upon them. We present a two stage architecture with the first stage being the localization stage and the second stage being the pose estimation stage, as shown in Figure 4.1.

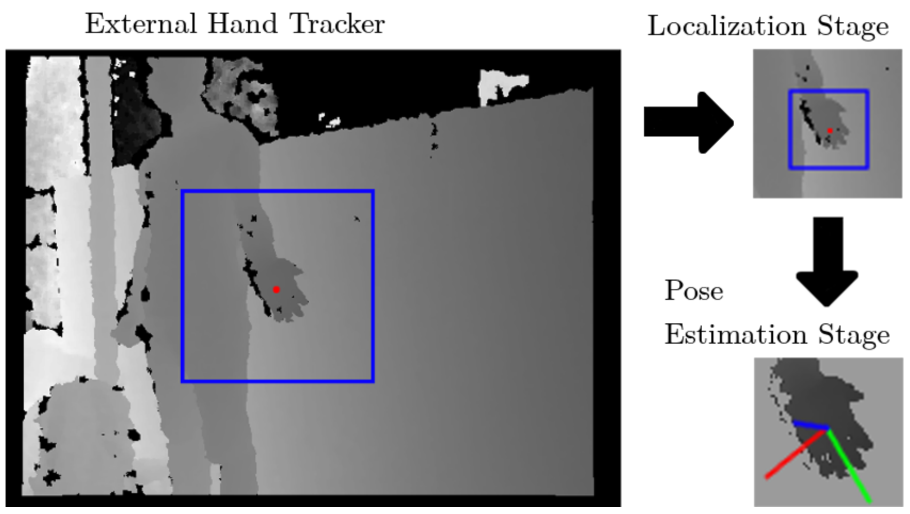


Figure 4.1: The hand orientation estimation pipeline: The external hand tracker gives an initial guess for the hand position which is used by the localization stage to determine the exact hand position. The pose estimation stage finally computes the hand orientation.

4.1 Localization Stage

The localization stage takes the proposed hand location from the external hand tracker and uses it as initial guess. This is important because the next stage needs a very exact estimate in order to predict the correct orientation. It turned out that even slight translations can lead to a significant drop in performance. Therefore the external tracker can only serve as a rough estimate.

4.1.1 Dataset Creation

We created a labeled dataset of approximately 160.000 images of left hands for training purposes: First, we took around 1600 depth images from different point of views varying the camera's height and distance to the hand. The images show hands of all kinds of poses using every possible degree of freedom.
Then we used our own keypoint annotation tool to label every image with the hand's center position which we defined as the center between middle finger root and ring finger root. In perspectives where this point could not be directly seen (e.g. if the hand is in a horizontal position) we chose the point which was directly in front of the hidden actual center.
Around this point in every image, we drew 100 quadratic bounding boxes with a size of 200×200 pixels, which were all randomly translated and rotated to augment the dataset, then cropped and downsized the images to 64×64 pixels (see Figure 4.2). The newly created images were randomly divided into training and test groups. A normalized pixel vector of the hand's center was also stored for every image.

4.1.2 Deep Network Training

We used the labeled dataset to train a convolutional neural network with four convolution layers (using ReLUs and max-pooling) and two fully-connected layers. The precision of the training was measured by the mean squared error between prediction and ground truth:

$$L(h, y) = \frac{1}{2} \|h(x) - y\|_2^2, \tag{4.1}$$

with $h(x)$ being the predicted value and y being the ground truth value. The architecture of our network is explained in detail in Figure 4.3. Informations about the framework and parameters we used for training are listed in Chapter 6.

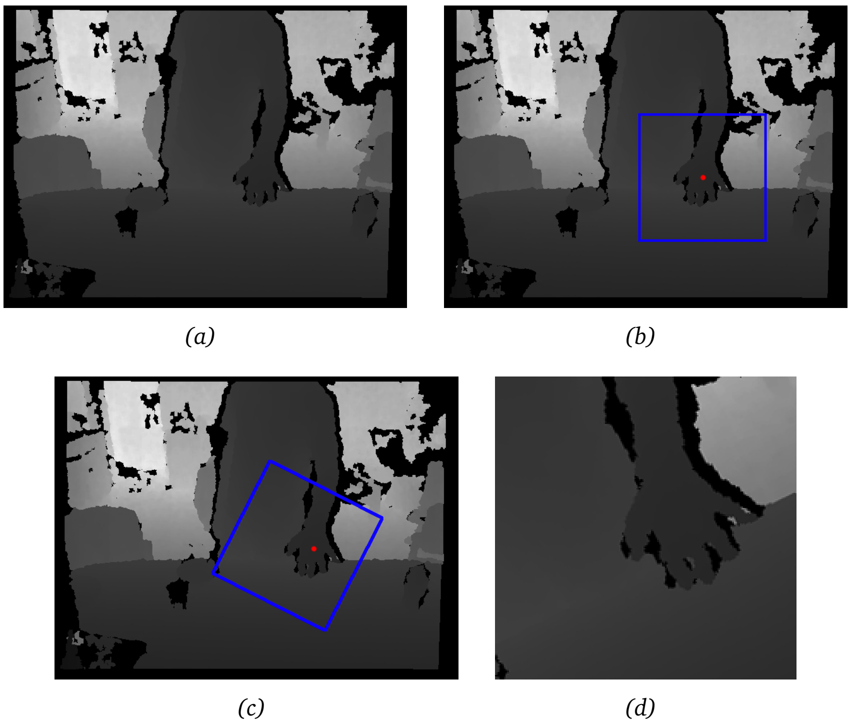


Figure 4.2: Dataset for the localization stage:
(a): Original depth image.
(b): Center point selected (red dot) and bounding box drawn (blue rectangle).
(c): Bounding box randomly shifted and rotated.
(d): Cropped training example.

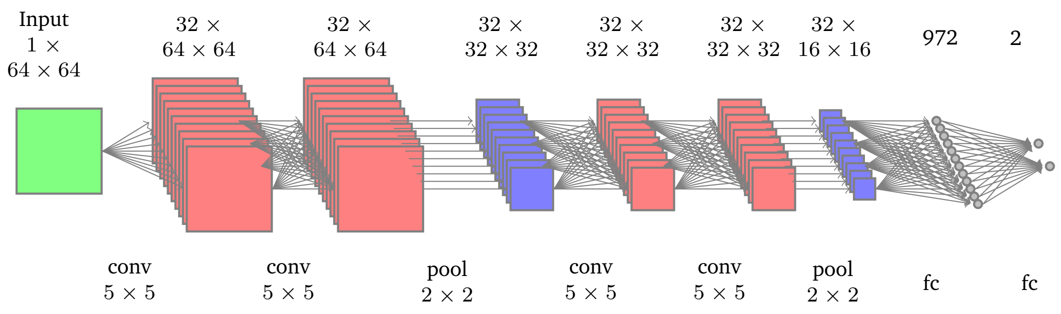


Figure 4.3: Network model for the localization stage:
Green: Input image (1 channel with 64×64 pixels).
Red: Feature maps after convolution with zero-padding of 2 and a stride of 1.
Blue: Feature maps after max-pooling with stride of 2.
Grey: Fully-connected layers.
Note that for simplicity we do not draw every connection and feature map.

4.1.3 Application

The trained model can be used for feature extraction. First the external body tracker gives us the initial guess for the position of the right and left hand as a 3D vector. We project the vector from world coordinates to 2D screen coordinates using the following conversion:

$$\begin{pmatrix} x^{(2D)} \\ y^{(2D)} \end{pmatrix} = \frac{f}{z^{(3D)}} \begin{pmatrix} x^{(3D)} \\ y^{(3D)} \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \end{pmatrix}, \tag{4.2}$$

with f and (c_x, c_y) denoting the camera's intrinsic parameters focal length and optical center. Around this position we draw a fixed-size bounding box which very likely contains the hand. The image in the bounding box is cut out, resized and mirrored in case of a right hand (we just trained with left hands). Now our network model uses it to predict the exact center position of the hand. In case of a right hand, the predicted point has to be horizontally flipped back.

4.2 Pose Estimation Stage

After the localization stage we have a reliable estimate of the hand position. The pose estimation stage now does the actual computation of the orientation. We developed two different approaches, the keypoint method and the quaternion method,

which work almost equally well and present both because either of them could be interesting for future scenarios.
Like in the previous stage, we created a dataset and used it to train a convolutional neural network. Both approaches use the same dataset, but with different labels. Figure 4.4 shows the notation for the coordinate system of the left hand.

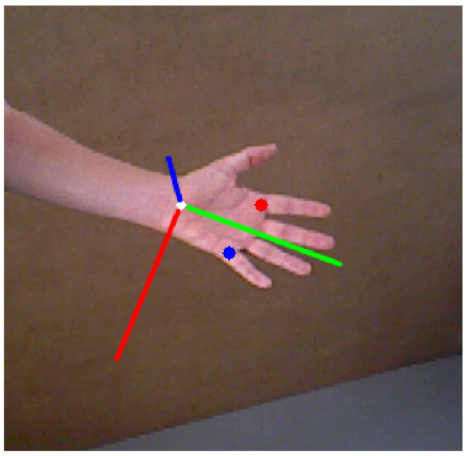


Figure 4.4: Left hand example image with coordinate system: the red axis denotes the x-axis, the green axis the y-axis and the blue axis the z-axis. Note that we use right-handed coordinate systems for both the left and the right hand, hence the x-axis is in the direction of the thumb for the right hands, but in the opposite direction for the left hand.

4.2.1 Keypoint Approach

Using the keypoint method we do not directly try to estimate the hand's orientation, but three keypoints on the hand which can subsequently be used to actually compute the explicit orientation with simple linear algebra. We chose the root of the index finger, the root of the little finger and the center of the wrist as keypoints (see Figure 4.4).

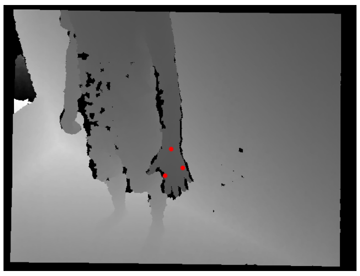
4.2.1.1 Dataset Creation

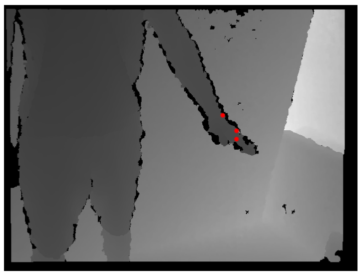
Creating the labeled depth image dataset for the pose estimation stage is not as straightforward in comparison to the localization stage. If the z-axis of the hand is facing towards the camera or in the opposite direction, all keypoints are visible. With our annotation tool, we can manually select those points in every depth image, read out the depth and project each point from screen coordinates to world coordinates.

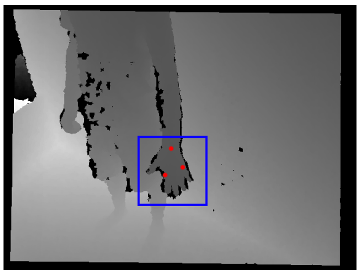
However, if the z-axis is facing down, up, left or right, it is more complicated because of self-occlusion (i.e. at least one of three points is not visible).
We used a two-camera setup to deal with this problem. The cameras were placed at different positions with overlapping fields of view, such that the palm or dorsum of the hand was always clearly visible to one of them. We calibrated the cameras using QR markers. After the calibration, we could annotate the points in one camera frame and transform the 3D vectors of our keypoints to the other camera frame, where the annotation would not have been possible otherwise. A positive side effect of this method is that we could obtain twice as many labeled images in the same time. Again, we only labeled left hands.
The points, which until now were located on the surface of the hand, were translated along the z-axis to be inside the hand. This is important in order to have consistent annotations from palm and dorsal side of the hand.
Around the center of the hand, we computed a bounding box in 3D space with a width and height of 0.24 m and transformed the vertices of the 3D bounding box to the screen using Equation 4.2. As a result the obtained 2D bounding box depended on the depth of the hand, so that for every depth, the hand occupies the same space. Then we normalized the depth to make sure that every hand lies in the same depth range by linearly shifting the whole depth image such that the center of the hand had the depth 0.27 m. The proportion of the hand was left unchanged. This specific value was chosen due to former experiments with an artificial hand dataset [25] whose images had this depth property. Furthermore we used thresholding to remove the background setting all depth values lower than 0.1 m and higher than 0.4 m to 1 m. We augmented the dataset with factor 50 for each image. Since we used two cameras, we got 100 labeled images for each annotation step which were randomly split into training and test examples.
We applied the following augmentation techniques to maximize the diversity of our dataset:

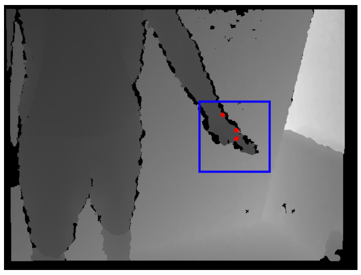
- random translation
- random rotation
- random depth shift

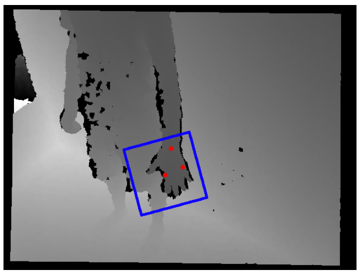
Figure 4.5 illustrates the generation of training examples in our two-camera setup.

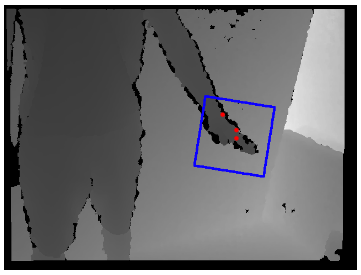

(a) Camera 1

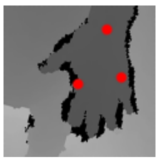

(b) Camera 2

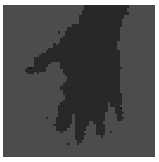

(c)

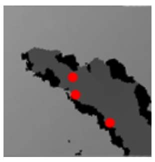

(d)


(e)


(f)


(g)


(h)


(i)

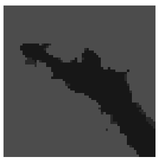

(j)

Figure 4.5: Dataset for the pose estimation stage:
(a) and (b): Same scene from two cameras. Key points are annotated in (a) and transformed to (b).
(c) and (d): Bounding box (blue rectangle).
(e) and (f): Bounding box randomly shifted and rotated.
(g) and (i): Cropped hand.
(h) and (j): Training example after thresholding and depth normalization.

We saved the keypoints as labels in the form

$$[x^{(1)} \quad y^{(1)} \quad z^{(1)}] \quad [x^{(2)} \quad y^{(2)} \quad z^{(2)}] \quad [x^{(3)} \quad y^{(3)} \quad z^{(3)}], \tag{4.3}$$

with x and y being the screen coordinates of the keypoint normalized with respect to the bounding box and z being the normalized depth. Originally we created a dataset with approximately 290.000 images. However, we realized that hand poses were not represented equally frequent which caused our detection to favor more common poses. In order to solve this issue, we randomly removed frequent orientations until they were almost uniformly distributed in the dataset. With about 87.000 images, the revised dataset contains considerably less images than the big dataset, but nevertheless achieved better results.

4.2.1.2 Deep Network Training

Like in the localization stage, we trained a convolutional neural network on the dataset using the mean squared error loss function (Equation 4.1). The network directly regresses the 3D keypoint locations. It consists of three convolution layers, three pooling layers and two fully-connected layers, all using ReLUs. Figure 4.6 depicts the network architecture.

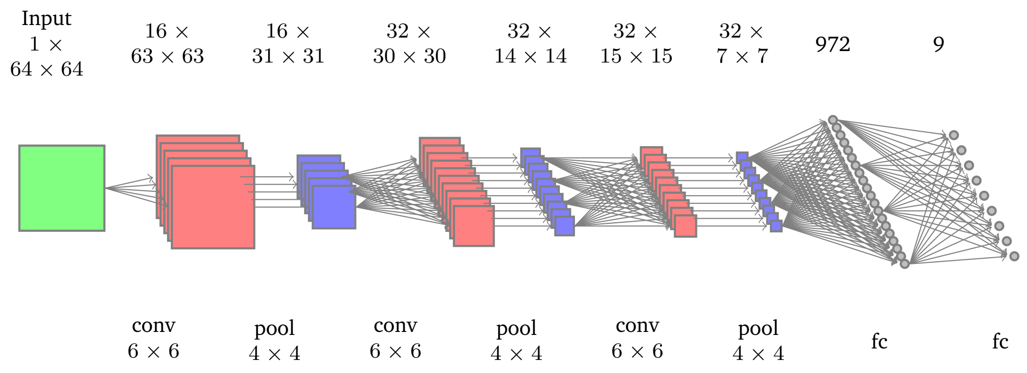


Figure 4.6: Network model for keypoint pose estimation: Green: Input image (1 channel with 64×64 pixels). Red: Feature maps after convolution with zero-padding of 2 and a stride of 1. Blue: Feature maps after max-pooling with stride of 2. Grey: Fully-connected layers. The last layer has 9 outputs (3 vectors with 3 components each. Note that for simplicity we do not draw every connection and feature map.

4.2.2 Quaternion Approach

By contrast to the previous method, for the quaternion approach we directly estimate the orientation of the hand, without having to compute it from keypoints. We represent the orientation using unit quaternions, which can express arbitrary 3D-rotations.

4.2.2.1 Dataset Creation

We used the same dataset as described for the keypoint approach, but changed the labels of the training examples. Hence the performance of both methods can be easily compared.
To convert the labels from keypoints to quaternions we first compute the base vectors x, y, z of the hand coordinate system from the keypoints (see Figure 4.4). Then we define a rotation matrix M which transforms the hand frame to the camera frame using the vectors of the hand coordinate system as columns:

$$M = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix} \tag{4.4}$$

From this matrix we compose Matrix K

$$K = \begin{pmatrix} x_1 - y_2 - z_3 & 0 & 0 & 0 \\ x_2 + y_1 & y_2 - x_1 - z_3 & 0 & 0 \\ x_3 + z_1 & y_3 + z_2 & z_3 - x_1 - y_2 & 0 \\ y_3 - z_2 & z_1 - x_3 & x_2 - y_1 & x_1 + y_2 + z_3 \end{pmatrix}, \tag{4.5}$$

and divide all entries by 3

$$K := \frac{K}{3}. \tag{4.6}$$

After computing the eigenvectors and eigenvalues of K, the quaternion q which describes the same rotation as M is the eigenvector v of K that corresponds to the largest eigenvalue λ.
The unit quaternion is calculated as follows:

$$normalize(q) = \frac{q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3}{\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}}. \tag{4.7}$$

4.2.2.2 Deep Network Training

Unlike the previously described network, in this approach the use of the mean squared error loss function is not possible because the euclidean norm is not an appropriate distance measurement for quaternion representations.
Two unit quaternions q and p represent the same orientation, if their dot product $\langle q,p\rangle = \sum_{i=1}^{4}(q_i p_i)$ equals 1 or -1. If the orientations vary highly, $\langle q,p\rangle$ is close to 0. We used this to define our own quaternion loss function

$$L(q,p) = 1 - \langle q,p\rangle^2, \tag{4.8}$$

where q denotes the predicted quaternion and p the actual quaternion.
Additionally, our network also consists of a normalization layer after the last fully-connected layer which normalizes the output in order to obtain unit quaternions. This step is necessary because the loss function is designed for unit quaternions only. The architecture of our network is illustrated in Figure 4.7.

4.2.3 Application

From the localization stage we obtain a precise estimate of the hand position.
Next, we draw a square bounding box around the hand in the depth image. The size of the bounding box again depends on the depth of the hand. We also shift, scale and threshold the depth as described earlier for the creation of the dataset (see Section 4.2.1.1).
Both networks can be used for feature extraction. They take the cropped bounding box region of the depth image as input and compute the respective representation of the hand pose. As with the previous stage, images of the right hand have to be mirrored before being fed into the networks. Likewise, the output also has to be interpreted subsequently.
In case of the keypoints, it is enough to horizontally mirror the points. In case of quaternions, we first have to compute the base vectors of the hand coordinate system, i.e. transform the quaternion to a rotation matrix and extract the column vectors. The resulting vectors can then be mirrored to get the actual vectors.
If we also compute the vectors of the keypoint coordinate system, we can easily compare the results of both methods.

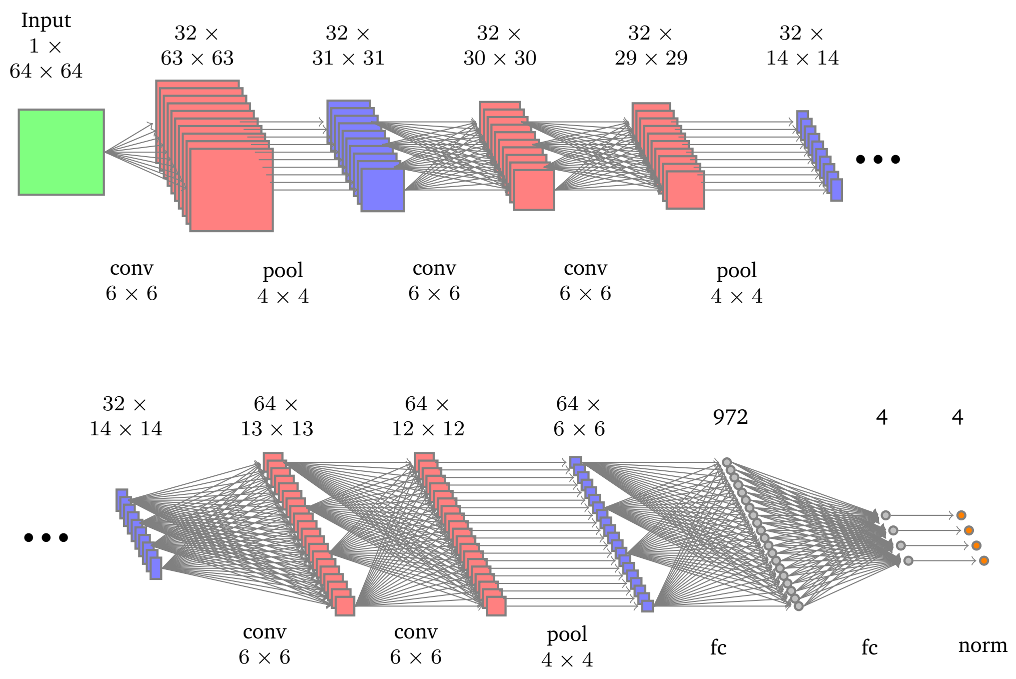


Figure 4.7: Network model for quaternion pose estimation:
For lack of space the figure is split in two.
Green: Input image (1 channel with 64×64 pixels).
Red: Feature maps after convolution with zero-padding of 2 and a stride of 1.
Blue: Feature maps after max-pooling with stride of 2.
Grey: Fully-connected layers. The last layer has 4 outputs.
Orange: Normalization layer, converts the output from the last fully-connected layer to unit quaternions. Note that for simplicity we do not draw every connection and feature map.

5 Experiments

5.1 Localization Stage

We tested the performance of the first stage of our method on an independent test dataset. The dataset was created as described in Section 4.1.1 and contains around 3000 labeled images. For each test example, the trained network computes an estimate of the center point of the hand. We define the error function as the pixel distance between the estimated point p and the ground truth point q:

$$error(p, q) = \frac{|p_x - q_x| + |p_y - q_y|}{2}. \tag{5.1}$$

The average error of our dataset is 4.2 with a standard deviation $\sigma = 2.6$ and 92.2% of the values are ≤ 8. Figure 5.1 shows the error distribution for the whole test dataset in a histogram.

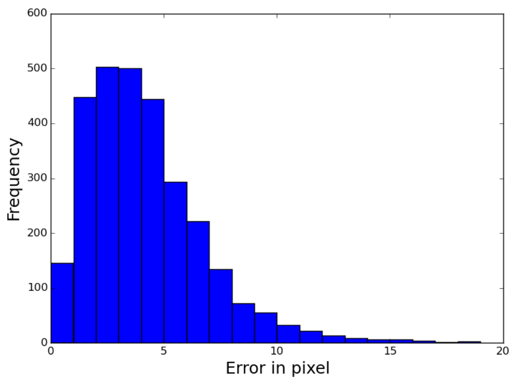


Figure 5.1: Histogram of errors: It shows the distribution of errors for the localization stage test dataset.

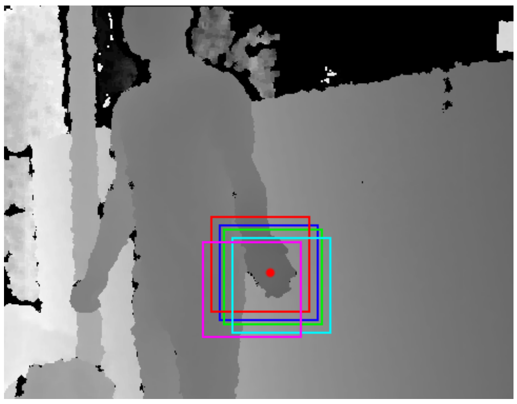


Figure 5.2: Creation of the test dataset with different shifts:
Blue bounding box: No shift.
Green bounding box: Shift of 4 pixels.
Red bounding box: Shift of 8 pixels.
Turquoise bounding box: Shift of 12 pixels.
Violet bounding box: Shift of 16 pixels.

5.2 Pose Estimation Stage

We performed a series of experiments to evaluate the performance of the pose estimation stage. We mainly focused on the comparison between the keypoint method and the quaternion method under different aspects in order to determine the advantages and disadvantages of the respective methods. For evaluation we created two independent test datasets.

The first one contains around 7800 images of hands with evenly distributed orientations. The dataset creation was similar to what we described in Section 4.2.1.1, but with some important changes.

In each annotation step, we augmented the depth image splitting it into 5 groups: No shift, shift of 4 pixels, shift of 8 pixels, shift of 12 pixels and shift of 16 pixels. The shift was performed in both x and y dimensions, with the respective direction (e.g. positive x-direction and negative y-direction) being randomly determined. Figure 5.2 illustrates this step for one test example. The purpose of this is to measure to what extend the networks can cope with inaccuracy of the previous stage.

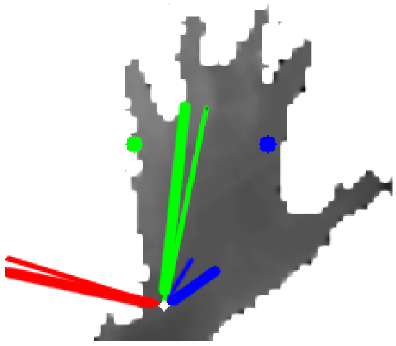


Figure 5.3: Depth image after feature extraction:
The thin lines denote the ground truth coordinate system computed from the labeled keypoints and the bold lines denote the predicted coordinate system from the quaternion network. In this case the error is approximately $10°$. The image appears pixelated because it is upscaled from 64×64 pixels for better visibility.

We defined the precision of the detection as the mean angle between the axes of the predicted coordinate system x, y, z and the actual coordinate system x', y', z' (see Figure 5.3). The angles are calculated as follows:

$$\alpha = \frac{180}{\pi} \arccos(\langle x, x' \rangle) \tag{5.2}$$

$$\beta = \frac{180}{\pi} \arccos(\langle y, y' \rangle) \tag{5.3}$$

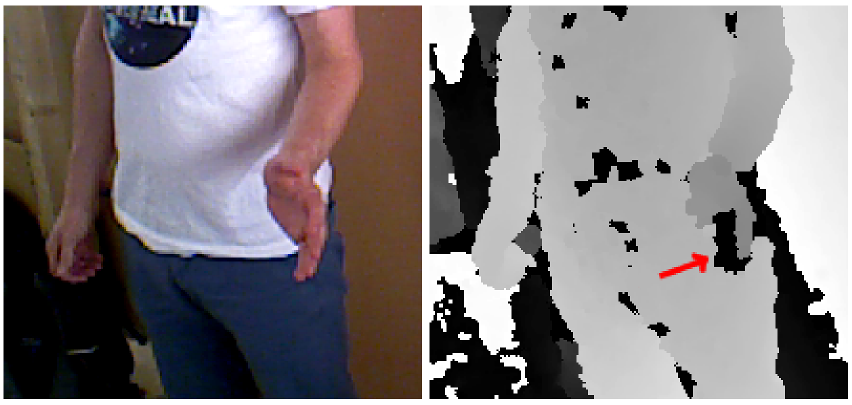
$$\gamma = \frac{180}{\pi} \arccos(\langle z, z' \rangle), \tag{5.4}$$

with $\langle \cdot, \cdot \rangle$ denoting the dot product.
The error function calculates the mean angle difference:

$$error(\alpha, \beta, \gamma) = \frac{\alpha + \beta + \gamma}{3}. \tag{5.5}$$

Smaller values for the error function mean higher precision. Figure 5.5 shows the average error of both networks for different shifts. The quaternion method achieves slightly better results than the keypoint method ($18.7°$ vs. $19.0°$ mean error for no shift and $18.8°$ vs. $19.5°$ for a shift of 4 pixels), with the difference becoming more noticeable for bigger shifts ($28.1°$ vs $31.6°$ for shifts of 12 pixels). The overall performance is not affected by a shift of 4 pixels, but significantly drops for images with a

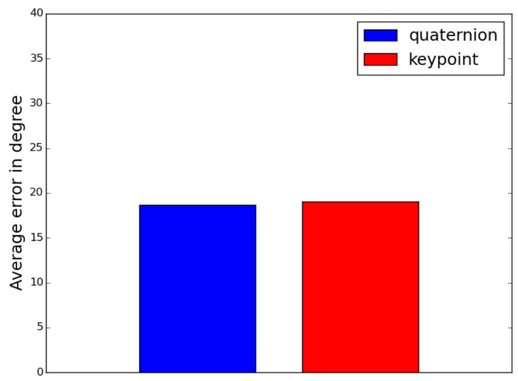
shift of 12 pixels.
Further, we investigated to what extend the hand orientation influences the precision of the prediction. For each hand in the dataset, we calculated the angle between the z-axis of the camera coordinate system and the z-axis of the hand coordinate system.
The angle is about 0° if the dorsum of the hand is facing the camera and about 180° if the palm is facing the camera. For hand poses where the hand's side faces the camera, the angle is around 90°. We divided the dataset in groups of similar orientations and compared the performance of the keypoint estimation and the quaternion estimation for every group. Figure 5.6 depicts the results of this experiment. Independent of the shift, the error is smallest if palm or dorsum directly face the camera and increases with less visibility. The disparity is most significant for images with a shift of 12 pixels, especially for the keypoint method. The increase in error for side views can mainly be explained by two reasons. First of all, if the individual fingers are not visible, there are less features available to be learned by the networks. Particularly the thumb proved to be an important feature to distinguish between front and backside of the hand. Secondly, the smaller the visible area of the hand is, the bigger are the difficulties of the depth sensor to correctly represent its shape because of not a number (NaN) values as illustrated in Figure 5.4.



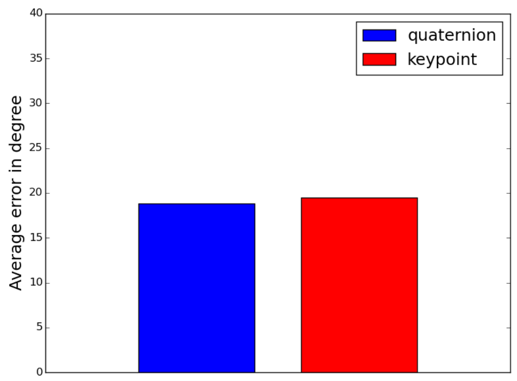
(a) RGB image *(b)* Depth image

Figure 5.4: Technical limitations of the depth sensor:
The hand in the depth image does not cover the same space as in the RGB image. Due to measurement errors, a part of the hand has NaN values (black pixels), which means that there is no depth information. This problem mainly occurs for side views of the hand.

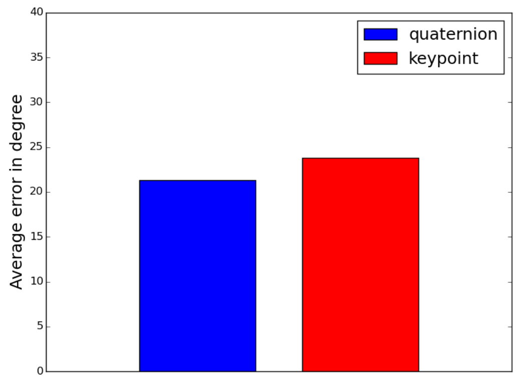
In order to have an overview about the distribution of errors we plotted histograms, which are shown in Figure 5.7. For no shifts or shifts of 4 pixels, the vast majority of orientations is predicted with an error of $30°$ or better, but if the shift increases a lot of orientations are estimated incorrectly. This again emphasizes the importance of a good performance of the localization stage.
In Figure 5.8 we compare the error distribution for all 5 shifts together in one histogram.
We recorded a second dataset with 1300 images with images of varying distance between camera and hand and divided the images into 10 groups of similar distances. Figure 5.9 shows that our method achieves decent precision for distances of more than 3 meters, with the keypoint method producing more stable results than the quaternion method. The high variance of the quaternion values could be due to the small dataset size.
Figures 5.10 and 5.11 show examples of images with estimated orientations of both methods. We selected examples where both methods work well, but also examples where one or both methods fail to estimate the correct orientation.



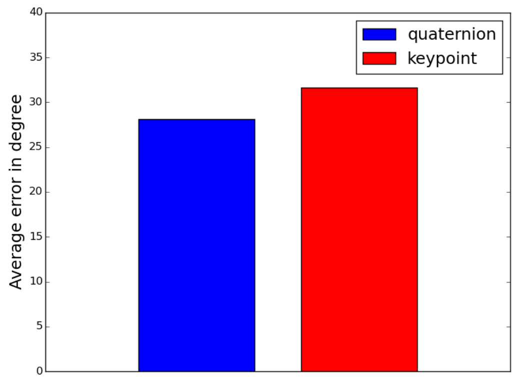
(a) No shift



(b) Shift of 4 pixels

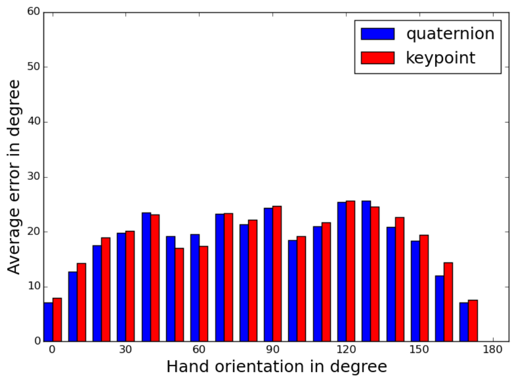


(c) Shift of 8 pixels

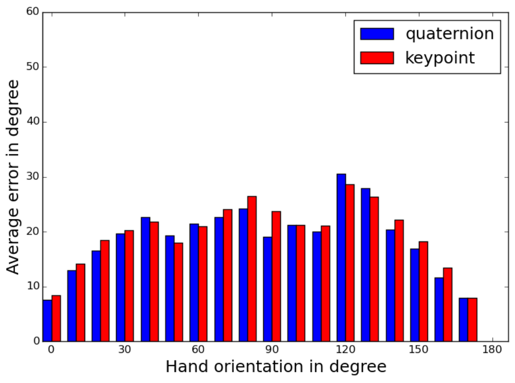


(d) Shift of 12 pixels

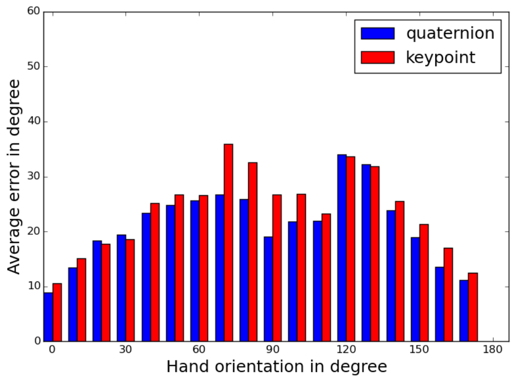
Figure 5.5: Average errors of feature extraction using the keypoint network and the quaternion network for different shifts.



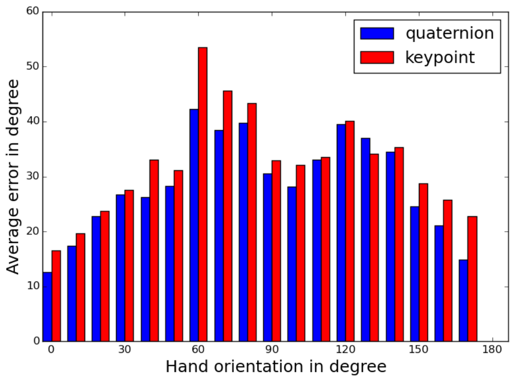
(a) No shift



(b) Shift of 4 pixels

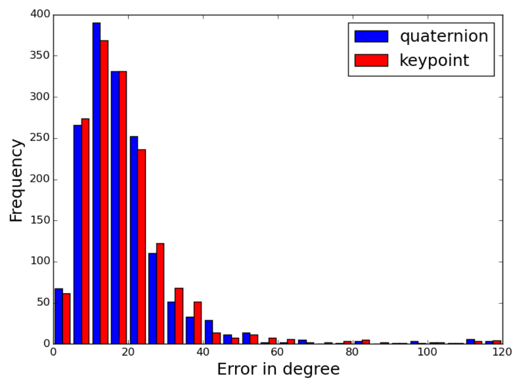


(c) Shift of 8 pixels

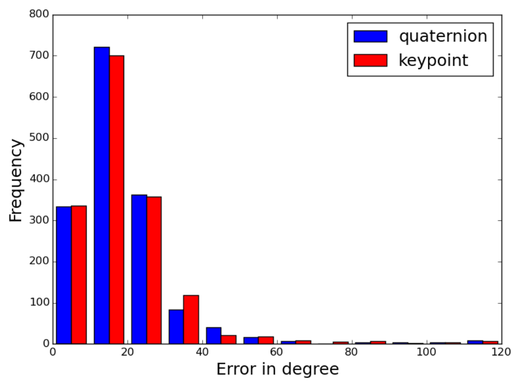


(d) Shift of 12 pixels

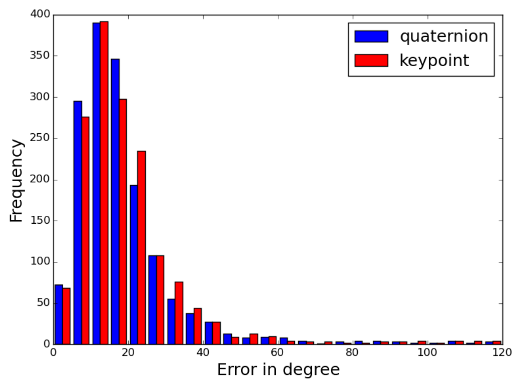
Figure 5.6: Average error for different hand orientations. Each bar represents the mean error of all test examples which lie in the respective 10° interval of similar hand orientations.



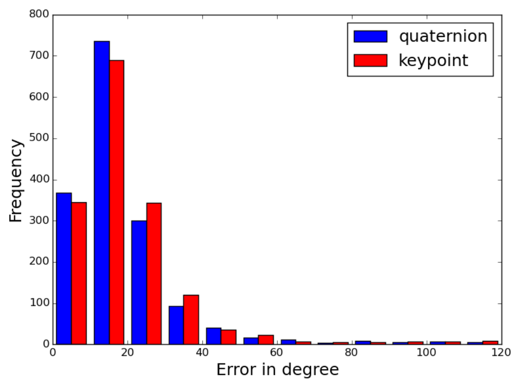
(a) No shift



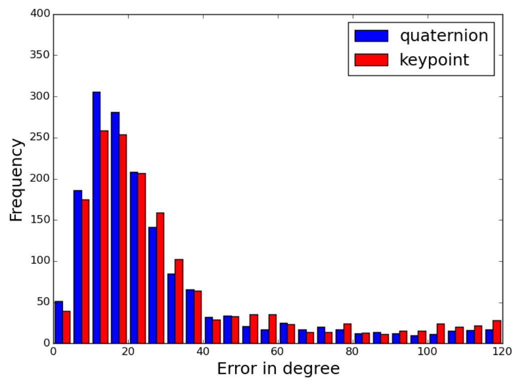
(b) No shift



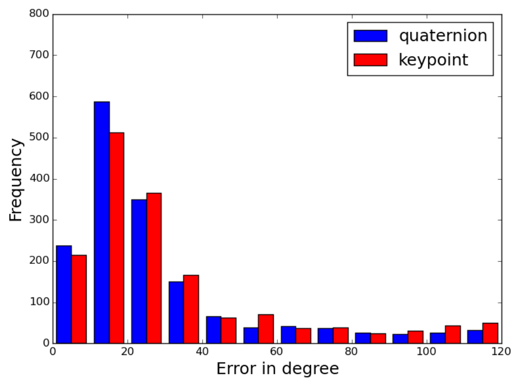
(c) Shift of 4 pixels



(d) Shift of 4 pixels

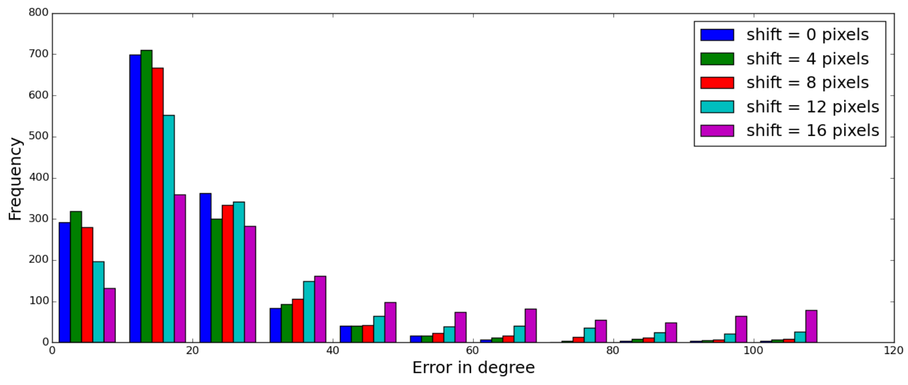


(e) Shift of 12 pixels

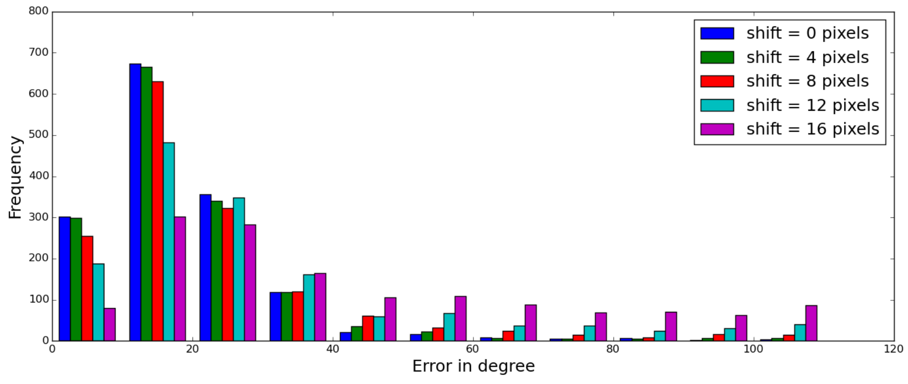


(f) Shift of 12 pixels

Figure 5.7: Error histograms with bin size of 5 in the left column and bin size of 10 in the right column.



(a) Quaternion network



(b) Keypoint network

Figure 5.8: Histogram of errors for various shifts.

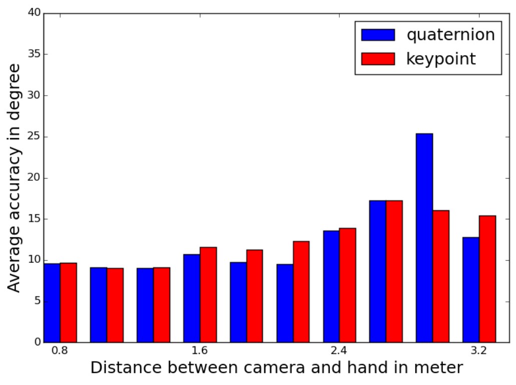
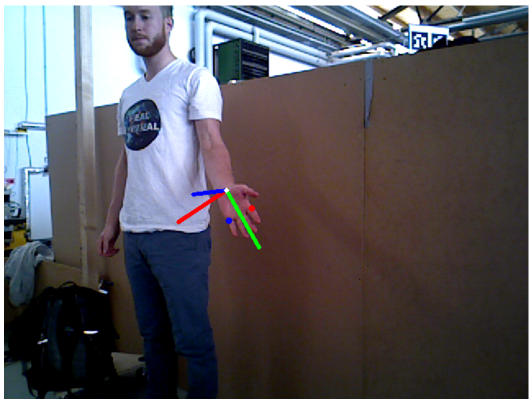
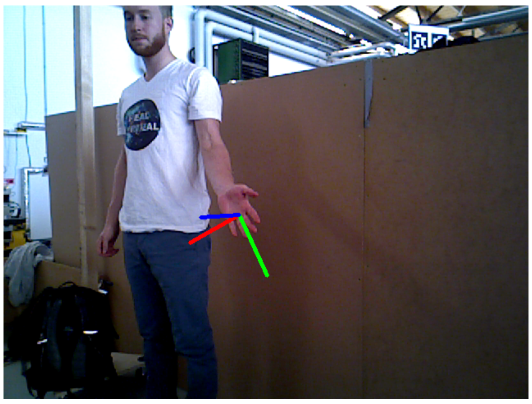
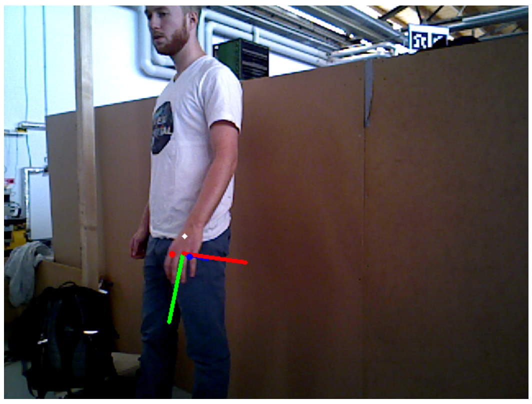
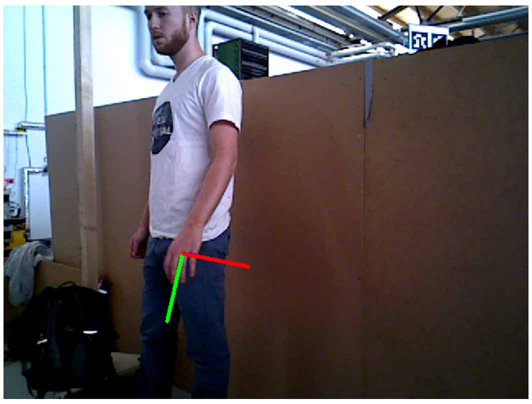


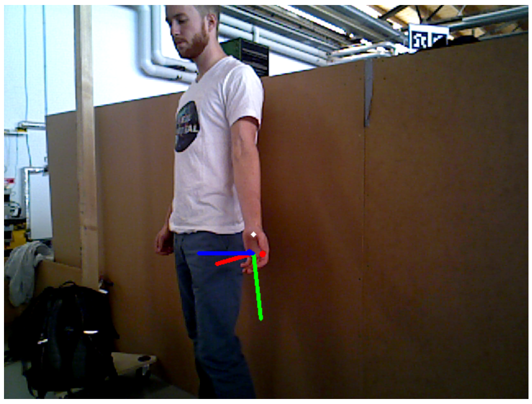
Figure 5.9: Mean error for different distances between camera and hand.


(a) Keypoint


(b) Quaternion


(c) Keypoint


(d) Quaternion


(e) Keypoint

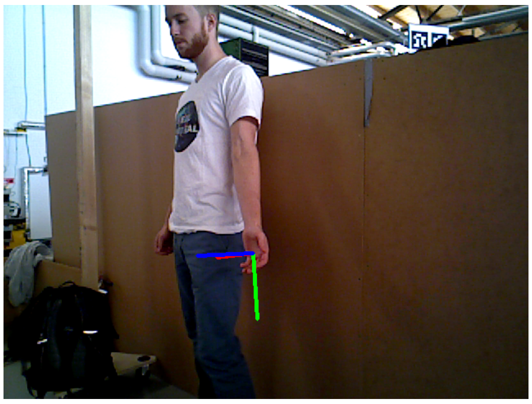
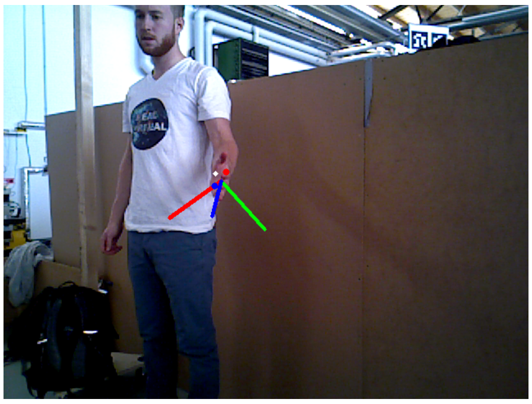
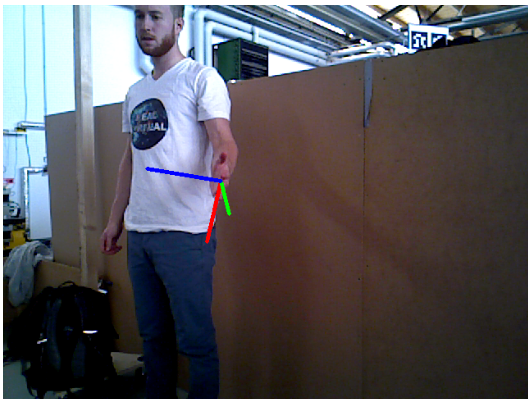
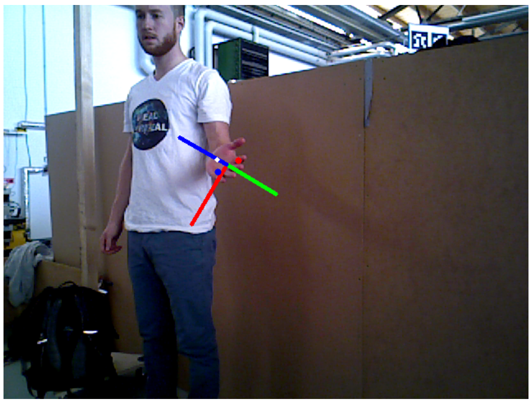
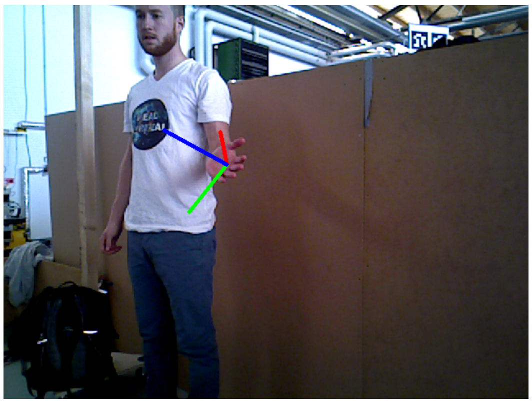

(f) Quaternion

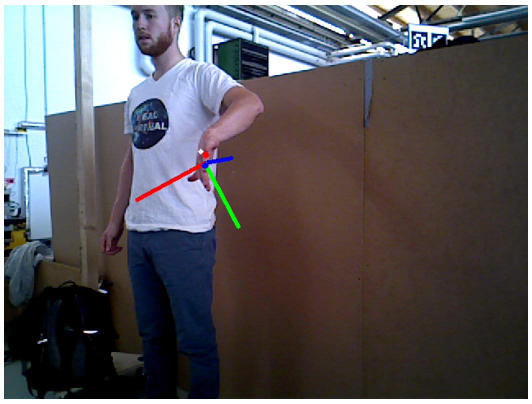
Figure 5.10: Example images with successful orientation estimation for both keypoint method (left column) and quaternion method (right column). We drew the coordinate systems in the RGB-images for better visibility.


(a) Keypoint: Fail


(b) Quaternion: Success


(c) Keypoint: Success


(d) Quaternion: Fail


(e) Keypoint: Fail

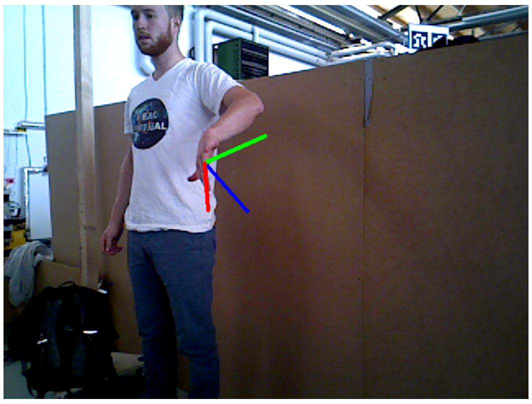

(f) Quaternion: Fail

Figure 5.11: Example images with (partially) unsuccessful orientation estimation. We drew the coordinate systems in the RGB-images for better visibility.

6 Implementation Details

6.1 Camera

We use the *Asus Xtion PRO LIVE* RGB-D camera to record depth images for the dataset and the hand pose estimation. The camera is similar to the widely know *Microsoft Kinect* camera, but it is smaller and does not need an external power supply. It uses a *PrimeSense* Sensor and has a field of view of $58°$ horizontally and $45°$ vertically. The camera supports different resolutions, we use a VGA resolution of 640×480 with 30Hz frame rate for both RGB and depth images. The depth range lies between 0.8 meters and 3.5 meters. It can be connected to the computer via USB [14].

Figure 6.1: Asus Xtion PRO LIVE [3]

6.2 GPU

A deep neural network training requires a lot of computational power. Using GPUs instead of CPUs gives a significant speed boost because linear algebra routines can be computed efficiently in parallel.
For our deep network training we use a *NVIDIA GeForce GTX Titan Black* GPU. It operates at a frequency of 889 MHz and has 6 GB RAM [1]. The *NVIDIA* driver version is 346.72 and we use *NVIDIA CUDA* 7.0 to accelerate the training.

6.3 CPU

In contrast to training, feature extraction should not require special hardware to run in real-time. We use an *Intel Core i5-4200* CPU with 2.5 GHz and achieve high frame rates, even though it is just an ordinary notebook processor. This demonstrates that our method is universally applicable.

6.4 Software

Our approach uses C++ and Python with various libraries.
First of all, the software framework which handles the camera-computer connection is *ROS (Robot Operating System)* [11], an open source library for robot applications. It provides packages like *OpenNI* which we use as camera driver and *OpenNI tracker* [9], the skeleton tracker that gives us the initial guess for the hand position. We also use the *tf (transformations)* package to interpret the output of the skeleton tracker.
We implemented the code for the generation of the dataset in C++ using the *OpenCV* library [8] for image processing. For the vector, matrix or quaternion representation as well as linear algebra calculations we use the *Eigen* library [4].
The two-camera setup was calibrated using *ArUco* markers and the corresponding marker detection library [2] plus the *Point Cloud Library (PCL)* [10] to get the transformation between the cameras.
We wrote a Python script to combine training images and the associated labels and chose *HDF5* [5], a library and file format for storing and managing data, as representation for the training examples.

6.5 Deep Network Training

We use *Caffe* [15], a deep learning framework developed by the *Berkeley Vision and Learning Center (BVLC)*. It is implemented in C++ and open source, so we could adapt it to fit our needs. For the quaternion pose estimation network we implemented a quaternion normalization layer and a quaternion loss layer. *Caffe* runs on both CPUs and GPUs and also provides useful tools for feature extraction or training evaluation.
The network architecture can be defined in *plaintext protocol buffer schema (prototxt)*, a file format developed by *Google*. Learned models are not saved in human-readable text anymore, they use the *binary protocol buffer (binaryproto) .caffemodel* file format. The convolutional network architecture consists of different layers, e.g. convolutional layers, with several parameters to define the behavior. Figure 6.2 shows an example of a convolutional layer. The training parameters are also defined in a *prototxt* file. Table 6.1 lists the parameters we used for training our networks.

Parameter	**Localization**	**Keypoint**	**Quaternion**
Iterations	300.000	300.000	450.000
Learning rate #1	0.01	0.01	0.001
Learning rate #2	0.05	0.05	0.005
Learning rate #3	0.001	0.001	0.0001
Learning rate #4	-	-	0.00005
Momentum	0.9	0.9	0.9
Weight decay	0.0005	0.0005	0.0005
Batch size training	256	256	256
Batch size test	64	128	64
# Training images	143029	78491	78491
# Test images	15778	8719	8719

Table 6.1: This table shows which parameters we used to train the deep networks. The learning rate was decreased every 100.000 iterations. For each network we use the weight configuration at the end of the respective training.

Caffe has the convenient feature that it creates snapshots of the learned weights at certain points. Therefore, a training can be easily paused and resumed, even with different parameters (e.g. a smaller learning rate).

```
layer {
   name: "conv1"
   type: "Convolution"
   bottom: "data"
   top: "conv1"
   param {
         lr_mult: 1
         decay_mult: 1
   }
   param {
         lr_mult: 2
         decay_mult: 0
   }
   convolution_param {
         num_output: 32
         pad: 2
         kernel_size: 6
         stride: 1
         weight_filler {
            type: "gaussian"
            std: 0.1
         }
         bias_filler {
            type: "constant"
            value: 1
         }
   }
}
```

Figure 6.2: An example of the definition of a convolutional layer: The layer takes the `data` blob as input (provided by the previous layer) and produces the `conv1` layer. The output has 32 feature maps that were created by 6×6 convolution kernels and a stride of 1. Zero-padding is applied, so the input is extended by two rows of zeros on each boundary. The weights are initialized randomly with a gaussian distribution and the bias terms are initialized constantly. `lr_mult` is a learning rate adjustment for weights or biases. Usually the bias learning rate is twice as large as the weight learning rate because this can lead to better convergence rates [7]. `decay_mult` is an adjustment for the weight decay. It is zero for the bias terms.

6.6 Feature Extraction

We use two different methods for feature extraction. To evaluate the performance of the trained networks, we use *Caffe's* feature extraction script to create a *LevelDB* file including the predicted labels of every test image. *LevelDB* is a fast key-value storage library [6].
We wrote a Python script which compares the actual label with the predicted label in the *LevelDB* file for each test image and thus measures the precision of the network (see Chapter 5).
As the real-time feature extraction requires a fast runtime, we implemented it in C++. It estimates the orientation of both hands at the same time for single images from a video stream with 30 frame per second on our CPU.

7 Conclusion

In this thesis we presented a method to estimate the hand's orientation from single depth images in real-time, which can be applied as an add-on to an arbitrary human body part tracker/detector. We propose a two-stage architecture where the first stage improves the predicted hand position of the external tracker and precisely locates the center point of the hand and the second stage estimates the hand's orientation. Both stages use deep convolutional neural networks trained on labeled real hand datasets. We developed an efficient method to label difficult hand poses (e.g. due to self-occlusion) using two calibrated depth cameras placed at different positions. Our augmentation pipeline allows to create big datasets in short time. We presented two approaches to estimate the hand's orientation, one based on keypoints and the second one based on quaternions.

We performed various experiments to evaluate the different aspects of our method. In more than 90% of cases, the point the localization stage predicts is less than 8 pixels away from the ground truth center point of the hand. Both orientation estimation methods can deal with uncertainty of the previous stage. Even with an error of 8 pixels, the quaternion approach and the keypoint approach have an error of only $21.4°$ and $23.8°$, respectively. If the error of the localization stage is higher, the performance significantly drops. Overall, the quaternion method achieved better results than the keypoint approach, particularly when we tested how they could cope with uncertainty from the localization stage. Not all hand poses can be estimated equally well. If neither palm nor dorsum of the hand face the camera, the error is considerably higher. Especially the keypoint method underperforms for such of poses. In summary, the quaternion method achieves a lower average estimation error and is more robust.

7.1 Future Work

Our approach is yet unable to deal with object interactions. Since this is an important feature in a lot of human-robot interaction scenarios, it is the major target of future improvements. The dataset could be extended with images of hands holding objects or being partially occluded. Until now our approach can only handle self-occlusions.

Even tough the camera is an RGB-D camera, we do not use color images. Color based hand segmentation could make the orientation estimation more stable, especially for hand poses where palm and dorsum are not visible and the depth sensor produces NaN values. Another approach to improve the performance for difficult poses would be giving up the even distribution of poses in the dataset extending it with more training examples of difficult poses. In addition, the keypoint method could be upgraded to not only estimate three keypoints, but multiple keypoints for every finger in order to estimate the fully articulated hand pose.

Bibliography

[1] NVIDIA GeForce GTX Titan Black. http://www.nvidia.de/object/geforce-gtx-titan-black-de.html#pdpContent=2. Accessed: 2015-08-14.

[2] ArUco. http://www.uco.es/investiga/grupos/ava/node/26. Accessed: 2015-08-15.

[3] Asus Xtion PRO LIVE. http://www.asus.com/de/3D-Sensor/Xtion_PRO_LIVE/. Accessed: 2015-08-24.

[4] Eigen Library. http://eigen.tuxfamily.org/index.php?title=Main_Page. Accessed: 2015-08-15.

[5] HDF5. https://www.hdfgroup.org/HDF5/. Accessed: 2015-08-15.

[6] LevelDB. https://github.com/google/leveldb. Accessed: 2015-08-18.

[7] Caffe Mnist Tutorial. http://caffe.berkeleyvision.org/gathered/examples/mnist.html. Accessed: 2015-08-15.

[8] OpenCV. http://opencv.org/, . Accessed: 2015-08-15.

[9] OpenNI Tracker. http://wiki.ros.org/openni_tracker, . Accessed: 2015-08-15.

[10] Point Cloud Library. http://docs.pointclouds.org/trunk/. Accessed: 2015-08-15.

[11] ROS (Robot Operation System). http://wiki.ros.org/. Accessed: 2015-08-15.

[12] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015. URL `http://www.iro.umontreal.ca/~bengioy/dlbook`.

[13] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[14] Anika Holtermüller and Jörg Plödereder. Tracking of persons with camera-fusion technology. 2012.

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[16] Alina Kuznetsova and Bodo Rosenhahn. Hand pose estimation from a single rgb-d image. In *Advances in Visual Computing*, pages 592–602. Springer, 2013.

[17] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN 0262018020, 9780262018029.

[18] Andrew Ng. Cs229 lecture notes. *CS229 Lecture notes*, 1(1):1–3, 2000.

[19] Andrew Ng. Sparse autoencoder. *CS294A Lecture notes*, 72, 2011.

[20] Markus Oberweger, Paul Wohlhart, and Vincent Lepetit. Hands deep in deep learning for hand pose estimation. *arXiv preprint arXiv:1502.06807*, 2015.

[21] Iason Oikonomidis, Nikolaos Kyriazis, Antonis Argyros, et al. Full dof tracking of a hand interacting with an object by modeling occlusions and physical constraints. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2088–2095. IEEE, 2011.

[22] Javier Romero, Hedvig Kjellström, Carl Henrik Ek, and Danica Kragic. Non-parametric hand pose estimation with object context. *Image and Vision Computing*, 31(8):555 – 564, 2013. ISSN 0262-8856. doi: http://dx.doi.org/10.1016/j.imavis.2013.04.002. URL `http://www.sciencedirect.com/science/article/pii/S0262885613000656`.

[23] B. Sapp and B. Taskar. Modec: Multimodal decomposable models for human pose estimation. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 3674–3681, June 2013. doi: 10.1109/CVPR.2013.471.

[24] Jamie Shotton, Toby Sharp, Alex Kipman, Andrew Fitzgibbon, Mark Finocchio, Andrew Blake, Mat Cook, and Richard Moore. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, 56(1): 116–124, 2013.

[25] Danhang Tang, Tsz-Ho Yu, and Tae-Kyun Kim. Real-time articulated hand pose estimation using semi-supervised transductive regression forests. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 3224–3231. IEEE, 2013.

[26] Jonathan Tompson, Murphy Stein, Yann Lecun, and Ken Perlin. Real-time continuous pose recovery of human hands using convolutional networks. *ACM Transactions on Graphics (TOG)*, 33(5):169, 2014.

[27] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in Neural Information Processing Systems*, pages 1799–1807, 2014.

List of Figures