Chair of Cyber Physical Systems

Master's Thesis

A Framework for Learning Visual and Tactile Correlation

Benjamin Schödinger, BSc

October 2022

A Framework for Learning Visual and Tactile Correlation

Master Thesis by Benjamin Schödinger[1]

[1]*benjamin.schoedinger@stud.unileoben.ac.at, m11770592, Montanuniversität Leoben, Austria*
Leoben, Austria, October 25, 2022

1st supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Elmar Rueckert
2nd supervisor: Vedant Dave, M.Sc.

Chair of Cyber-Physical-Systems
Montanuniversität Leoben, Austria

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 21.10.2022

[illegible]

Unterschrift Verfasser/in
Benjamin Schödinger

ACKNOWLEDGEMENTS

I would like to express my gratitude to my primary supervisor Vedant Dave MSc, who guided me, always gave great advice on how to overcome the problems throughout this project and patiently answered all the questions I had. I would also like to thank Univ.-Prof. Dr. Elmar Rueckert who made this thesis possible and greatly advised me on how to present a thesis properly.

ABSTRACT

Tactile data is an important source of information for applications in fields such as object manipulation or object recognition. However, the process of gathering the tactile data can be inconvenient and time consuming. For example a robotic manipulator would have to grasp the object to be moved every time to gather the tactile information and then again to finally pick it up. This thesis proposes a way to overcome this kind of issue by implementing a method to predict what the tactile feedback sensors would measure when touching an object at a given position, based on two dimensional visual data. Therefore, visual-tactile data pairs were gathered to train a Convolutional Neural Network that takes images of objects with the positions of interest marked as the input and the force vector as the output. To improve performances, the edges of the input images were extracted using the Canny algorithm, a new architecture was developed and the training process optimised with the Bayesian Optimisation algorithm. An evaluation strategy was developed and a test set built, to be able to effectively compare the different models to each other. The result is a framework that is capable of understanding the spacial relationship between tactile sensors and surfaces but lacks in accuracy, as a result of noisy data. The noise is caused by inaccurate sensors and a sub-optimal acquisition strategy.

KURZFASSUNG

Taktile Daten sind eine wichtige Informationsquelle für Anwendungen in Bereichen wie der Objekt Manipulation oder der Objekt Erkennung, aber die Sammlung der Daten kann Zeitintensiv und problematisch sein. Zum Beispiel, ein Manipulator müsste jedes Mal das zu bewegende Objekt greifen um die taktilen Informationen zu sammeln und dann noch einmal um es endgültig aufzuheben. Diese Masterarbeit schlägt ein System vor, das hilft dieses Problem zu umgehen, indem es die Taktilen Daten, die Sensoren messen würden, wenn sie ein Objekt an einer gewissen Stelle berühren, basierend auf zweidimensionalen visuellen Informationen. Dafür wurden visuell-taktile Datenpaare gesammelt, um ein Convolutional Neural Network zu trainieren, das Bilder von Objekten als Input nimmt und die Kraftvektoren ausgibt. Um bessere Ergebnisse zu erzielen, wurden die Kanten in den Bildern mit dem Canny Algorithmus extrahiert, eine neue Architektur entwickelt und der Training Prozess mit dem Bayesian Optimisation Algorithmus optimiert. Es wurde eine Evaluations Strategie entwickelt und ein Test Set eingerichtet, um die verschiedenen Modelle effektiv miteinander vergleichen zu können. Das Resultat ist ein System, das die räumlichen Beziehungen zwischen taktilen Sensoren und Oberflächen versteht, aber an Genauigkeit zu wünschen übriglässt. als Folge von ungenaue Daten. Die schlechte Qualität der Daten ist auf ungenaue Sensoren und die suboptimale Strategie der Datensammlung zurückzuführen.

Contents

List of Figures

List of Tables

1 Introduction

For us humans it comes naturally to use our senses to acquire information about the world around us. When we pick up an object our tactual perception provides us with numerous aspects related to the objects material such as roughness, compliance or coldness (Bergmann Tiest, 2010). In fact, we can effortlessly distinguish numerous categories of materials such as textiles, stones, liquids etc. and even recognise materials within each class solely by looking at them. Additionally we are capable of predicting how an object would feel when touched and what the physical properties like density or thermal conductivity would be (Fleming, 2014). This skill, namely knowing how something feels purely based on vision, can also be very useful in the world of robotics because it can provide tactile information without the necessity to establish physical contact with the object, which can be inconvenient and time consuming. This thesis proposes a method to teach computers the perception of touch. Specifically this project implements a framework which predicts the feedback the tactile sensors would measure when touching an object at a given position with 2D images as the input.

1.1 Motivation

Tactile data is a valuable source of information for multiple purposes such as object manipulation or object recognition. In robot grasping it is used to predict the stability (Dang and Allen, 2012) or assess the quality of a grasp and subsequently adjust the configuration (Chebotar et al., 2016; Hyttinen, Kragic, and Detry, 2017). Furthermore the manipulation process can be monitored to detect slipping events and readjust the gripping force using tactile sensors (Dong et al., 2019). For recognizing objects, tactile data can be used as an additional source of information to improve the accuracy (Liu et al., 2017; Zhang et al., 2021) or as input for a framework trained with visual data (Falco et al., 2017), which can be imagined as looking for something in a box without seeing the inside, based on knowledge of the appearance of the object.

In some applications of tactile information it can be an obstacle and an inconvenience to gather the data. For example consider the case of an manipulator that takes objects off of a conveyor belt. Before the robot picks up the object it needs to decide how hard and where to grasp. If physical contact had to be established every time before something can be manipulated, that could cost a considerable amount of time. Consequently if the robot had the ability to predict what it would "feel" when it touches something, just as we humans are capable of, it could accelerate the process.

1.2 Related Work

The most relevant and inspiring paper for this thesis was published by Zapata-Impata et al. (2021). They proposed a method to predict the response of the tactile sensors by using 3D point clouds. They use the PointNet (Qi et al., 2017) architecture to extract features from the 3D input, combine them with the coordinates of the points of interest and then use fully connected layers to map from the features to the sensor feedback. This work is one of very few which actually aim for generating tactile data as if produced by a sensor. Later they improved their framework by training a Generative Adversarial Network (developed by Goodfellow et al. (2020)) to generate fake training data (Zapata-Impata and Gil, 2020).

It is more common to try to categorize the surfaces of objects by describing its properties. The approach is still similar but the goal is different. For example Takahashi and Tan (2019) implemented a method to estimate the tactile properties such as the roughness or hardness of a surface based on images by using an encoder-decoder network. Therefore they trained the network to predict the tactile sensor feedback and then use the latent variables to estimate the surface properties. Purri and Dana (2020) also implemented a method to estimate the tactile properties using a GAN and a viewpoint selector to find the best subset of images.

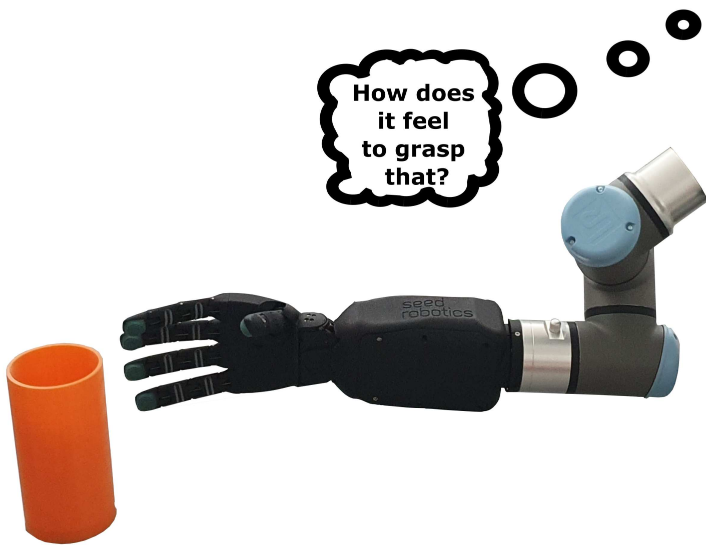
Another approach of mapping visual data to a tactile response was done by Shin et al. (2019). They implemented a method using a combination of CNNs and RNNs to infer contact forces between an object and a special tool based on videos.

In some cases the goal is more specific than simply producing tactile data. Pinto and Gupta (2016) trained a framework that suggests grasp configurations based on visual data. Therefore they implemented a Convolutional Neural Network which takes an image of the object to grasp as the input and gives a 18 dimensional vector where each dimension represents the likelihood of whether the center of the object is graspable at different configurations $(0, 10, ..., 170°)$. They gathered the data by picking up objects multiple times at different positions and used the tactile data to decide whether a grasp was successful or not.

Many works utilize GelSight sensors which provide three dimensional surface data of the touched object based on the deformation of the grasping surface, captured by an optical sensor and are therefore somewhat related to visual data. Therefore these sensors are often used for cross-modal data generation. Lee, Bollegala, and Luo (2019) implemented a method to generate visual images from tactile images, captured with a GelSight sensor, of cloth textures and vice versa using GANs. Li et al. (2019) also used the GelSight sensors to implement a cross-modal prediction model, which finds the touched position in a scene based on the tactile data or predicts the tactile feedback based on visual input.

1.3 Thesis Outlook

In this thesis first the most important background methods are described and explained in Section 2, so a reader without any knowledge in machine learning can follow the work done. Section 2 covers the basics of Feedforward Neural Networks to get a grasp of how neural networks operate, explains Convolutional Neural Networks, the training process, what Data Augmentation is and describes the Bayesian Optimization and Canny Edge Detection algorithms. Next the equipment used to gather the needed data is introduced, followed by a detailed description of the acquisition process, the data structure and possible inaccuracies for both visual and tactile information (Sections 3.1 & 3.2). A separate algorithm was implemented, called the Bounding Box Network, to generate some of the visual data variations and is described in Section 3.3. In Section 4 the strategy and the implemented parameters to evaluate the main algorithm is explained. In Section 5 the choosing process of the best image variation, the proposed architecture and the optimisation of the training process are described. The last Section concludes and discusses the thesis and gives an outlook for possible future work.



2 Fundamentals and Background Methods

This section introduces the basics and some additional information about neural networks, the Bayesian Optimization algorithm and the Canny Edge Detection algorithm.

2.1 Neural Networks

The heart of the framework implemented in this thesis is a Convolutional Neural Network which typically consists of convolutional, pooling and in the end fully connected layers to classify or regress. It uses activation functions and has to be trained using, for example, gradient descent with the gradients computed by back-propagation. This subsection briefly explains all the above and some more, so a reader without prior knowledge in this field is able to understand the work done in this thesis. All the information in this subsection is summarized from Goodfellow, Bengio, and Courville (2016).

2.1.1 Feedforward Neural Networks

The goal of Feedforward Neural Networks is to approximate some function f^*. For example a classifier maps an input $\boldsymbol{x}$ to a category y: $y = f^*(\boldsymbol{x})$. The neural network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the parameters $\boldsymbol{\theta}$ that result in the best function approximation. They are called networks because they are typically composed of many different functions. For example $f(\boldsymbol{x})$ could be formed of three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain, representing the first, second and third layer, resulting in $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. The overall length of the chain gives the depth of the model. The final layer of a neural network is called the output layer.

To reach the goal of matching $f(\boldsymbol{x})$ to $f^*(\boldsymbol{x})$ the network has to be trained. The training data is usually a set of examples $\boldsymbol{x}$ accompanied by labels $y \approx f^*(\boldsymbol{x})$ which specify what the values of the output layers should be. The behaviour of the hidden layers (the layers between input and output) is not defined by the training data. It is the learning algorithms task to decide how to change those layers to produce the desired output and implement the best possible approximation of f^*. Each hidden layer is typically vector-valued and their dimensionality determines the width of the network.

For better understanding lets describe a neural network with one hidden layer containing two units two inputs and one output shown in Figure 1. The hidden layer can be described as $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{\theta}) = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ where $\boldsymbol{x}$ is the input, $\boldsymbol{W}$ the weights and $\boldsymbol{c}$ the biases. The hidden layer is then used as the input for the second layer which computes the output: $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$. Note that the weights here are a vector and the bias a scalar because there is only one output of the last layer in contrast to the hidden layer where the weights are a matrix and the biases a vector. Finally the complete model can be described as $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, \boldsymbol{b}) = f^{(2)}(f^{(1)}(\boldsymbol{x})$.

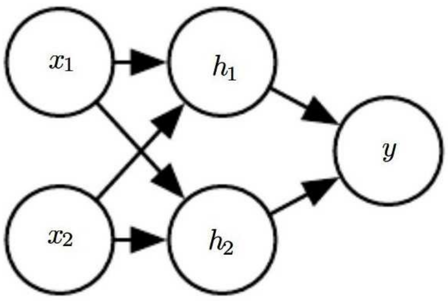


Figure 1: *Illustration of a feedforward neural network with two inputs one hidden layer with two units and one output. Source: Goodfellow, Bengio, and Courville (2016)*

If all layers are linear functions than the whole network would remain a linear function of its input. To overcome this restriction most neural networks use nonlinear activation functions in their hidden layers. In most cases the Rectified Linear Unit or ReLU is recommended and is defined by the activation function $g(z) = max\{0, z\}$.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks are a a specialized kind of neural networks for processing data that has a known, grid-like topology like image data which can be thought of as 2D grid of pixels. As the name suggests these kind of networks employ a mathematical operation called convolution (in German: Faltung), typically denoted with an asterisk($*$). How this operation is applied on data such as images is best explained with an illustration (Figure 2): The kernel moves along the input, which in many cases is an image, and computes the feature map. It can be thought of as a detector with the task of finding small and meaningful features.

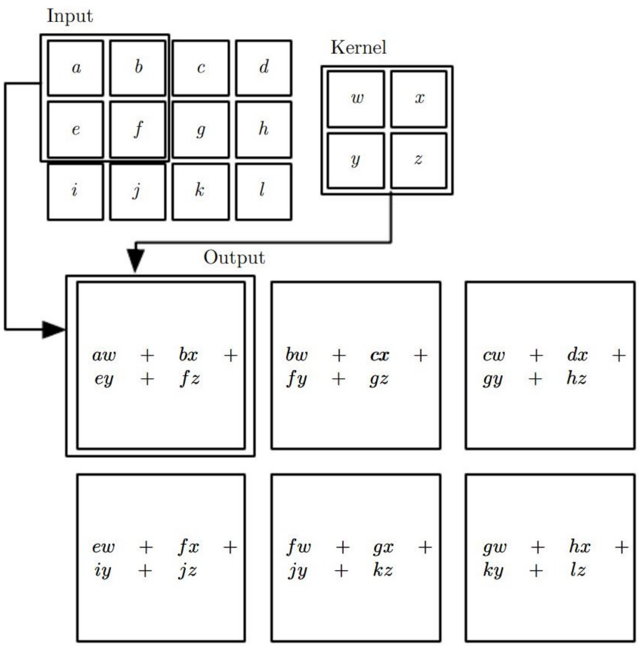


Figure 2: *The convolution operation applied on two dimensional data.*
Source: Goodfellow, Bengio, and Courville (2016)

In traditional neural networks each output interacts with and has separate parameters for every input. This results in $m \times n$ interactions and parameters with m as the number of inputs and n as the number of outputs. Convolutional Neural Networks however have sparse interactions. This is accomplished by making the kernel much smaller than the input which means that fewer parameters need to be stored, the computing requires fewer operations and improves the statistical efficiency. Additionally the kernel typically uses the same parameters at every position (called parameter sharing) of the input, which reduces the storage requirements even further. The total amount of computations with k as the number of kernel parameters is then $k \times n$. Parameter sharing causes the model to have a convenient side effect called equivariance to translation. Explained with the example of an image it means that convolution creates a 2D map of where certain features appear in the input and if this feature moves in the input its representation in the output moves by the same amount.

In a typical convolutional network one layer consists of three stages. In the first stage multiple convolutions are performed in parallel to produce a set of linear activations. This simply means that different kernels are used to produce different activations from the same input. Next the outputs of the convolutional stage are run through nonlinear activation functions. In the third stage a function is applied which replaces the output of the net at a certain location with a summary statistic of the nearby outputs, called pooling function. This stage helps to make the representation approximately invariant to small translations of the input. Invarinace to translation means that if the input is translated by a small amount the values of the most pooled outputs do not change. This is useful if we need to know if a feature is present but not exactly where it is. For example if we try to find a face in an image it is not necessary to know the exact positions of the eyes, it is sufficient to know that there is one on the left side and on the right side of the face.

2.1.3 Training of a Neural Network

Most deep learning algorithms involve optimisation of some sort, which refers to the task of maximizing or minimizing some function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$. The function to be minimized has different names under different circumstances but will be referred to as cost function or loss function in this thesis. This cost function quantifies the accuracy of the models output by comparing it with the labels of the training or validation set. There are many cost functions for different kinds of outputs but in this thesis the neural network is used for regression, so the Mean Squared Error (MSE) loss is used. The MSE is shown in Equation 1, where $\hat{\boldsymbol{y}}$ are the predictions of the model and $\boldsymbol{y}$ are the labels.

$$MSE = \frac{1}{m}\sum_i (\hat{\boldsymbol{y}} - \boldsymbol{y})_i^2 \tag{1}$$

Suppose we have a function $y = f(x)$ where both y and x are real numbers. The derivative of this function is $f'(x)$ and gives the slope of $f(x)$ at the point x. The slope yields information of how x has to change to make an improvement in y. If we want to minimize $f(x)$ we can do so by moving x with the opposite sign of the derivative and hopefully reach a local or even a global minimum of the function. This method is called gradient descent and is visualized in Figure 3.

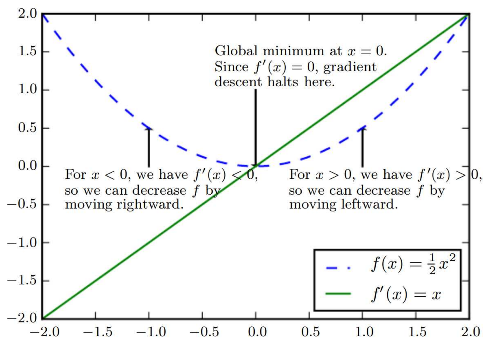


Figure 3: *Visualization of the gradient descent algorithm. Source: Goodfellow, Bengio, and Courville (2016)*

In the case the function has multiple inputs a new point closer to a minimum can be computed with Equation 2 where $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ are the partial derivatives of the function and ϵ the learning rate. Minus the second term because we want to minimize the function and therefore move with the opposite sign of the derivative. The learning rate is a positive scalar determining the size of the step and can have a big influence on the performance of the model.

$$\boldsymbol{x'} = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{2}$$

For optimizing neural networks usually algorithms like stochastic gradient descent are used. This algorithm splits the training set into equally sized mini batches $\mathbb{B} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}\}$ and applies gradient descent after each mini batch. The estimated gradient can be computed as in Equation 3 where $\boldsymbol{\theta}$ are the parameters of the model and $L(\boldsymbol{x}^i, y^i, \boldsymbol{\theta})$ is per example cost function.

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^i, y^i, \boldsymbol{\theta}) \tag{3}$$

The stochastic gradient descent follows the gradient downhill and updates the parameters accordingly as shown in Equation 4.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g} \tag{4}$$

In many cases more advanced learning algorithms, which have adaptive learning rates, are used for optimizing neural networks. The "Adam" algorithm is one of those and is the preferred optimizer in this thesis.

Before the model can be trained with gradient descent, the gradient has to be computed. This is typically done by a method called back-propagation, which got its name by the direction the information flows through the model. When using the model to compute an output the information starts at the input and then flows forward. During training the result of the forward propagation is usually a cost which is then used by the back-propagation algorithm to flow backwards through the model to compute the gradient by applying the chain rule with a specific order of operations, which is highly efficient. Figure 4 briefly describes the chain rule.

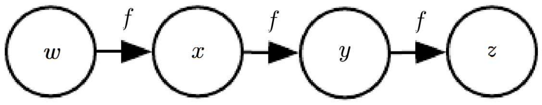


Figure 4: *Let $w \in \mathbb{R}$ be the input to the chain. The same function $f : \mathbb{R} \rightarrow \mathbb{R}$ is used at every step of the chain: $x = f(w)$, $y = f(x)$, $z = f(y)$. To compute $\frac{\partial z}{\partial w}$ the chain rule is needed:*

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} = f'(y)f'(x)f'(w)$$

Source: Goodfellow, Bengio, and Courville (2016)

2.1.4 Dataset Augmentation

The most effective way to achieve better generalization is to train the model with more data but in practice data is limited. One possibility to overcome this problem is by generating fake data and adding it to the training set. This can be as simple as rotating or translating the image and can greatly improve generalization. However, one must be careful not to apply a transformation that would change the correct outcome of the input. For example if the task is to recognize characters in images and the input would be mirrored then a "d" would be changed to a "b" and consequently falsify the training data.

2.2 Bayesian Optimization for Hyperparameter Optimization

Most machine learning algorithms come with many hyperparameters that influence the behaviour of the algorithm in many aspects. Some of them affect the time and memory cost of running the algorithm, others have an important role in the generalization performance of the model recovered by the training process. There are two main approaches in choosing the hyperparameters: manually or automatically. The manual approach requires a good understanding of what the hyperparameters do and how they influence the algorithm. The automated methods reduce the need to understand, but comes with an increased computational cost. Some well known approaches are Grid- or Random Search.(Goodfellow, Bengio, and Courville, 2016)

The Bayesian Optimization algorithm is a tool which has become very popular for automating the search for the optimal hyperparameters in neural networks. It is designed for black-box optimisation, which means that there is no information on the behaviour of the function other than the input and the output. The appealing difference to exhaustive approaches like Grid Search is that the Bayesian Optimization actively tries to find the most promising values for the parameters and therefore is more efficient and in many cases even outperforms other state of the art global optimization algorithms. (Snoek, Larochelle, and Adams, 2012)

The Bayesian Optimization consists of two main components: a Bayesian statistical model for modeling the objective function and an acquisition function for deciding where to sample next. The statistical model is generally a Gaussian Process and provides a posterior Bayesian probability distribution that describes potential values for the objective function $f(x)$ at a candidate point x. Every time an observation of $f(x)$ at a new point was made, the posterior is updated. The acquisition function measures the value that would be generated by evaluation of the objective function at a new point x, based on the current posterior distribution over f. An illustration of the here described method is shown in Figure 5. (Frazier, 2018)

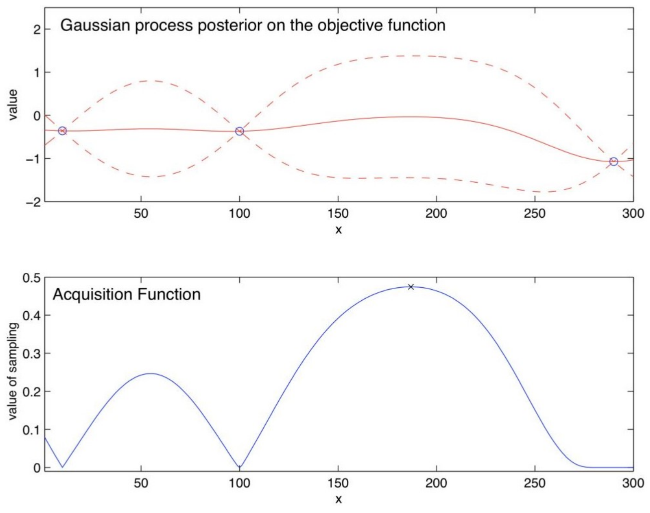


Figure 5: *Illustration of the Bayesian Optimization maximizing an objective function. The top graph shows the estimate (solid red line) and the uncertainty (dashed red line) obtained using the Gaussian Process regression based on three data points (blue dots). The bottom graph shows the acquisition function which is used to choose the next sample at the point that maximizes the acquisition function. Source: Frazier (2018)*

For an in depth explanation of Gaussian Processes, refer to Rasmussen and Williams (2005). Frazier (2018) also describes commonly used acquisition functions. The Bayesian Optimization algorithm implemented in this thesis was developed by Nogueira (2014–).

2.3 Edge Detection Algorithm

In this thesis the edge detection algorithm proposed by Canny (1986) and implemented in the open source library "OpenCV" (Bradski, 2000) is used. The summary in this subsection is from Bradski (2000) if not stated otherwise with some complementary information from additional sources.

The Canny-Algorithm is a multi-stage algorithm. Due to edge detection being susceptible to noise in images the first step is to remove the noise with a Gaussian filter. This step, also called smoothing, computes an average of the surrounding pixels with the pixel $[x, y]$ weighted according to Equation 5 where $d = \sqrt{(x - x_c)^2 + (y - y_c)^2}$ is the distance to the center pixel $[x_c, y_c]$ and σ is the standard deviation of the noise. (Shapiro, Stockman, et al., 2001)

$$g(x, y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{d^2}{2\sigma^2}} \tag{5}$$

The first derivatives of the smoothened image in the vertical (G_x) and horizontal (G_y) direction are then computed with a Sobel kernel. The resulting edge gradient G and its direction θ are then determined using the Equations 6 and 7.

$$G = \sqrt{G_x^2 + G_y^2} \tag{6}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{7}$$

The gradient direction is then used to remove any unwanted pixels which may not be part of an edge. This is achieved by checking every pixel if its response is higher than the two neighbouring ones on either side of it along the gradient direction and suppressing the ones which are not. This method is called non-maximum suppression and is visualized in Figure 6. (Shapiro, Stockman, et al., 2001)

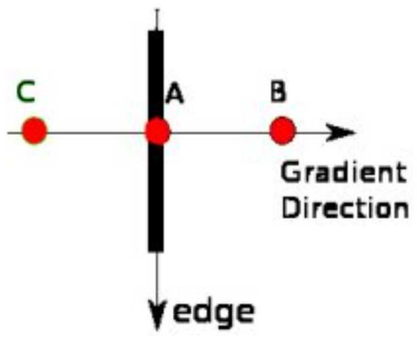


Figure 6: *Point A is on the edge and the points B and C are in gradient directions, so A is checked if it forms a local maximum with B and C. Source: Bradski (2000)*

In the last stage of the algorithm called hysteresis thresholding it is decided which edges should be kept or discarded. This is done by choosing two threshold values. Lets call them $maxVal$ and $minVal$. If the magnitude of the gradient is greater than $maxVal$ the pixel is always considered to be part of an edge. If the magnitude is smaller than $minVal$ the pixel is always discarded. Now, if the contour starts above but falls under $maxVal$ along the way it is still considered an edge as long as the gradient doesn't fall under $minVal$. See Figure 7 for a visualized explanation.

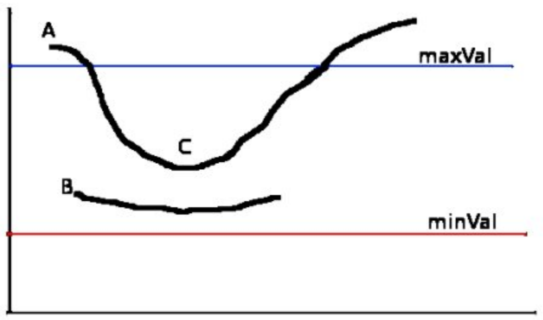


Figure 7: *The edge consisting of A and C is considered an edge even though C falls under $maxVal$ because it is one continuous contour. B is under $maxVal$, so discarded. Source: Bradski (2000)*

3 Experimental Setup & Data Processing

This section describes the equipment used for gathering the data, explains the acquisition process, how the raw visual and tactile data were prepared for the use in the framework and discusses possible inaccuracies.

3.1 Experimental Setup

The needed equipment for this thesis are the data gathering setup and a computer preferably with a GPU to train the neural network. The setup to collect the training data consists of four main components shown in Figure 8:

1. smartphone camera
2. RH8D manipulator by seed robotics
3. FTS-3 tactile pressure sensor by seed robotics
4. UR3e robot arm by UNIVERSAL ROBOTS

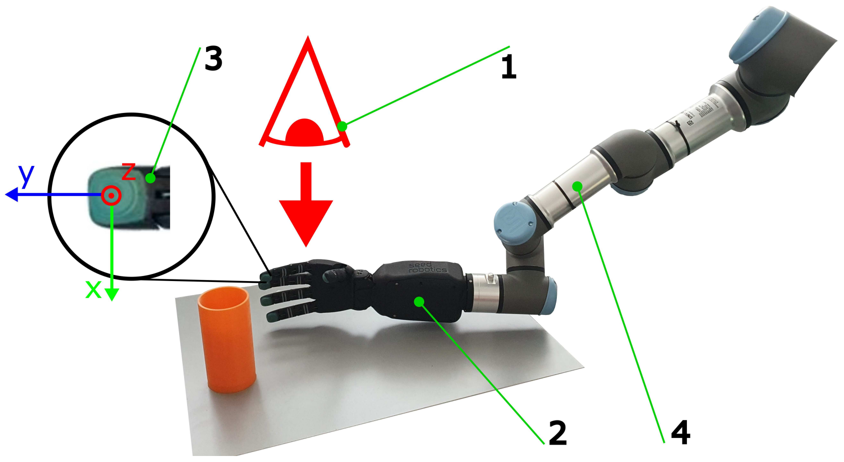


Figure 8: *This setup is used to gather the training and evaluation data. The red eye symbolizes the camera and looks down on the fingers. The coordinate system at position 3 shows the positive direction of the measured forces where the z direction points out of the plane.*

The purpose of the arm is to hold the hand and bring it in the right position. The hand carries the tactile sensors which are located on the fingertips of the robotic hand. In the course of this thesis only the index finger and the thumb are used. In Figure 8 in position 3 a tactile sensor and its coordinate system is shown. This coordinate system is true for the index finger but for the thumb the positive y points in the opposite direction. In Table 1 some specifications of the tactile sensors are listed.

Table 1: *Sensor Specifications*

Standard Range	$0 - 10N$
Standard Range Resolution	$1mN$
Non linearity	2.5%
Hysteresis	2%
Sampling Frequency	$50Hz$

The computer used for training the models has the specifications listed in Table 2. The neural networks are built and trained using the PyTorch library (Paszke et al., 2019) in Python.

Table 2: *Computer Specifications*

GPU	NVIDIA GeForce GTX 1060 6GB
CPU	Intel Core i5-8600K
RAM	16GB of DDR4
OS	WINDOWS

3.2 The Data

The success of a framework based on a neural network is highly dependent on the used training data. To be able to evaluate the quality of the gathered data it is necessary to understand the acquisition process and subsequently how accurate the information actually is. This chapter describes the details of the acquisition, the possible inaccuracies of the process and the data itself for both visual and tactile information.

The Table 3 shows how much data was gathered and generated. The number of grasp configurations is the total number how many different positions were touched. For example if five objects were each touched in five positions, the number of grasp configurations would be 25. The number of grasps describes the total amount of tactile data gathered. To capture the noise of the measurement each grasp configuration is touched multiple times, so if each of the 25 grasp configurations was touched 5 times the total number of grasps would be 125. Each grasp configuration has a corresponding image. To augment the data set these images were each rotated 20 times which results in $162 * 20 = 3240$ visual-tactile data pairs in the case of the training data.

Table 3: *The Amount of Data*

	Grasp Configs.	Grasps	Visual Data after Augmentation
Training Data	162	950	3240
Test Data	34	218	680

3.2.1 Visual Data

The visual information is the input for the framework which then predicts the tactile feedback. In the course of this thesis multiple approaches were tried with the idea of focusing more on the important parts of the image. This means to try to remove as much information from the images that isn't needed or highlight the parts essential for the prediction. These approaches are illustrated and described in Figure 9.

Acquisition Process

The base photographs are taken with a smartphone mounted on a tripod photographing from a top view. This means the robot hand approaches the object orthogonal to the view direction.

• **Figure 9 a)**: The base photographs are first cropped to be square so future processing steps are easier and cause less complications and resized to have the dimension 227×227 because the AlexNet architecture, which was used initially, is designed for it. Also when building Convolutional Neural Networks the initial image size dictates how many layers are needed to reduce the feature space to a small enough size so the first fully connected layer isn't too wide. The wider a fully connected layer is the more GPU memory is needed and consequently can quickly exceed the capabilities of the used hardware. For example if the output of the last convolutional or pooling layer has the dimension $[256, 24, 24]$ the needed input features for the fully connected layer would be $256 * 24 * 24 = 147456$. Lets have 20000 outputs. Each output has its own weight for each input which results in $147456 * 20000 \approx 3 * 10^9$ weights which stored as float32 (4 bytes) take up approximately 12GB of space only for the weights. Starting

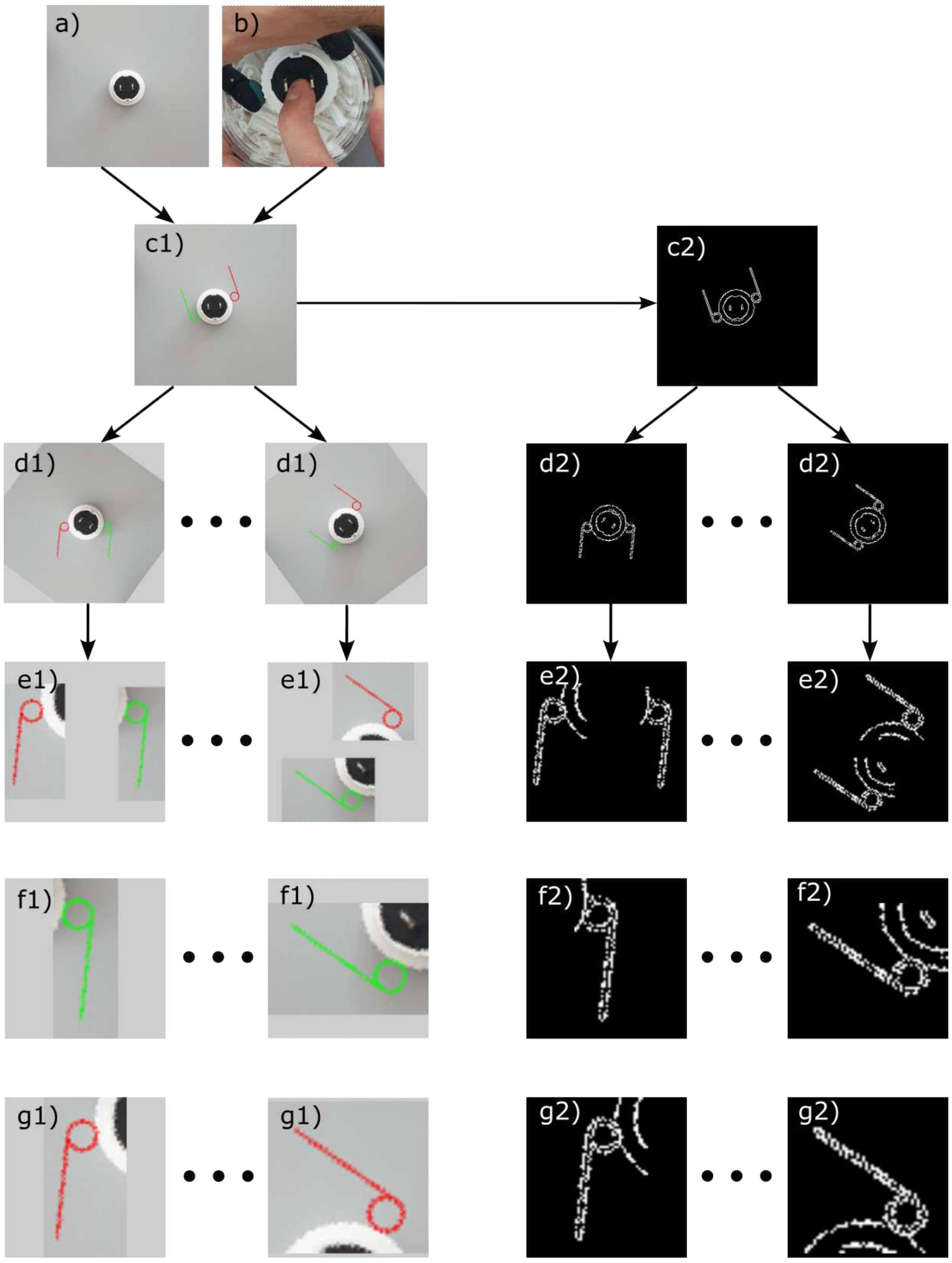


Figure 9: ***a) Base photograph****: The original photograph cropped to be square and resized to* 227 × 227.
b) Data gathering photograph*: A screenshot of the tactile gathering process used to edit the fingers into the base photograph.*
c) Base image*: The position of the fingers edited into the base photograph.*
d) Rotated base image*: The base image was rotated multiple times for data augmentation purposes.*
e) Cropped image*: Focus on both fingers.*
f) Index image*: Focus only on the index finger.*
g) Thumb image*: Focus only on the thumb.*
The images with the index 2 *represent the edges of the images with the index* 1.

with the image dimension 227×227 allows the reduction of the feature space to a small enough size without the necessity to make the network too deep. Additionally the needed storage space for the images themselves is much smaller.

• **Figure 9 c1) and d1)**: Next the positions of the fingers (green: index finger, red: thumb) were manually edited into the base photograph and rotated multiple times as way of data augmentation. This image variant is the first to be used as an input for the framework and serves as the base for the following variants.

• **Figure 9 e1)**: The idea for the cropped image was to remove as much unnecessary information as possible but still keep the relative positions of the fingers. Therefore the Bounding-Box Network (Section 3.3) was used to find the fingers in the images and then crop everything beyond them.

• **Figure 9 f1) and g1)**: Assuming that to be able to predict the tactile feedback of the index finger, the position of the thumb is not needed to be known and vice versa, the decision was made to use images focusing on one finger as an input. Again the Bounding-Box Network was used to find the fingers and then crop everything outside the bounding box.

• **Figure 9 c2) - g2)**: Because the main idea of this thesis is to predict the tactile feedback based on the geometry of the object, it seemed to make sense to only consider information about the edges of the object. For this purpose the Canny-Algorithm was used to extract the edges of the images and use these as the input for the network. To further improve this method the optimal threshold values for every image was determined. This not only improves the performance of the model due to better training data but also reduces the amount of unusable data due to bad edge extraction.

All images were resized to 227×227 after cropping. The last step is to filter out the images where either the bounding box is in the wrong position or the result of the edge extraction is not satisfactory.

Data Structure

To be able to effectively build a Convolutional Neural Network with PyTorch it is crucial to understand how it handles image data. When loading an image into the program and transforming it to a tensor the dimensions are usually $[channels, height, width]$. The channels, in the case of an RGB image, are the intensities of the color red, green and blue in the range of $[0.0, 1.0]$. The image height and the width are the number of the vertical and horizontal pixels respectively. For example the tensor for a 3x3 yellow square in PyTorch would look like this:

$$\begin{bmatrix}\begin{bmatrix}[1.0 & 1.0 & 1.0]\\ [1.0 & 1.0 & 1.0]\\ [1.0 & 1.0 & 1.0]\end{bmatrix} & \begin{bmatrix}[1.0 & 1.0 & 1.0]\\ [1.0 & 1.0 & 1.0]\\ [1.0 & 1.0 & 1.0]\end{bmatrix} & \begin{bmatrix}[0.0 & 0.0 & 0.0]\\ [0.0 & 0.0 & 0.0]\\ [0.0 & 0.0 & 0.0]\end{bmatrix}\end{bmatrix}$$

In the case of the edge images c2) - g2) in Figure 9) there are no other colors than black and white, so when transformed to a tensor the data has only one channel which holds information about the gray scale. This has to be considered when building the neural network because the number of input channels have to be defined correctly.

Possible Inaccuracies

Due to the fact that the fingers were edited manually into the base photograph comparing to the data gathering photograph there will be human error in the visual data. Also, the shape of the edited fingers differs to robots finger shape.

In the case of the edge images the lighting can cause shadows which look similar to the edges of light colored objects and therefore difficult for the Canny algorithm to distinguish. This can lead to edges where there are none.

3.2.2 Tactile Data

The tactile data is the feedback when the object is touched with the robot hand and what the network should predict. The goal for this thesis is to measure different tactile feedbacks for varying shapes of surfaces touched. In particular the ratio of the forces in y- and z-direction are of interest because they are the forces influenced by the shapes visible in the input images. In contrary the forces in the x-direction are induced by the object shape orthogonal to the image plane which isn't visible in the images therefore it is not expedient to try to predict these forces with this setup.

The measured forces were reduced to the unit vector, because the absolute forces induced by the touch are highly dependent on how hard fingers were pressed against the object but, the interesting information lies in the direction of the force vector. In theory this means the tactile data is not influenced by the strength of the grasp but by the shape of the touched surface.

Each grasp, meaning same object and position, was repeated several times to capture the noise of the measurements. These multiple measurements together make a set which can be visualized as a cone as shown in Figure 10. These cones are later used as the targets for evaluation purposes for the model rather than the individual measurements of the set.

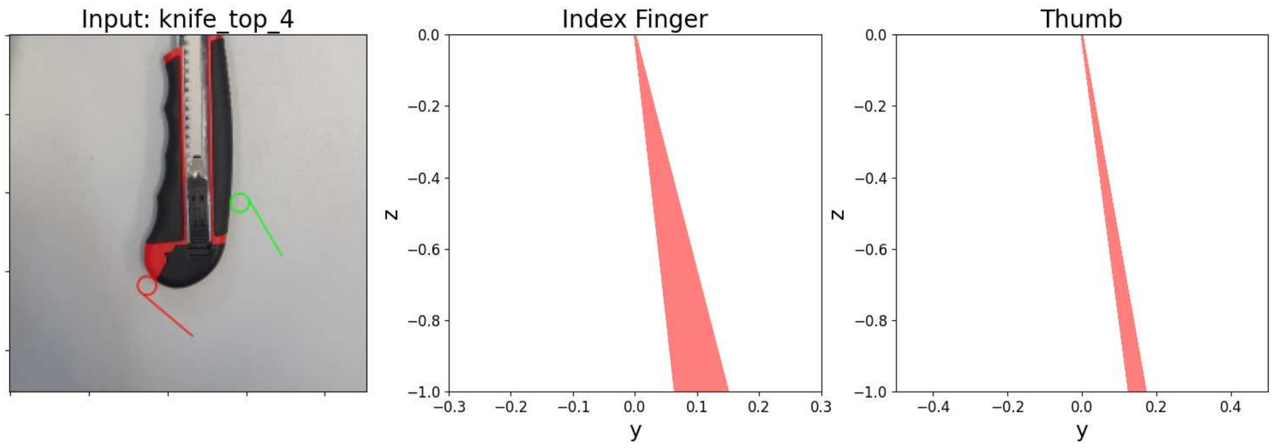


Figure 10: *The middle and right graph show an example of the cone in which the multiple measurement for the index finger and thumb lie for the grasp configuration shown in the image on the left.*

Acquisition Process

The data was gathered by manually pressing the index finger and thumb against the desired positions on the object as shown in Figure 11. This strategy was chosen because it is faster and easier to touch the object at the wanted position than to use the motors in the robot hand to grasp the item, because the fingers tend to not stay straight.

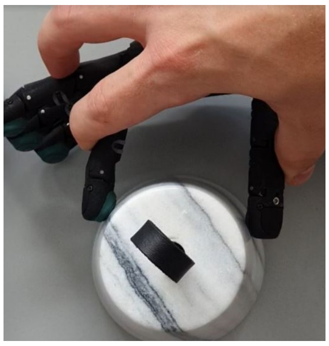


Figure 11: *This illustration shows how the tactile data is gathered.*

As mentioned each grasp was repeated several times. Because the robotic system continuously saves the tactile data, all grasps are together in one time series, so the raw data was split up into the individual grasps. This was done by saving the data in a new array every time the force in z direction of the index finger would fall under a certain threshold. At this point each grasp is still a small time series. Next, one time point for each small time series was chosen to represent the grasps. The point chosen is the first peak bigger than the mean value of the individual grasp time series of the index finger in z direction. This method seems to generally find a good point that is closer to the value that the forces plateau at than the mean is. This procedure is visualized in Figure 12.

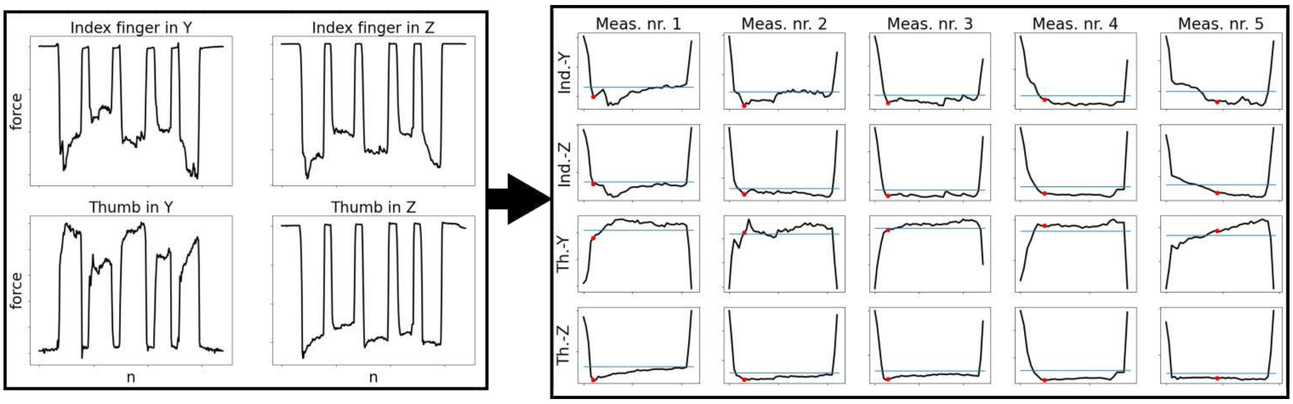


Figure 12: *On the left side the raw time series of the forces in y- and z-direction for the thumb and the index finger are shown. The forces go back to roughly zero every time the grasp is released. There the data is split into the individual grasps. The graphs on the right side show each peak with the mean as the blue line and the representative point for the grasp as the red point.*

At this point the graphs in Figure 12 are used to discard the grasps which are, for whatever reason, not usable. The good grasps are then reduced to the unit vector, so they don't yield information about the force applied but only about direction, using Equation 8 where $\boldsymbol{r}$ is the direction vector of the force $\boldsymbol{F}$.

$$\boldsymbol{r} = \begin{bmatrix} r_y \\ r_z \end{bmatrix} = \frac{\begin{bmatrix} F_y \\ F_z \end{bmatrix}}{\sqrt{F_y^2 + F_z^2}} = \frac{\boldsymbol{F}}{|\boldsymbol{F}|} \tag{8}$$

Data Stucture

The raw data is saved into csv files. From there the required information is extracted processed and then stored in xlsx files. This data format was chosen because it is easy to manipulate manually and automatically and easier to read than for example csv files. Each grasp is saved in a new line where the columns are: image name, r_y (index finger), r_z (index finger), r_y (thumb), r_z (thumb). Consequently all the data is in one file.

Possible Inaccuracies & Data Discussion

It is certain that gathering the tactile data by manually pressing the robot fingers against the surfaces induces forces into the system which falsify the results to some degree. Additionally during the process the robot hand itself can deform and cause deviations.

The idea of this thesis was to measure the reaction forces when touching an object with the robot fingers. The reaction force is always orthogonal to the surface touched. This means in the case pictured in Figure 13 the y part of the force induced by the surface would be in the negative y direction of the index finger. In reality the measured force has a positive y part which means that there has to be a different explanation for the occurring force. The best guess is that during the grasp the finger moves a little bit, pulls the fingertip and consequently induces a positive force in the y direction. Luckily the tactile feedback varies with different surface shapes and therefore is usable for this thesis.

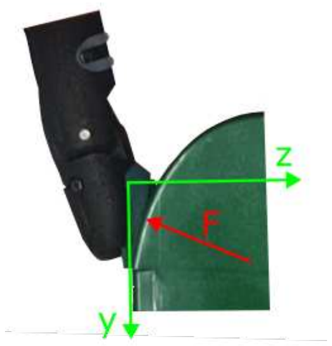


Figure 13: *In this illustration the index finger is shown touching an object in a way where a negative force in the y direction, based on the reaction force of the surface is expected. The two green arrows represent the coordinate system of the finger and the red arrow the expected reaction force orthogonal to the surface.*

3.3 The Bounding Box Network

For some of the visual data types the bounding boxes of the edited index finger and thumb need to be found. For this task a Convolutional Neural Network was implemented which takes the images as in Figure 9 c1) and d1) and gives the positions of the bounding boxes.

The used architecture for the neural network is from Krizhevsky, Sutskever, and Hinton (2012), which was initially built for classifying images. Because the network is used for predicting bounding box positions, which is a regression problem, the output layer had to be changed to 8 linear units to fit the task. This architecture was not built with this kind of task in mind therefore isn't ideal, but because the results were sufficient for the use in this thesis and this isn't the main problem to solve the time wasn't invested to optimise the model.

The training data was manually made by positioning the index finger and thumb at random positions in the base image and writing down the coordinates and sizes of the bounding boxes. This was repeated 20 times for each image. Next the images with the fingers were rotated 3 times by 90 and the bounding box parameters adapted accordingly for data augmentation purposes. The images were only rotated by $90°$ because otherwise the edges of the bounding boxes wouldn't be strictly horizontal or vertical anymore and that would complicate the transformation process. The bounding box parameters are the coordinates of the top left corner x and y, the width w and the height h and are measured in pixels. The transformation of the parameters, where the subscript old indicates the parameters before and the subscript new after the rotation, is described as follows:

$$x_{new} = y_{old} \qquad y_{new} = 227 - (x_{old} + w_{old})$$
$$w_{new} = h_{old} \qquad h_{new} = w_{old}$$

This process is visualized in Figure 14. Note that usually both fingers are in the images. The transformation for the thumb is analog to the index finger. In fact, because the network is trained with both fingers always in the images, the network will only work when both are present.

In total 39 different base photographs (Figure 9 a)) were used and then the fingers edited into them at random positions 20 times. These base images were then each rotated 3 times by 90 which results in $39 \times 20 \times 4 = 3120$ images used for training the neural network. Four images had to be removed, so actually 3116 were available. The training parameters for the Bounding-Box Network are shown in Table 4. The epochs aren't stated exactly because the training process was stopped when the loss of the last batch of an epoch fell under 1 which was usually at around 150 epochs.

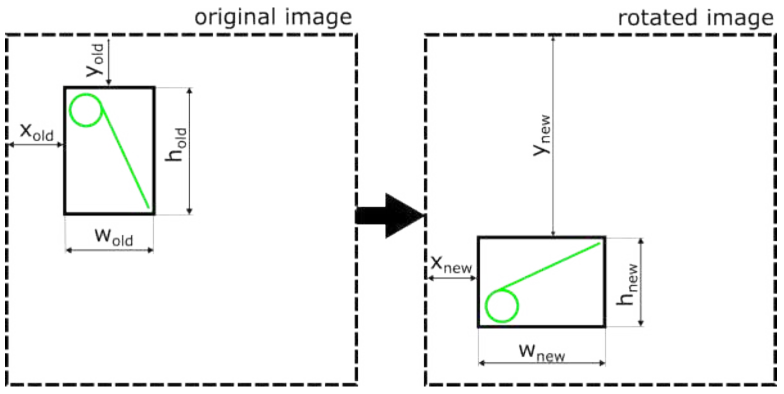


Figure 14: *The data for the bounding box network is augmented by rotating the images* 3 *times by* 90 *counter-clockwise. This image shows how the parameters for the index finger change by the transformation. Note that usually the index finger and the thumb are in the image.*

Table 4: *Bounding Box Network Training Parameters*

Network Architecture	AlexNet (Krizhevsky, Sutskever, and Hinton, 2012)
Loss Function	Mean Square Error
Optimizer	Adam
Learning Rate	10^{-4}
Weight Decay	10^{-5}
Batch Size	5
Epochs	~ 150

For the evaluation of the model the data set for training the tactile prediction network was used, which at the time of the evaluation, consisted of 197 base images which were then each rotated 10 times, resulting in 1970 images. Because some of the base photographs were also used for training the bounding box model, the evaluation is split into known and novel photographs. The accuracy of the predictions are measured by counting the unusable bounding boxes and calculating the percentage of the usable model outputs. The model gives the bounding boxes for the index finger and the thumb and in most cases the prediction fails for one but not both. Therefore accuracies for both fingers, index finger and thumb are computed and presented in Table 5.

Table 5: *Bounding Box Model Performance*

	Novel	**Known**
Both Fingers	95.8%	91%
Index Finger	99.2%	98.2%
Thumb	96.6%	92.8%

To better understand the results from Table 5 some additional information about the behaviour of the bounding box model should be considered. The performance of the model is very dependent on the base photograph for the input because it is sensitive towards shapes and colors similar to the fingers. For example if there are red thin lines in the image, the network is likely to misclassify the thumb and therefore give wrong results. This means that the model doesn't necessarily have to be worse at finding the thumbs but there could simply be more often red lines than green lines in the images, which lead to wrong results. The same logic can be applied to the performance differences for novel and known photographs. Nonetheless the overall results are satisfactory for the use in this thesis.

4 Evaluation Strategy

The framework is evaluated with a test set consisting exclusively of base images not used in the training set. The test set is a mix of mostly novel and some known objects but in all cases the specific grasp location was not seen before. In the cases where the tactile data for both the index finger and the thumb are predicted, usually only the output for the index finger is evaluated. This is done to reduce the amount of evaluation information, so it is easier to interpret the results. The index finger is a better choice here because the amount of tactile measurements with positive y parts is close to the amount of negative y parts, compared to the thumb where most of them are positive. However, if the results for the thumb are interesting they will be mentioned.

To be able to effectively evaluate the performance of the model, several parameters are used:

• **% of predictions in the correct y-direction**: The output of the model for each finger is a vector with two entries and represents the y and z part of the force reduced to the unit length. The z-value of the force direction is in all cases negative because when touching an object the surface always pushes into the fingertip. In contrary the y-value is highly dependent on the shape of the surface or on the direction the finger approaches the surface from (at least for the index finger) and lies in the most cases in the range $[-0.25, 0.25]$. Consequently a parameter was implemented which counts the predictions that are in the correct y-direction because it is a first indicator that the model "understands" the relationship between surface shape and finger orientation. Note that in some cases the label data has both positive and negative points and therefore the prediction is regarded as correct for positive and negative predictions.

• **% of predictions in the cone**: The cone represents the label data and is an indicator for how accurate it is. Since it is counter intuitive to expect the model to be more precise than the data itself, the decision was made to consider the predictions which lie in the label cone as the best possible outcome. This parameter gives the percentage of the predictions that lie in the cone.

• **mean/median of the errors for predictions not in the cone and overall**: As the name suggests these parameters give the mean and median of the errors as described in Section 4.1 for the test set. To get more information these parameters are computed for all predictions and for predictions only outside the cone. Initially only the mean was used but due to some prediction errors being extremely high the results could be misleading. Lets consider a random example model where the mean error is 555.2% and the median error 61.0%, both only for predictions outside of the label cone. This discrepancy is mostly caused by a small number of tests which have an extremely high error. The highest error in this case is 35960% and is solely responsible for an increase of the mean of 89%. Those extreme values usually occur when the model predicts a y very close to zero because the error basically describes how far off the prediction is from the label in multiples of the prediction. In Figure 15 the distribution of the errors for this case is visualized and shows that most of them are in the range $[10\%, 100\%]$. Consequently the median is considered as more meaningful than the mean in the evaluation, since the mean gives a wrong idea of how well a model actually performs. In the final results the mean errors won't be considered at all to prevent a false impression of the results but will be used in the performance comparison tables in the appendix as an additional parameter.

• **mean/median of the inaccuracies for the predictions to the cone edge and the cone mean**: These parameters are called inaccuracies, so they are more easily distinguishable from the errors but basically are errors themselves. As mentioned before the errors give high values if the y part of the predictions are very small and therefore increase the mean unproportionally. The inaccuracies are an additional parameter to overcome that problem and have more information about a models performance and are computed as in Section 4.2. The inaccuracies basically describe how far off the prediction is from the cone edge or the mean of the cone in multiples of the width of the cone.

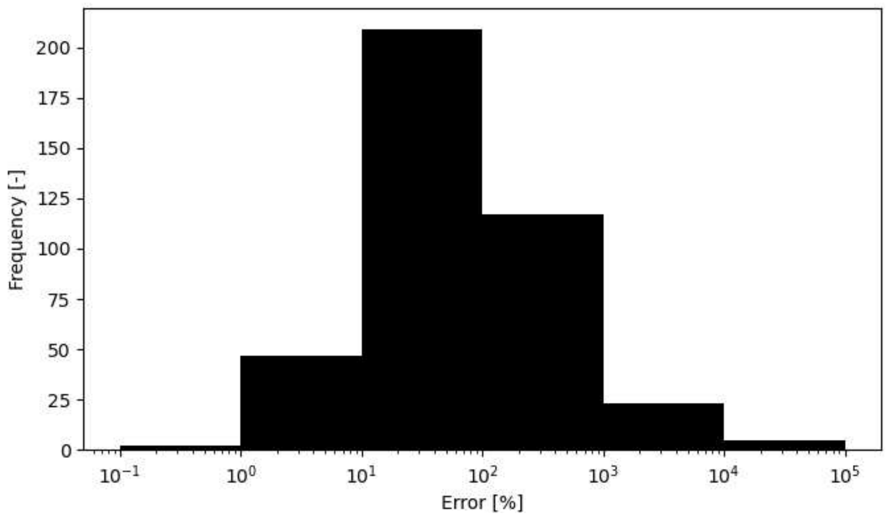


Figure 15: *Example error distribution for the predictions outside the cone.*

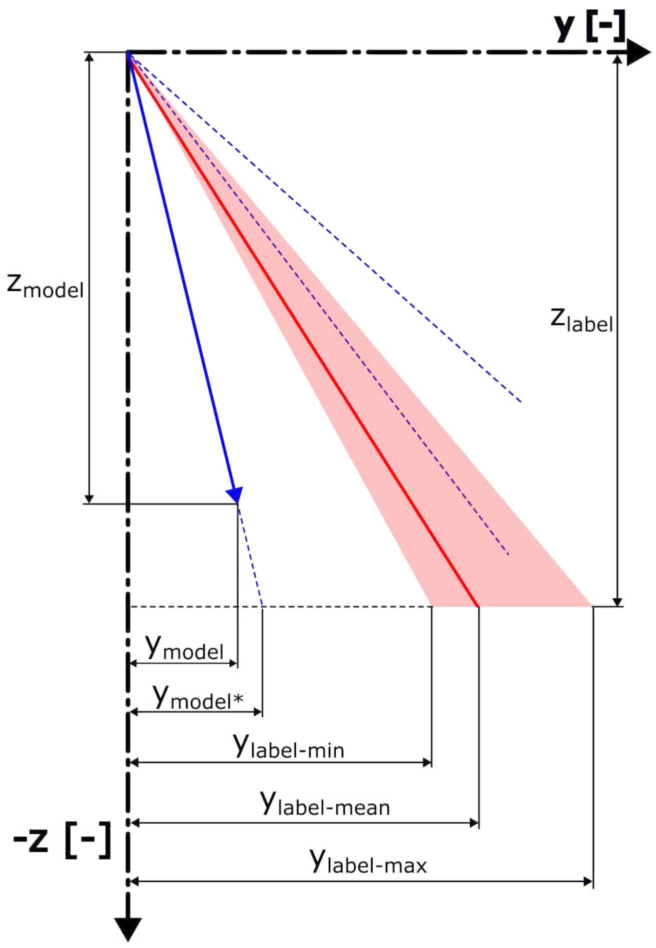


Figure 16: *This image shows an example output of the model (blue arrow), the label data represented as the red cone and the mean of the label data as the red line. The dashed blue lines show the possible cases for the error calculation where the most right is the case $m_{model} < m_{labelmax}$ the most left is $m_{model} > m_{labelmin}$ and the line in the cone has zero error.*

4.1 Error Computation

To be able to quantify the error of the model output and compare different models, a method to calculate the error was implemented. The idea of this method is to consider a prediction that is inside the cone of the label data as good and therefore as zero error and calculate how far off the predictions are from the cone. The strategy for this method is to bring the z-part of the model output and the label data on the same level and then compare the y-parts. The new y of the model is then called y_{model^*} and is visualized in Figure 16. With the slopes calculated as in the Equations 9

$$m_{labelmin} = \frac{y_{labelmin}}{z_{label}} \qquad m_{labelmax} = \frac{y_{labelmax}}{z_{label}} \qquad m_{model} = \frac{y_{model}}{z_{model}} \tag{9}$$

y_{model^*} is then calculated as in Equation 10.

$$y_{model^*} = m_{model} z_{label} \tag{10}$$

The error in the case of y_{model^*} being smaller than $y_{labelmin}$ can be calculated as in Equation 11. The calculation when y_{model^*} is bigger than $y_{labelmax}$ is analogous.

$$error(y_{model^*} < y_{labelmin}) = \frac{|y_{labelmin} - y_{model^*}|}{|y_{model^*}|} = \frac{\left|(y_{labelmin} - y_{model^*})\frac{1}{z_{label}}\right|}{\left|y_{model^*}\frac{1}{z_{label}}\right|} = \frac{|m_{labelmin} - m_{model}|}{|m_{model}|} \tag{11}$$

The error could have been defined by using the slopes right away but this approach was taken to make it more clear how to interpret it. The error is basically how far the prediction is from the cone measured in multiples of the prediction. Equation 12 summarizes all of the above.

$$error = \begin{cases} \frac{|m_{labelmin} - m_{model}|}{|m_{model}|} & \text{if } m_{model} > m_{labelmin}, \\ \frac{|m_{labelmax} - m_{model}|}{|m_{model}|} & \text{if } m_{model} < m_{labelmax}, \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Note that $y_{model^*} < y_{labelmin} \hat{=} m_{model} > m_{labelmin}$ and $y_{model^*} > y_{labelmax} \hat{=} m_{model} < m_{labelmax}$.

4.2 Inaccuracy Computation

For the computations of the inaccuracies Equations 9 and 10 are needed again. Two types are introduced: The first type measures the distance to the cone edge and is counted as zero if the prediction falls into the cone and the second type measures the distance to the mean of the cone. Note that the mean is not necessarily the middle of the cone since the cone only describes the the outer edges of the label data.

Inaccuracy to the Cone Edge

Again, lets consider the case of y_{model^*} being smaller than $y_{labelmin}$. In this case the inaccuracy to the cone edge can be computed as in Equation 13.

$$\begin{aligned} inaccuracy_{edge}(y_{model^*} < y_{labelmin}) &= \frac{|y_{labelmin} - y_{model^*}|}{|y_{labelmax} - y_{labelmin}|} \\ = \frac{\left|(y_{labelmin} - y_{model^*})\frac{1}{z_{label}}\right|}{\left|(y_{labelmax} - y_{labelmin})\frac{1}{z_{label}}\right|} &= \frac{|m_{labelmin} - m_{model}|}{|m_{labelmax} - m_{labelmin}|} \end{aligned} \tag{13}$$

The computation for all the cases is summarized in Equation 14. Again, note that $y_{model^*} < y_{labelmin} \hat{=} m_{model} > m_{labelmin}$ and $y_{model^*} > y_{labelmax} \hat{=} m_{model} < m_{labelmax}$.

$$inaccuracy_{edge} = \begin{cases} \frac{|m_{labelmin} - m_{model}|}{|m_{labelmax} - m_{labelmin}|} & \text{if } m_{model} > m_{labelmin}, \\ \frac{|m_{labelmax} - m_{model}|}{|m_{labelmax} - m_{labelmin}|} & \text{if } m_{model} < m_{labelmax}, \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

Inaccuracy to the Cone Mean

The computation of the inaccuracy to the mean of the cone is basically analogous to the inaccuracy to the cone edge but one new variable is needed which is described in Equation 15.

$$m_{labelmean} = \frac{y_{labelmean}}{z_{label}} \tag{15}$$

Since it always measure the distance to the mean of the cone there is only one case. Consequently the inaccuracy is computed as in Equation 16.

$$inaccuracy_{mean} = \frac{|m_{labelmean} - m_{model}|}{|m_{labelmax} - m_{labelmin}|} \tag{16}$$

5 Using CNNs for Learning Visual-Tactile Correlation

The main algorithm in this thesis is responsible for predicting the tactile feedback of the sensors based on 2D images. It is a Convolutional Neural Network because the task is a part of computer vision and they have been proven effective for these kinds of problems numerous times before (Yoo, 2015). Throughout this thesis many variations of input images were tried to get the best possible results with the idea to focus more and more on the important information. After the input format was chosen a new architecture for the neural network was built and the hyperparameters optimized. This section describes this process in detail and quantifies the most important performances of the different models and approaches. For more information on the results and detailed descriptions of all the architectures refer to the Section 6.

5.1 Choosing the Best Image Variation

The different image variations described in Section 3.2.1 were initially tested with the AlexNet architecture and then compared to find the best performing one. For detailed results refer to Table 7. The Color-Cropped, the Edge-Cropped and the Edge-Index images had overall the best results but were relatively close to each other. Since this information wasn't enough to choose the best option, more tests were done with the SchoeConv100 - V3(FC160) and the SchoeConv227 - V1 architectures to determine which has the best potential (Table 8 & 9). The conclusion was that the Edge-Index images performed the best overall and therefore are the used image variation for this thesis.

5.2 Building the Architecture

The basic idea for the implemented architecture was that the kernels of each consecutive convolutional layer cover a bigger area than the one before. This is implemented by pooling layers which reduce the size of the feature maps by half but the kernels stay the same size for every layer. For explanation lets consider the example shown in Figure 18. The first layer recognizes small features like straight lines or segments of a ring. In the next layer the kernels cover a bigger area and start to combine the previous features to a ring and a longer line. The kernels of the third layer now cover an even bigger area and combine the ring and the straight line to a finger. In the end the fully connected layers consider all the features and the relative positions to each other to give a prediction about the tactile response. This explanation only serves as food for thought but does not actually describe how the neural network comes to a result. The architectures based on this idea are named SchoeConv.

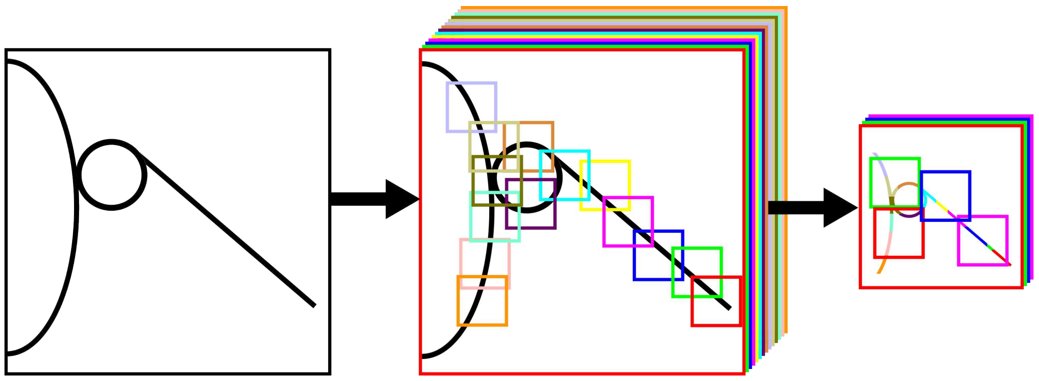


Figure 17: *Left image: Example input image. Middle image: The colored squares represent where the different kernels have the best results. Each kernel produces its own feature map. Right image: The feature maps are now smaller but the kernel size stays the same, consequently the kernels cover a bigger area.*

After the basic idea for the neural network was developed the specifics of the architecture had to be determined. This was done by experimenting with different kernel sizes, number of feature maps, number of layers, etc. Additionally some more or less random architectures were also tested to make sure that the implemented idea performs as intended. The training process for every network was the same to ensure comparability. The configuration is shown in Table 14. During training after each epoch the model was evaluated on the test set, so after the process was finished the best performing epoch could be chosen. The results of the experiments are shown in Table 10 & 11. The architectures SchoeConv100 all use image inputs of the size 100×100. They were used to determine a good configuration of the kernel size and number of feature maps without being too computationally expensive, compared to the SchoeConv227 which use inputs of the size 227×227. The best performing ratio of input size to kernel size was then taken and transferred to an input size of 227. Later the SchoeConv100s were also used to find a good strategy for the Bayesian Optimization before committing to the bigger models. The experiments led to the following conclusions:

- zero-padding seems to improve the performance
- more feature maps increase the overall results
- input image size of 227 is better than 100
- wider fully connected layers don't necessarily improve the performance
- a bigger kernel can be disadvantageous

The best performing architecture was the SchoeConv227 - V2 and is visualized in Figure 18.

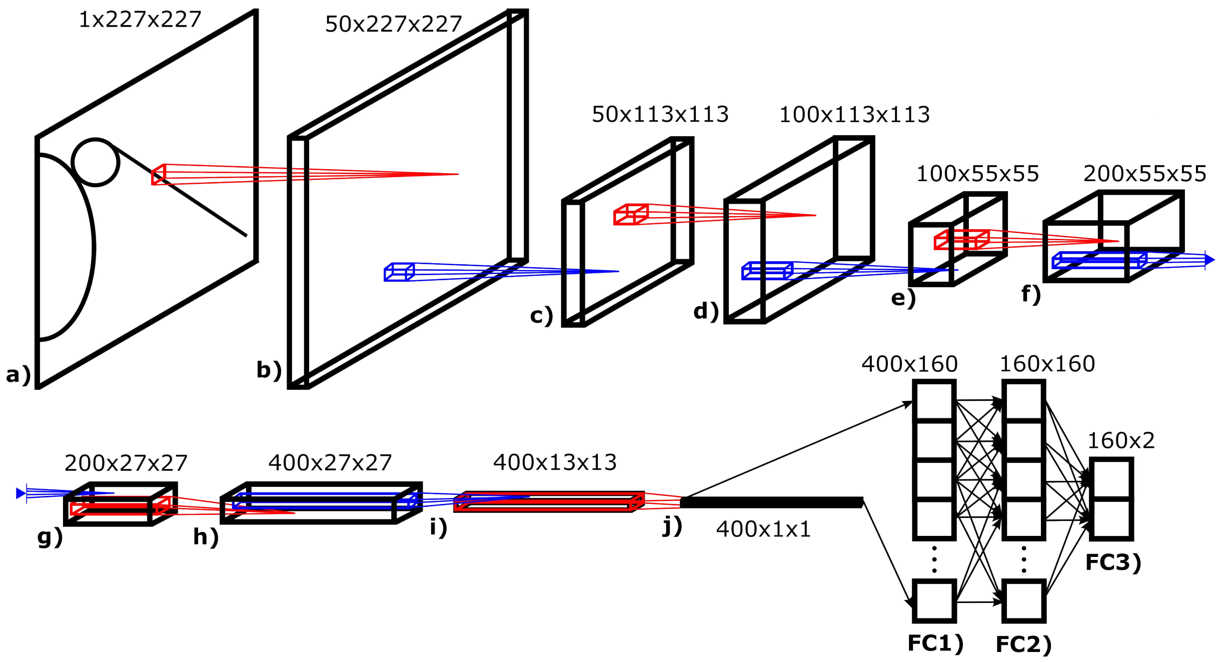


Figure 18: *In this illustration SchoeConv227 - V2 is visualized. The dimensions for the convolutional layers a) to j) are* $[channels \times height \times width]$ *and for the fully connected layers FC1) to FC3)* $[input \times output]$.
a) to b): *Convolution with 50 kernels of the size* 13×13*, stride of* 1 *and padding of* 6.
b) to c): *Max-Pooling with a kernel of the size* 3×3 *and stride of* 2.
c) to d): *Convolution with 100 kernels of the size* 13×13*, stride of* 1 *and padding of* 6.
d) to e): *Max-Pooling with a kernel of the size* 3×3 *and stride of* 2.
e) to f): *Convolution with 200 kernels of the size* 13×13*, stride of* 1 *and padding of* 6.
f) to g): *Max-Pooling with a kernel of the size* 3×3 *and stride of* 2.
g) to h): *Convolution with 400 kernels of the size* 13×13*, stride of* 1 *and padding of* 6.
h) to i): *Max-Pooling with a kernel of the size* 3×3 *and stride of* 2.
i) to j): *Convolution with 400 kernels of the size* 13×13*, stride of* 1 *and padding of* 6.
j) to FC1): *Reshaping of the features to fit as an input for the first fully connected layer.*
FC1) & FC2): *Fully connected layers of width* 160.
FC3): *Output layer of size* 2.

5.3 Optimizing the Hyperparameters & Final Results

Until this point the hyperparameters for the training process were always the same (Table 14). The parameters of interest here are the batch size, the learning rate and the weight decay. The Bayesian Optimization algorithm is designed to maximise the output of a black-box function by tweaking the given input variables. In this case the function to be optimised is the training process. For this to work it is necessary for the training process to return a number, ideally a value that represents how well the generated model performs on the test set. This is visualized in Figure 19.

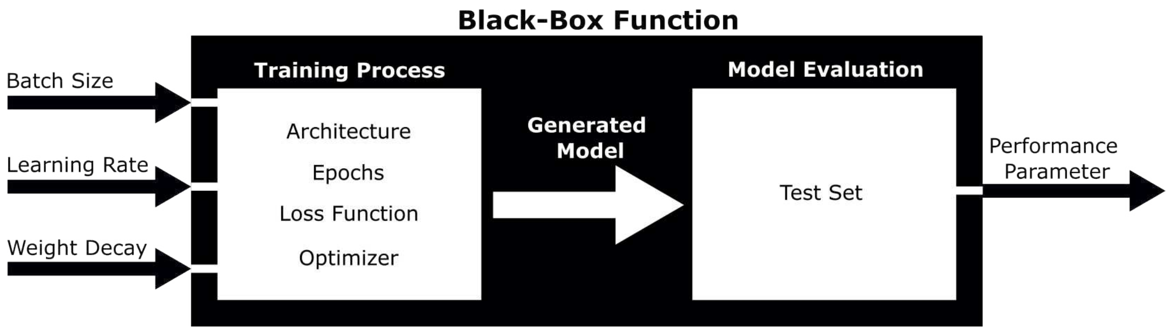


Figure 19: *This illustration visualizes the black-box function the Bayesian Optimization algorithm is supposed to maximise. The training process generates a model with the suggested hyperparameters. This model is then evaluated on the tests set and one of the performance parameters is then the output of the black-box function. This means the Bayesian Optimization algorithm tries to maximise the performance parameter by adjusting the input.*

As stated in Section 5.2, the best performing architecture is the SchoeConv227 - V2 but it is relatively big, consequently computationally expensive to train. To find a good parameter to use as the output for the black-box function in a reasonable amount of time, the SchoeConv100 - V3(FC160) architecture was used as a surrogate doing optimisation test runs. In those test runs the number of epochs was always set to 5 to save some time and the models from previous training processes usually didn't really improve after five epochs. The number of iterations for the Bayesian Optimisation algorithm was set to 50 with 10 random initialisation iterations. The median overall Error, the median error for predictions not in the cone and the % of predictions in the cone were tried as black-box function outputs. After the algorithm was finished, the best performing hyperparamter configuration was used to then train the architecture properly. The detailed results are shown in Table 12. Generally it can be said, that the performances of the models didn't improve by much but the median overall error as the output for the black-box function generated the best results by a small margin. Based on this information the Bayesian Optimisation algorithm was applied to the SchoeConv227 - V2 architecture with the median overall error as the target. The performance improved for most of the evaluation parameters. The results for the comparison to the performances before the optimization are shown in Table 13.

The Bayesian Optimization was the last step of the main algorithm implementation. The final performance of the framework is in Table 6.

Table 6: *Performance of the Best architecture (SchoeConv227 - V2) Optimised.*

% in Correct y	93.5	**Median Error Not in Cone [%]**	39	**Mean Inaccuracy to Cone-Edge [%]**	45.2
% in Cone	44.1	**Median Inaccuracy to Cone-Edge [%]**	9.5	**Mean Inaccuracy to Cone-Mean [%]**	82.5
Median Error Overall [%]	6.7	**Median Inaccuracy to Cone-Mean [%]**	57.1	**Mean MSE**	2.1e-3

6 Evaluation Results

To find the best possible configuration of input image variation and network architecture, numerous tests have been done. This section presents all evaluation results in detail and shows the performance improvement with each step. The best and therefore most important results are highlighted in green. The training parameters for the tests described in the Sections 6.1 and 6.2 are listed in Table 14.

6.1 Comparison of the Image Variations

As mentioned in Section 5.1, all the different image variations were first tried with the AlexNet (Table 7) architecture. At this point the Colored Cropped Image yields the best results and the Edge Cropped and Edge Index Image are a close second and third rank. Interestingly the change of performance from the colored to the edged images stays relatively equal in the case of the Rotated Base Image but increases drastically in for the Index Image.

Table 7: *Performances of the Image Variations - Tested with AlexNet*

Image Variation	**Color - Rotated Base**	**Color - Cropped**	**Color - Index**	**Edge - Rotated Base**	**Edge - Cropped**	**Edge - Index**
Best Epoch	11	11	6	10	19	4
% in Correct y	70	83.8	52.9	72.5	83.9	86.3
% in Cone	24.6	34.6	5.9	23.2	35.3	35.4
Median Error Overall [%]	71.3	18.5	263	88.8	22.3	25.6
Median Error Not in Cone [%]	120.2	68.2	268	130	61.5	73.5
Mean Error Overall [%]	858	150	304	255	216	168
Mean Error Not in Cone [%]	1137	229	323	332	334	260
Median Inaccuracy to Cone-Edge [%]	66.8	26.2	121	70.2	33.9	34
Median Inaccuracy to Cone-Mean [%]	115	72.2	167	118	79.6	78.3
Mean Inaccuracy to Cone-Edge [%]	115	75.9	136	109	66.2	61.1
Mean Inaccuracy to Cone-Mean [%]	157	115	183	150	105	99.8
Mean MSE	4.4e-3	2.6e-3	4.9e-3	4.2e-3	2.8e-3	2.4e-9

Since the Colored Cropped, the Edge Cropped and the Edge Index Images performed similarly in the test with the AlexNet architecture, more tests were done with the, for this thesis implemented, architecture idea. Table 8 shows the results of an architecture with input size of 100×100 and Table 9 with input size 227×227. The Edge Index Image performs the best in both cases. Surprisingly, the Colored Cropped Image achieved bad results with for a bigger input size.

Table 8: *Performances of the best Image Variations - Tested with SchoeConv100 - V3(FC160)*

Image Variation	Color - Cropped	Edge - Cropped	Edge - Index
Best Epoch	17	8	8
% in Correct y	86.6	82	91.6
% in Cone	37.6	32.6	41.7
Median Error Overall [%]	13.1	25.4	9.27
Median Error Not in Cone [%]	54.7	64	60.5
Mean Error Overall [%]	179	143	157
Mean Error Not in Cone [%]	287	212	270
Median Inaccuracy to Cone-Edge [%]	19.2	36.9	14
Median Inaccuracy to Cone-Mean [%]	68.1	80.8	60.1
Mean Inaccuracy to Cone-Edge [%]	65.5	68.4	53.3
Mean Inaccuracy to Cone-Mean [%]	103	108	89.9
Mean MSE	2.6e-3	2.7e-3	2.2e-3

Table 9: *Performances of the best Image Variations - Tested with SchoeConv227 - V1*

Image Variation	Color - Cropped	Edge - Cropped	Edge - Index
Best Epoch	2	7	18
% in Correct y	52.1	84.3	94
% in Cone	8.7	34.4	41.7
Median Error Overall [%]	389	21.2	7.34
Median Error Not in Cone [%]	464	58.3	43.2
Mean Error Overall [%]	509	150	104
Mean Error Not in Cone [%]	558	228	179
Median Inaccuracy to Cone-Edge [%]	106	29.8	11.2
Median Inaccuracy to Cone-Mean [%]	160	75.5	58.4
Mean Inaccuracy to Cone-Edge [%]	132	65.5	46.3
Mean Inaccuracy to Cone-Mean [%]	178	104.3	82.4
Mean MSE	5.2e-3	2.7e-3	1.9e-3

6.2 Comparison of the Architectures

To find the best configuration for the architecture idea explained in Section 5.2 numerous experiments were done. Table 10 and Table 11 show the performances of all the architectures tried in the course of this thesis. Overall the SchoeConv227 - V2 showed the best results.

Table 10: *Architecture Performances 1/2*

Model	**AlexNet**	**AlexNet - 1×1**	**RandomConv**	**SmallConv**	**SchoeConv100 - V1**	**SchoeConv100 - V2**	**SchoeConv100 - V3(FC160)**	**SchoeConv100 - V3(FC160) - 4×4**	**SchoeConv100 - V3(FC480)**
Input Size	227	227	227	227	100	100	100	100	100
Best Epoch	4	6	10	13	19	10	8	7	18
% in Correct y	86.3	86.6	79.2	76.5	83.2	87.2	91.6	92.6	89.8
% in Cone	35.4	37.6	28.8	26.4	35.4	36.9	41.7	40.4	41.4
Median Error Overall [%]	25.6	18.5	41.4	52.3	23.3	13.3	9.27	12	11.8
Median Error Not in Cone [%]	73.5	65.8	87.2	107	82.2	54.3	60.5	51.4	64.9
Mean Error Overall [%]	169	156	527	1282	164	170	157	175	210
Mean Error Not in Cone [%]	261	251	740	1742	254	270	270	294	358
Median Inaccuracy to Cone-Edge [%]	34	26.6	42	47.9	33.4	22.9	14	17.6	17.3
Median Inaccuracy to Cone-Mean [%]	78.3	73.5	89.6	96.4	79.2	73	60.1	65.9	64.6
Mean Inaccuracy to Cone-Edge [%]	61.1	61.6	84.7	145	68.9	60.3	53.3	53.3	54.1
Mean Inaccuracy to Cone-Mean [%]	99.8	99.3	125.2	104	109	98.5	89.9	90.9	90.9
Mean MSE	2.4e-3	2.6e-3	3.2e-3	4e-3	2.5e-3	2.3e-3	2.2e-3	2.3e-3	2.1e-3

Table 11: *Architecture Performances 2/2*

Model	**SchoeConv100 - V4**	**SchoeConv100 - V5**	**SchoeConv227 - V1**	**SchoeConv227 - V2**
Input Size	100	100	227	227
Best Epoch	3	7	18	11
% in Correct y	91	92.4	94	94.1
% in Cone	39.6	42	41.7	41.2
Median Error Overall [%]	10.9	7.96	7.34	7.26
Median Error Not in Cone [%]	51.5	53.1	43.2	41.7
Mean Error Overall [%]	120	135	104	293
Mean Error Not in Cone [%]	198	233	179	497
Median Inaccuracy to Cone-Edge [%]	15	12.5	11.2	9.8
Median Inaccuracy to Cone-Mean [%]	60.4	60.8	58.4	58.4
Mean Inaccuracy to Cone-Edge [%]	50.6	47.1	46.3	44.9
Mean Inaccuracy to Cone-Mean [%]	87.2	83.7	82.4	81.2
mean MSE	2.3e-3	1.9e-3	1.9e-3	2.0e-3

6.3 Performance Improvement by Bayesian Optimisation

To find the best performance parameter as an output for the Black-Box function, the smaller SchoeConv100 - V3(FC160) architecture was used as a surrogate, due to a smaller computational cost. The results of the tried parameters are shown in Table 12. The Median Overall Error achieved the best performance by a small margin.

Table 12: *Performances after Bayesian Optimisation - SchoeConv100 - V3(FC160)*

Comparison of Performance Parameters

Black-Box Output	**Before Optim. for Comparison**	**Median Error Overall**	**Median Error Not in Cone**	**% in Cone**
Batch Size	5	3	5	3
Learning Rate	1e-4	1.02e-4	1.7e-4	8.5e-5
Weight Decay	1e-5	2.79e-6	2.15e-6	1.5e-6
Input Size	100	100	100	100
Best Epoch	8	3	3	12
% in Correct y	91.6	93.1	91.9	91.6
% in Cone	41.7	40.9	40.3	43.2
Median Error Overall [%]	9.27	9.74	9.3	9.98
Median Error Not in Cone [%]	60.5	42.1	49.5	64.5
Mean Error Overall [%]	157	133	118	127.2
Mean Error Not in Cone [%]	270	225	198	224
Median Inaccuracy to Cone-Edge [%]	14	14.6	16.7	14.8
Median Inaccuracy to Cone-Mean [%]	60.1	60.4	62.6	60.4
Mean Inaccuracy to Cone-Edge [%]	53.3	49	51.5	52.4
Mean Inaccuracy to Cone-Mean [%]	89.9	86.3	89.7	89.5
mean MSE	2.2e-3	2.2e-3	2.3e-3	2.1e-3

After the performance parameter was chosen, the last step of the thesis was to apply the Bayesian Optimisation algorithm to the SchoeConv227 - V2 architecture. The comparison of the performances before and after the optimisation are shown in Table 13. The values highlighted in green are also the final results of this thesis.

Table 13: *Performances after Bayesian Optimisation - SchoeConv227 - V2*

Performance Parameter was the Overall Median Error

SchoeConv227 - V2	**Before Optim. for Comparison**	**Optimised**
Batch Size	5	4
Learning Rate	1e-4	4.6e-5
Weight Decay	1e-5	8.2e-6
Input Size	227	227
Best Epoch	11	5
% in Correct y	94.1	93.5
% in Cone	41.2	44.1
Median Error Overall [%]	7.26	6.7
Median Error Not in Cone [%]	41.7	39
Mean Error Overall [%]	293	95.5
Mean Error Not in Cone [%]	497	171
Median Inaccuracy to Cone-Edge [%]	9.8	9.5
Median Inaccuracy to Cone-Mean [%]	58.4	57.1
Mean Inaccuracy to Cone-Edge [%]	44.9	45.2
Mean Inaccuracy to Cone-Mean [%]	81.2	82.5
mean MSE	2.0e-3	2.1e-3

7 Conclusion

This thesis implemented a framework that predicts how the response tactile sensors would feel when grasping an object based on 2D visual data. For this purpose a new dataset was built, which consisted, after augmentation, of about 3200 training and 680 test visual-tactile data pairs. Numerous variations for the visual data were tried to find the best fit and the force vectors of the tactile data were reduced to unit length to only consider the direction of the forces. A separate neural network was trained to find the bounding boxes of the fingers in the input images for generating some of the image variations. An evaluation strategy with several performance parameters was developed, to be able to confidently evaluate and compare the different approaches and architectures. The main algorithm is a Convolutional Neural Network and its architecture was developed for the purpose of this project. Finally the best performing architecture was optimised with the Bayesian Optimization algorithm.

The following two research questions were addressed in this thesis.

Does the framework successfully predict the sensor feedback?
The framework undeniably understands the relationship between relative positioning of the sensors to the object and the resulting tactile feedback. This can be said because the model predicted the y part of the force in 93.5% of the times in the correct direction and this is highly dependent on how the object was grasped. The accuracy on the other hand is not ideal. Only 44.1% of the predictions are actually in the label data cone, which is, considering how wide these cones can be, not bad but not a satisfactory result either. The possible causes for the inaccuracy are discussed in Section 7.1.

Was the equipment used suitable for this task?
The main problem was the robot hand. When grasping the objects the hand would, if not careful, deform and therefore induce forces into the tactile sensors and consequently falsify the measurements. Also the shape of the tactile sensors was a limiting factor in many cases, because if the object was touched to far off the center of the sensors, the tactile feedback would be close to zero. In combination with the small range of motion of the robot fingers, the possible sizes of objects to choose from was very limited. Additionally the sensors themselves are not very accurate.

7.1 Discussion

In this subsection the decisions made in this thesis are revisited and possible improvements are suggested. Also the performance of the model is discussed in more detail and the likely causes for the unsatisfactory results determined.

The Tactile Data
As mentioned, the quality of the tactile data is partly dependent on the used hardware but also on the acquisition process itself. The tactile data was gathered by manually squeezing the fingers against the object. Naturally this is not ideal, because our hand will always induce some force into the system, so this is likely to be another source of inaccuracy in the data. Unfortunately, using the robot hands internal motors to establish the contact is not only tedious and time consuming but also makes it very difficult to grasp the object at the needed positions. In summary it can be said that the data is not very accurate.

The Visual Data
The position of the fingers was defined by manually editing them into the images. This method is inaccurate and likely in part responsible for the unsatisfactory performance of the framework. Additionally, it is very time consuming to build a data set in this way.

The data set was augmented by rotating the images. This was done, so the predictions would be independent of the orientation of the fingers, meaning from which direction they approach the object. In practice, the predictions are close to each other but vary enough to have a significant impact on the performance. A possible solution would be to turn the images, so the fingers always have the same orientation.

The Amount of Data
The best way of improving the generalization of a model is to use more data. The amount of data used for training the models in this thesis is likely not enough, consequently the performance would probably be better if more data was gathered. Nonetheless, the model preforms well on novel as well as on known objects. This can be due to the fact that the shapes of the objects are similar in many cases but seem to be, at least partly, due to generalization of the model.

Limitations
For this framework to function properly, the input data needs to be quite specific:

- The background in the base images should be similar to the training data, otherwise the performance could worsen.
- The angle and distance from which the objects are photographed relative to the fingers always need to be the same, because the view angle wasn't considered in this framework.
- The framework only predicts two dimensional forces.
- The grasped surface needs to be rigid, because the framework hasn't been trained with soft objects.

7.2 Future Work

The first step to improve this framework should be to gather more data. To make the acquisition more efficient, a system could be implemented that automates the process of finding the object, grasping it in multiple positions and finally describing the configuration. Pinto and Gupta (2016) uses a similar system to teach a robot how to grasp. To be able to effectively predict three dimensional forces, the features of multiple views could be considered by, for example, naively concatenating the feature spaces or use a more developed approach as proposed by Zhang et al. (2020). Soft objects are not considered in the current state of the framework. To be less restricted with the choice of possible objects to touch, the framework could be extended with a system that provides information about stiffness.

The suggestions so far aim on improving the current framework. Other possibilities for future work could be to use the framework as a support for other tasks such as object manipulation or object recognition.

Bibliography

Bergmann Tiest, Wouter M. (2010). “Tactual perception of material properties”. In: *Vision Research* 50.24. Perception and Action: Part I, pp. 2775–2782. ISSN: 0042-6989. DOI: https://doi.org/10.1016/j.visres.2010.10.005. URL: https://www.sciencedirect.com/science/article/pii/S0042698910004967.

Bradski, G. (2000). “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools*.

Canny, John (1986). “A Computational Approach to Edge Detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6, pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

Chebotar, Yevgen et al. (2016). “Self-supervised regrasping using spatio-temporal tactile features and reinforcement learning”. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1960–1966. DOI: 10.1109/IROS.2016.7759309.

Dang, Hao and Peter K. Allen (2012). “Learning grasp stability”. In: *2012 IEEE International Conference on Robotics and Automation*, pp. 2392–2397. DOI: 10.1109/ICRA.2012.6224754.

Dong, Siyuan et al. (2019). “Maintaining Grasps within Slipping Bounds by Monitoring Incipient Slip”. In: *2019 International Conference on Robotics and Automation (ICRA)*, pp. 3818–3824. DOI: 10.1109/ICRA.2019.8793538.

Falco, Pietro et al. (2017). “Cross-modal visuo-tactile object recognition using robotic active exploration”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5273–5280. DOI: 10.1109/ICRA.2017.7989619.

Fleming, Roland W. (2014). “Visual perception of materials and their properties”. In: *Vision Research* 94, pp. 62–75. ISSN: 0042-6989. DOI: https://doi.org/10.1016/j.visres.2013.11.004. URL: https://www.sciencedirect.com/science/article/pii/S0042698913002782.

Frazier, Peter I (2018). “A tutorial on Bayesian optimization”. In: *arXiv preprint arXiv:1807.02811*.

Goodfellow, Ian et al. (2020). “Generative adversarial networks”. In: *Communications of the ACM* 63.11, pp. 139–144.

Goodfellow, Ian J., Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press.

Hyttinen, Emil, Danica Kragic, and Renaud Detry (2017). “Estimating tactile data for adaptive grasping of novel objects”. In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pp. 643–648. DOI: 10.1109/HUMANOIDS.2017.8246940.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25.

Lee, Jet-Tsyn, Danushka Bollegala, and Shan Luo (2019). ““Touching to see” and “seeing to feel”: Robotic cross-modal sensory data generation for visual-tactile perception”. In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 4276–4282.

Li, Yunzhu et al. (2019). “Connecting Touch and Vision via Cross-Modal Prediction”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Liu, Huaping et al. (2017). “Visual–Tactile Fusion for Object Recognition”. In: *IEEE Transactions on Automation Science and Engineering* 14.2, pp. 996–1008. DOI: 10.1109/TASE.2016.2549552.

Nogueira, Fernando (2014–). *Bayesian Optimization: Open source constrained global optimization tool for Python*. URL: https://github.com/fmfn/BayesianOptimization.

Paszke, Adam et al. (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pinto, Lerrel and Abhinav Gupta (2016). “Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours”. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3406–3413. DOI: 10.1109/ICRA.2016.7487517.

Purri, Matthew and Kristin Dana (2020). "Teaching cameras to feel: Estimating tactile physical properties of surfaces from images". In: *European Conference on Computer Vision*. Springer, pp. 1–20.

Qi, Charles R. et al. (2017). "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Rasmussen, Carl Edward and Christopher K. I. Williams (2005). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press. ISBN: 026218253X.

Shapiro, Linda G, George C Stockman, et al. (2001). *Computer vision*. Vol. 3. Prentice Hall New Jersey.

Shin, Hochul et al. (2019). "Sequential Image-Based Attention Network for Inferring Force Estimation Without Haptic Sensor". In: *IEEE Access* 7, pp. 150237–150246. DOI: 10.1109/ACCESS.2019.2947090.

Snoek, Jasper, Hugo Larochelle, and Ryan P Adams (2012). "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems* 25.

Takahashi, Kuniyuki and Jethro Tan (2019). "Deep visuo-tactile learning: Estimation of tactile properties from images". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 8951–8957.

Yoo, Hyeon-Joong (2015). "Deep convolution neural networks in computer vision: a review". In: *IEIE Transactions on Smart Processing and Computing* 4.1, pp. 35–43.

Zapata-Impata, Brayan S and Pablo Gil (2020). "Prediction of tactile perception from vision on deformable objects". In.

Zapata-Impata, Brayan S. et al. (2021). "Generation of Tactile Data From 3D Vision and Target Robotic Grasps". In: *IEEE Transactions on Haptics* 14.1, pp. 57–67. DOI: 10.1109/TOH.2020.3011899.

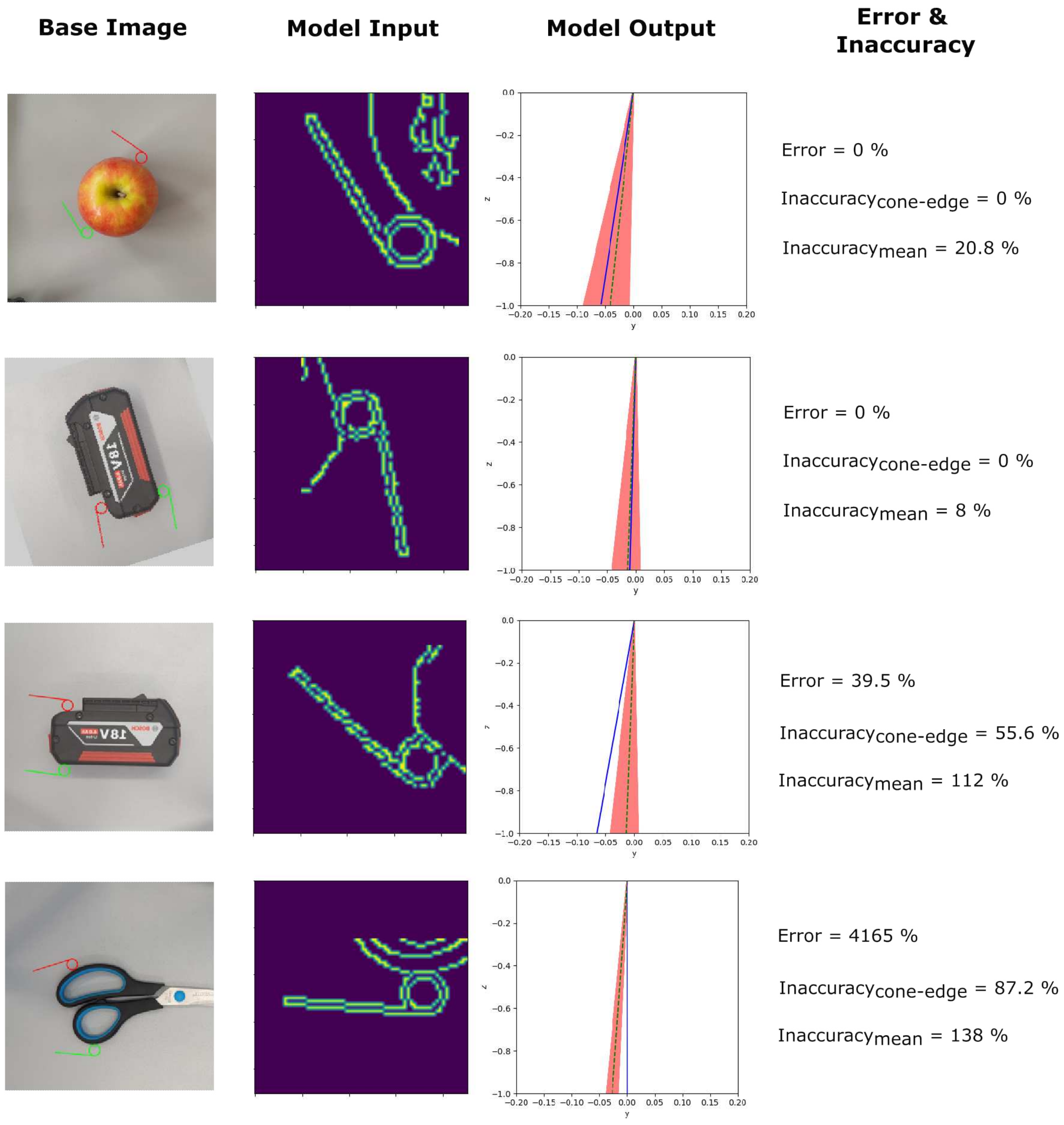
Zhang, Jinxin et al. (2020). "Collaborative weighted multi-view feature extraction". In: *Engineering Applications of Artificial Intelligence* 90, p. 103527.

Zhang, Tao et al. (2021). "Visual-Tactile Fused Graph Learning for Object Clustering". In: *IEEE Transactions on Cybernetics*, pp. 1–15. DOI: 10.1109/TCYB.2021.3080321.

A APPENDIX

A.1 Example Model Predictions

This subsection shows examples of the final model predictions and their errors and inaccuracies. The purpose of this is to get a better feeling of how to interpret these performance parameters. In the graphs the red cone represents the label data, the green dashed line the mean of the label data and the blue line the prediction.



Error = 0 %

$\text{Inaccuracy}_{\text{cone-edge}}$ = 0 %

$\text{Inaccuracy}_{\text{mean}}$ = 20.8 %

Error = 0 %

$\text{Inaccuracy}_{\text{cone-edge}}$ = 0 %

$\text{Inaccuracy}_{\text{mean}}$ = 8 %

Error = 39.5 %

$\text{Inaccuracy}_{\text{cone-edge}}$ = 55.6 %

$\text{Inaccuracy}_{\text{mean}}$ = 112 %

Error = 4165 %

$\text{Inaccuracy}_{\text{cone-edge}}$ = 87.2 %

$\text{Inaccuracy}_{\text{mean}}$ = 138 %

Figure 20: *Some examples of model predictions with their errors and inaccuracies.*
The second and third example are the same grasp but the image is rotated, still the model gives different outputs. This example shows why the data augmentation didn't work as intended in some cases.
The fourth example has a very big error, even though the absolute distance between prediction and cone isn't. This example shows why the mean error parameter can be misleading at times.

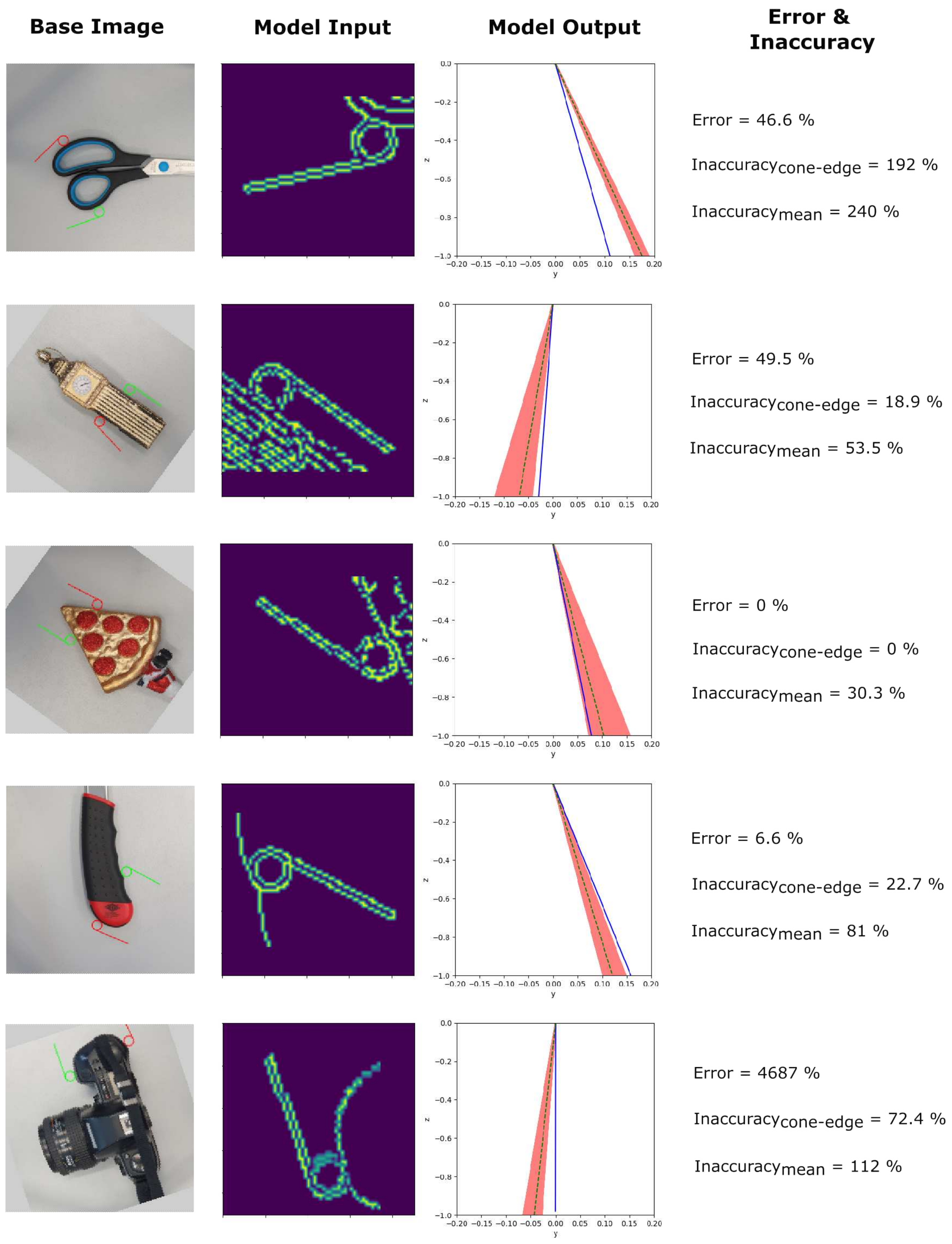


Figure 21: *More prediction examples.*

A.2 Architectures & Hyperparameters

In this section the architectures used in the course of this thesis are described. These exact configurations are used for the Edge Index Finger Images, so the number of input channels is one and the output features is two. These two variables can change depending on the used image variation.

As mentioned before, the training parameters were for the comparison of the image variations and the different architectures always the same:

Table 14: *Training Parameters for Image Variation and Network Architecture Comparison*

Loss Function	Mean Square Error
Optimizer	Adam
Learning Rate	10^{-4}
Weight Decay	10^{-5}
Batch Size	5
Epochs	20

AlexNet

Remark		The original AlexNet (Krizhevsky, Sutskever, and Hinton, 2012) architecture.				
Input Image Size		227×227				
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	11	4	1	96	0
	maxpool	3	2			
	ReLU(conv)	5	1	96	256	2
	maxpool	3	2			
	ReLU(conv)	3	1	256	384	1
	ReLU(conv)	3	1	384	384	1
	ReLU(conv)	3	1	384	256	1
	maxpool	3	2			
Regression		**features in**	**features out**			
	ReLU(FC)	9216	4096			
	ReLU(FC)	4096	4096			
	FC	4096	2			

AlexNet - 1×1

Remark	The AlexNet architecure but the size of the feature map is reduced to 1×1.					
Input Image Size	227×227					
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	11	4	1	96	0
	maxpool	3	2			
	ReLU(conv)	5	1	96	256	2
	maxpool	3	2			
	ReLU(conv)	3	1	256	384	1
	ReLU(conv)	3	1	384	384	1
	ReLU(conv)	3	1	384	256	1
	maxpool	3	2			
	ReLU(conv)	6	1	256	256	0
Regression		**features in**	**features out**			
	ReLU(FC)	256	160			
	ReLU(FC)	160	160			
	FC	160	2			

RandomConv

Remark	Quite random architecture.					
Input Image Size	227×227					
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	5	2	1	20	0
	maxpool	4	2			
	ReLU(conv)	3	2	20	40	0
	maxpool	3	2			
	ReLU(conv)	3	2	40	80	0
	maxpool	3	2			
Regression		**features in**	**features out**			
	ReLU(FC)	320	160			
	ReLU(FC)	160	160			
	FC	160	2			

SmallConv

Remark	Small architecture.					
Input Image Size	227×227					
Feat. Extrac.		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	5	3	1	20	0
	maxpool	3	3			
	ReLU(conv)	4	3	20	40	0
	maxpool	2	3			
Regression		**features in**	**features out**			
	ReLU(FC)	360	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv100 - V1

Remark	Idea explained in Sec. 5.2. No padding. Kernel size of 5.					
Input Image Size	100×100					
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	5	1	1	30	0
	maxpool	2	2			
	ReLU(conv)	5	1	30	60	0
	maxpool	2	2			
	ReLU(conv)	5	5	60	120	0
	maxpool	2	2			
	ReLU(conv)	5	1	120	240	0
	ReLU(conv)	5	1	240	240	0
Regression		**features in**	**features out**			
	ReLU(FC)	240	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv100 - V2

Remark	Idea explained in Sec. 5.2. With padding. Kernel size of 5.					
Input Image Size	100×100					
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	5	1	1	30	2
	maxpool	2	2			
	ReLU(conv)	5	1	30	60	2
	maxpool	2	2			
	ReLU(conv)	5	1	60	120	2
	maxpool	3	2			
	ReLU(conv)	5	1	120	240	2
	maxpool	2	2			
	ReLU(conv)	6	1	240	240	0
Regression		**features in**	**features out**			
	ReLU(FC)	240	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv100 - V3(FC160)

Remark	Idea explained in Sec. 5.2. With padding. Kernel size of 7.					
Input Image Size	100×100					
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	7	1	1	30	3
	maxpool	2	2			
	ReLU(conv)	7	1	30	60	3
	maxpool	2	2			
	ReLU(conv)	7	1	60	120	3
	maxpool	3	2			
	ReLU(conv)	7	1	120	240	3
	maxpool	2	2			
	ReLU(conv)	6	1	240	240	0
Regression		**features in**	**features out**			
	ReLU(FC)	240	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv100 - V3(FC160) - 4×4

Remark	Same as SchoeConv100 - V3(FC160) but reduces feature map to 4×4.					
Input Image Size	100×100					
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	7	1	1	30	3
	maxpool	2	2			
	ReLU(conv)	7	1	30	60	3
	maxpool	2	2			
	ReLU(conv)	7	1	60	120	3
	maxpool	3	2			
	ReLU(conv)	7	1	120	240	3
	maxpool	3	3			
Regression		**features in**	**features out**			
	ReLU(FC)	3840	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv100 - V3(FC480)

Remark		Idea explained in Sec. 5.2. With padding. Kernel size of 7.				
Input Image Size		100 × 100				
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	7	1	1	30	3
	maxpool	2	2			
	ReLU(conv)	7	1	30	60	3
	maxpool	2	2			
	ReLU(conv)	7	1	60	120	3
	maxpool	3	2			
	ReLU(conv)	7	1	120	240	3
	maxpool	2	2			
	ReLU(conv)	6	1	240	240	0
Regression		**features in**	**features out**			
	ReLU(FC)	240	480			
	ReLU(FC)	480	480			
	FC	480	2			

SchoeConv100 - V4

Remark		Idea explained in Sec. 5.2. With padding. Kernel size of 9.				
Input Image Size		100 × 100				
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	9	1	1	30	4
	maxpool	2	2			
	ReLU(conv)	9	1	30	60	4
	maxpool	2	2			
	ReLU(conv)	9	1	60	120	4
	maxpool	3	2			
	ReLU(conv)	9	1	120	240	4
	maxpool	2	2			
	ReLU(conv)	6	1	240	240	0
Regression		**features in**	**features out**			
	ReLU(FC)	240	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv100 - V5

Remark		Idea explained in Sec. 5.2. With padding. Kernel size of 7. More feature maps.				
Input Image Size		100 × 100				
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	7	1	1	50	3
	maxpool	2	2			
	ReLU(conv)	7	1	50	100	3
	maxpool	2	2			
	ReLU(conv)	7	1	100	200	3
	maxpool	3	2			
	ReLU(conv)	7	1	200	400	3
	maxpool	2	2			
	ReLU(conv)	6	1	400	400	0
Regression		**features in**	**features out**			
	ReLU(FC)	400	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv227 - V1

Remark		Idea explained in Sec. 5.2. Like SchoeConv100 - V3 just for bigger input.				
Input Image Size		227 × 227				
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	13	1	1	30	6
	maxpool	3	2			
	ReLU(conv)	13	1	30	60	6
	maxpool	3	2			
	ReLU(conv)	13	1	60	120	6
	maxpool	3	2			
	ReLU(conv)	13	1	120	240	6
	maxpool	3	2			
	ReLU(conv)	13	1	240	240	0
Regression		**features in**	**features out**			
	ReLU(FC)	240	160			
	ReLU(FC)	160	160			
	FC	160	2			

SchoeConv227 - V2

Remark		Idea explained in Sec. 5.2. Like SchoeConv100 - V5 just for bigger input.				
Input Image Size		227×227				
Feature Extraction		**kernel size**	**stride**	**channels in**	**channels out**	**padding**
	ReLU(conv)	13	1	1	50	6
	maxpool	3	2			
	ReLU(conv)	13	1	50	100	6
	maxpool	3	2			
	ReLU(conv)	13	5	100	200	6
	maxpool	3	2			
	ReLU(conv)	13	1	200	400	6
	maxpool	3	2			
	ReLU(conv)	13	1	400	400	0
Regression		**features in**	**features out**			
	ReLU(FC)	400	160			
	ReLU(FC)	160	160			
	FC	160	2			

A.3 Code

This appendix consists of the code used to implement this thesis' framework.

A.3.1 Tactile Data Preprocessing

The code used for the preprocessing of the tactile data.

Needed Packages for the Tactile Data Preprocessing Code

```python
import os
import matplotlib.pyplot as plt
import csv
import numpy as np
import math
import openpyxl
```

Function for Extracting Data From Raw CSV File

```python
def extract_data(raw_data_path, normalized):
    """
    This function extracts the raw, measured data from the files and returns
    the forces from the index finger and thumb seperately.
    The force can be normalized. This means that at EACH time step the forces
    in y and z direction are divided by the resultant force of that time step.
    """

    index = [[], []]
    thumb = [[], []]

    with open(raw_data_path, 'r') as file:
        csv_reader = csv.reader(file)
        next(csv_reader) # skip the header

        for row in csv_reader:
            y_index, z_index = int(row[18]), int(row[19])
            y_thumb, z_thumb = int(row[15]), int(row[16])

            if normalized:
                index_raw = (y_index, z_index)
                thumb_raw = (y_thumb, z_thumb)
                y_index, z_index, y_thumb, z_thumb = \
                    normalize_force(index_raw, thumb_raw)

            index[0].append(y_index)
            index[1].append(z_index)
            thumb[0].append(y_thumb)
            thumb[1].append(z_thumb)

    return index, thumb
```

Function for Reducing Force Vectors to Unit Length

```python
def normalize_force(index_raw, thumb_raw):
    """
    Reduces forces to unit vectors, so they don't yield information about the
    magnitude of the force.
    Inputs should be a tuple: (fy, fz)
    """
    index_res = math.sqrt(index_raw[0]**2 + index_raw[1]**2)
    if index_res == 0: index_res = 1 # prevent divison by 0
    y_index = index_raw[0]/index_res
    z_index = index_raw[1]/index_res

    thumb_res = math.sqrt(thumb_raw[0]**2 + thumb_raw[1]**2)
    if thumb_res == 0: thumb_res = 1 # prevent divison by 0
    y_thumb = thumb_raw[0]/thumb_res
    z_thumb = thumb_raw[1]/thumb_res

    return y_index, z_index, y_thumb, z_thumb
```

Function for Splitting the Raw Data Into Smaller Time Series

```python
def split_data(filepath, threshold, normalized):
    """
    Returns a list of tuples of measurements from one file
    [([y1],[z1]), ([y2],[z2]), ...]
    The measuremens where z_index is below threshold will be ignored.
    Everytime the z_index falls below the threshold a new split begins.
    """
    index, thumb = extract_data(filepath, normalized=False)

    y_index_list, z_index_list = [], []
    y_thumb_list, z_thumb_list = [], []
    all_splits = []

    for y_index, z_index, y_thumb, z_thumb in \
    zip(index[0], index[1], thumb[0], thumb[1]):

        z_index_ref = z_index

        if normalized:
            f_index = (y_index, z_index)
            f_thumb = (y_thumb, z_thumb)
            y_index, z_index, y_thumb, z_thumb = \
                normalize_force(f_index, f_thumb)

        # As long as z_index is above threshold values will be appended,
        # because it's still the same split
        if abs(z_index_ref) > threshold:
            y_index_list.append(y_index)
            z_index_list.append(z_index)
            y_thumb_list.append(y_thumb)
            z_thumb_list.append(z_thumb)
        else:
            # Once below the split it will be added to the list of splits.
            one_split = \
                (y_index_list, z_index_list, y_thumb_list, z_thumb_list)

            y_index_list, z_index_list =  [], []
            y_thumb_list, z_thumb_list =  [], []

            # This prevents appending empty splits.
            if len(one_split[0]) != 0:
                all_splits.append(one_split)

    return all_splits
```

Function for Choosing Points to Represent Each Split

```python
def choose_points(file_path, threshold, normalized=False):
    """
    Chooses a point of each split.
    The chosen point is the first peak of z_index after its mean.
    """
    # splits to use to choose the right points
    all_splits_ref = split_data(file_path, threshold, normalized=False)
    # splits to use to return the chosen points
    all_splits = split_data(file_path, threshold, normalized=normalized)

    z_means = [] # mean for choosing points
    for split in all_splits_ref:
        z_means.append(np.mean(split[1]))

    means = [] # means for returning
    for split in all_splits:
        mean = (np.mean(split[0]), np.mean(split[1]), \
                np.mean(split[2]), np.mean(split[3]))
        means.append(mean)

    # choose the first peak that is bigger than the mean in z direction of the
    # index finger
    points = []
    for split_ref, split, z_mean in zip(all_splits_ref, all_splits, z_means):

        z_index = split_ref[1]

        for i in range(1, len(z_index) - 1):
            # if positive peak
            if z_index[i - 1] < z_index[i] > z_index[i + 1] \
                and z_index[i] > z_mean > 0:
                # with index in the end for visualization purposes
                points.append((split[0][i], split[1][i], split[2][i], \
                               split[3][i], i))
                break

            # if negative peak
            if z_index[i - 1] > z_index[i] < z_index[i + 1] \
                and z_index[i] < z_mean < 0:
                points.append((split[0][i], split[1][i], split[2][i], \
                               split[3][i], i))
                break

    return points, means
```

Function for Choosing Points to Represent Each Split

```python
def choose_points(file_path, threshold, normalized=False):
    """
    Chooses a point of each split.
    The chosen point is the first peak of z_index after its mean.
    """
    # splits to use to choose the right points
    all_splits_ref = split_data(file_path, threshold, normalized=False)
    # splits to use to return the chosen points
    all_splits = split_data(file_path, threshold, normalized=normalized)

    z_means = [] # mean for choosing points
    for split in all_splits_ref:
        z_means.append(np.mean(split[1]))

    means = [] # means for returning
    for split in all_splits:
        mean = (np.mean(split[0]), np.mean(split[1]), \
                  np.mean(split[2]), np.mean(split[3]))
        means.append(mean)

    # choose the first peak that is bigger than the mean in z direction of the
    # index finger
    points = []
    for split_ref, split, z_mean in zip(all_splits_ref, all_splits, z_means):

        z_index = split_ref[1]

        for i in range(1, len(z_index) - 1):
            # if positive peak
            if z_index[i - 1] < z_index[i] > z_index[i + 1] \
                and z_index[i] > z_mean > 0:
                # with index in the end for visualization purposes
                points.append((split[0][i], split[1][i], split[2][i], \
                               split[3][i], i))
                break

            # if negative peak
            if z_index[i - 1] > z_index[i] < z_index[i + 1] \
                and z_index[i] < z_mean < 0:
                points.append((split[0][i], split[1][i], split[2][i], \
                               split[3][i], i))
                break

    return points, means
```

Function for Writing the Chosen Points Into the xlsx File

```python
def save_chosen_data_points(raw_data_path, good_splits, threshold):
    """
    Saves the chosen data points into the labels_base.xlsx file.
    One chosen data point is the first peak after the mean of a split.
    The data will be split where the z_index value falls below threshold.
    good_splits is a list of the indices of the splits to choose a data point
    from.
    The forces will be saved  in reduced form!
    """

    # get the points to save
    points, means = choose_points(raw_data_path, threshold, normalized=True)

    # open workbook, create workbook object
    file_path = os.path.join(os.getcwd(), "tactile_data", "labels_base.xlsx")
    wb = openpyxl.load_workbook(file_path)
    # create sheet object
    sheet = wb.active

    image_name = raw_data_path.split(os.sep)[-1].split(".")[0]

    for idx in good_splits:
        # normalize the data
        f_index = (points[idx][0], points[idx][1])
        f_thumb = (points[idx][2], points[idx][3])
        f_norm = normalize_force(f_index, f_thumb)

        f_norm = [str(x) for x in f_norm]
        data_to_write = [image_name]
        data_to_write.extend(f_norm)

        sheet.append(data_to_write)

    wb.save(file_path)
```

Function for Creating a Labels File - Connect Visual and Tactile Data

```python
def make_labels_for_rotated_pics(test):
    """
    This function creates a labels file for all the rotated pictures.
    The rotated pictures are simply the base images rotated and then saved
    as new pictures to make the dataset bigger. Therefor a new labels file is
    needed.
    """

    if test:
        image_folder_path = \
            os.path.join(os.getcwd(), "tactile_data", "pictures_test")
        labels_path = \
            os.path.join(os.getcwd(), "tactile_data", "labels_test.xlsx")
    else:
        image_folder_path = \
            os.path.join(os.getcwd(), "tactile_data", "pictures_training")
        labels_path = \
            os.path.join(os.getcwd(), "tactile_data", "labels_training.xlsx")

    # get a list of content of the folder
    images_list = os.listdir(image_folder_path)
    # this step filters out the folders in the directory -> only files
    images_list = [x for x in images_list \
                   if os.path.isfile(os.path.join(image_folder_path, x))]

    labels_base_path = \
        os.path.join(os.getcwd(), "tactile_data", "labels_base.xlsx")
    # open workbook
    wb = openpyxl.load_workbook(labels_base_path)
    # create sheet object
    sheet = wb.active
    # new workbook in which the new data will be saved
    wb_new = openpyxl.Workbook()
    sheet_new = wb_new.active
    sheet_new.append(["image", "y_index", "z_index", "y_thumb", "z_thumb"])

    for image in images_list:
        # remove "_rotby_xx_i.png"
        image_short = image.split("_rotby_")[0]

        for row in sheet.iter_rows(values_only=True):
            if row[0] == image_short:
                data_to_write = [image.split(".")[0]]
                data_to_write.extend(row[1:])
                sheet_new.append(data_to_write)

    wb_new.save(labels_path)
```

A.3.2 Image Preprocessing

The code used for the preprocessing of the visual data.

Needed Packages for the Image Preprocessing Code

```python
import os
from torchvision import transforms
import torch
from alexnet import AlexNet
from PIL import Image
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import openpyxl
import cv2 as cv
```

Function that Yields the Rotated Image Tensors and Bounding Boxes

```python
def get_pictures_and_bounding_box(device, pics_folder_path, pad=None,
                                  box_model_path=None):
    """
    Yields the image tensors, the corresponding bounding boxes and image path.
    Rotates every image 20 times.
    If box_model_path == None no bounding boxes will be yielded.
    """

    if box_model_path != None:
        # load the model which will be used to compute the bounding boxes
        model = AlexNet(outputs=8).to(device)
        model.load_state_dict(torch.load(box_model_path,
                                         map_location=torch.device(device)))

    # get a list of content of the folder
    pics_folder_files = os.listdir(pics_folder_path)
    # this step filters out the folders in the directory -> only files
    pic_names = [x for x in pics_folder_files \
                 if os.path.isfile(os.path.join(pics_folder_path, x))]

    # Color fill value. Color of the area outside of the image visible
    # when rotated.
    # Quite important because performance of the bounding model varies alot
    # with this.
    fill = 0.81

    for pic_name in pic_names:
        pic_path = os.path.join(pics_folder_path, pic_name)
        image = Image.open(pic_path)

        # convert Image object to tensor
        to_tensor = transforms.ToTensor()
        image_tensor = to_tensor(image)

        # RGB-D image. D information lost when picture edited,
        # so it's removed here.
        image_tensor = image_tensor[0:3]

        # resize to make sure all pictures are the same size
        resize = transforms.Resize([227,227])
        image_tensor = resize(image_tensor)

        for angle in range(0,360, 18):
            # rotate the image by angle; fill mkaes the outside white
            # the results of the bounding box are much better when outside
            # is grey instead of black!
            image_tensor_rot = transforms.functional.rotate(image_tensor,
                                                            angle, fill=fill)

            pic_name_rot = pic_name.split(".")[0] + "_rotby_" + str(angle) + \
                                                                        ".png"

            if box_model_path != None:
                # add a dimension which is supposed to be the batch size.
                # this is necessary, so the model works
                image_tensor_rot = image_tensor_rot[None, :, :, :]

                # compute the bounding box
                bbox = model(image_tensor_rot.to(device))[0]
                bbox = bbox.cpu().detach().numpy()

                # remove dimension for further use
                image_tensor_rot = image_tensor_rot[0]

                # add a pad to the bounding box for more information
                x_index = round(bbox[0]) - pad
                y_index = round(bbox[1]) - pad
                w_index = round(bbox[2]) + 2 * pad
                h_index = round(bbox[3]) + 2 * pad
                x_thumb = round(bbox[4]) - pad
                y_thumb = round(bbox[5]) - pad
                w_thumb = round(bbox[6]) + 2 * pad
                h_thumb = round(bbox[7]) + 2 * pad

                # make sure the bounding box isn't outside the image
                # index
                if x_index < 0: x_index = 0
                if y_index < 0: y_index = 0
                if w_index + x_index > 227: w_index = 227 - x_index
                if h_index + y_index > 227: h_index = 227 - y_index
                # thumb
                if x_thumb < 0: x_thumb = 0
                if y_thumb < 0: y_thumb = 0
                if w_thumb + x_thumb > 227: w_thumb = 227 - x_thumb
                if h_thumb + y_thumb > 227: h_thumb = 227 - y_thumb

                bbox = (x_index, y_index, w_index, h_index,
                        x_thumb, y_thumb, w_thumb, h_thumb)

                yield image_tensor_rot, pic_name_rot, bbox

            else:
                yield image_tensor_rot, pic_name_rot
```

Function for Saving the Base Images Simply Rotated

```
def color_simply_rotate(device, pics_folder_path, test):
    """
    Saves the orginal images rotated 20 times.
    If test is True the pictures will be saved in the pictures_test folder.
    """
    for image_tensor_rot, pic_name_rot in \
        get_pictures_and_bounding_box(device, pics_folder_path):

        # transform tensor to image and save the picture
        to_pil = transforms.ToPILImage()
        image = to_pil(image_tensor_rot)

        if test: folder_name = "pictures_test"
        else: folder_name = "pictures_training"

        image_path_to_save = os.path.join(os.getcwd(), "tactile_data",
                                          folder_name, pic_name_rot)

        image.save(image_path_to_save)
```

Function for Calculating the Padding Needed to Make the Images Square

```python
def get_pad(img_shape):
    """
    Gives the needed padding to make the image square.
    """
    # if height greater than width
    if img_shape[1] > img_shape[2]:
        # [left, top, right, bottom]

        diff = 227 - img_shape[2]

        if diff % 2 == 0:
            pad_left = int(diff/2)
            pad_right = int(diff/2)
        else:
            pad_left = int(diff/2) + 1
            pad_right = int(diff/2)

        pad_img = [pad_left, 0, pad_right, 0]

    # if height smaller than width
    elif img_shape[1] < img_shape[2]:
        # [left, top, right, bottom]

        diff = 227 - img_shape[1]

        if diff % 2 == 0:
            pad_top = int(diff/2)
            pad_bottom = int(diff/2)
        else:
            pad_top = int(diff/2) + 1
            pad_bottom = int(diff/2)

        pad_img = [0, pad_top, 0, pad_bottom]

    else:
        pad_img = [0,0,0,0]

    return pad_img
```

Function for Saving Colored Images which are Cropped Beyond the Fingers

```python
def color_crop_beyond_fingers(device, box_model_path, pics_folder_path, pad,
                              test):
    """
    Creates pictures in which everything beyond the bounding boxes is cropped
    and the space between the fingers is gray.
    The pictures will then be used to train the visual to tactile network.
    Out of one picture 10 will be generated by rotating it.
    The purpose is to make the training set larger.

    pad: The number of pixels added around the bounding box for more picture
    information.
    """
    fill = 0.81

    for image_tensor_rot, pic_name_rot, bbox in \
        get_pictures_and_bounding_box(device, pics_folder_path, pad=pad,
                                      box_model_path=box_model_path):

        x_index, y_index, w_index, h_index, \
            x_thumb, y_thumb, w_thumb, h_thumb = bbox

        # make everything ouside the bounding boxes grey:
        # the color was taken from the input images
        # and is supposed to be as similar to the background as possible
        for channel in image_tensor_rot:
            for y in range(227):
                for x in range(227):
                    if not (((y_index <= y <= y_index + h_index) and
                             (x_index <= x <= x_index + w_index)) or
                            ((y_thumb <= y <= y_thumb + h_thumb) and
                             (x_thumb <= x <= x_thumb + w_thumb))):
                        channel[y][x] = fill

        # crop the image, so the bounding box are the outer edges
        # first find the outer edges
        # if the index finger is more to the left than the thumb
        if x_index < x_thumb: crop_left = x_index
        else: crop_left = x_thumb

        # if the index is higher up than thumb
        if y_index < y_thumb: crop_top = y_index
        else: crop_top = y_thumb

        # if the right border of the index is more right than the border
        # of the thumb
        if x_index + w_index > x_thumb + w_thumb:
            crop_width = x_index + w_index - crop_left
        else:
            crop_width = x_thumb + w_thumb - crop_left

        # if the bottom border of the index is lower than the thumbs
        if y_index + h_index > y_thumb + h_thumb:
            crop_height = y_index + h_index - crop_top
        else:
            crop_height = y_thumb + h_thumb - crop_top

        # crop
        image_tensor_rot = transforms.functional.crop(image_tensor_rot,
          top=crop_top, left=crop_left, height=crop_height, width=crop_width)

        # resize
        # size must be smaller than max_size but it's important that
        # max_size is 227. So this is a little "hack".
        # this makes sure that the bigger side isn't bigger than 227
        # This is needed for the Pad step because there it is assumed
        # that the longer side is 227.
        # Only when the picture is already square the image will be 226x226 but
        # that is not a problem because the dataload resizes it 227x227 anyway
        resize = transforms.Resize(size=226, max_size=227)
        image_tensor_rot = resize(image_tensor_rot)

        # pad, so it's square
        img_shape = image_tensor_rot.shape
        pad_img = get_pad(img_shape)

        padding = transforms.Pad(padding=pad_img, fill=fill,
                                 padding_mode="constant")
        image_tensor_rot = padding(image_tensor_rot)

        # transform tensor to image and save the picture
        to_pil = transforms.ToPILImage()
        image = to_pil(image_tensor_rot)

        if test: folder_name = "pictures_test"
        else: folder_name = "pictures_training"

        image_path_to_save = os.path.join(os.getcwd(), "tactile_data",
                                          folder_name, pic_name_rot)
        image.save(image_path_to_save)
```

Function for Saving Colored Images of One Finger

```
def color_one_finger(device, thumb, box_model_path, pics_folder_path, pad,
                     test):
    """
    Creates color images of the index finger only.
    """
    # approximately the color of the table the objects lie on
    fill = 0.81

    for image_tensor_rot, pic_name_rot, bbox in \
        get_pictures_and_bounding_box(device, pics_folder_path, pad=pad,
                                      box_model_path=box_model_path):

        x_index, y_index, w_index, h_index, x_thumb, y_thumb, w_thumb, h_thumb =
            bbox

        if thumb:
            image_tensor_rot = transforms.functional.crop(image_tensor_rot,
                    top=y_thumb, left=x_thumb, height=h_thumb, width=w_thumb)
        else:
            image_tensor_rot = transforms.functional.crop(image_tensor_rot,
                    top=y_index, left=x_index, height=h_index, width=w_index)

        # resize
        # size must be smaller than max_size but it's important that
        # max_size is 227.
        resize = transforms.Resize(size=226, max_size=227)
        image_tensor_rot = resize(image_tensor_rot)

        # pad, so it's square
        img_shape = image_tensor_rot.shape
        pad_img = get_pad(img_shape)

        padding = transforms.Pad(padding=pad_img, fill=fill,
                                 padding_mode="constant")
        image_tensor_rot = padding(image_tensor_rot)

        # transform tensor to image and save the picture
        to_pil = transforms.ToPILImage()
        image = to_pil(image_tensor_rot)

        if test: folder_name = "pictures_test"
        else: folder_name = "pictures_training"

        image_path_to_save = os.path.join(os.getcwd(), "tactile_data",
                                          folder_name, pic_name_rot)
        image.save(image_path_to_save)
```

Function that Returns the Edge Image of the Corresponding Color Image

```python
def get_edge_image(pic_name_rot, pics_folder_path):
    """
    This function returns the edge image of the corresponding color image.
    """
    fill = 0

    # extract the base pic name and the rotation angle
    pic_name = pic_name_rot.split(".")[0].split("_rotby")[0]
    rot_angle = int(pic_name_rot.split(".")[0].split("_")[-1])

    # open file with threshold values
    thresfile_path = os.path.join(os.getcwd(), "tactile_data", \
                                  "Canny_threshold.xlsx")
    wb = openpyxl.load_workbook(thresfile_path)
    sheet = wb.active
    # find the threshold value
    for row in sheet.iter_rows(values_only=True):
        if row[0] == pic_name:
            low_thres = int(row[1])
            high_thres = int(row[2])

    # open the image with opencv and get the edges
    pic_path = os.path.join(pics_folder_path, pic_name + ".png")
    img_edges = cv.imread(pic_path)
    img_edges = cv.Canny(img_edges,low_thres, high_thres)

    # transform to tensor
    to_tensor = transforms.ToTensor()
    image_tensor_edges = to_tensor(img_edges)

    # resize to make sure all pictures are the same size
    resize = transforms.Resize([227,227])
    image_tensor_edges = resize(image_tensor_edges)

    # rotate edge image
    image_tensor_edges_rot = transforms.functional.rotate(image_tensor_edges,
                                                          rot_angle, fill=fill)

    return image_tensor_edges_rot
```

Function for Saving the Base Edges Images Simply Rotated

```python
def edges_simply_rotate(device, pics_folder_path, test):
    """
    Saves the base edge images rotated 20 times.
    If test is True the pictures will be saved in the pictures_test folder.
    """
    for image_tensor_rot, pic_name_rot in \
        get_pictures_and_bounding_box(device, pics_folder_path):

        # get the edge image of image_tenso_rot as tensor
        image_tensor_edges_rot = get_edge_image(pic_name_rot=pic_name_rot,
                                                pics_folder_path=pics_folder_path)

        # transform tensor to image and save the picture
        to_pil = transforms.ToPILImage()
        image = to_pil(image_tensor_edges_rot)

        if test: folder_name = "pictures_test"
        else: folder_name = "pictures_training"

        image_path_to_save = os.path.join(os.getcwd(), "tactile_data",
                                          folder_name, pic_name_rot)

        image.save(image_path_to_save)
```

Function for Saving Edge Images which are Cropped Beyond the Fingers

```python
def color_crop_beyond_fingers(device, box_model_path, pics_folder_path, pad,
                              test):
    """
    Creates pictures in which everything beyond the bounding boxes is cropped
    and the space between the fingers is gray.
    The pictures will then be used to train the visual to tactile network.
    Out of one picture 10 will be generated by rotating it.
    The purpose is to make the training set larger.

    pad: The number of pixels added around the bounding box for more picture
    information.
    """
    fill = 0.81

    for image_tensor_rot, pic_name_rot, bbox in \
        get_pictures_and_bounding_box(device, pics_folder_path, pad=pad,
                                      box_model_path=box_model_path):

        x_index, y_index, w_index, h_index, \
            x_thumb, y_thumb, w_thumb, h_thumb = bbox

        # make everything ouside the bounding boxes grey:
        # the color was taken from the input images
        # and is supposed to be as similar to the background as possible
        for channel in image_tensor_rot:
            for y in range(227):
                for x in range(227):
                    if not (((y_index <= y <= y_index + h_index) and
                             (x_index <= x <= x_index + w_index)) or
                            ((y_thumb <= y <= y_thumb + h_thumb) and
                             (x_thumb <= x <= x_thumb + w_thumb))):
                        channel[y][x] = fill

        # crop the image, so the bounding box are the outer edges
        # first find the outer edges
        # if the index finger is more to the left than the thumb
        if x_index < x_thumb: crop_left = x_index
        else: crop_left = x_thumb

        # if the index is higher up than thumb
        if y_index < y_thumb: crop_top = y_index
        else: crop_top = y_thumb

        # if the right border of the index is more right than the border
        # of the thumb
        if x_index + w_index > x_thumb + w_thumb:
            crop_width = x_index + w_index - crop_left
        else:
            crop_width = x_thumb + w_thumb - crop_left

        # if the bottom border of the index is lower than the thumbs
        if y_index + h_index > y_thumb + h_thumb:
            crop_height = y_index + h_index - crop_top
        else:
            crop_height = y_thumb + h_thumb - crop_top

        # crop
        image_tensor_rot = transforms.functional.crop(image_tensor_rot,
          top=crop_top, left=crop_left, height=crop_height, width=crop_width)

        # resize
        # size must be smaller than max_size but it's important that
        # max_size is 227. So this is a little "hack".
        # this makes sure that the bigger side isn't bigger than 227
        # This is needed for the Pad step because there it is assumed
        # that the longer side is 227.
        # Only when the picture is already square the image will be 226x226 but
        # that is not a problem because the dataload resizes it 227x227 anyway
        resize = transforms.Resize(size=226, max_size=227)
        image_tensor_rot = resize(image_tensor_rot)

        # pad, so it's square
        img_shape = image_tensor_rot.shape
        pad_img = get_pad(img_shape)

        padding = transforms.Pad(padding=pad_img, fill=fill,
                                 padding_mode="constant")
        image_tensor_rot = padding(image_tensor_rot)

        # transform tensor to image and save the picture
        to_pil = transforms.ToPILImage()
        image = to_pil(image_tensor_rot)

        if test: folder_name = "pictures_test"
        else: folder_name = "pictures_training"

        image_path_to_save = os.path.join(os.getcwd(), "tactile_data",
                                          folder_name, pic_name_rot)
        image.save(image_path_to_save)
```

Function for Saving Edge Images of One Finger

```
def edges_crop_beyond_fingers(device, box_model_path, pics_folder_path,
                              pad, make_small, test):
    """
    Similar to pics_crop_beyond_fingers() but with edge images and the space
    between is black.
    """
    fill = 0

    for image_tensor_rot, pic_name_rot, bbox in \
        get_pictures_and_bounding_box(device, pics_folder_path, pad=pad,
                                      box_model_path=box_model_path):

        # get the edge image of image_tenso_rot as tensor
        image_tensor_edges_rot = get_edge_image(pic_name_rot=pic_name_rot,
                                        pics_folder_path=pics_folder_path)

        # crop
        x_index, y_index, w_index, h_index, \
            x_thumb, y_thumb, w_thumb, h_thumb = bbox

        # make everything ouside the bounding boxes black
        for channel in image_tensor_edges_rot:
            for y in range(227):
                for x in range(227):
                    if not (((y_index <= y <= y_index + h_index)
                         and (x_index <= x <= x_index + w_index))
                         or ((y_thumb <= y <= y_thumb + h_thumb)
                         and (x_thumb <= x <= x_thumb + w_thumb))):
                        channel[y][x] = fill

        # crop the image, so the bounding boxes are the outer edges
        # first find the outer edges
        # if the index finger is more to the left than the thumb
        if x_index < x_thumb: crop_left = x_index
        else: crop_left = x_thumb

        # if the index is higher up than thumb
        if y_index < y_thumb: crop_top = y_index
        else: crop_top = y_thumb

        # if the right border of the index is more right than
        # the border of the thumb
        if x_index + w_index > x_thumb + w_thumb:
            crop_width = x_index + w_index - crop_left
        else:
            crop_width = x_thumb + w_thumb - crop_left

        # if the bottom border of the index is lower than the thumbs
        if y_index + h_index > y_thumb + h_thumb:
            crop_height = y_index + h_index - crop_top
        else:
            crop_height = y_thumb + h_thumb - crop_top

        # crop
        image_tensor_edges_rot = transforms.functional.crop(
            image_tensor_edges_rot, top=crop_top, left=crop_left,
            height=crop_height, width=crop_width)

        resize = transforms.Resize(size=226, max_size=227)
        image_tensor_edges_rot = resize(image_tensor_edges_rot)

        # pad, so it's square
        img_shape = image_tensor_edges_rot.shape
        pad_img = get_pad(img_shape)

        padding = transforms.Pad(padding=pad_img, fill=fill,
                                 padding_mode="constant")
        image_tensor_edges_rot = padding(image_tensor_edges_rot)

        # transform tensor to image and save the picture
        to_pil = transforms.ToPILImage()
        image = to_pil(image_tensor_edges_rot)

        if test: folder_name = "pictures_test"
        else: folder_name = "pictures_training"

        image_path_to_save = os.path.join(os.getcwd(),
                                  "tactile_data", folder_name, pic_name_rot)
        image.save(image_path_to_save)
```

A.3.3 The Network: Architecture, Dataset, Training, Optimization & Evaluation

The code used to build the neural network, feed it with data, train and evaluate it. In this appendix only the code for the AlexNet architecture will be shown, since the principal is for all models the same and only the parameters change.

Code for Buliding the AlexNet Architecture

```
import torch.nn as nn
import torch.nn.functional as F

class AlexNet(nn.Module):
    def __init__(self, linear_width=4096, outputs=4, in_channels=3):
        super().__init__()
        "output_size = (pic − kernel + 2*pad)/stride + 1"
        self.conv1 = nn.Conv2d(in_channels=in_channels, out_channels=96,
                               kernel_size=11, stride=4, padding=0)

        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)

        self.conv2 = nn.Conv2d(in_channels=96, out_channels=256,
                               kernel_size=5, stride=1, padding=2)

        self.conv3 = nn.Conv2d(in_channels=256, out_channels=384,
                               kernel_size=3, stride=1, padding=1)

        self.conv4 = nn.Conv2d(in_channels=384, out_channels=384,
                               kernel_size=3, stride=1, padding=1)

        self.conv5 = nn.Conv2d(in_channels=384, out_channels=256,
                               kernel_size=3, stride=1, padding=1)

        self.fc1 = nn.Linear(in_features=9216, out_features=linear_width)
        self.fc2 = nn.Linear(in_features=linear_width, out_features=linear_width
            )
        self.fc3 = nn.Linear(in_features=linear_width, out_features=outputs)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.maxpool(x)
        x = F.relu(self.conv2(x))
        x = self.maxpool(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.relu(self.conv5(x))
        x = self.maxpool(x)
        x = x.reshape(x.shape[0], −1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Code of the Dataset Class

```python
from torch.utils.data import Dataset
import os
# installed Pillow instead of PIL due to error, works fine too
from PIL import Image
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from torchvision import transforms
import numpy as np
import openpyxl


class Dataset(Dataset):

    def __init__(self, size, test):

        self.size = size
        self.test = test

        if test:
            self.image_folder_name = "pictures_test"
            labels_path = os.path.join(os.getcwd(), "tactile_data",
                                       "labels_test.xlsx")
        else:
            self.image_folder_name = "pictures_training"
            labels_path = os.path.join(os.getcwd(), "tactile_data",
                                       "labels_training.xlsx")

        # open labels file
        wb = openpyxl.load_workbook(labels_path)
        sheet = wb.active

        self.data = []
        for row in sheet.iter_rows(min_row=2, values_only=True): # skip header
            self.data.append(row)


    def __len__(self):
        return len(self.data)


    def __getitem__(self, idx):

        pic_name = self.data[idx][0]
        image_path = os.path.join(os.getcwd(), "tactile_data",
                                  self.image_folder_name, pic_name + ".png")
        image = Image.open(image_path)

        # do transformations
        to_tensor = transforms.ToTensor()
        image_tensor = to_tensor(image)

        # just to be sure all the pictures are the same size
        resize = resize = transforms.Resize(self.size)
        image_tensor = resize(image_tensor)

        # label has to be returned as numpy array otherwise the dataloader
        # gives a list of tensors and not a tensor of lists
        label = [float(x) for x in self.data[idx][1:]]
        label = np.asarray(label)

        return image_tensor, label, pic_name
```

The Needed Packages for the Remaining Functions

```
from dataset_tactile import Dataset
import simple_conv
from torch.utils.data import DataLoader
from torch.utils.data.sampler import RandomSampler, SequentialSampler
import torch.nn as nn
import torch.optim as optim
import torch
import matplotlib.pyplot as plt
import numpy as np
import os
import openpyxl
import random
from PIL import Image
from PIL import ImageFile
from bayes_opt import BayesianOptimization
from bayes_opt.logger import JSONLogger
from bayes_opt.event import Events
from bayes_opt.util import load_logs
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

Function for Training a Model

```python
def train_model(device, batch_size, model, finger_cfg, pic_size, epochs,\
                lr=0.0001, l2=0.00001):
    """
    Train a given model. The model is an input of the function.
    Not the model name, the actual model object.
    """

    dataset = Dataset(size=pic_size, test=False)
    loader = DataLoader(dataset, batch_size=batch_size,
                        sampler=RandomSampler(dataset),
                        num_workers=6)  # num_workers in 1060=6, on 3090=20

    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=l2)

    losses = []
    for epoch in range(1,epochs+1):
        for i_batch, sample_batch in enumerate(loader):
            img_batch, label_batch, pic_name_batch = sample_batch
            img_batch = img_batch.to(device)
            label_batch = label_batch.to(device)

            # get the labels for the needed finger configuration
            if finger_cfg == "index":
                label_batch = label_batch[:,0:2]
            elif finger_cfg == "thumb":
                label_batch = label_batch[:,2:4]

            # compute model output
            optimizer.zero_grad()
            output = model(img_batch)

            # both output and label have to be the same data type
            # for the backward function to work -> .float()
            loss = criterion(output.float(), label_batch.float())
            loss.backward()
            optimizer.step()

        losses.append(loss.cpu().detach().numpy())
        print(loss.cpu().detach().numpy())

        save_model_as =  "model_" + finger_cfg + "_" + str(batch_size) + \
                        "_" + str(epoch) + ".pth"


        if epoch % 1 == 0:
            torch.save(model.state_dict(), save_model_as)
            test_model(device=device, model=model,
                       finger_cfg=finger_cfg,
                       pic_size=pic_size, save_info=True)
```

Function for Evaluating a Model

```
def test_model(device, model, finger_cfg, pic_size, save_info=False,\
               make_histo=False, print_info=True):
    """
    Tests a given model on the test set.
    Counts the number of predictions in the correct y direction and the
    number of predictions in hte cone.
    Computes the mean and the median for the errors and the inaccuracies.
    Returns the percentages.
    """

    data = Dataset(size=pic_size, test=True)
    loader = DataLoader(data, batch_size=1,
                        sampler=SequentialSampler(data),
                        num_workers=4)

    losses = [] # MSE losses over all epochs
    num_tests = 0
    num_right_y_dir = 0
    num_y_neg_dir = 0
    num_in_cone = 0
    rel_errors = []
    range_errors_to_mean = []
    range_errors_to_coneedge = []
    pic_name_last = ""

    for i, sample in enumerate(loader):
        img, lab, pic_name = sample
        img, lab = img.to(device), lab.to(device)

        # if the last image is the same one as this time skip, because that
        # would change the results
        if pic_name_last == pic_name:
            pic_name_last = pic_name
        else:
            pic_name_last = pic_name

            num_tests += 1

            # compute output
            output = model(img)

           # get the labels for the needed finger configuration
            if finger_cfg == "index":
                lab = lab[:,0:2]
                # the cone data in y
                label_cone, label_mean = make_data_cone(pic_name[0])[0:2]
            elif finger_cfg == "thumb":
                lab = lab[:,2:4]
                label_cone, label_mean = make_data_cone(pic_name[0])[2:4]
            else:
               # if both fingers are in the image still only use the index
               # finger for the evaluation since it is more meaningful than
               # the thumb
               lab = lab[:,0:2]
               label_cone, label_mean = make_data_cone(pic_name[0])[0:2]
```

```
                output = output[:, 0:2]

            # MSE losses
            criterion = nn.MSELoss()
            loss = criterion(output.float(), lab.float())
            losses.append(loss.cpu().detach().numpy())

            # check how many go in the right y direction
            # if the label has both negative and positive values then the
            # the prediction is correct in any way
            # if the product of label and output is bigger than 0 then
            # they have the same sign
            if ((min(label_cone[0]) * output[0][0]) > 0) \
            or ((max(label_cone[0]) * output[0][0]) > 0):
                num_right_y_dir += 1

            # check how many go in negative y
            if min(label_cone[0]) < 0:
                 num_y_neg_dir += 1

            # get data cone
            # !=0 because the make_data_cone function puts a 0 between every
            # every entry, so a cone will be plotted. They need, to be removed.
            y = [y for y in label_cone[0] if y != 0]
            z = -1

            # compute the outer edges of the cone and the slope of the model
            # prediction
            # compute the y range of the cone; so how inaccurate the data is
            out_ratio = output[0][0]/output[0][1]
            max_ratio = max(y) / z
            min_ratio = min(y) / z
            y_range = abs(max(y) - min(y))

            # the error of the prediction to the mean of the test data
            # relative to the width of the cone
            range_error_to_mean = abs(out_ratio*(-1) - label_mean)/y_range
            range_error_to_mean = range_error_to_mean.cpu().detach().numpy()
            range_errors_to_mean.append(range_error_to_mean)

            # check if output is in cone
            # if not compute the errors
            if max_ratio < out_ratio < min_ratio:
                num_in_cone += 1
                rel_error = 0
                range_error_to_coneedge = 0
            else:

                if out_ratio < max_ratio:
                    # error in multiples of the prediction
                    rel_error = (out_ratio-max_ratio)/out_ratio
                    rel_error = rel_error.cpu().detach().numpy()

                    # error in multiples of the data cone range
                    range_error_to_coneedge = \
                        abs(out_ratio*(-1) - max_ratio*(-1))/y_range
                    range_error_to_coneedge = \
                        range_error_to_coneedge.cpu().detach().numpy()

                if out_ratio > min_ratio:
                    # error in multiples of the prediction
                    rel_error = (min_ratio-out_ratio)/out_ratio
                    rel_error = rel_error.cpu().detach().numpy()

                    # error in multiples of the data cone range
                    range_error_to_coneedge = \
                        abs(out_ratio*(-1) - min_ratio*(-1))/y_range
                    range_error_to_coneedge = \
                        range_error_to_coneedge.cpu().detach().numpy()

            rel_errors.append(abs(rel_error))
            range_errors_to_coneedge.append(range_error_to_coneedge)

    rel_errors.sort()
    range_errors_to_coneedge.sort()
    range_errors_to_mean.sort()

    perc_num_y_neg_dir = num_y_neg_dir / num_tests
    perc_right_y_dir = num_right_y_dir / num_tests
    perc_num_in_cone = num_in_cone/num_tests
    # median
    median_rel_error_all = rel_errors[int(num_tests/2)]
    median_rel_error = rel_errors[int((num_tests-num_in_cone)/2+num_in_cone)]
    # mean error
    mean_rel_error_all = sum(rel_errors)/num_tests
    mean_rel_error = sum(rel_errors)/(num_tests-num_in_cone)
    # range errors
    mean_range_error_coneedge = sum(range_errors_to_coneedge)/num_tests
    mean_range_error_to_mean = sum(range_errors_to_mean)/num_tests
    median_range_error_coneedge = range_errors_to_coneedge[int(num_tests/2)]
    median_range_error_to_mean = range_errors_to_mean[int(num_tests/2)]

    # avg MSE loss
    avg_MSE_loss = sum(losses)/len(losses)

    if print_info:
        print(str(num_tests) + " tests in total.")
        print(str(round(perc_num_y_neg_dir*100, 1)) + \
              "%" + " in negative y direction.")
        print(str(round(perc_right_y_dir*100, 1)) + \
              "%" + " predicted in the right y direction.")
        print(str(round(perc_num_in_cone*100,1)) + \
              " % of the predictions are in the cone.")
        print("The median of the errors of the predictions not in the cone is "
           \
              + str(round(median_rel_error*100,1)) + "%.")
        print("The median of the errors is " + \
              str(round(median_rel_error_all*100,1)) + "%.")
        print("Predictions not in cone are, in average, off by " \
              + str(round(mean_rel_error*100, 1)) + "%")
        print("Overall error: " + str(round(mean_rel_error_all*100, 1)) + \
              "% (predictions in cone count as 0% error)." )
        print("The error to the MEAN of the cone measured in multiples of the"
               + " width of the cone is in average " +
```

```python
               str(round(mean_range_error_to_mean*100, 1)) + "%.")
        print("The median is " + str(round(median_range_error_to_mean*100, 1))
               + "%.")
        print("The error to the EDGE of the cone measured in multiples of the"
               + " width of the cone is in average "  + \
               str(round(mean_range_error_coneedge*100, 1)) + "%.")
        print("The median is " + str(round(median_range_error_coneedge*100, 3))
               + "%.")
        print("Average MSE loss: " + str(round(avg_MSE_loss, 4)))

    if save_info:

        file_path = os.path.join(os.getcwd(), "test_info_" \
        + finger_cfg + ".txt")

        with open(file_path,"a") as f:
            f.write(str(perc_num_y_neg_dir))
            f.write(";")
            f.write(str(perc_right_y_dir))
            f.write(";")
            f.write(str(perc_num_in_cone))
            f.write(";")
            f.write(str(median_rel_error_all))
            f.write(";")
            f.write(str(median_rel_error))
            f.write(";")
            f.write(str(mean_rel_error_all))
            f.write(";")
            f.write(str(mean_rel_error))
            f.write(";")
            f.write(str(mean_range_error_coneedge))
            f.write(";")
            f.write(str(mean_range_error_to_mean))
            f.write(";")
            f.write(str(avg_MSE_loss))
            f.write("\n")

    if make_histo:
        rel_errors = [x*100 for x in rel_errors]
        plt.ylabel("Frequency [-]")
        plt.xlabel("Error [%]")
        plt.xscale("log")
        plt.hist(rel_errors, [0.1,1,10,100,1000,10000,100000], color="black")
        plt.show()

    # this return is for the bayesian optimization purposes
    # (-median_rel_error_all) will be the value to be maximized
    return median_rel_error_all
```

Function that Makes the Label Cone and Computes the Label Mean

```
def make_data_cone(pic_name):
    """
    Makes a cone in which the measurements of the y- and z-direction lie.
    And gives the mean of the measurements.
    """
    labels_path = os.path.join(os.getcwd(), "tactile_data", "labels_base.xlsx")

    # pic name is with "rotby_xx". Not needed here.
    pic_name = pic_name.split("_rotby")[0]

    # read from file
    wb = openpyxl.load_workbook(labels_path)
    sheet = wb.active
    all_tac = []
    for row in sheet.iter_rows(min_row=1, values_only=True):
        if row[0] == pic_name:
            one_tac = row[1:]
            one_tac = [float(x) for x in one_tac]
            all_tac.append(one_tac)

    # find the edge ratios: ratio = F_y/F_z
    max_ratio_index = min_ratio_index = all_tac[0][0]/all_tac[0][1]
    max_ratio_thumb = min_ratio_thumb = all_tac[0][2]/all_tac[0][3]
    # and compute the mean of the measurements
    y_labels_index = []
    y_labels_thumb = []
    for one_tac in all_tac:
        # index
        ratio_index = one_tac[0]/one_tac[1]
        y_labels_index.append(ratio_index*(-1)) # *(-1) to compute y at z=-1
        if ratio_index < min_ratio_index:
            min_ratio_index = ratio_index
        if ratio_index > max_ratio_index:
            max_ratio_index = ratio_index
        # thumb
        ratio_thumb = one_tac[2]/one_tac[3]
        y_labels_thumb.append(ratio_thumb*(-1)) # *(-1) to compute y at z=-1
        if ratio_thumb < min_ratio_thumb:
            min_ratio_thumb = ratio_thumb
        if ratio_thumb > max_ratio_thumb:
            max_ratio_thumb = ratio_thumb

    # now compute the means
    y_index_mean = sum(y_labels_index)/len(y_labels_index)
    y_thumb_mean = sum(y_labels_thumb)/len(y_labels_thumb)

    # make a bunch of points in the cone at z=-1 with varying ys, so the plot
    # looks like a surface
    n = 100

    # index
    z_index = [-1]*n
    # put 0 between every entry, so a cone will be plotted
    z_index = \
        [item for items in zip(z_index, [0] * len(z_index)) for item in items]
    ratio_range_index = np.linspace(min_ratio_index, max_ratio_index, n)
    y_index = [y*(-1) for y in ratio_range_index]
    y_index = \
        [item for items in zip(y_index, [0] * len(y_index)) for item in items]

    #thumb
    z_thumb = [-1]*n
    # put 0 between every entry, so a cone will be plotted
    z_thumb = \
        [item for items in zip(z_thumb, [0] * len(z_thumb)) for item in items]
    ratio_range_thumb = np.linspace(min_ratio_thumb, max_ratio_thumb, n)
    y_thumb = [y*(-1) for y in ratio_range_thumb]
    # put 0 between every entry, so a cone will be plotted
    y_thumb = \
        [item for items in zip(y_thumb, [0] * len(y_thumb)) for item in items]

    return (y_index, z_index), y_index_mean, (y_thumb, z_thumb), y_thumb_mean
```

Function of the Training Process used for the Bayesian Optimization

```python
def train_model_for_opt(lr, l2, batch_size_opt):
    """
    Function used for optimizing the training process with the Bayesian
    Optimization algorithm.
    """
    device = "cuda:0"
    model = simple_conv.Simple_conv_4_6(in_channels=1, out_features=2).to(device
        )
    finger_cfg = "index"
    pic_size = 100
    epochs = int(round(epochs_opt))
    batch_size = int(round(batch_size_opt))


    dataset = Dataset(size=pic_size, test=False)
    loader = DataLoader(dataset, batch_size=batch_size,
                        sampler=RandomSampler(dataset),
                        num_workers=6)  # num_workers in 1060=6, on 3090=20

    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=l2)

    for epoch in range(1,epochs+1):
        losses = []
        for i_batch, sample_batch in enumerate(loader):
            img_batch, label_batch, pic_name_batch = sample_batch
            img_batch = img_batch.to(device)
            label_batch = label_batch.to(device)

            # get the labels for the needed finger configuration
            if finger_cfg == "index":
                label_batch = label_batch[:,0:2]
            elif finger_cfg == "thumb":
                label_batch = label_batch[:,2:4]

            # compute model output
            optimizer.zero_grad()
            output = model(img_batch)

            # both output and label have to be the same data type
            # for the backward function to work -> .float()
            loss = criterion(output.float(), label_batch.float())
            loss.backward()
            optimizer.step()

            losses.append(loss.cpu().detach().numpy())
        #print(np.mean(losses))

        eval_par = test_model(device=device, model=model,
                              finger_cfg=finger_cfg,
                              pic_size=pic_size, save_info=False,
                              print_info=False)

    return -eval_par
```

Function of the Training Process used for the Bayesian Optimization

```python
def tune_hyperparameters():
    """
    Tunes the learning rate, weight decay and the batch size.
    """
    pbounds = {
        "lr": (1e-5, 1e-3),
        "l2": (1e-7, 1e-5),
        "batch_size_opt": (2,6),
        }

    optimizer = BayesianOptimization(
       f = train_model_for_opt,
       pbounds = pbounds,
       verbose = 2)

    logger = JSONLogger(path="./bayes_opt_new.json")
    optimizer.subscribe(Events.OPTIMIZATION_STEP, logger)

    optimizer.maximize(init_points=10, n_iter=40)
```

A.3.4 The Bounding Box Network

The code used for the Bounding Box Network excluding the training function because it is very similar to the training function of the main algorithm.

The Code of the Dataset Class

```python
from torch.utils.data import Dataset
import os
# installed Pillow instead of PIL due to error, works fine too
from PIL import Image
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
from torchvision import transforms
import numpy as np
import matplotlib.pyplot as plt
import openpyxl



class Dataset(Dataset):
    def __init__(self):

        labels_path = os.path.join(os.getcwd(), "bounding_box_data",
                                   "labels_rotated.xlsx")

        # open labels file
        wb = openpyxl.load_workbook(labels_path)
        sheet = wb.active

        self.label_data = []
        for row in sheet.iter_rows(min_row=1, values_only=True):
            self.label_data.append(row)

    def __len__(self):
        return len(self.label_data)

    def __getitem__(self, idx):

        pic_path = os.path.join(os.getcwd(),"bounding_box_data",
                       "pictures_rotated", self.label_data[idx][0] + ".png")
        image = Image.open(pic_path)

        # do transformations
        to_tensor = transforms.ToTensor()
        image_tensor = to_tensor(image)
        # RGB-D image. D information lost when picture edited,
        # so it's removed here.
        image_tensor = image_tensor[0:3]
        # pictures should be that size anyway, just to be sure
        resize = transforms.Resize(227)
        image_tensor = resize(image_tensor)

        x_index, y_index, width_index, height_index = \
            int(self.label_data[idx][1]),\
            int(self.label_data[idx][2]),\
            int(self.label_data[idx][3]),\
            int(self.label_data[idx][4])

        x_thumb, y_thumb, width_thumb, height_thumb =\
            int(self.label_data[idx][5]),\
            int(self.label_data[idx][6]),\
            int(self.label_data[idx][7]),\
            int(self.label_data[idx][8])

        # label has to be returned as numpy array otherwise the dataloader
        # gives a list of tensors and not a tensor of lists
        label = np.asarray((x_index, y_index, width_index, height_index,
                            x_thumb, y_thumb, width_thumb, height_thumb))

        pic_name = pic_path.split("/")[-1].split(".")[0]

        return image_tensor, label, pic_name
```

The Code for the Data Augmentation

```
import openpyxl
import os
from PIL import Image


def rotate(bounding_box_path):
    """
    This function rotates all the pictures 3 times and calculates the new
    bounding boxesfor each new picture. This means out of 1 picture 3 more
    will be generated.
    The information will be automatically written into a different xlsx file.
    """

    # open workbook, create workbook object
    wb = openpyxl.load_workbook(os.path.join(bounding_box_path,
                                             "labels_base.xlsx"))
    # create sheet object
    sheet = wb.active
    # new workbook in which the new data will be saved
    wb_new = openpyxl.Workbook()
    sheet_new = wb_new.active

    # Note cells start from 1,not 0!
    # get number of rows
    max_row = sheet.max_row

    for i in range(2, max_row+1): # start from 2 to skip header
        a_i = "A" + str(i)
        i_i = "I" + str(i)
        # this is a tuple of shape [n_cell_objects,]; filled with cell objects
        row_data = sheet[a_i:i_i][0]
        old_row_data = (row_data[0].value, row_data[1].value,
                        row_data[2].value, row_data[3].value,
                        row_data[4].value, row_data[5].value,
                        row_data[6].value, row_data[7].value,
                        row_data[8].value)

        # open image
        root_img_path = os.path.join(bounding_box_path, "pictures_base",
                                     row_data[0].value + ".png")
        root_img = Image.open(root_img_path)

        # do the rotation 3 times.
        # The root image is always the last state of rotation.
        # root -> rotated1 -> rotated2 ...
        for j in range(4):
            # rotate image
            rotated_img = root_img.rotate(90)
            new_img_name = row_data[0].value + "_rotated_" + str(j)
            # save new image
            rotated_img.save(os.path.join(bounding_box_path,
                                  "pictures_rotated", new_img_name + ".png"))

            # calculate the new coordinates of the bounding boxes
            # index
            x_new_index = old_row_data[2] # = y_old
            # y_new = 227 - (x_old + w_old)
            y_new_index = 227 - (old_row_data[1] + old_row_data[3])
            w_new_index = old_row_data[4] # = h_old
            h_new_index = old_row_data[3]  # = w_old
            # thumb
            x_new_thumb = old_row_data[6]  # = y_old
            # y_new = 227 - (x_old + w_old)
            y_new_thumb = 227 - (old_row_data[5] + old_row_data[7])
            w_new_thumb = old_row_data[8]  # = h_old
            h_new_thumb = old_row_data[7]  # = w_old

            new_row_data = (new_img_name, x_new_index, y_new_index,
                            w_new_index, h_new_index,
                            x_new_thumb, y_new_thumb,
                            w_new_thumb, h_new_thumb)

            # write rotated data into file
            sheet_new.append(new_row_data)

            # make new to old for next iteration
            root_img = rotated_img
            old_row_data = new_row_data

    wb_new.save(os.path.join(bounding_box_path, "labels_rotated.xlsx"))
```