VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Hand Pose Estimation from RGBD Images

Odhad polohy ruky v RGBD obrazech

2020 Marek Šimoník

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student:	**Bc. Marek Šimoník**
Study Programme:	N2647 Information and Communication Technology
Study Branch:	2612T025 Computer Science and Technology
Title:	Hand Pose Estimation from RGBD Images Odhad polohy ruky v RGBD obrazech
The thesis language:	English

Description:

The aim of this thesis is to develop an application that will be able to detect hands in a RGBD image and estimate the 3D positions of the most dominant hand's joints. The application will display the estimated 3D joint positions overlaid on top of the input RGBD image.

Complete the following steps in your thesis:
1. Study the available approaches to hand pose estimation [1, 2, 3, 4].
2. Implement a solution that is able to estimate hand pose from a RGBD image based on the approaches.
3. Test performance of the implementation.
4. Create documentation of the implemented solution.

References:

[1] Adrian Spurr, Jie Song, Seonwook Park, Otmar Hilliges: Cross-modal Deep Variational Hand Pose Estimation. https://arxiv.org/abs/1803.11404
[2] Bardia Doosti: Hand Pose Estimation: A Survey. https://arxiv.org/abs/1903.01013
[3] Jonathan Taylor, Vladimir Tankovich, Danhang Tang, Cem Keskin, David Kim, Philip Davidson, Adarsh Kowdle, and Shahram Izadi: Articulated distance fields for ultra-fast tracking of hands interacting. http://doi.acm.org/10.1145/3130800.3130853
[4] Abhishake Kumar Bojja, Franziska Mueller, Sri Raghu Malireddi, Markus Oberweger, Vincent Lepetit, Christian Theobalt, Kwang Moo Yi, Andrea Tagliasacchi: Handseg: A dataset for hand segmentation from depth images. http://arxiv.org/abs/1711.05944

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Mgr. Ing. Michal Krumnikl, Ph.D.**

Date of issue: 01.09.2019

Date of submission: 30.04.2020

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA
Fakulta elektrotechniky a informatiky
1D16
OSTRAVA

Jan Platoš [illegible]

doc. Ing. Jan Platoš, Ph.D.
Head of Department

prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, May 12, 2020 [illegible]
. . . .

I hereby agree to the publishing of the master's thesis as per s. 26, ss. 9 of the Study and Examination Regulations for Master's Degree Programmes at VŠB – Technical University of Ostrava.

Ostrava, May 12, 2020 .... [illegible]

I would like to thank my supervisor Mgr. Ing. Michal Krumnikl, Ph.D. for consultations and for providing access to computational resources.

Abstrakt

Je překvapivé, že i s přibývajícím rozšířením mobilních aplikací využívající virtuální reality, uživatélé stále interagují s těmito aplikacemi pomocí dotyků obrazovky namísto ovládání virtuálního obsahu přímo pomocí rukou v prostoru. K vytvoření této úrovně interaktivity je zapotřebí stabilní a robustní algoritmus pro odhadování polohy ruky. Tato diplomová práce proto slouží jako studie možných přístupů k problému odhadu polohy ruky a sestává ze dvou částí; ze segmentace a z odhadu parametrů modelu ruky.

Klíčová slova: odhad polohy ruky, sémantická segmentace, hluboké učení, neuronové sítě, CNN, PSO

Abstract

It is surprising that even with increasing ubiquitousness of Augmented Reality applications on mobile devices, users nowadays still interact with said applications via on-screen controls, rather than controlling presented environment directly with their hands in real-world. To enable this degree of interactivity, a stable and robust hand pose estimation pipeline is needed. This thesis therefore serves as a study of a possible approach to hand pose estimation that consists of two parts; segmentation and estimation of hand model parameters.

Key Words: hand pose estimation, semantic segmentation, deep learning, neural networks, CNN, PSO

Contents

List of symbols and abbreviations

API	–	Application Programming Interface
AR	–	Augmented Reality
CNN	–	Convolutional Neural Network
CPU	–	Central Processing Unit
CUDA	–	Compute Unified Device Architecture
DoF	–	Degrees of Freedom
FCN	–	Fully Convolutional Network
GPU	–	Graphics Processing Unit
GT	–	Ground Truth
HPE	–	Hand Pose Estimation
MSE	–	Mean Squared Error
PEL	–	Permutation Invariant Layer
PSO	–	Particle Swarm Optimization
RGB	–	Red Green Blue
RGBD	–	Red Green Blue Depth
SDF	–	Signed Distance Field
SL	–	Structured Light
ToF	–	Time-of-Flight
VAE	–	Variational Auto Encoder
VR	–	Virtual Reality
mult-adds	–	multiply-add operations
px	–	Pixel

List of Figures

List of Tables

Listings

1 Introduction

Hand pose estimation has been chosen as the topic of this master thesis in order to explore feasibility of one of the possible ways of solving the problem on mobile devices. Accurate hand pose estimation in real time is a vital component of *Virtual Reality* (VR) and *Augmented Reality* (AR) systems. There are many approaches to solving this problem with different goals in mind; ultrasound/radar hand gesture recognition [9] for e.g. automotive industry [10], precise tracking for dataset creation via magnetic gloves [11] or gloves with stretchable capacitive sensors [12], and optical methods, to name some of them.

The optical variant of this task is a non-intrusive way of solving this problem, which is wanted for AR systems where wearing a glove or holding a controller is unacceptable, especially for casual usage. Optical systems can estimate hand pose using either RGB or depth images. In order to capture absolute hand pose in 3D space and aim for greater accuracy, depth image-based pose estimation is preferred and will be used in this thesis. Data capture was done using the TrueDepth camera of iPhone X, which is a *Structured Light* (SL) depth sensor.

Since the input depth images contain not only hands, but also background and other foreground objects (e.g. arms, faces, etc.), it is important to extract the portions of the input image that contain hands, such process is called *semantic segmentation*. After hands are extracted from the input image, estimation of hand joints can begin.

This thesis is divided into 4 parts; The first part explains how the semantic segmentation task has been implemented (Section 3). Output of this step is a heatmap that for each pixel estimates whether it belongs to a left hand, right hand or background.

In the second part, a more traditional computer vision approach to the *Hand Pose Estimation* (HPE) task is described (Section 4), where point cloud of the most dominant hand in the input image is being fitted to an *Signed Distance Field* (SDF) hand model via iterative optimization process.

Part three (Section 5) presents a brief overview of a few selected Machine Learning-based alternatives to the traditional computer vision HPE approach. The described techniques consist of deep learning architectures based on *Variational Auto Encoders* (VAEs), *Convolutional Neural Networks* (CNNs) and permutation-invariant networks with point cloud inputs.

The final part (Section 6) selects the most promising architecture from Section 5 — as neural network-based approach significantly surpassed the SDF-based approach (see Section 4) in precision — and proposes changes to its architecture that yield more than two times less parameters and floating point operations being used, while preserving the achieved *Mean Squared Error* (MSE) on the MSRA dataset [13].

Section 6 also presents an application which implements the whole HPE pipeline consisting of the semantic segmentation and subsequent Hand Pose Estimation using the proposed neural network architecture.

2 Related Work

Depth map-based hand pose estimation has been an active area of research and as such there are various techniques it can be solved with. In the research phase of this thesis, when I was learning about techniques that could be used for hand pose estimation, I tried Pose-REN [14], which guesses hand pose based on input depth image of a hand (the hand needs to be segmented and cropped from the depth image beforehand). It utilizes convolutional layers to learn features of individual fingers and then passes them into Fully Connected layers that regress the hand pose. Although the network did a good job regressing hand pose, I found it not to be fast enough for purposes of this thesis and I wanted to explore other possibilities with better potential for improvement.

A novel approach called Point-to-Pose Voting by Shile Li and Dongheui Lee [4] uses point clouds of a hand as the input. In order to ensure permutation-invariant feature computation, they generate features for each input point using global maximum operations in addition to the point's features. Even though the authors report very low error on well-known datasets while maintaining low runtime speeds, I could not replicate the results in my own implementation of the paper (authors did not provide their implementation and some parts of the paper were confusing for me), which is why I abandoned this approach.

A robust hand pose estimation method from depth images was presented by Kuo Du et. al. [5]. They use multi-task learning in their CNN architecture to predict hand pose from 96×96 px depth map hand crops. The authors made their Tensorflow implementation public, which allowed me to prove feasibility of this method and potential for improvement. This method therefore serves as the basis of the proposed architecture in Section 6.

Taylor et. al. [15] used a *Fully Convolutional Network* (FCN) to segment hands from depth image. After hands have been segmented, they used *Random Decision Forests* (RDF) to obtain initial guess of 6D hand pose (global position and rotation) and then used skinned tetrahedral mesh to warp *Signed Distance Field* (SDF) model of hand, generating an articulated hand model that they used to fit the segmented hands. I adopted the idea of using SDFs in Section 4, but instead of using them for warping tetrahedral meshes, I used them to directly compute loss metric based on distance between each point of a hand point cloud and the SDF model itself.

A more robust hand segmentation using FCNs (compared to [15]) was presented in *HandSeg* [16], that also uses the hourglass model, but instead of MaxPooling layers it uses strided convolution and skip connections, which are essential for achieving fast forward passes & maintaining high accuracy. Moreover, the authors also presented a framework for semantic segmentation dataset creation, which I used to produce my own dataset (refer to Section 3.2 for details). After implementing a customized version of the network, it was deemed suitable for usage in this thesis (in terms of model simplicity and runtime speed), which is why I did not research alternative solutions.

3 Hand Segmentation

The first step in hand pose estimation pipeline is identifying the parts of the input depth image I that contain hands — this process is called *semantic segmentation.* In this Section, the process of custom dataset creation will be described together with the architecture of neural network that is used for hand segmentation in this thesis.

There is a trade-off between speed and quality of segmentation; there are methods like Random Decision Forests, which are fast to run, but are not as precise in more complicated cases where multiple interacting hands are involved as neural networks are [15], which are sufficiently precise, but often not fast enough for real-time processing. Fortunately, neural networks architectures are often designed with scaling in mind, which allows for graceful degradation of neural network capacity and performance. Since this is also the case with the neural network architecture selected for the hand segmentation task in this thesis — HandSeg [16] — a scaled-down architecture of this network is being used in this thesis.

Both RDF and neural networks are supervised machine learning approaches and as such require a labeled dataset of training examples to be made in order for them to learn to segment objects. Each training example of such dataset comprises of a depth image I and segmentation map S pair. The segmentation map determines for each pixel of I whether it belongs to left hand, right hand or background.

The TrueDepth camera of Apple's iPhone X (a SL depth sensor) has been used to obtain depth images. Unfortunately, most publicly available RGBD hand datasets (e.g. [17, 13, 18]) were captured by *Time of Flight* (ToF) type of sensors that produce depth images with different kind of (systematic) noise compared to SL sensors and thus models learned on ToF dataset are unlikely to perform well on SL images. Because of that a new dataset has been created. The difference between SL and ToF noise in a real-world scene can be seen in Figure 1 and is further discussed in Section 3.1.

Throughout this thesis, only depth map pixels within the range $R_D = [0.15; 1.5]$ meters are considered valid, because depth measurements outside this range are significantly noisy. Given the nature of SL sensors, some pixels might not have depth value computed by the firmware for them (e.g. glass/metallic surfaces or pixels near depth discontinuity). Such pixels have `float` value of `NaN` and are also considered invalid. These invalid measurements occur mainly at the edge of objects and are present because the Structured Light sensor firmware cannot recreate depth with enough confidence due to discontinuity in captured Infrared (IR) dot pattern that the sensor emits onto scene.

3.1 Noise Difference between SL and ToF sensors

Noise statistics for a SL sensor (Kinect$^{\mathrm{SL}}$) and for a ToF sensor (Kinect$^{\mathrm{ToF}}$) can be seen in Figure 2. The noise of Kinect$^{\mathrm{SL}}$ is mainly caused by quantization; the IR pattern that is projected onto scene is captured with relatively small resolution (640×480 px in case of Apple's TrueDepth

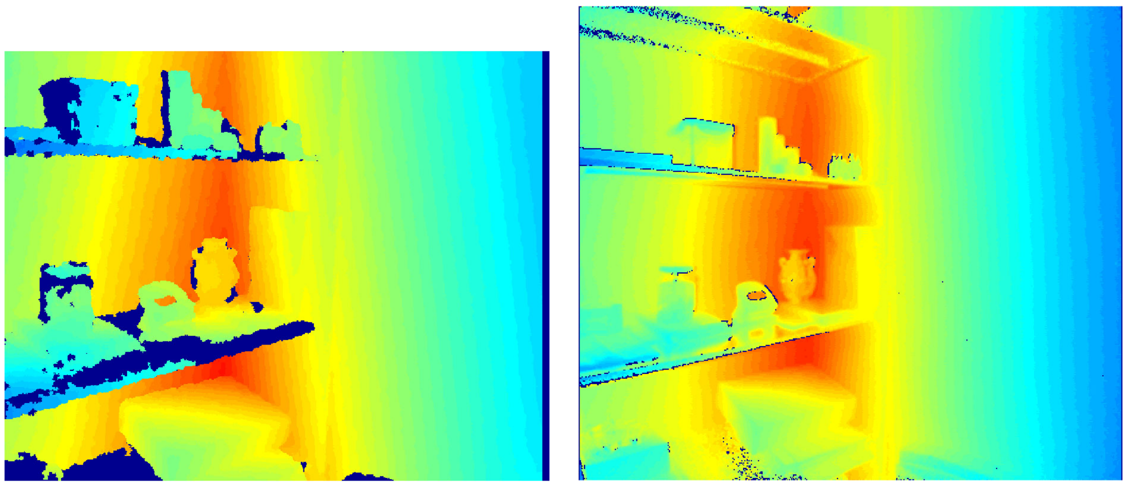

(a) Kinect 1 (Structured Light sensor) (b) Kinect 2 (Time of Flight sensor)

Figure 1: Comparison between a SL sensor and a ToF sensor [1]

camera), which causes ambiguity and shadow-like artifacts with invalid depth measurements when computing depth values. Kinect$^{\text{ToF}}$ computes distances by mixing optical signals with reference signals and its systematic error stems mainly form approximation of said signals [19]. For in-depth overview of differences between SL and ToF sensors, refer to [19].

3.2 Dataset Creation

As the most tedious part of a hand segmentation dataset creation is the labeling process, there were attempts to semi-automate it; for example in [20], synchronized RGBD video is being captured of a person performing hand gestures with their hands painted with different colours for the left and the right hand.

Exploiting the fact that the RGB and Depth components of the RGBD stream are overlaid, the color (RGB) information is used to automatically segment the painted hands, which produces a rough segmentation map that can be manually edited to correct labeling imperfections if needed, enabling semi-automatic dataset creation. The depth component of the RGBD stream is not affected by the presence of paint on the person's hands.

Painting hands is a relatively intrusive technique that was improved in [16] by replacing paint with coloured surgical gloves. As surgical gloves tightly fit the contours of a hand, depth maps are not noticeably affected.

A new dataset has been created using the iPhone X's TrueDepth Camera (SL sensor), following the process of [16]. Several 640 $\times$ 480 pixel RGBD videos were created, capturing a person wearing coloured surgical gloves while performing various hand gestures. The RGBD videos were then processed via a Python script into a dataset comprising of depth map I and segmentation map S pairs that could be used for training a machine learning model.

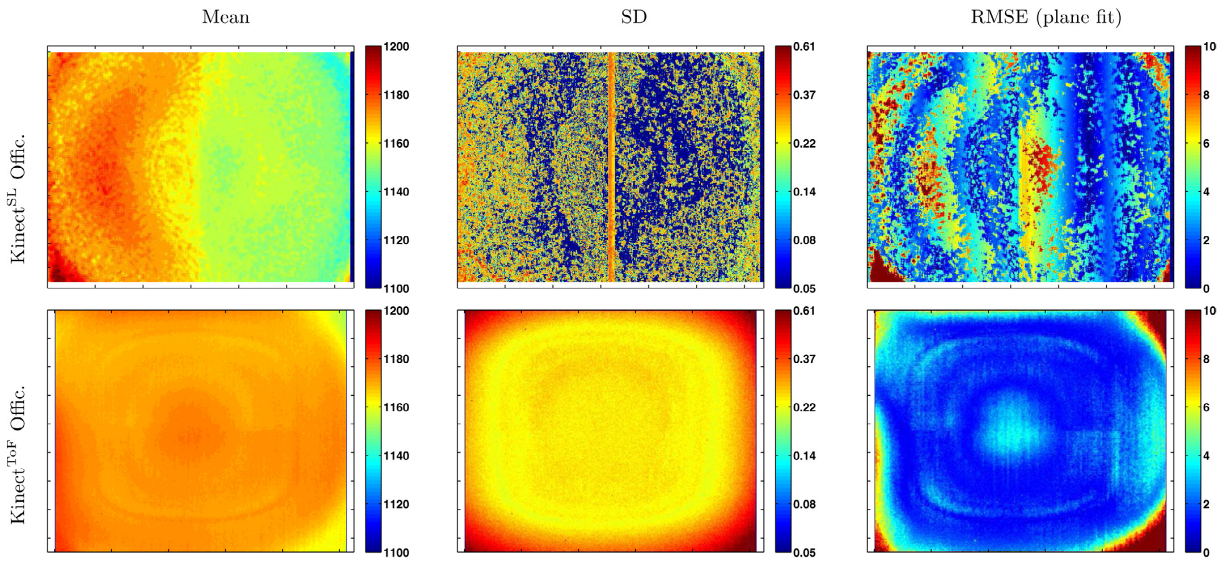


Figure 2: Comparison of SL and ToF depth measurements noise
Statistics of 200 frame for Kinect^{SL} and Kinect^{ToF} acquiring a planar wall (values in mm): Mean (left col.), standard deviation (middle col.) and RMSE with respect of a fitted plane (right) for the Kinect official drivers. [19]

The RGBD video is processed in the following way: all RGB frames J_{RGB} are converted into HSV colour space (to enable easier segmentation) producing HSV frame J_{HSV}. For the first few J_{HSV} frames of the video, both left and right hands are manually segmented (using OpenCV's implementation of GrabCut) in order to compute the range of HSV colors $R_L = [R_{L0}; R_{L1}]$ and $R_R = [R_{R0}; R_{R1}]$ for the left and right hand glove's colour respectively, where R_{L0}, R_{L1}, R_{R0} and R_{R1} are points in the HSV colour space.

Once both colour ranges are computed, the semi-automatic segmentation can begin; a J_{HSV} frame is thresholded into two binary masks M_L, M_R by using the R_L and R_R colour ranges respectively. Both masks are usually noisy with many false positives, therefore for each mask, median blur operator is applied followed by detection of connected components, keeping only the largest component. Next, both masks are combined into segmentation map $S \in \mathbb{L}^{W \times H}$, where $\mathbb{L} \in \{0, 1, 2\}$ and $0, 1, 2$ represent the background, left hand and right hand pixels respectively. Overlapping pixels from M_L and M_R are considered as background in S as well as pixels with depth value outside R_D or equal to `NaN`.

The automatically segmented map S is manually edited in case it is not sufficiently precise. Each frame is manually inspected and after (or rather, if) deemed valid, is passed into the dataset collection. The dataset consists of 33 079 samples. Example of one frame from the (unlabeled) dataset can be seen in Figure 3.

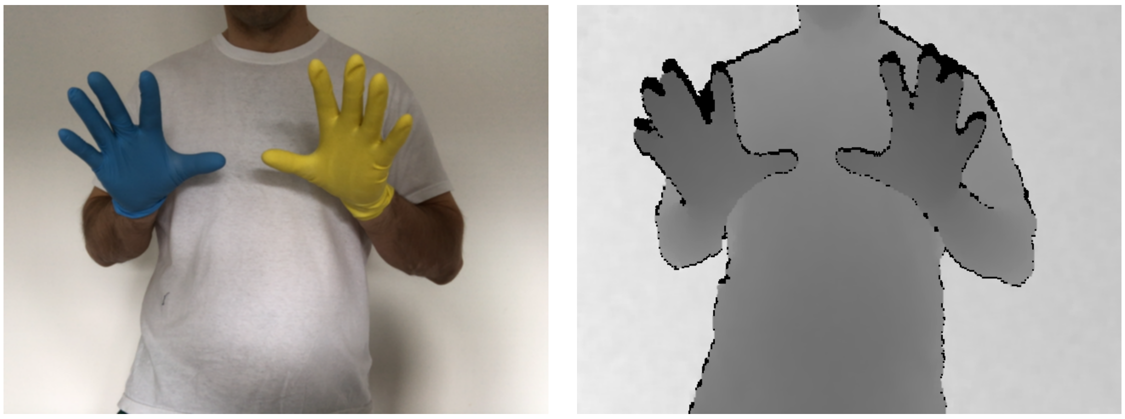

(a) RGB image (b) Depth image

Figure 3: A sample frame from unlabeled dataset (RGBD frame)

3.3 Fully Connected Neural Network Segmenter

The task of segmenting depth image I into segmentation map S is solved by the FCN architecture of [16] — it is a hourglass model based on the U-Net architecture [21].

Although the resolution of dataset images is 640×480 pixels, when processing depth images in real-time stream, the depth images are scaled down to 320×240 pixels in order to speed up the segmentation process. Therefore the network has also been learning with two times scaled-down dataset.

The architecture of the network can be seen in Figure 4. At the input, there is a post-processed 320×240 px `float` depth image. The post-processing is done per pixel in the following manner:

$$px_{\text{new}} = \begin{cases} \frac{px_{\text{old}} - R_{D0}}{R_{D1} - R_{D0}}, & \text{if } px_{\text{old}} \text{ within the range } R_D \\ -1, & \text{if } px_{\text{old}} \text{ outside the range } R_D \text{ or if } \texttt{NaN} \end{cases} \tag{1}$$

Where px_{old} is the original pixel value from the non-post-processed depth map, px_{new} is the post-processed value and $R_D = [R_{D0}; R_{D1}] = [0.15; 1.5]$ is the range of valid depth values in meters. In case the pixel value is within valid depth range R_D, the value is normalized into $[0; 1]$, else the pixel is assumed to be background and its value is transformed to -1.

The network can be also thought about as a kind of autoencoder (except that at the output is a segmentation map instead of the input image).

The encoder part of the network is composed of four blocks; each block contains three layers: 2-strided convolutional layer, batch normalization layer (except for the first block) and leaky ReLU layer. With each successive block the number of feature maps doubles and the resolution quarters (i.e. halves in both dimensions).

The decoder part is composed of three blocks and one transposed convolutional layer. Each block contains transposed convolutional layer (kernel size 3×3), batch normalization layer and

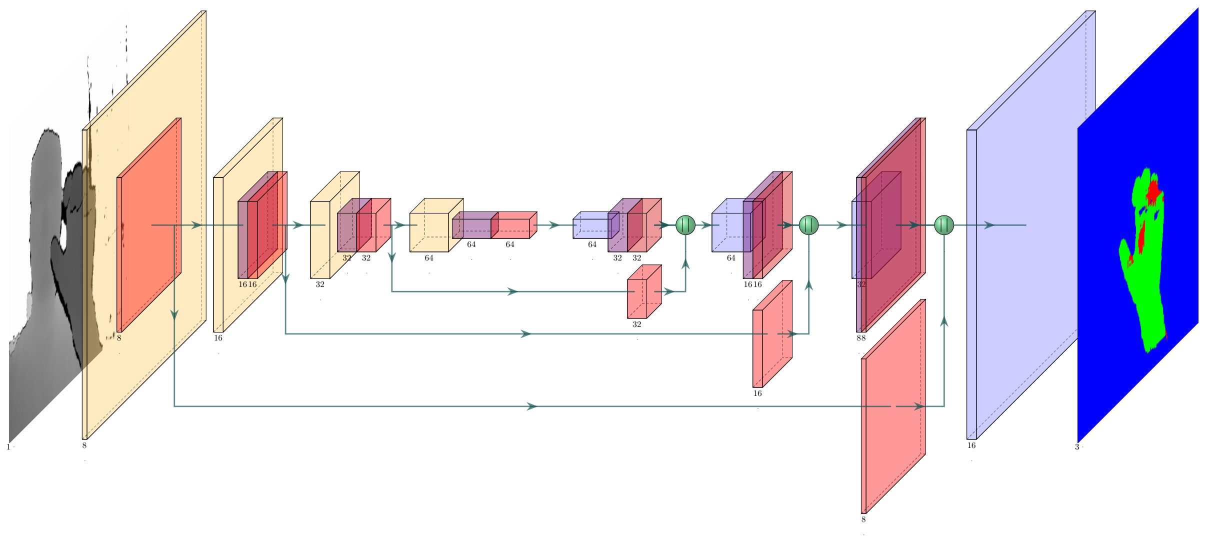


Figure 4: FCN Segmenter Architecture

ReLU layer. With each successive block, the resolution of image quadruples and the number of *output* features halves, however, due to *concatenation* of feature maps of the previous decoder block with corresponding output of encoder block, the number of *input* features of successive decoder block is double of the current one's output (please refer to Figure 4 for architecture diagram).

The output of the decoder is 3-channel image $\bar{S}_{l,i,j}$ where each channel l represents activations for different segmented class (background, left hand, right hand). Activations of any two channels are not mutually exclusive and can overlap. Therefore in order to produce the segmentation map S, it is needed to decide which label $(0, 1, 2)$ should be used for particular pixel. In this thesis, the final class of a pixel is determined as $S_{i,j} = \operatorname{argmax}_l \bar{S}_{l,i,j}$.

The raw segmentation map S usually contains isolated islands and discontinuities. To fix this, median blur is applied together with detection of connected components after which only the largest component is retained.

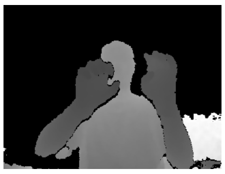
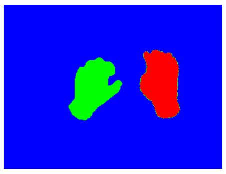
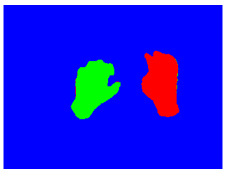
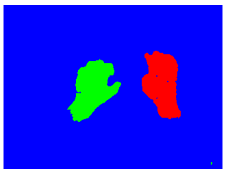
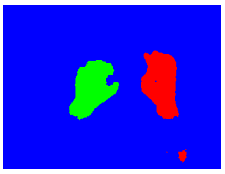
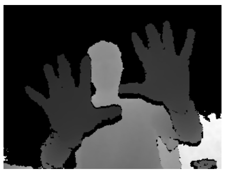
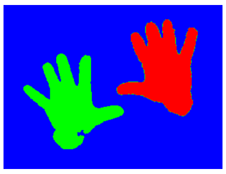
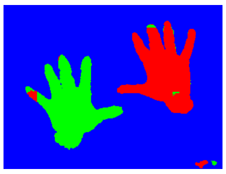
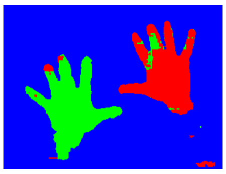
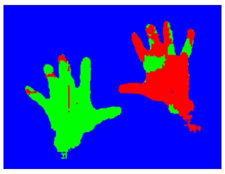
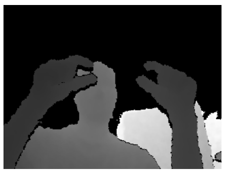
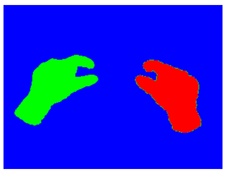
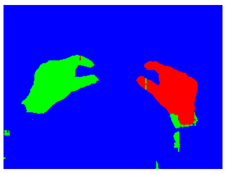
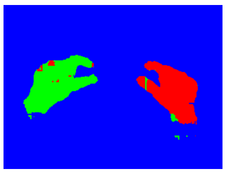
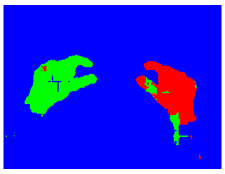
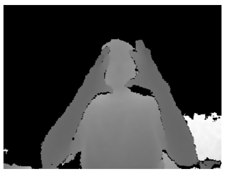
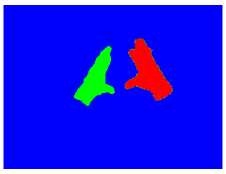
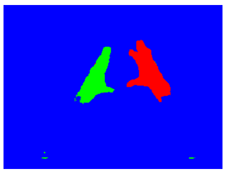
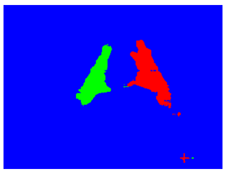
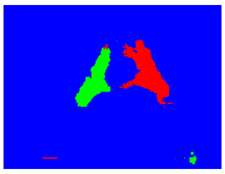
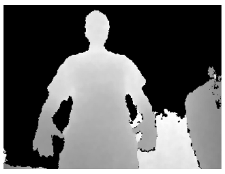
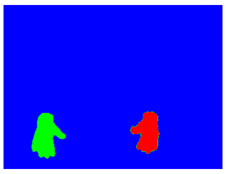
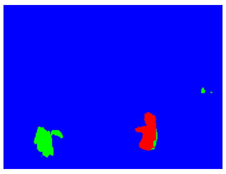
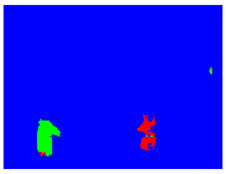
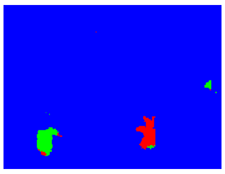
Input Image	Ground Truth	64 layers	16 layers	8 layers
				
				
				
				
				

Table 1: Comparison of different number of convolutional feature maps in encoder's first block (no post-processing applied)

Listing 1: HandSeg Network implementation in PyTorch

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

class UpConvolution(nn.Module):
  def __init__(self, in_depth, out_depth, kernel_size, stride=2):
    super(UpConvolution, self).__init__()
    self.deconv = nn.ConvTranspose2d(in_depth, out_depth, kernel_size, stride, padding=1)
    self.bn = nn.BatchNorm2d(out_depth)

  def forward(self, x, upper_layer_activation):
    x = self.deconv(x)
    x = F.relu(self.bn(x))
    x = torch.cat((upper_layer_activation, x), dim=1)  # concat upper layer activation
    return x

class HandSegNet(nn.Module):
  def __init__(self):
    super(HandSegNet, self).__init__()
    ncc = [8, 16, 32, 64]  # number of convolution channels

    # the encoder
    self.conv_1 = nn.Conv2d(1, ncc[0], kernel_size=3, stride=2, padding=1)
    self.conv_2 = nn.Conv2d(ncc[0], ncc[1], kernel_size=3, stride=2, padding=1)
    self.bn_2   = nn.BatchNorm2d(ncc[1])
    self.conv_3 = nn.Conv2d(ncc[1], ncc[2], kernel_size=3, stride=2, padding=1)
    self.bn_3   = nn.BatchNorm2d(ncc[2])
    self.conv_4 = nn.Conv2d(ncc[2], ncc[3], kernel_size=3, stride=2, padding=1)
    self.bn_4   = nn.BatchNorm2d(ncc[3])

    # the decoder
    self.up_1 = UpConvolution(ncc[3]*1, ncc[2], 4)
    self.up_2 = UpConvolution(ncc[2]*2, ncc[1], 4)
    self.up_3 = UpConvolution(ncc[1]*2, ncc[0], 4)
    self.out  = nn.ConvTranspose2d(ncc[0]*2, 3, kernel_size=4, stride=2, padding=1)

  def forward(self, x):
    # the encoder part with saved activations
    x1 = F.leaky_relu(self.conv_1(x))
    x  = self.bn_2(self.conv_2(x1))
    x2 = F.leaky_relu(x)
    x  = self.bn_3(self.conv_3(x2))
    x3 = F.leaky_relu(x)
    x  = F.relu(self.bn_4(self.conv_4(x3)))

    # the decoder part, using the saved activations
    x = self.up_1(x, x3)
    x = self.up_2(x, x2)
    x = self.up_3(x, x1)
    x = self.out(x)

    return x
```

The final version of the network uses 8 feature maps in the first block of the encoder. This number was selected empirically after comparing the run times of different architectures on iPhone X; the only the 8 feature maps version was able to run in real-time on this device, therefore larger architectures were discarded from consideration. The forward pass of the network with 8 input feature maps took approximately 25 ms on iPhone X. Tests on iPad Pro 2018 with A12X SoC proved the running times improved roughly ten times. This speedup can be accounted to utilization of a special coprocessor called Neural Engine designated for AI acceleration. Comparison of segmentation quality between various number of first block's feature maps can be seen in Table 1.

3.3.1 Implementation and Training

The network was implemented in Python using the PyTorch framework. The implementation can be seen in Listing 1. Following [16], Adam optimizer with the following parameters was used: $\text{lr} = 0.0002$, $\beta_1 = 0.5, \beta_2 = 0.999$. The network was trained with batch size 32.

For each batch, a randomly selected transformations for data augmentation purposes were used for every training sample; random rotations (in the range of $[-45; 45]$ degrees), random translations (in the ranges of $[-20; 52]$ percent of image width and $[-20; 20]$ percent of image height) and horizontal flips (with corresponding change of labels).

3.3.2 Intuition Behind HandSeg

The network's ability to segment the input depth map and distinguish between the left and the right hand is possible thanks to downsampling (in the X (width) and Y (height) dimension) of convolutional feature maps; although the size of the convolutional kernels is 3×3 pixels in all layers, a 3×3 pixel window located in layer n can be thought of as "seeing" 6×6 pixel window from the previous layer $n-1$. This is due to downsampling by factor of 2 (via 2-strided convolution) — a 1×1 pixel in layer n is equivalent of 2×2 pixels in layer $n-1$.

In the deeper layers of the network, a 3×3 pixel region "covers" significant portion of the original input image, which helps the network to decide how to label a particular pixel according to its surroundings. Decision whether a pixel belongs to the right hand can be made based e.g. on the orientation of forearm, which cannot be determined using a simple 3×3 convolution on the input depth image.

3.4 Performance

In order to increase the speed of segmentation and to save the per-frame compute time for pose estimation, the network variant with the least amount of convolutional feature maps (8) is being used. Although the segmentation quality is not optimal, it is sufficient for purposes of this thesis. When segmenting a 320×240 px depth image, the network variant with 8 feature maps took approx. 22 ms for forward pass on the iPhone X (using GPU). However, when testing the same network on late 2018 iPad Pro 10.5”, the speed has increased approx. nine times to 2-3 ms. This speedup was possible thanks to Apple’s Neural Engine coprocessor which allows to execute CoreML models.

For segmentation quality assessment, a testing dataset consisting of 3 309 samples has been used. The assessment consists of precision, recall and F1 score computation (for each pair of predicted and *Ground Truth* (GT) segmentation map, confusion matrix is computed based on per-pixel values; confusion matrices of all samples are accumulated into global confusion matrix from which the metrics are computed):

$$\begin{aligned}\text{Precision} &= \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}\\ \text{Recall} &= \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}\\ \text{F1} &= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}\end{aligned}$$

Graphs of these metrics for the 4 variants of the neural network (8, 16, 32 or 64 convolutional filters in the first layer of the CNN) can be seen in Figures 5, 6 and 7. Each Figure contains two subfigures — one for segmentation quality of each hand.

The lower values of those metrics can be attributed to the fact that the network has been trained on a small dataset (33 079 samples) with not enough variety and not challenging environment. Another factor are imperfectly aligned segmentation labels.

To increase performance of hand segmentation, it might be worthwhile to perform quantization of the neural network; quantization usually allows for significant performance increase with the trade-off in form of worse precision. Alternatively, the efficient DeepLabV3+ [22] network’s architecture could be used instead. The problem with these efficient architectures is that their building blocks (e.g. point-wise convolution) are not yet implemented efficiently by some of the deep learning frameworks and platforms, which makes their improvement only theoretical so far.

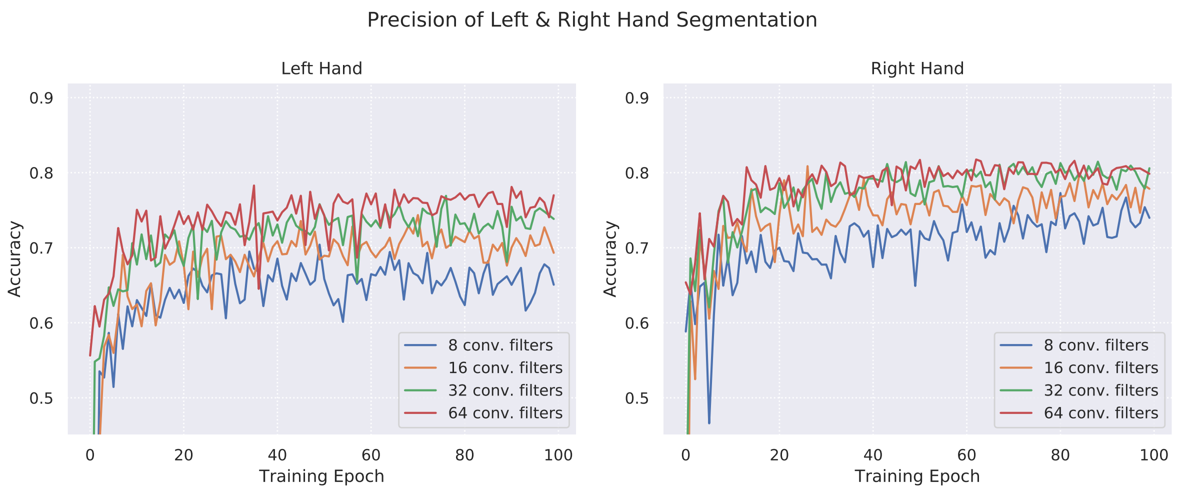


Figure 5: Precision of segmentation computed over the 3 309-sample testing dataset.

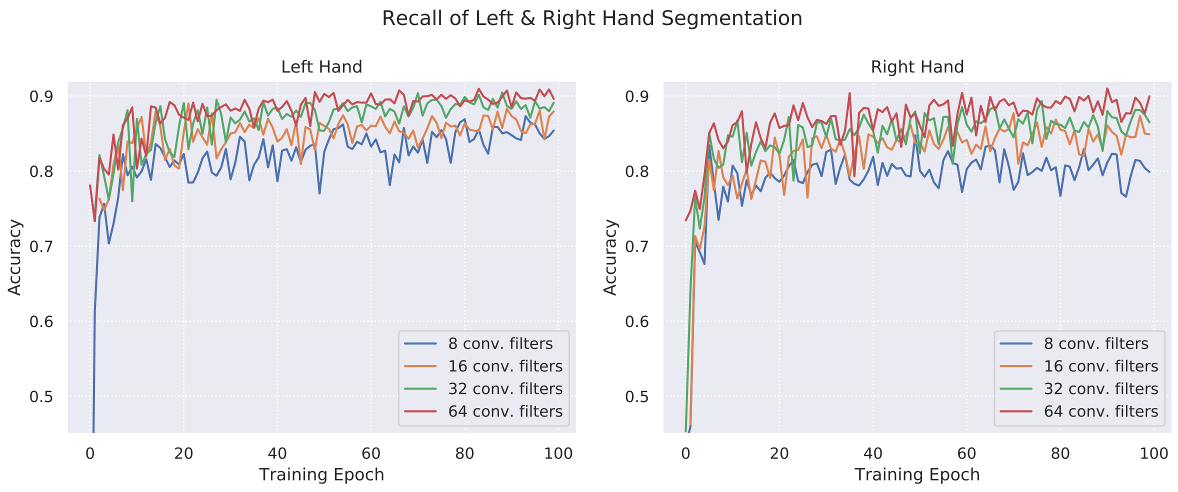


Figure 6: Recall of segmentation computed over the 3 309-sample testing dataset.

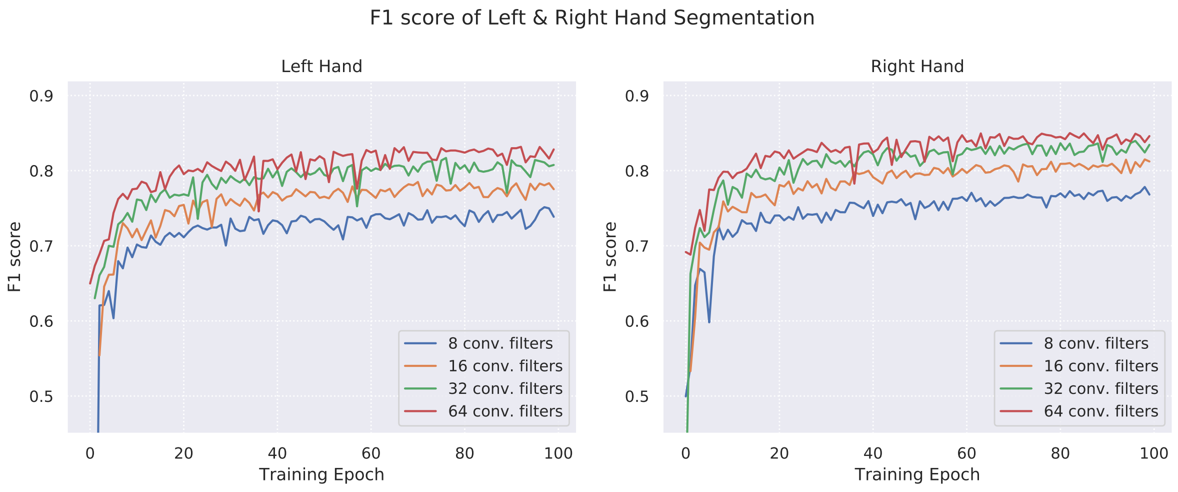


Figure 7: F1 score of segmentation computed over the 3 309-sample testing dataset.

4 Non-Machine Learning Hand Pose Estimation

After obtaining segmented hand (as described in the previous Section), we can proceed with estimating its pose. In this Section, it will be described how the traditional computer vision pipeline of Hand Pose Estimation was implemented. First, the process of point cloud extraction from cropped depth map of hand will be shown. Then we will see how the SDF hand model was created and how it is used in for iterative *Particle Swarm Optimization* (PSO) that achieves fitting of the point cloud to the SDF model.

The traditional — non-Machine Learning based — Hand Pose Estimation part of this thesis comprises of a C++ application that utilizes the trained HandSeg network (more details in Section 3) to segment image and then estimates the hand model's parameters. Together the two tasks form a hand pose estimation pipeline as depicted in Figure 8. The C++ application takes a dataset of depth maps as the input and performs offline optimization of hand parameters for each of the frames.

Once the HandSeg CNN has been trained, the second part of Hand Pose Estimation could begin — obtaining 26 parameters of SDF hand model based on point cloud of segmented hand. To simplify the optimization process, it is assumed that only single hand is in the input depth image — if there are more hands detected, only the one that occupies the largest amount of pixels (without considering the pixels' depth, because the larges hand is usually the closest to the camera due to perspective projection) is considered for pose optimization.

As can be seen from Figure 8, the main loop of pose estimation starts with segmentation of hands, followed by selecting a single hand, projecting the points of the hand into a point cloud and finally fitting the point cloud to the SDF hand model to obtain parameters.

In the next sections follows description of individual stages of the C++ application.

4.1 Hand Segmentation and Hand Point Cloud Generation

In the beginning of the pipeline, an input image I of size 320×240 px is fed forward through the HandSeg network, producing segmentation map S. The segmentation map is post-processed by detecting and keeping only the largest connected component of non-background pixels, keeping only what is likely to be the most dominant hand in the input image. The segmentation mask is then used to select hand-pixels from depth map. Those pixels (containing depth measurement) are then projected into 3D camera-space, forming a point cloud that can be further used to optimize parameters of the SDF hand model (more details in Section 4.1.2).

The hand segmentation map is noisy (due to low number of convolutional feature maps) and therefore not all pixels of a segmented hand are labeled correctly (see e.g. the segmentation map in second row and the last column of Table 1). Chirality of the hand is decided based on the number of left-hand and right-hand labels. Because the SDF model depicts *left* hand and does not allow specifying chirality, the point cloud needs to be flipped accordingly if needed.

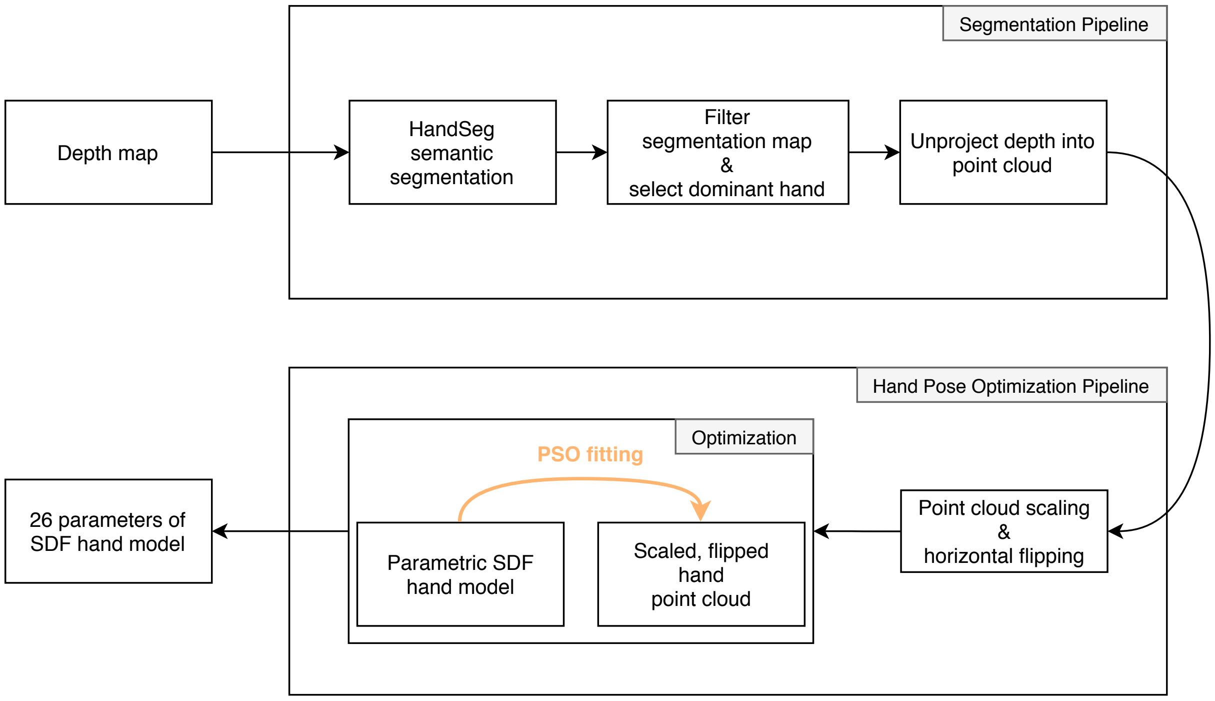

Segmentation Pipeline
Depth map
HandSeg semantic segmentation
Filter segmentation map & select dominant hand
Unproject depth into point cloud
Hand Pose Optimization Pipeline
Optimization
PSO fitting
26 parameters of SDF hand model
Parametric SDF hand model
Scaled, flipped hand point cloud
Point cloud scaling & horizontal flipping

Figure 8: Pipeline of the C++ application

Scale of the model also cannot be specified[1], therefore the point cloud needs to be scaled to match the size of the model instead. Scale of the model is not included as one of the parameters of optimization function, but is rather estimated empirically for a particular hand.

4.1.1 Limiting the Number of Points

Point cloud of the hand was initially produced from the 320×240 px depth image, but the number of hand-pixels was between 9 000 to 10 000 in a typical frame of the testing dataset. Such amount of points caused slow evaluation of the penalty function and slow rendering; it is therefore beneficial to reduce the number of points in point cloud. Number of points can be limited by various techniques — e.g. by filtering the number of points based on occupancy of voxels in regular grid. To simplify point reduction, I downscaled the depth map by factor of $s = 4$ in both dimensions (also updated the intrinsic matrix K accordingly) and generated point cloud from this smaller depth map.

4.1.2 Projecting Depth Map Into Point Cloud

To obtain point cloud from segmented pixels of depth map $I \in \mathbb{R}^{W \times H}$ (where $H = 480$ and $W = 640$ in this thesis), the pixels need to be projected into camera-space via the 3×3 *inverse*

[1]The SDF function of the model uses non-linear functions to model certain parts of the hand (palm, finger phalanges, etc.) and linear downscaling of individual parts have not produced linearly downscaled model because the parts are "connected" via a smoothing function — more can be read in Section 4.3.

intrinsic matrix K^{-1} of depth camera. However, the matrix K is to be used only with the original $W \times H$ px depth map I captured via iPhone X's TrueDepth camera. To project points from a scaled down depth map $I^\alpha \in \mathbb{R}^{\alpha W \times \alpha H}$ (where α is the scaling factor), the intrinsic matrix K needs to be adjusted accordingly. In the equation 2 you can see the original matrix K for $W \times H$ px image and the adjusted matrix K_α to be used with scaled down depth map of size $\alpha W \times \alpha H$ px.

$$K = \begin{bmatrix} f_x & 0 & t_x \\ 0 & f_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \qquad K_\alpha = \begin{bmatrix} \alpha f_x & 0 & \alpha t_x \\ 0 & \alpha f_y & \alpha t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

where:

f_x : focal length (in pixels) for computation of the X axis coordinate
f_y : focal length (in pixels) for computation of the Y axis coordinate ($f_x = f_y$ iff camera produces square pixels)
t_x : the X coordinate of the optical center (in pixels)
t_y : the Y coordinate of the optical center (in pixels)

An intrinsic matrix K_α is used to project a 3D point from camera-space to 2D image space. To obtain 3D point in camera-space from a depth pixel, the opposite transformation needs to be performed and to do this, the inverted intrinsic matrix K_α^{-1} is to be used as can be seen from equation 3.

$$P_{i,j} = \begin{bmatrix} \alpha f_x & 0 & \alpha t_x \\ 0 & \alpha f_y & \alpha t_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} j \\ i \\ {I^\alpha}_{i,j} \end{bmatrix}, \tag{3}$$

where:

$0 \leq i <$ image height : vertical pixel coordinate (origin in the top-left corner)
$0 \leq j <$ image width : horizontal pixel coordinate (origin in the top-left corner)

4.2 Signed Distance Fields

The hand model [2] has been modeled using a combination of Signed Distance Field functions. Signed Distance Fields (SDF) are used to define/represent geometry in implicit manner as opposed to the classical explicit approach of polygonal geometry. A 3D Signed Distance Field is a function $s\colon \mathbb{R}^3 \mapsto \mathbb{R}$ that for an arbitrary point $p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}$ defines *the shortest* distance to the surface:

$$s(p) = \begin{cases} > 0, & \text{if } p \text{ is outside of every object} \\ 0, & \text{if } p \text{ is exactly on the surface of an object} \\ < 0, & \text{if } p \text{ is inside of an object} \end{cases} \tag{4}$$

For example, the SDF function for a ball with radius $r \in \mathbb{R}$ located at position $q \in \mathbb{R}^3$ could be written as $s(p) = \|p - q\|_2 - r$. Note that meanwhile polygonal model can express only surfaces, SDFs on the other hand allow to also express volume (due to the fact they are signed). SDFs are commonly used to express surfaces of implicit functions and complex shapes (such as fractals). They have many advantages; it is simple, for example, to compute normals in arbitrary point. In practice, normals are computed via central differences by the function $\hat{n}_s(p)$ (using a small delta step $\epsilon \in \mathbb{R}$):

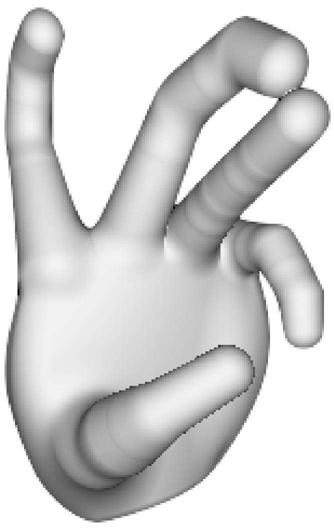
$$\begin{aligned} \epsilon_x &= \begin{bmatrix}\epsilon & 0 & 0\end{bmatrix}^T, \quad \epsilon_y = \begin{bmatrix}0 & \epsilon & 0\end{bmatrix}^T, \quad \epsilon_z = \begin{bmatrix}0 & 0 & \epsilon\end{bmatrix}^T \\ n_s(p) &= \frac{1}{2\epsilon}\begin{bmatrix} s(p+\epsilon_x) - s(p-\epsilon_x) \\ s(p+\epsilon_y) - s(p-\epsilon_y) \\ s(p+\epsilon_z) - s(p-\epsilon_z) \end{bmatrix} \\ \hat{n}_s(p) &= \frac{n_s(p)}{\|n_s(p)\|_2} \end{aligned} \tag{5}$$

The articulated hand model (see Figure 9) has been created as a combination of the following SDF functions that can be found in [23]: Ellipsoid (sdEllipsoid(P_i)), Cone (sdRoundCone(P_i)) and Box (rbox(P_i)). Combination of two SDF functions into a single one can be done by selecting the smaller function value, e.g. the expression $\min(\text{rbox}(P_i), \text{sdRoundCone}(P_i))$ combines two SDF functions — Box and Cone and computes the value of the combined function's value in point P_i.

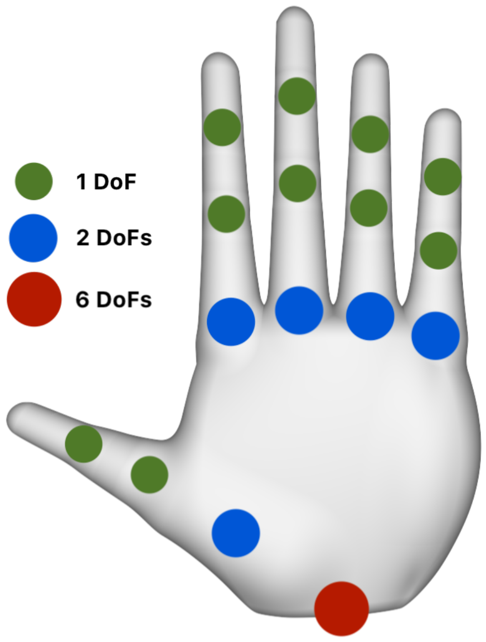
4.3 SDF Hand Model

The articulated hand model of left hand (see in Figure 9) used in this thesis — whose source code can be found in [2] — contains 26 *Degrees of Freedom* (DoFs). There are 3 DoFs for global hand position, 3 DoFs for global hand rotation and for each of the 5 fingers 4 DoFs for rotation in the three joints: 2 DoFs in the bottom phalanx and one DoF for both of the rest joints.

Every finger (including thumb) is made of three phalanxes that are represented by the SDF rounded cone function with varying diameters and lengths. The palm of the hand is made of a beveled box SDF function with applied nonlinear transformations (using trigonometric functions) that deform its shape to look more like a palm. All the partial SDF functions are merged into a single SDF function by performing a smooth $\min(\cdot,\cdot)$ over each sub-function.



(a) Model under a particular set of parameters



(b) Overview of degrees of freedom

Figure 9: Articulated SDF hand model [2]

Each finger's properties — length of phalanxes, their thickness and location — can be easily adjusted by changing model's geometry parameters. All of the 26 degrees of freedom has specified its limits, e.g. a finger cannot be bent 180° backwards. Self-intersection is not being checked.

The constants controlling the shape of the hand model were not determined so that the resulting hand shape would replicate a particular real-life hand; I guessed them by iteratively adjusting them until the result of their combination resembled a simplified, generic-looking hand [2].

4.4 Penalty Function

The core of the Hand Pose Optimization task is determining the parameters of hand model. The non-Machine Learning approach tries to estimate hand pose from point cloud of vertices projected from segmented depth map. A point cloud $L^{N \times 3} \in \mathbb{R}$ is a set of N vertices that represent the *surface* of a hand viewed under a particular angle. The goal of the penalty function $f(C, L) \mapsto \mathbb{R}$ is to produce a non-negative value that measures how well does a point cloud $L^{N \times 3}$ align with the articulated hand model under specific parameter configuration $C \in \mathbb{R}^{26}$.

The decision to use Signed Distance Fields as the way of expressing articulated hand model was motivated mainly by the fact that SDFs allow to implicitly compute the shortest distance from a point to the surface of a model, unlike with polygonal models, where such computation can be fairly involved.

Enforcing the requirement of close alignment of points to the surface of the SDF model, the penalty function can be expressed as the mean of absolute distances of point cloud vertices to the closest point on the model's surface;

$$f\left(C, L\right) = \frac{1}{N}\sum_{i=1}^{N} \left|\mathrm{sdHand}\left(C, L_i\right)\right| \tag{6}$$

where:

sdHand$(\cdot)$	:	the SDF function of the hand model found at [2]
L_i	:	i^{th} vertex of the hand's point cloud
C	:	vector of 26 hand parameters describing the SDF hand model pose

The closer the vertices are to the hand model, the smaller its value is going to be. In the beginning of the optimization process, user should ideally have their hand in the "default" position (open palm facing the camera, as in Figure 9b) for the algorithm to be able to begin tracking. Since there is not a robust approximate reinitialization of hand parameters performed before optimization begins, it is beneficial for quality of the tracking to assume certain starting conditions.

The penalty function used in this thesis comprises only of a single term — the mean error of point cloud-SDF hand model distances. To further improve quality of optimization, additional constraint terms such as self-intersection and time-wise hand pose smoothness penalties should be added.

4.5 Optimization Process

The penalty function f consists of multiple local minimums and is therefore difficult to find the global minimum (or a sufficiently close local minimum). Thanks to the relatively small size of point cloud, it would be feasible to approximate the gradient ∇f by central differences. However, given that f comprises of nested functions, also the analytical gradient could be computed. This means that methods like Levenberg-Marquardt algorithm could be used for non-linear least squares fitting.

For the sake of simplicity I decided to use evolutionary algorithms — in particular *Particle Swarm Optimization* (PSO), based on Taylor et. al. [24]. These algorithms do not need the gradient ∇f, but they rather only evaluate the penalty functions f itself. All parameters — except for the 3 DoFs for global hand position — are enforced to lie within the range $[0; 1]$.

For the first depth frame in the sequence of captured hand, the optimization starts by computing the mean position of the point cloud that is used as the initial guess of the hand position (3 DoFs), other parameters are left with the default values (for the resting pose of the hand that can be seen in Figure 9b). In the subsequent frames, the optimization algorithm uses hand parameters computed in the previous frame as one of its initial guesses.

Because only a small inter-frame movement of hand is assumed in this thesis, the hand parameters C^t in frame t should be close to the parameters C^{t-1} of the previous frame $t-1$. The initial guesses for frame t are therefore samples from uniform distribution whose mean are the parameters C^{t-1} and radius is a metaparameter of the algorithm. From empirical tests, the radius $r = 0.3$ has been found to work reasonably well for the 23 parameters (excluding 3-DoF global position, for which the radius is based on the scale of the hand).

Optimizing all 26 parameters at once with the default hand parameters proved to produce unstable, jittery hand configurations that were not smooth enough between frames (pose reinitialization to mean position of the point cloud was used too). One of the main factors behind jittery hand configurations seemed to be the fact that the hand model is thick (i.e. it is not a surface, but rather a volume). To mitigate this behavior, initialization of parameters from previously found values was used.

Second key factor of this problem was that the optimization was performed over all 26 parameters at once. To make the optimization more robust, I decided to optimize only selected parameters individually, therefore the following parameters are optimized sequentially:

1. global hand position (3 DoFs)
2. global hand rotation (3 DoFs)
3. thumb rotations (4 DoFs)
4. index finger rotations (4 DoFs)
5. middle finger rotations (4 DoFs)
6. ring finger rotations (4 DoFs)
7. little finger rotations (4 DoFs)
8. optimize all parameters at once (26 DoFs)

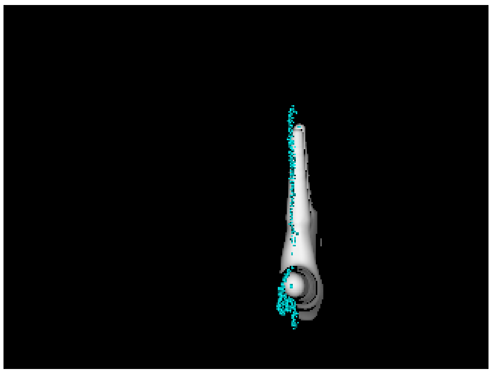
Wrapping-up the optimization process, in the beginning, the current parameters vector C^t is set to the values of the previous one C^{t-1}. Next follows partial optimization (optimizing e.g. only global hand position while having all others parameters "frozen") that sequentially optimizes all parameters mentioned in the list above.

Further experimentation with changing the search radius r for particular sets of optimized parts needs to be performed (e.g. there might be different preferable radii for hand position compared to search radii for angles of the index finger).

4.6 Implementation Details

To perform semantic segmentation, the C++ *Application Programming Interface* (API) of PyTorch is used to load the trained model (which was trained in Python). Computation of the segmentation map is performed on CPU (in the time of writing of this thesis, GPU acceleration was not available) and the raw segmentation image is post-processed using OpenCV to select the most dominant hand.

The application utilizes CUDA to implement rendering of the SDF hand model together with the point cloud projected from segmented hand, which are rendered into the same image (a CUDA kernel has been used) as can be seen in Figure 10. OpenMP has been used for trivial


(a) Side view

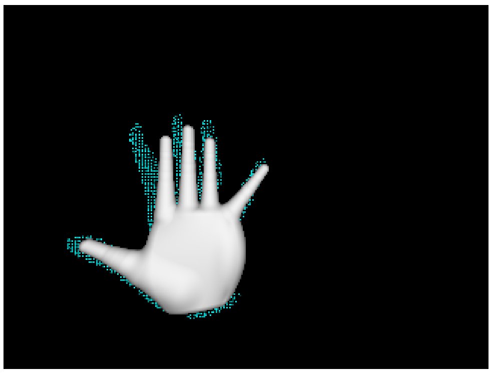

(b) Frontal view

Figure 10: Rendering of the SDF model fitted the point cloud of segmented hand model

Table 2: Mean errors between ground truth and predicted fingertips positions

Sequence name	flexex1	fingercount	random	adbadd
Mean error [cm]	8.25	10.50	16.62	13.66

parallelization of the PSO algorithm. As an alternative to PSO, the SOMA [25] algorithm was also tried. Both PSO and SOMA showed similar results, but in the end, the PSO algorithm was chosen for its simplicity.

4.7 Performance

Hand pose estimation using naive PSO implementation on CPU (parallelized with OpenMP) has proven not to be usable under real-time processing constraint. The speed varies greatly depending on the number of optimization steps and number of vertices in segmented-hand point cloud. As an example, when doing 20 PSO iterations with 10 particles for firstly for pose, then for rotation and subsequently 20 iterations with 15 particles for each individual finger, the pose optimization took approx. 170 ms to compute ($\approx$ 450 points in point cloud) on a computer with Intel i7-6700K CPU and 16 GB of RAM.

For testing purposes, the pose estimation algorithm has been tested on some sequences of the Dexter 1 dataset [26], where the metric is mean error in centimeters of distances from ground-truth to predicted fingertip positions. The results of evaluation can be seen in Table 2. Quality of optimization depends heavily on chosen PSO metaparameters.

5 A Brief Overview of Deep Learning-based Hand Pose Estimation

In order to improve upon the traditional HPE method (described in Section 4) in terms of performance and robustness, I decided to search for an alternative approach among Deep Learning-based architectures. Several candidates, which will be described below, were considered and the most promising one was selected as the base architecture for further architectural improvement, that will be described in Section 6. This section therefore serves as a brief study of the available Deep Learning architectures.

Classical Computer Vision approaches rely on explicitly designed algorithms which are based on observations made by Computer Vision researchers who tend to come up with solutions that are elegant, but not always the best performing, which is where Deep Learning helps to improve performance of Computer Vision models — by being able to find and utilize latent statistical properties of the input data. In order to find these latent properties, a sufficiently large and diverse set of example input data needs to be observed.

In case of depth-based Hand Pose Estimation, the input data can be e.g. in form of depth images of cropped hands and the task of a Deep Neural Network is then to estimate the correct 3D hand joints corresponding to the cropped image. There exist several public datasets suitable for the task of Hand Pose Estimation from depth data, but in this thesis, only the MSRA dataset [13] is used due to its diversity; it consists of 9 subjects performing 17 gestures each with up to 500 frames per gesture. Although the MSRA dataset was captured using a ToF sensor (Intel's Creative Interactive Gesture Camera), networks that were trained on this datasets performed well even on data from Structured Light sensor of the Apple's TrueDepth camera.

Deep Learning Hand Pose Estimation can be categorized into Generative, Discriminative and hybrid solutions. Generative architectures model distributions of hand poses, while Discriminative models regress them. The following subsections describe different types of models that were evaluated while searching for the most promising approach.

5.1 Generative approach to HPE

Generative approaches differ from discriminative ones by being able to sample new data from learned distribution. To test this way of HPE, I selected paper [3] by Adrian Spurr et. al. This paper exploits observation that a hand in a particular pose can be expressed in different modalities; the same hand configuration can be represented by an RGB image, depth map and 3D joints of the hand. The neural network encodes each of the modalities (RGB, depth, joints) into a shared (low-dimensional) embedding called latent space Z. The mapping is done using multi-encoder-decoder architecture where each modality has its own encoder and decoder, which creates many-to-many mapping (see Figure 11). This lets us encode one modality into the latent space and decode a different one. Let us say that a depth image I_D is encoded into a latent code

$z \in Z$ via the depth image encoder subnetwork. In order to obtain the corresponding hand joints J, we will need to use the joint decoder subnetwork which takes the latent code z as input and outputs $J \in \mathbb{R}^{Mx3}$, where M is the number of hand joints. We can similarly decode the same latent code z into RGB image representing the hand. Such RGB image is only hallucination of the network and it is based on the distribution that the net learned during the training process.

In an ordinary encoder-decoder architecture, the embedding space Z' does not have any regularization applied to it during training, which may result in the embedding space having arbitrary, discontinuous, structure that fits well only for the training dataset, but does not generalize well for novel samples. Such models do not allow smooth, continuous interpolation between two points ${z'}_1 \in Z'$ and ${z'}_2 \in Z'$ from the latent space Z'. To mitigate this problem, the Variational Auto Encoder framework is used. Its latent space Z is smoother compared to Z'; a VAE encoder does not encode its input (e.g. depth image I) into a single point in Z, but rather into infinite number of sample points that are represented by a normal distribution in the Z space (defined by mean and variance) — a VAE encoder therefore outputs two tensors for a given input (e.g. I); mean and variance. During the training process, it will be randomly sampled from this mean and variance, which will produce a different concrete latent code $\bar{z}_i \in Z$ for the same input I in each epoch i. This new $\bar{z}_i$ is then used as the input to an encoder (an encoder's input is single latent variable, not normal distribution inside the latent space Z). Using a different latent variable z_i in each epoch enforces smoothness of the latent space and therefore enables meaningful interpolation between two latent codes (the distributions of two different encoded inputs can overlap). The training loss of VAEs is then composed of two terms: the sum of traditional mean squared error and Kullback-Leibler divervence (KL divergence). By applying different weights to these two terms, we can control the trade-off between smoothness of the latent space Z and reconstruction quality. Refer to [3] for more details.

5.1.1 Experiments

In order to assess feasibility of VAEs, I tried to implement the method described by [3]. I had difficulties making the model learn; even trying to overfit it for a small number of samples (10) turned out to be a problem. Due to this failure, I chose not to pursue this method further.

5.2 HPE with Point Cloud Input

Depth images can be thought of as 2.5D representations of captured objects and may not fully convey spatial features due to being compressed inside an image. Moreover, different depth cameras can produce different images of the same scene due to having varying intrinsic parameters. To overcome this issue and enable to extract 3D features that hopefully represent the captured hand in more expressive way, point clouds can be used as the network's input. However, point clouds are *ordered* sets of vertices and given a 3D point cloud $P \in \mathbb{R}^{N \times 3}$, where N is the number of points, there are $N!$ ways in which the points can be ordered. In order to solve this problem

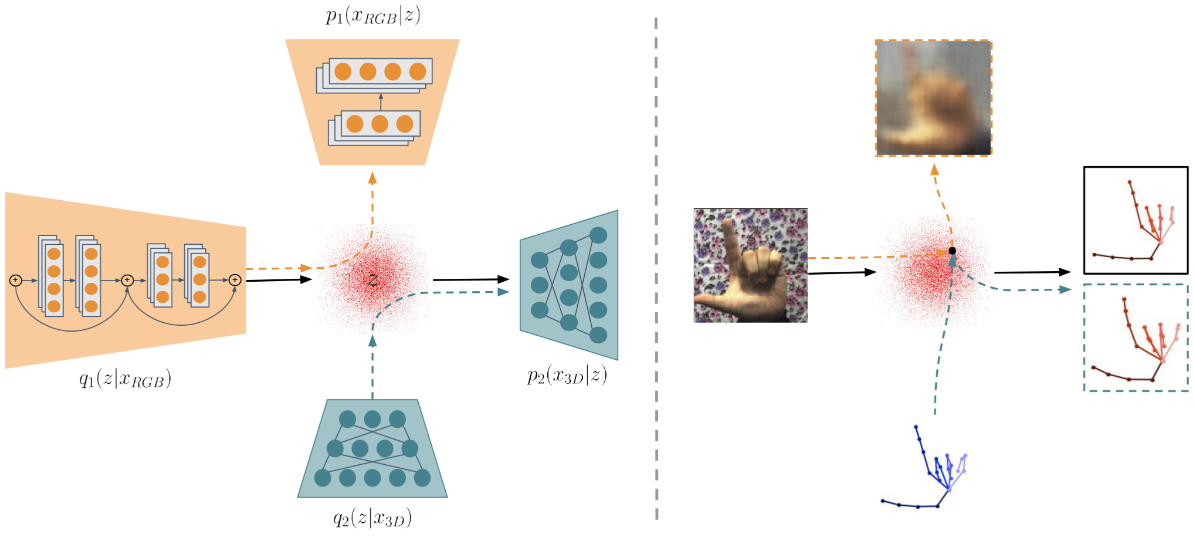


Figure 11: Cross-modal Deep Variational Hand Pose Estimation architecture [3].

and extract permutation-invariant features that can be later used for hand pose estimation, Shile Li and Dongheui Lee presented residual permutation-invariant blocks [4]. Overview of this architecture can be seen in Figure 12.

In this architecture, an input point cloud $P \in \mathbb{R}^{N \times 3}$ with arbitrary number of points[2] is passed through the feature extracting subnetwork consisting of Residual *Permutation Invariant Layer* (PEL) blocks. This subnetwork extracts 1024 features for each vertex of the point cloud and passes them into the regression subnetwork which regresses two matrices; pose estimates $\bar{j} \in \mathbb{R}^{N \times J}$ and importance terms (i.e. weighting matrix) $G \in \mathbb{R}^{N \times J}$, where N is the number of points and J is the number of joint parameters ($J = 21 \cdot 3 = 63$). Pose estimates $\bar{j}$ are then weighted by the the weighting matrix G, which results into $j \in \mathbb{R}^J$ In other words, each point $p \in P$ contributes to computation of the position of each regressed joint via the pose estimates $\bar{j}$ weighted by G (hence point-to-pose voting).

The basic building block of this architecture is the Permutation-Invariant Layer (PEL) and the authors of [4] describe it as:

$$x' = \sigma\left(\beta + \left(x I_{K_{\text{in}}} \Lambda + I_{K_{\text{in}}} x_{\max} \Gamma\right) W\right)$$

where:

[2]Authors tested 256, 512, 1024 and 2048 points and found that using point cloud of size $N = 1024$ yields the best results. Performance of the network gracefully degrades with decreasing number of points.

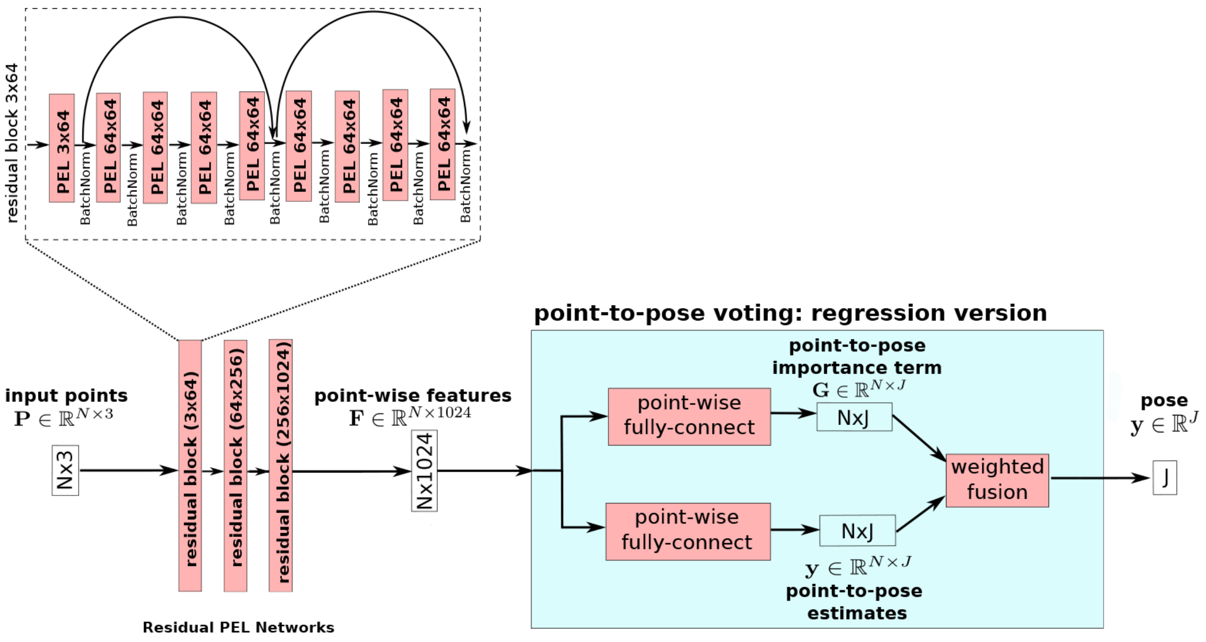

residual block 3x64
PEL 3x64
PEL 64x64
BatchNorm
point-to-pose voting: regression version
input points
$\mathbf{P} \in \mathbb{R}^{N\times 3}$
Nx3
residual block (3x64)
residual block (64x256)
residual block (256x1024)
point-wise features
$\mathbf{F} \in \mathbb{R}^{N\times 1024}$
Nx1024
Residual PEL Networks
point-wise fully-connect
point-to-pose importance term
$\mathbf{G} \in \mathbb{R}^{N\times J}$
NxJ
weighted fusion
$\mathbf{y} \in \mathbb{R}^{N\times J}$
point-to-pose estimates
pose
$\mathbf{y} \in \mathbb{R}^{J}$
J

Figure 12: Point-to-Pose Voting architecture [4].

$\sigma(\cdot)$	: activation function (Sigmoid)
$x \in \mathbb{R}^{N\times K_{\text{in}}}$	: layer input
$x' \in \mathbb{R}^{N\times K_{\text{out}}}$	: layer output
$\Lambda \in \mathbb{R}^{K_{\text{in}}}$	: weighting term for the computed point's feature
$\Gamma \in \mathbb{R}^{K_{\text{in}}}$	: weighting term for the global maximum
$x_{\max} \in \mathbb{R}^{K_{\text{in}}}$	: column-wise maximum of x
$W \in \mathbb{R}^{K_{\text{in}}\times K_{\text{out}}}$	: weighting matrix (serves as fully-connected layer)

Due to the weights W being shared with all points and usage of global maximum over the input points, the layer achieves permutation invariance so that arbitrarily ordered points of the same point cloud can still produce the same output.

5.2.1 Experiments

In order to try how this promising method works on real data, I implemented it PyTorch. Unfortunately, I was not able to understand details of some parts of the pipeline (e.g. how exactly the input data should be constructed and normalized) which is the reason I was unable to replicate the paper's results. Evaluation of the network variant with 256 input points, the error on testing part of the MSRA dataset was over 17 mm (current state-of-the-art methods accomplish error in the range $7-8$mm). Moreover, after trying the trained model on live RGBD stream, the results were not satisfactory. This can be attributed to different sensor type of the depth camera used while creating the MSRA dataset — on which the network was trained — and to type of depth camera the live test was performed with (Time of Flight vs. Structured Light);

difference in the type of noise and increased probability of incorrect normalization seemed to disable the architecture to generalize well on unknown inputs.

Although this architecture is elegant and very promising, my inability to replicate paper's results made me continue my search for the most ideal network design.

5.3 CNN-based HPE

Convolutional Neural Networks (*CNNs*) have been known to effectively extract features from images and CNN-based architectures are among the leading methods in computer vision tasks benchmarks. Apart from RGB images, CNNs have also been used for processing depth images in regression tasks; one of the latest works in this area is *CrossInfoNet* [5].

This discriminative model consists of 3 subnetworks; the Feature Extraction part, which serves as the backbone and uses 4 ResNet blocks [27] to extract general (common) features from the depth map, next there is the Feature Refinement section that splits into two branches to extract specialized features for the palm area and fingers area of the hand. Finally, there is the Regressor subnetwork that consists of a few fully-connected layers that combine the two branches into single one and regress the whole hand joints. Overview of the architecture can be seen in Figure 13.

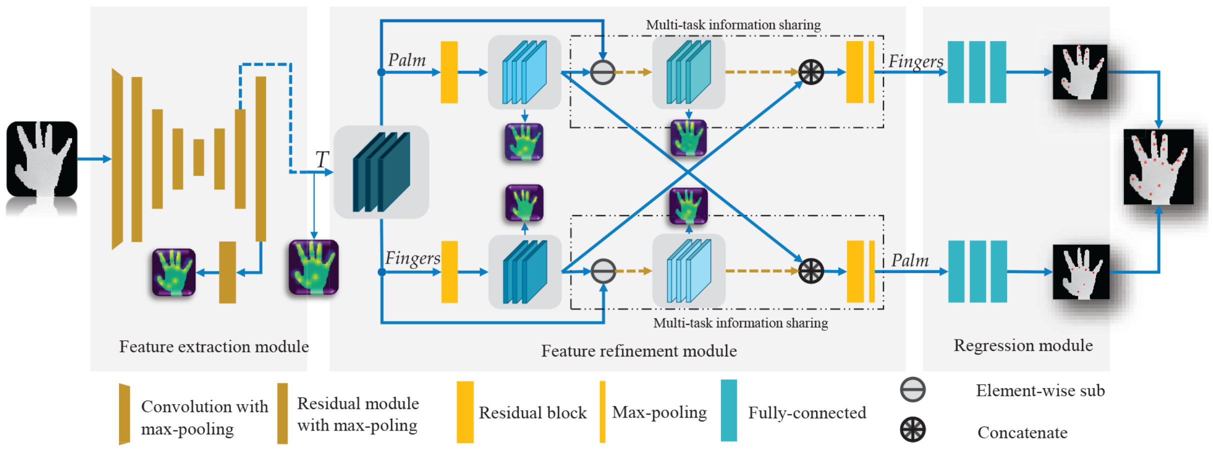

Multi-task information sharing
Palm
Fingers
T
Fingers
Palm
Multi-task information sharing
Feature extraction module
Feature refinement module
Regression module
Convolution with max-pooling
Residual module with max-poling
Residual block
Max-pooling
Fully-connected
Element-wise sub
Concatenate

Figure 13: CrossInfoNet architecture [5].

5.3.1 Feature Extraction subnetwork

Input to this network is a 96×96 px cropped depth image of a hand, normalized into the range $[-1; 1]$ with the hand's center of mass being in the origin (the origin being in the middle of the depth image). The Feature Extraction subnetwork passes 256 feature maps of size 6×6 (let us refer to them as $T \in \mathbb{R}^{256 \times 6 \times 6}$) into the Feature Refinement subnetwork. Moreover, the Feature Extraction module also contains auxiliary CNN block after computation of the T features, which outputs 2D heatmaps of all the regressed hand joints. These auxiliary heatmaps (one for each joint) are not utilized in further computation and are therefore used only during training as a constraints in the loss function to guide the network to learn more relevant features for the hand pose estimation task.

5.3.2 Feature Refinement subnetwork

The Feature Refinement module takes the common features T as input and creates two CNN branches (based on residual blocks) that compute specialized features F_P for the palm joints (6 joints for the MSRA dataset) and features F_F for the fingers joints (16 joints for the MSRA dataset). To help the network learn meaningful representations, authors use cross connection; after extracting F_P and F_F, these specialized features are then subtracted from the common features T, giving rise to $\bar{F_P} = T - F_F$ and $\bar{F_F} = T - F_P$. These complementary features are then concatenated with their corresponding non-complementary counterparts, making ${F_P}' = F_P \oplus \bar{F_P}$ and ${F_F}' = F_F \oplus \bar{F_F}$, where $\oplus$ denotes concatenation of features.

After ${F_P}'$ and ${F_F}'$ are computed, another CNN residual blocks are applied to both of these features. The resulting maps ${F_P}'$ and ${F_F}'$ are then passed to the Regressor subnetwork.

Before cross connection, auxiliary heatmaps of the palm and fingers are computed from F_P and F_F. Similar to the Feature Extraction part, these heatmaps are not used for computation, but serve solely to constrain the network to learn better features (by including heatmap loss while training). Computation of auxiliary heatmaps can be seen as multi-task learning; forcing the network to extract features that can be used for both joint regression and heatmaps computation helps the network to generalize better.

5.3.3 Regressor subnetwork

The Regressor module is the last subnetwork and it produces the final hand joints. It consists solely of fully-connected layers with ReLU activations and dropout. Firstly, there are two fully-connected layers for each of the palm and fingers branches. Then there is auxiliary fully-connected layer for both branches that regresses palm joints and fingers joint respectively. The size of the fully-connected layers is 1024.

Before the auxiliary layer, the palm and fingers features are concatenated into a single feature map of size 2048 — merging the palm and fingers branches into a single one — which is then fed into the last fully-connected layer that regresses the final hand joints.

5.3.4 Training

The loss function L is composed of 6 individual terms which can be divided into two types; heatmap loss and regression loss. The regression loss consists of 3 MSE (Mean Squared Error) terms; the palm joints term, the fingers joints term and the whole hand joints term. All of them are summed.

The heatmap loss also consists of summation of 3 MSE terms, the constrained elements are: the auxiliary whole hand heatmap from the Feature Extraction subnetwork and two auxiliary heatmaps from the Feature Refinement subnetwork; the palm heatmaps and the fingers heatmaps.

The loss function can be described as $L = 0.01 \cdot heatmap\_loss + regression\_loss$. During training, online data augmentation is used in form of random rotations, scaling and translation — all with corresponding adjustment of the hand joint coordinates.

Adam optimizer is used with learning rate 0.001 and weight decay 0.96. The network is trained for 110 epochs with batch size 128. The MSRA dataset [13] — on which the network was trained — consists of 9 subjects performing 17 gestures each. As the dataset is not divided into training and testing parts, the error is computed in leave-one-out style by averaging the results of 9 trained models (each of the 9 models uses 8 subjects for training and the remaining one for testing).

5.3.5 Experiments

As the authors of [5] provided Tensorflow implementation of their architecture, it was easy to test feasibility of the CrossInfoNet architecture. Due to it being implemented in Tensorflow and also to fully understand the architecture, input image normalization and data augmentation, I reimplemented the architecture into a more modern and well-structured framework; PyTorch.

During the reimplementation, I noticed there are slight discrepancies between the architecture described in the paper [5] and the actual code; the author's Tensorflow implementation of the Regressor subnetwork did not include as many dense layers. In my PyTorch reimplementation, the architecture was able to achieve 8.19 mm mean joint error on the MSRA dataset (error averaged over 9 models in leave-one-out style, like authors did) as opposed to the 7.86 mm error mentioned in the paper. *Note again that the paper described slightly different architecture than the one found in the implementation.*

After trying the trained model on live RGBD stream (Structured Light depth sensor), I found that the network was able to generalize exceptionally well and even though it was trained on the MSRA dataset (which was captured with a Time of Flight depth sensor), the predicted joint positions were convincingly accurate and stable.

The great generalization capability of this architecture coupled with its superior results (compared to networks I tried before) made me choose to try improving it.

6 Proposed Hand Pose Estimation Method

By researching some of the available Deep Learning methods used for solving the Hand Pose Estimation problem — as mentioned in Section 5 — and eventually finding the most promising one to be the CrossInfoNet network [5], I decided to try reducing its computational complexity and number of parameters by iteratively changing its architecture, which will be the topic of this Section. Improved architecture will be presented together with a demo application implementing the whole HPE pipeline.

In order to achieve this, I reimplemented the authors' Tensorflow implementation of [5] into PyTorch to establish a reference version, which all my experimental iterative changes to the architecture could be compared to. Note that the architecture in the Tensorflow implementation and the paper's [5] description of it differ; the paper describes more fully-connected layers in the Regression subnetwork, more auxiliary heatmap computations in the Feature Refinement subnetwork and does not mention presence of convolutional layers in skip connections inside the Feature Extraction subnetwork[3]. Refer to Section 5.3 for brief overview of the CrossInfoNet architecture.

I also noticed difference between the average joint error on the MSRA dataset reported in the paper (7.86mm) versus computed by my reimplementation (8.19mm). It is likely that these differences occurred due to the above mentioned architectural differences and also due to possible errors introduced while reimplementing into PyTorch.

Despite this performance difference, I decided to take the PyTorch reimplementation's 8.19mm mean joint error as the target error to retain with the proposed architecture, while reducing the number of parameters and multiply-add operations. Number of parameters of the original model is 23.937×10^6 and the number of multiply-add operations is 571.304×10^6.

The CrossInfoNet architecture uses ResNet [27] blocks throughout its architecture. ResNet networks were introduced in 2015 and are too computationally demanding to be run at above-real-time rates on mobile devices. The most prominent occurrence of ResNet blocks is in the Feature Extractor subnetwork of CrossInfoNet (which serves as the backbone); it uses 4 ResNet blocks to extract features of the input depth image, which is equivalent to the ResNet-50 variant[4].

In order to make a lightweight network, the number of *multiply-add operations* (mult-adds) needs to be reduced. By inspecting which sections of CrossInfoNet require the most mult-adds, we can see that it is the Feature Extraction subnetwork, as expected (349.007×10^6 mult-adds). Following is the Feature Refinement subnetwork with 201.132×10^6 mult-adds and lastly the Regression subnetwork with 21.165×10^6 mult-adds. It is therefore clear that by replacing the ResNet blocks with a more effective building block, we could achieve significant speedup.

[3]These convolutional layers are used to adjust number of feature maps to allow skip connections via addition.

[4]There are several variants of ResNet which differ in the number of used layers. The variant with least capacity is ResNet18. The most common number of layers used in ResNet architectures is 18, 34, 50, 101 and 152.

Such replacement candidate could be the latest backbone architecture used in mobile networks — MobileNetV3 [7].

The next two sections describe building blocks of the MobileNetV3 architecture (upon which the new Feature Extractor and Refiner subnetworks are based) together with the used activation functions. The later sections describe the final improved HPE network in detail with performance reports, used data augmentation and implementation details.

6.1 MobileNet

At the time of writing of this thesis, the state-of-the-art in mobile CNN backbones was the MobileNetV3 network [7]. This architecture has evolved throughout the last three years and so far three version have been published ([6], [28], [7]). Similarly to ResNet, the architecture is modular and allows to control the performance/accuracy trade-off by modifying number of used layers and building blocks.

In the following sections, the key ideas on which the MobileNet architectures are built are going to be described.

6.1.1 Depth-wise Separable Convolutions

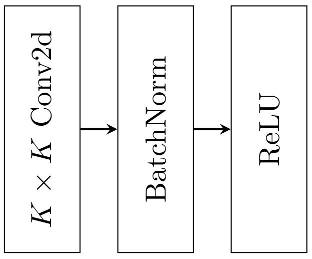
The first MobileNet paper [6] introduced architecture based on depth-wise separable convolutions. To better understand its impact, let us first review the way ordinary 2D convolutions work in the context of deep neural networks.

6.1.1.1 Ordinary Convolutions For the sake of simplicity, let us consider convolutions with stride 1 and padding $(K-1)/2$, where K is the convolutional kernel size (assuming square kernel). These convolutions take a tensor (multi-channel image) $T_{in} \in \mathbb{R}^{C_{in} \times H \times W}$ — where W is width, H is height and C_{in} is number of channels of the input image — as their input. Output of the convolution operation is a tensor $T_{out} \in \mathbb{R}^{C_{out} \times H \times W}$.

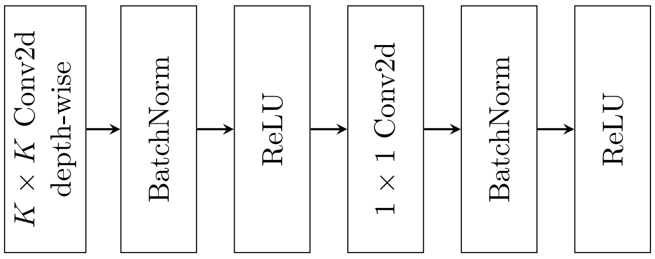
For the computation of T_{out}, C_{out} convolutional kernels $W_i \in \mathbb{R}^{C_{in} \times K \times K}$ are required. These kernels W_i slide in the X and Y axes of T_{in} and each kernel produces single channel of the output image T_{out}. Refer to Figure 14 for overview.

6.1.1.2 Depth-wise Separability To reduce the number of parameters and increase performance, we can replace single ordinary convolutional layer by two layers. In the first step, we apply depth-wise separable convolution with kernel size K. This is done by convolving each individual channel of the input image with kernel of size $K \times K$ (thus there is C_{in} $K \times K$ learnable kernels). Then nonlinearity is applied (e.g. ReLU). This step ensures that features are extracted in the spatial dimension.

Feature extraction in the channel dimension is ensured in the second step by ordinary 1×1 convolution on the spatially filtered features from the first step. A nonlinearity (e.g. ReLU)



(a) Ordinary convolution module



(b) Depth-wise separable convolution module

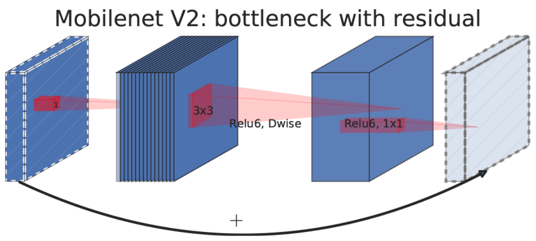
Figure 14: Comparison between a ordinary and depth-wise convolution modules [6].

is also applied after afterwards. The difference between ordinary convolution and depth-wise separable convolution modules can be seen in Figure 14.

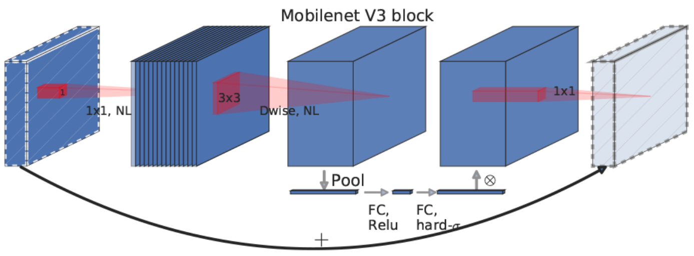
Depth-wise separable convolution modules dramatically reduce the number of parameters and mult-adds (8-9 times less computation) with only a small decrease in accuracy [6].

6.1.2 Inverted Bottleneck with Residual

The basic building block in the first MobileNet [6] architecture is the depth-wise convolutional module (see Figure 14b). The whole architecture is made of single ordinary convolution module (see Figure 14a), followed by 13 depth-wise convolutional modules and average pooling layer. As MobileNets are used for classification, the very last layer is a fully-connected one. However, the hand pose estimation task does not need to classify objects, but it rather utilizes MobileNet as a feature-extracting backbone, therefore the average pooling and fully-connected layers at the end are omitted.



(a) MobileNetV2 bottleneck block [7]



(b) MobileNetV3 bottleneck block [7]

Figure 15: Comparison between MobileNetV2's and MobileNetV3's bottleneck modules [7].

6.1.2.1 MobileNetV2 [28] introduced a more lightweight building block; the *Inverted Bottleneck with Residual.* The bottleneck module performs 1×1 ordinary convolution — which expands the number of channels (most often six times in MobileNetV2) — followed by batch normalization and ReLU activation. Then 3×3 depth-wise convolution followed by batch normalization and ReLU activation is applied. In the final step, another 1×1 convolution (with batch normalization) is performed, which serves as projection that reduces the six-times expanded number of layers to a target number of layers (significantly smaller than number of features after expansion).

The expand-convolve-project design of bottleneck can be thought of as if the input data were compressed and the expansion part represented decompression. After that, the data are convolved and finally they are compressed again by the projection part.

The depth-wise convolution can be strided in order to reduce spatial dimension of the computed feature maps. In case it is not strided and if at the same time the number of input and output channels of the bottleneck block is the same, skip connection between the input an the output is performed via addition operation. There is 17 of these modules in the MobileNetV2 architecture. See Figure 15a for overview of the bottleneck module and refer to [28] for overview of the whole architecture.

6.1.2.2 MobileNetV3 [7] improves upon its predecessor by adding attention mechanism into the bottleneck design, by replacing ReLU with the Hard Swish activation function (refer to Section 6.2.2) in deeper bottleneck modules (ReLU is used only in the first three bottleneck layers of the Small variant of MobileNetV3) and by changes in the classification part of the network. However, because the classification part of the architecture is not used in the hand pose estimation task, it can be ignored for the purposes of this thesis.

The attention mechanism added into the bottleneck is Squeeze-and-Excitation module [8]. It is applied to the feature maps produced by the depth-wise convolution module, just before the projection stage, as can be seen from Figure 16. Squeeze-and-Excitations are drop-in enhancements that can be added into every convolutional network. Given an input image $U \in \mathbb{R}^{C \times H \times W}$, they apply global average pooling $F_{sq}(\cdot)$ on U, which outputs $C \times 1 \times 1$ tensor (this means there is single weight for each channel). Next, those C weights are passed into a primitive encoder-decoder network represented by two fully-connected layers activated by ReLU ($F_{sq}(\cdot)$ in Figure 16). This resulting C weights are then used to weight the original input image U, that is, each channel of U is multiplied by respective computed weight.

This attention method learns to weight feature maps that are relevant for processing of the current image. Refer to Figure 15b for overview of the MobileNetV3 bottleneck module.

6.2 Activation Functions

Activation functions are non-linear functions that prevent networks from collapsing into simple linear transformations. In the following sections, the functions used in this thesis are described.

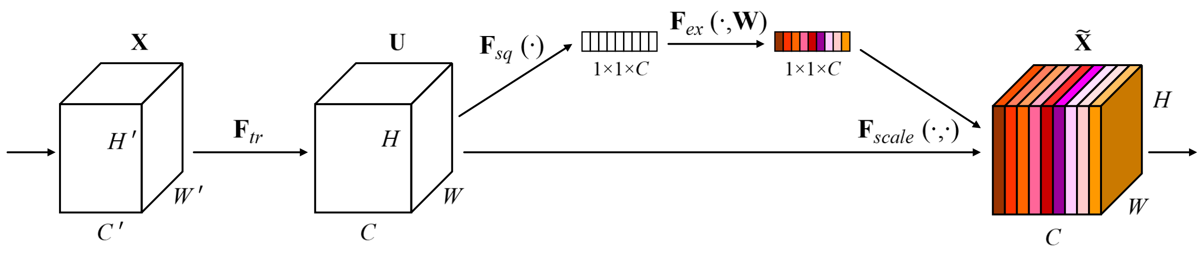


Figure 16: Squeeze-and-Excitation Module [8]
Input image $X \in \mathbb{R}^{C' \times H \times W}$ is convolved into tensor U, for which its channel weights are computed by performing global average pooling $F_{sq}(\cdot)$, followed by two fully-connected layers with ReLU activations. The resulting weight vector is used to multiply each corresponding channel of U by $F_{scale}(\cdot, \cdot)$.

The HPE task uses modified MobileNetV3 backbone for feature extraction — there are two activation functions used in the MobileNetV3 architecture: ReLU and Hard Swish [7]. Due to ReLU's computational simplicity, it is used in the initial layers of the architecture, while Hard Swish — being slightly more computationally demanding (see more below) — is used in deeper layers (due to smaller spatial size of feature maps).

Among many newer activation functions, there exists one which provides increased accuracy that I decided to test; Mish [29]. Next follows description of each of these activation functions.

6.2.1 ReLU

ReLU [30] is a widely used, computationally efficient activation function. It is bounded from below and unbounded from above. As noted by [29], it is not smooth, which may be disadvantageous during optimization at training time. Refer to Equation 7 and Figure 17 for its definition and plot.

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

6.2.2 Hard Swish

The Hard Swish function [7] is a less computationally demanding approximation of the Swish function [31]. Like ReLU, it is bounded only from below, which according to [7] produces significantly better results compared to ReLU and is quantization-friendly. Its definition can be seen in Equation 9. Refer to Figure 17 for comparison with other activation functions.

$$\mathrm{ReLU6}(x) = \begin{cases} 6, & \text{if } x > 6 \\ x, & \text{if } 0 \le x \le 6 \\ 0, & \text{if } x < 0 \end{cases} \tag{8}$$

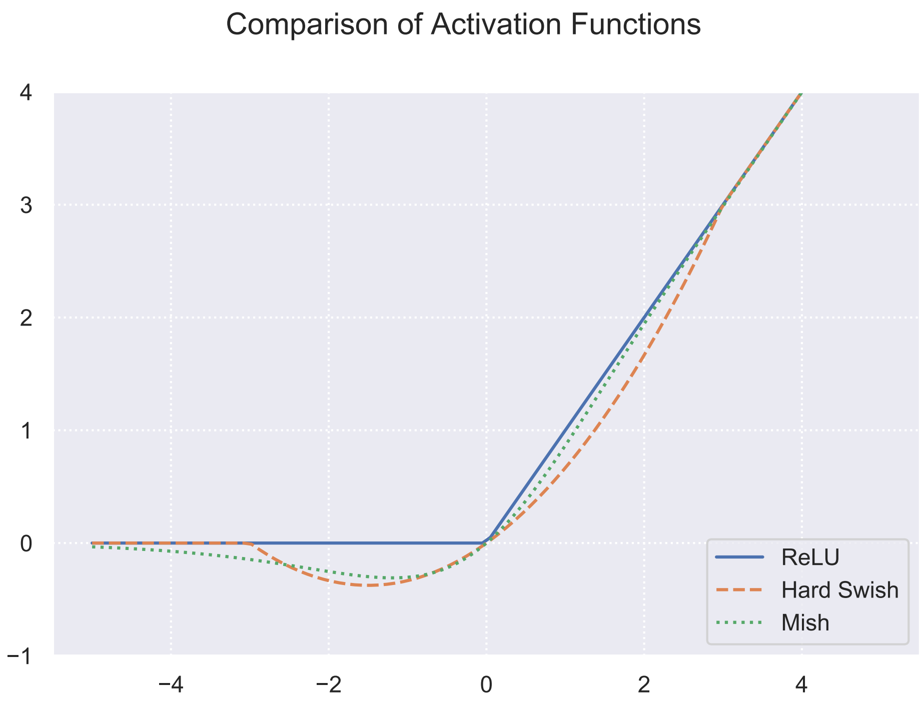


Figure 17: Comparison of activation functions used in implementation of this thesis.

$$\text{H-Swish}\left(x\right) = x\frac{\text{ReLU6}\left(x+3\right)}{6} \tag{9}$$

6.2.3 Mish

The Mish activation function [29] is the newest one of the three and it provides the best accuracy throughout all major architectures like ResNet, MobileNetV2 and Inception v3. The function is smoother and provides increased stability of the loss graph while training [29]. Its drawback is increased per-epoch time. One possible future performance improvement would be introduction of a Hard Mish variant, that — like Hard Swish — would be a piecewise linear approximation. See Equation 10 for its definition and Figure 17 for comparison with other activation functions.

$$\text{Mish}\left(x\right) = x\tanh\left(\ln\left(1+e^{x}\right)\right) \tag{10}$$

6.3 Final Architecture

Now that the main building blocks — CrossInfoNet (see Section 5.3) and MobileNetV3 (see Section 6.1) — have been overviewed, the proposed Hand Pose Estimation neural network architecture, dubbed *CrossInfoMobileNet*, will be described.

CrossInfoMobileNet, like CrossInfoNet [5], consists of three subnetworks; Feature Extractor, Feature Refiner and Joint Regressor. The added value of CrossInfoMobileNet over its predecessor is performance increase ($\approx$ 2.37 times less mult-adds) and smaller model size ($\approx$ 2.21 times less parameters). It achieves this by modifying all three subnetworks to use more efficient building blocks. Note that CrossInfoMobileNet is based on the CrossInfoNet authors' Tensorflow implementation rather than on the paper [5]; there are slight differences between those two. Refer to Section 5.3.5 for more details.

I started searching for a more effective architecture by applying incremental changes to my PyTorch reimplementation of CrossInfoNet. Over 27 different architectures were tested in the search for the most feasible one. I have eventually adjusted all three subnetworks of CrossInfoNet — namely the Feature Extractor, Feature Refiner and the Joint Regressor.

In the following sections, it will be described what the revised subnetworks consist of and how do they differ compared to CrossInfoNet. For graphical overview of CrossInfoMobileNet, refer to Figure 18. It is recommended to revisit Figure 18 while reading the following sections to get a better idea of how the described modules are structured.

6.3.1 Feature Extractor

The Feature Extractor module is the entry point of CrossInfoMobileNet; it takes a 96×96 px cropped depth image of a hand (see Section 6.4 for more details) as the input, which is then fed into a bottleneck-like architecture. The Feature Extractor outputs 168 feature maps of size 12×12 px that serve as the input to the Feature Refiner module. A bottleneck architecture can be thought of as a two-part process; compression of the input into a lower-dimensional representation and decompression into desired modality.

The Feature Extractor consists of three parts: modified MobileNetV3 for general feature extraction (compression part of the bottleneck), feature upscaling, which outputs the 168 feature maps of size 12×12 px (decompression part of the bottleneck) and auxiliary heatmap generation task.

The first step is extraction of general features from the input image via a modified MobileNetV3 architecture[5] — this can be seen as the compression part of the bottleneck architecture. MobileNetV3 contains 11 bottleneck blocks (see Section 6.1.2) — let us refer to i^{th} block as B_i — and each of them outputs more feature maps than the previous block while reducing the spatial dimension.

In the decompression part, the outputs $B_1 \in \mathbb{R}^{16\times24\times24}, B_3 \in \mathbb{R}^{24\times12\times12}, B_8 \in \mathbb{R}^{48\times6\times6}$ and $B_{11} \in \mathbb{R}^{96\times3\times3}$ are two times upscaled and used for establishing skip connections (like in a U-Net architecture [21]). The decompression part consists of 3 upscale-convolve-concatenate blocks.

Finally, there is the auxiliary heatmap generation task; heatmaps are generated by MobileNet Bottleneck submodule and one convolutional layer, generating 21 heatmaps of size 24×24 px —

[5]There are two variants of MobileNetV3; MobileNetV3-Small and MobileNetV3-Large [7]. Those two differ in the number of used Bottleneck modules (11 vs. 17) — see Section 6.1 for explanation. After trying both variants inside of the final version of the HPE network, it was found that counterintuitively, the Large variant performed worse than the Small one, therefore a modified MobileNetV3-Small architecture was used for general features extraction.

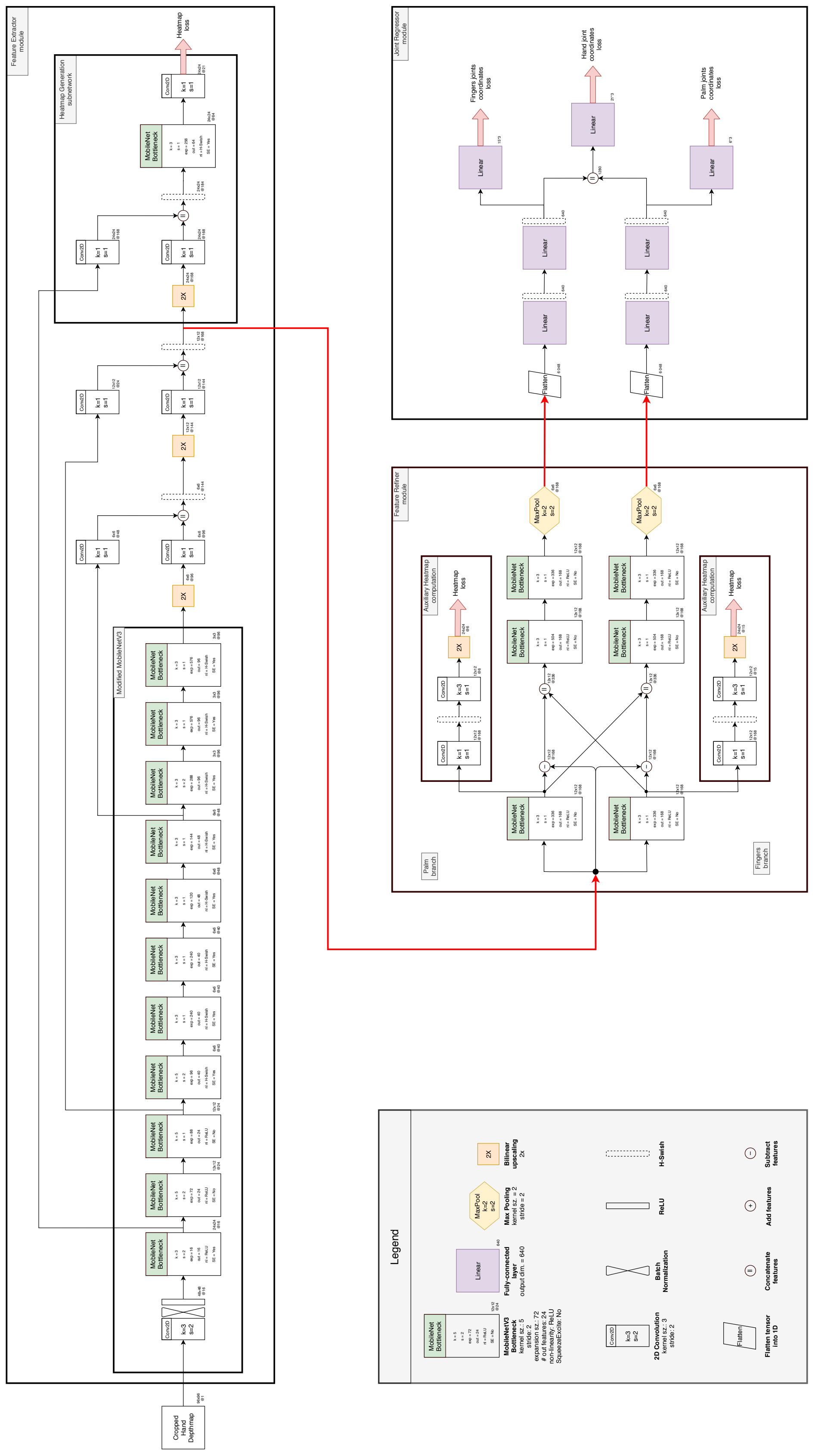

Feature Extractor module
Heatmap Generation subnetwork
Modified MobileNetV3
MobileNet Bottleneck
Conv2D
k=3 s=2
k=1 s=1
2X
Heatmap loss
Cropped Hand Depthmap
Feature Refiner module
Auxiliary Heatmap computation
Palm branch
Fingers branch
MaxPool k=2 s=2
Joint Regressor module
Flatten
Linear
Fingers joints coordinates loss
Hand joint coordinates loss
Palm joints coordinates loss
Legend
MobileNetV3 Bottleneck kernel sz.: 5 stride 2 expansion sz.: 72 # out features: 24 non-linearity: ReLU SqueezeExcite: No
Fully-connected layer output dim. = 640
Max Pooling kernel sz.: 2 stride = 2
Bilinear upscaling 2x
2D Convolution kernel sz.: 3 stride: 2
Batch Normalization
ReLU
H-Swish
Flatten tensor into 1D
Concatenate features
Add features
Subtract features

Figure 18: CrossInfoMobileNet architecture overview.

one heatmap per predicted joint. Each of the 21 heatmaps predicts for each pixel the likelihood of occurrence of projected 3D joint at that position. The heatmaps are used only during training as a component of the loss function to allow the network to learn more relevant features. This technique is called multi-task learning and the reasoning behind it is that if we first extract common, general features and then use those features to learn task-specific features, then the network should learn more relevant/meaningful features for both tasks and be less prone to overfitting.

Refer to Figure 18 to see how the Feature Extractor architecture looks like.

6.3.1.1 Comparison to CrossInfoNet The Feature Extractor module was the first part that I started to modify; in CrossInfoNet, the Feature Extractor module uses ResNet-50 residual blocks for feature extraction, which accounts for 61% of mult-adds of the whole network. As those blocks were not designed to run seamlessly on mobile devices, I decided to replace them by a more recent and a more efficient design; MobileNetV3 [7].

The modified Feature Extractor of CrossInfoMobileNet uses 78.6% less mult-adds (74.4×10^6 vs. 349.0×10^6) and 28% less parameters (1.0×10^6 vs. 1.4×10^6) compared to CrossInfoNet's version.

6.3.2 Feature Refiner

The features obtained from Feature Extractor (168 features of size 12×12 px) — let us denote them as F_E — describe the whole image of the hand. Authors of CrossInfoNet [5] split the task of predicting hand joints into two subtasks; they divide the whole hand's joints into two sets for which specialized feature maps are then computed in the Feature Refiner module, so that they can be regressed separately;

- Palm joints — the wrist joint and the 5 metacarpal joints that are not as mobile as the other joints. These 5 joints are located on roughly the same imaginary plane created by the palm. These six joints are all the (blue) joints with 2 DoFs and the red joint with 6 DoFs in Figure 9b.
- Fingers joints — the rest of the joints, those are the 10 green joints with 1 DoF in Figure 9b together with additional 5 joints for all fingertips. There is therefore 15 fingers joints in this set.

The Feature Refiner comprises of two branches that compute features specific to palm joints and fingers joints. Each branch first passes F_E into a MobileNet bottleneck (let us denote output of this MobileNet bottleneck as F_{P1} in the palm branch and as F_{F1} in the fingers branch), after which it splits again into two subbranches:

- Auxiliary heatmap generation subbranch — in order to constrain the MobileNet Bottleneck to learn features relevant to palm and fingers joints respectively, this branch — consisting of

2 convolutional layers followed by two times upscaling — produces only heatmaps specific for either the palm or fingers joints. It outputs 6 (for the palm branch) or 15 (for the fingers branch) heatmaps of size 24×24 px that are then being used only during training in the loss function.

- Feature maps generation subbranch — this is the primary subbranch whose task is to generate joint-specific features that will be later used in the Joint Regressor module. Let us describe this subbranch on the concrete example of the palm branch (the fingers branch follows the same procedure). The first step is generation of complementary features $\bar{F}_{F1} = F_E - F_P$ (by subtracting the palm feature maps F_P from the common feature maps F_E, we obtain the complementary finger feature maps $\bar{F}_{F1}$).

  In the next step, cross-connection is performed: we concatenate the complementary maps $\bar{F}_{F1}$ with the (non-complementary) palm feature maps F_F from the fingers branch. These concatenated finger features are then passed into two consequent MobileNet bottlenecks and 2×2 Max Pooling with stride 2 is then performed on the resulting features, which will yield 168 finger feature maps of size 6×6 px to be used in the Joint Regressor module. Similar operations are performed on the fingers branch, which will yield palm feature maps.

Feature Refiner outputs two sets of feature maps; there is 168 features of size 6×6 px in both of them — the first set contains feature maps specific to the palm joints and the second set is specialized for the fingers joints. Both outputs are then passed into the Joint Regressor module. It is vital to see Figure 18 to understand architecture of Feature Refiner.

6.3.2.1 Comparison to CrossInfoNet The auxiliary heatmap generation subbranch is unchanged. However, CrossInfoMobileNet replaces CrossInfoNet's bottleneck submodule by a more efficient MobileNet Bottleneck. The CrossInfoNet's bottleneck consists of three times Convolution-Batch Normalization-ReLU layers[6], for overview of the MobileNet bottleneck, refer to Section 6.1.2. CrossInfoMobileNet also uses 2 MobileNet bottleneck after the cross-connection, while CrossInfoNet uses only one bottleneck.

The modified Feature Refiner of CrossInfoMobileNet uses 21.6% less mult-adds (157.4×10^6 vs. 201.1×10^6) and 20.8% less parameters (1.08×10^6 vs. 1.36×10^6) compared to CrossInfoNet's version.

6.3.3 Joint Regressor

This is the final module of CrossInfoMobileNet, its inputs are the fingers feature maps $F_{rF} \in \mathbb{R}^{168 \times 6 \times 6}$ and palm feature maps $F_{rP} \in \mathbb{R}^{168 \times 6 \times 6}$ outputted by the Feature Refiner module. The Joint Regressor is splitted into two branches (one branch for the palm and fingers joints

[6]Note that the bottleneck is not inverted (unlike in MobileNet).

respectively) that are joined at the end to produce a single, unified prediction of the whole hand joints.

Each branch firstly flattens the feature maps from a $\mathbb{R}^{168\times6\times6}$ tensor into $\mathbb{R}^{6\,048}$ vector that is then passed into a fully-connected layer with H-Swish activation that outputs vector of size 640[7]. The exactly same fully-connected layer with H-Swish activation is repeated once more. The branch is now splitted into two subbranches; the first subbranch contains fully-connected layer that outputs either palm joints prediction ($\mathbb{R}^{6\times3}$) or fingers joints prediction ($\mathbb{R}^{15\times3}$) respectively. These partial joint predictions are used only during training in the loss function.

The second subbranch concatenates the 640 features from the current branch with the other branch's respective 640 features into a single feature vector of size 1280 (merging the two branches together), which is is then passed into the final fully-connected layer that outputs all 21 hand joints together ($\mathbb{R}^{21\times3}$).

6.3.3.1 Comparison to CrossInfoNet

CrossInfoMobileNet uses 640 features instead of 1024 features found in CrossInfoNet. Moreover, CrossInfoMobileNet replaces ReLU activations with H-Swish and removes dropout layers.

The modified Feature Refiner of CrossInfoMobileNet uses 58.9% less mult-adds (8.68×10^6 vs. 21.16×10^6) and also 58.9% less parameters (8.68×10^6 vs. 21.16×10^6) compared to CrossInfoNet's version.

6.4 Network Input

Input to CrossInfoMobileNet is a 96×96 px normalized depth image of a hand. To obtain this normalized image patch from input image, we first need to detect where the Center-of-Mass (CoM) of the hand is in the image. This is done by utilizing the segmented depth image (see Section 3) and finding CoM of the most dominant hand.

Once the 3D coordinates of CoM in world space are obtained, we compute the 3D coordinates of vertices of an imaginary cube with edge size 175 mm, centered at the CoM. Then we project those 8 vertices onto the 2D image, we create bounding box out of the 8 vertices and crop the hand inside of the bounding box. Because the cropped hand region can be non-square, the cropped hand image is scaled down so that its longer edge is 96 px wide, respecting its aspect ratio and centering it inside the final 96×96 px input image. The cropped hand is therefore not distorted.

The depth values of cropped patch are normalized to be within the range $[-1; +1]$ with the origin being at the location of the CoM. Invalid depth values are mapped to be on the far side of the imaginary cube.

[7]The CrossInfoNet's Joint Regressor architecture uses inner layer size of 1024. By experimenting with the number of layers in CrossInfoMobileNet's Joint Regressor, it was found that lowering the number of layers to 640 still provided the same error on the MSRA dataset, which is why 640 layers has been selected for CrossInfoMobileNet. See Table 3 for impact of layer size in Joint Regressor to the MSRA error value.

6.5 Data Augmentation

On-line data augmentation is applied during training to prevent overfitting and to make the network more robust. There are 4 data augmentation settings that are randomly applied on each image of every batch during training:

- None — the original hand image can be unchanged.
- Movement — the Center of Mass can be offsetted by an offset within the range $[-3; +3]$ centimeters in each dimension.
- Scaling — the image can be scaled by $\pm 5\%$.
- Rotation — 2D rotation of the image in the range of $[-180; 180]$ degrees.

The 3D hand joints are adjusted accordingly to correspond to the data augmentation performed on the hand image.

6.6 Loss function

The loss function L comprises of 6 terms in total (heatmap loss and regression loss);

$$\begin{aligned} L &= 0.01 \cdot L_{\text{heatmaps}} + L_{\text{regression}} \\ L_{\text{heatmaps}} &= L_{\text{FE\_HandHeatmaps}} + L_{\text{FR\_PalmHeatmaps}} + L_{\text{FR\_FingersHeatmaps}} \\ L_{\text{regression}} &= L_{\text{JR\_PalmJoints}} + L_{\text{JR\_FingersJoints}} + L_{\text{JR\_HandJoints}} \end{aligned}$$

where:

$L_{\text{FE\_HandHeatmaps}}$	: 21 heatmaps for all joints computed in the Feature Extractor module
$L_{\text{FR\_FingersHeatmaps}}$	: 15 heatmaps for fingers joints computed in the Feature Refiner module
$L_{\text{FR\_PalmHeatmaps}}$	: 6 heatmaps for palm joints computed in the Feature Refiner module
$L_{\text{JR\_PalmJoints}}$	: 21 3D world coordinates of the whole hand's joints computed in Joint Regressor module
$L_{\text{JR\_FingersJoints}}$	: 15 3D world coordinates of the fingers joints computed in Joint Regressor module
$L_{\text{JR\_HandJoints}}$	: 6 3D world coordinates of the palm joints computed in Joint Regressor module

Refer to red arrows in Figure 18 to see at what stages in the network are the 6 components of the loss function computed.

The heatmap losses are computed as a simple Mean Squared Error between the estimated and GT heatmaps. Regression loss is implemented via Mean Squared Error between predictions and GT 3D locations as well.

Ground Truth heatmaps are generated on-line during training due to randomized application of data augmentation techniques. To generate a heatmap for single joint, we first create a 24×24 px image initialized to 0. Then the Ground Truth (GT) 3D location of that joint is projected onto the 96×96 px cropped depth image patch of hand and at that location a single pixel is set. Then 3×3 px Gaussian Blur is applied to that place to ensure that the heatmap does not contain sharp gradients. This process is repeated for each of the 21 joints.

6.7 Training

The network was trained using the Adam optimizer with learning rate $lr = 10^{-3}$, weight decay 10^{-5} and learning rate decay of 0.97 per epoch. The best accuracy was achieved by using epoch size 128 while training on two GPUs in parallel (each GPU processed batch of size 64).

The experimental architectures were trained on two hardware configurations; the first system had 4× NVIDIA GTX 1070 and the second system was the IT4Innovations' Barbora cluster with 4× NVIDIA V100 cards. Per-epoch training time using the Barbora cluster was about three times smaller compared to the first system. Per-epoch training time for the CrossInfoMobileNet on the first system was 76 seconds and the networks have been trained for 130 epochs.

The network has been trained on the MSRA dataset [13]. Training metadata such as per-epoch time, training error, testing error and the best epoch were saved into TensorBoard logs in order to compare experimental architectures to each other and to the reference CrossInfoNet implementation. An example of graph with testing error comparing CrossInfoNet and CrossInfoMobileNet can be seen in Figure 19, which depicts error on the testing dataset (consisting of the first subject of the MSRA dataset in this particular case — the networks are trained on the rest of the MSRA dataset, i.e. on subjects 2-9) after a training epoch ends.

It can be seen that CrossInfoMobileNet is more stable during the training process for subjects 2-9 of the MSRA dataset compared to CrossInfoNet; it is hypothesised that the stability of the training process can be partially credited to the use of H-Swish activation function (see Section 6.2.2), which produces smoother output landscape manifold (see Figure 5 of [29]) compared to ReLU-based architecture (such as CrossInfoNet), that is easier to optimize.

Note that the testing error varies depending on which parts of the dataset the networks are trained on and it can happen that both CrossInfoNet's and CrossInfoMobileNet's reported testing errors are not going to be stable during training on subjects of the MSRA dataset different from 2-9.

6.8 Performance

As can be seen from Table 3, the CrossInfoMobileNet architecture has ≈ 2.37 times less mult-adds and is ≈ 2.21 times smaller than its predecessor CrossInfoNet while retaining the same accuracy on the MSRA dataset.

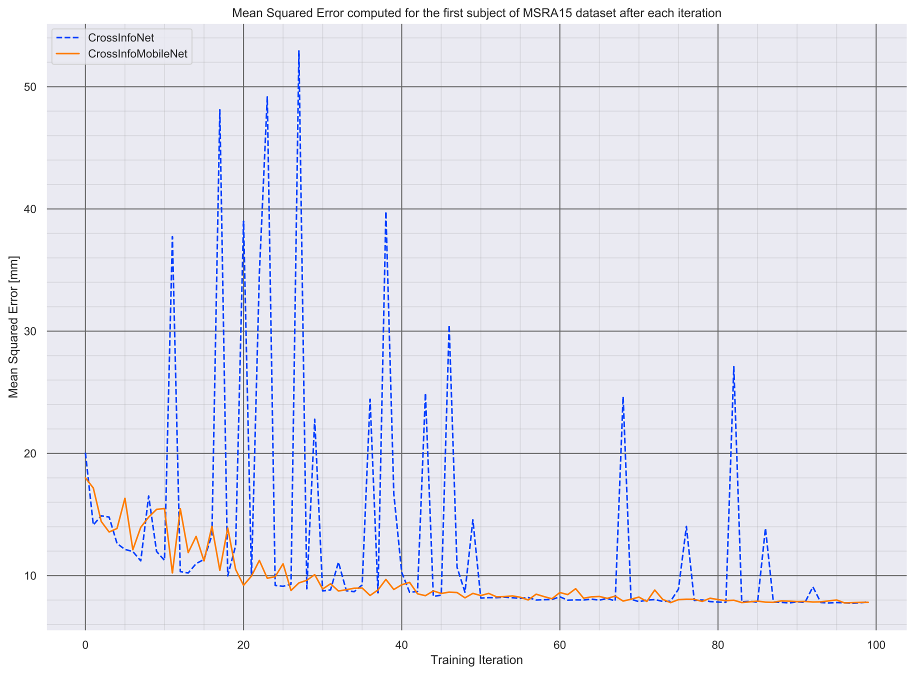


Figure 19: MSE for the first subject of MSRA15 dataset after each iteration during training.

In the first row of the table, there are results for my reference PyTorch reimplementation of CrossInfoNet, the next three rows show the final version of CrossInfoMobileNet with and without dropout in Joint Regressor subnetwork. The rest of the rows are CrossInfoMobileNet variants with varying number of neurons of fully-connected layers in the Joint Regressor part.

The final version of CrossInfoMobileNet is the one without dropout and with 640 neurons in Joint Regressor's fully-connected layers. As can be seen from Table 3, dropout negatively influences accuracy on the MSRA dataset. To see the influence, compare CrossInfoMobileNet variants without dropout and with two different dropout probabilities (in Table 3).

Performance tests showed that it takes in average about 0.94 milliseconds per sample for my CrossInfoNet PyTorch reimplementation to evaluate one sample. It takes in average only 0.60 milliseconds per sample for the proposed final version of CrossInfoMobileNet, which means the speed has been improved 1.56 times.

Testing was performed on a system with NVIDIA GTX 1070, Intel i7 7600k and 16 GB of RAM, PyTorch 1.5 on Ubuntu 19.04 was used to do the testing. The elapsed time measurements were obtained using CUDA Events to ensure that only kernel execution times (and not memory transfers) are being considered. The average duration reported above was computed in the following way: it was iterated through all frames of the first subject of the MSRA dataset by batches of size 128. For each batch the elapsed time was measured by the means of CUDA Events. These partial, per-batch measurements were then averaged to obtain the kernel execution duration per one input depth image.

[8]The error is computed in leave-one-out fashion by taking the average of 9 models' errors. Each model is trained

Table 3: Performance overview of CrossInfoNet and CrossInfoMobileNet variants.

# neurons denotes the number of neurons in fully-connected layers of Joint Regressor subnetwork, *Error* is average joint error on the MSRA dataset [13] measured in millimeters[8], *# params* is the number of architecture's parameters in millions, *# MADs* denote number of multiply-add operations and *Size* is size of a model in megabytes. The p value in CrossInfoMobileNet-dp represents probability of dropout (models without p do not use dropout).

Model Name	**# neurons**	**Error**	**# params.**	**# MADs**	**Size**
CrossInfoNet	1024	8.196 622	23 936 648	571.304 480	91.479
CrossInfoMobileNet	640	8.192 249	10 768 376	240.580 968	41.306
CrossInfoMobileNet-d0.4	640	8.483 366	10 768 376	240.580 968	41.306
CrossInfoMobileNet-d0.6	640	8.927 258	10 768 376	240.580 968	41.306
CrossInfoMobileNet-1024	1024	8.192 479	16 765 304	246.576 360	64.182
CrossInfoMobileNet-512	512	8.255 707	8 900 472	238.713 576	34.180
CrossInfoMobileNet-128	128	8.390 644	3 689 976	233.504 616	14.304
CrossInfoMobileNet-64	64	8.518 030	2 878 904	232.693 800	11.166

6.9 Testing

A demo Python application has been made that uses the Hand Segmentation model presented in Section 3 to segment out the most dominant hand from a scene and crop it out of the image. The cropped hand is then fed into the CrossInfoMobileNet network to estimate 3D join positions of the hand. The estimated 3D joints are projected onto the cropped hand image and shown to the user. Screenshot of the application can be seen in Figure 22.

For more detailed overview of success and failure cases, refer to Table 4 (note that all images in the Table except "Cropped hand" were cropped around the hand to show the hand in detail); the first five samples represent the success cases where the pipeline was able to segment hand and infer its joints.

Even though the hand segmentation process is not pixel-perfect due to the smallest (and fastest) variant of the HandSeg neural network being used (see Section 3), the results are mostly sufficient as the only goal of the segmentation process in this pipeline is to obtain rough centroid position of the most dominant hand in the input depth image and decide whether it is left or right hand (chirality of hand is chosen based on the number of pixels of the hand belonging into the left- or right-hand class).

Let us review the selected failure cases in Table 4; sample 6 is an example of hand centroid detection failure; in such cases, the centroid can be located outside of the surface of the hand (as is the case with centroid of torus) or the depth map may have invalid depth measurement (`NaN`)

on 8 subjects and tested on the ninth one (the MSRA dataset consists of 9 subjects performing 17 gestures each). Average of all 9 results is made, which is reported in Table 3.

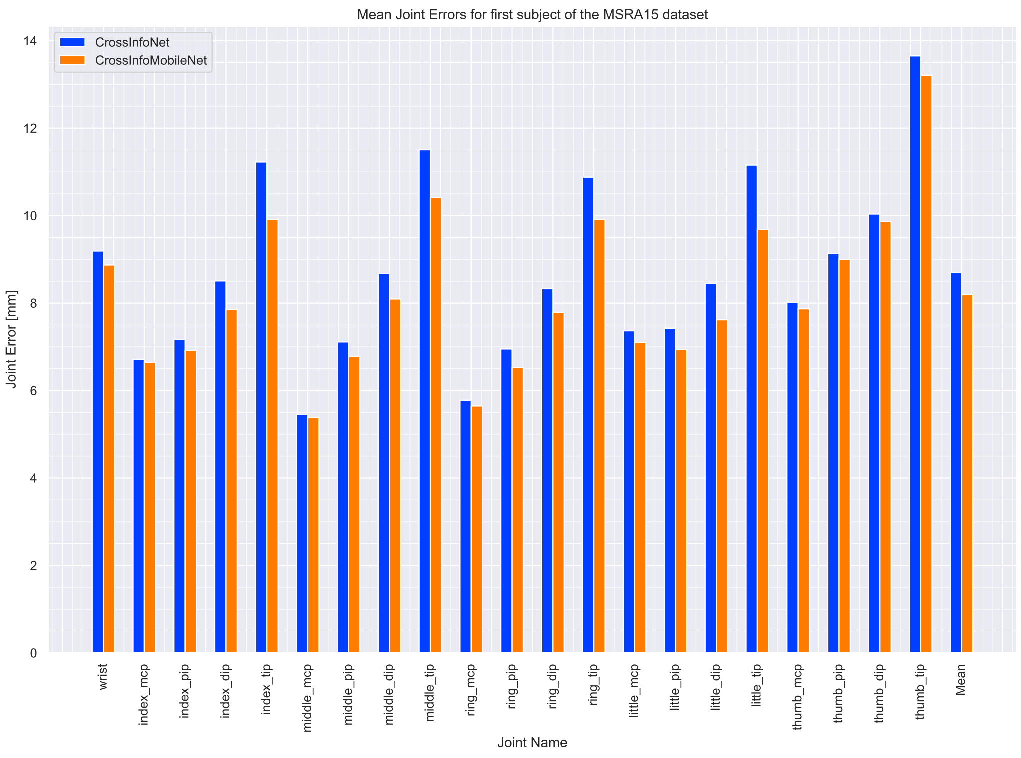


Figure 20: Mean Joint Errors of CrossInfoNet and CrossInfoMobileNet

at the centroid's position. Sample 7 shows that due to occlusions, it is impossible to correctly infer the occluded the finger joints because only the back of the hand is at the input of the network. Finally, on sample 8, it can be seen that when the fingers are closely together in open palm configuration, the network has difficulties discerning features of the individual fingers and sees them as a planar surface, thus failing to correctly estimate joint positions.

On the other hand, rows 1-5 in Table 4 demonstrate that the network is able to detect joints with enough precision in most cases for real-time usage in entertainment applications.

Figure 20 shows comparison of mean joint errors for each joint (testing was done only for the first subject of the MSRA dataset, testing on other subjects may produce different results) and compares CrossInfoNet and CrossInfoMobileNet in this regard. Percentage of samples of the testing dataset for which both networks produced errors under given thresholds can be seen in Figure 21. From both of these charts, it can be seen that CrossInfoMobileNet produces better results for subject 1 of the MSRA dataset. Testing in leave-one-out fashion (i.e. training on 8 subjects of the MSRA datasets and testing on the remaining one and doing this for all 9 subjects) shows that CrossInfoMobileNet produces only insignificantly better Mean Squared Error compared to my CrossInfoNet reimplementation when averaging results over all subjects, as can be seen from Table 3.

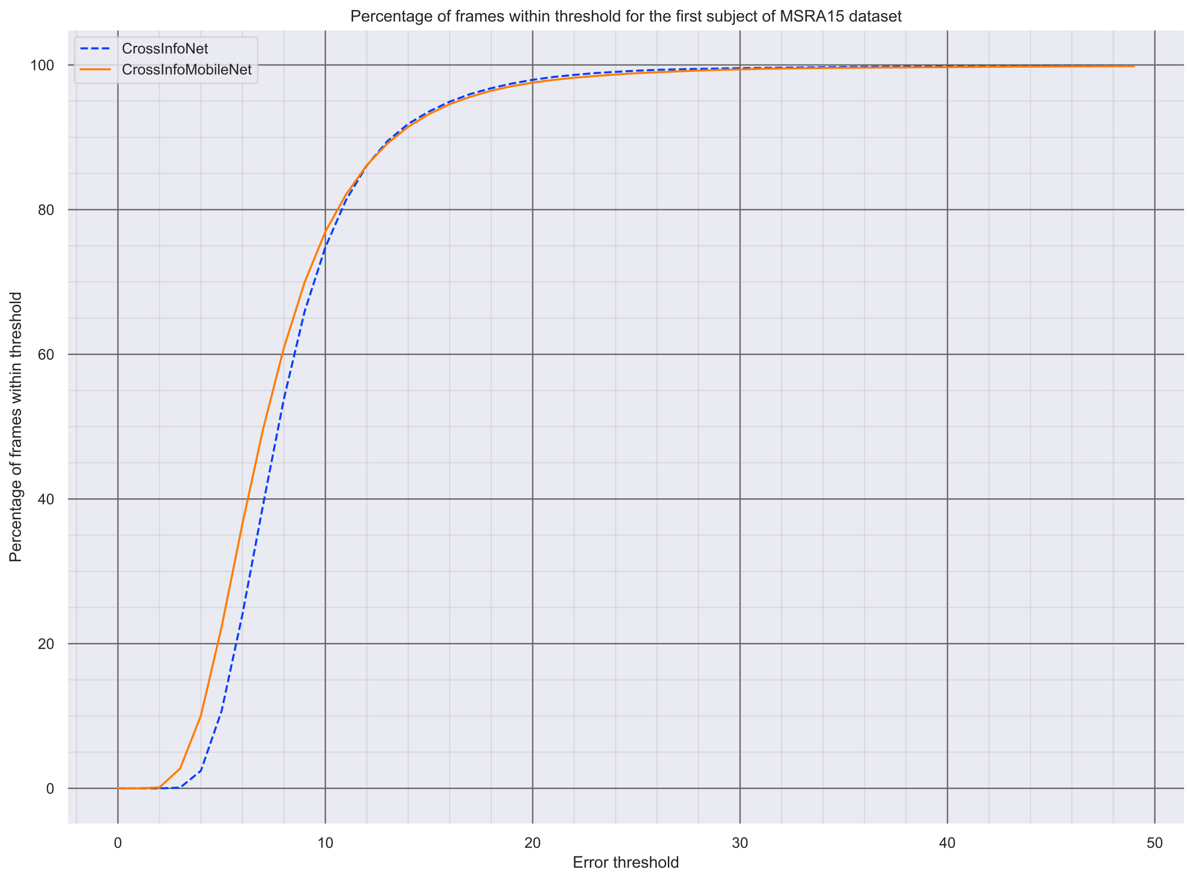


Figure 21: Percentage of frames within threshold for the first subject of MSRA15 dataset

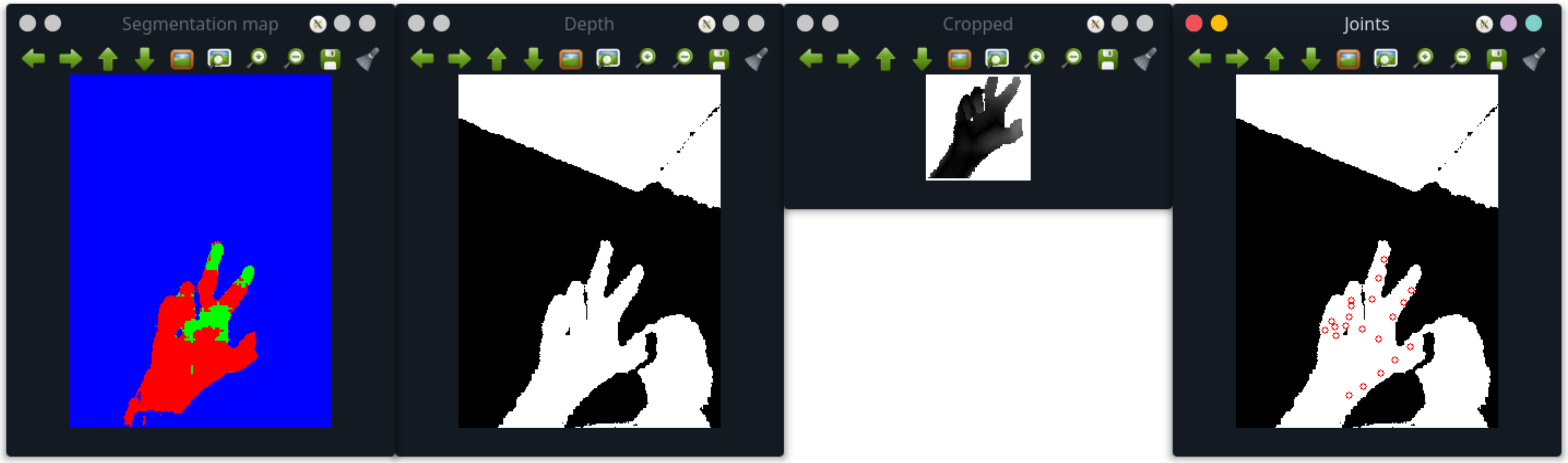
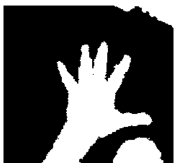
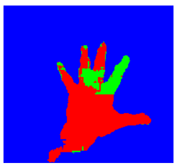
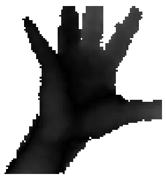
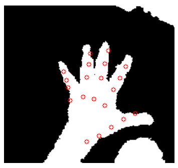
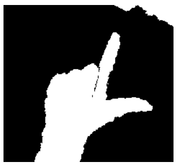
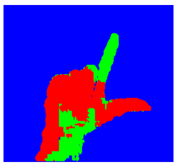
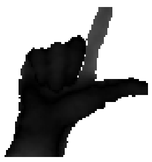
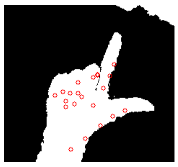
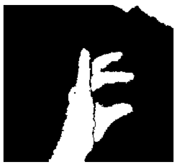
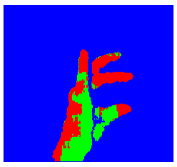
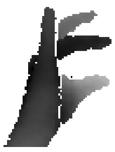
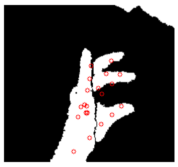
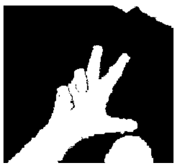
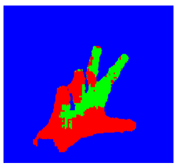
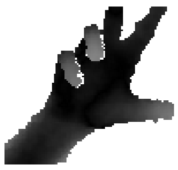
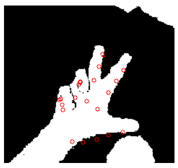
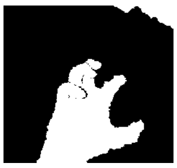
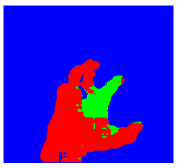
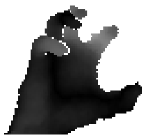
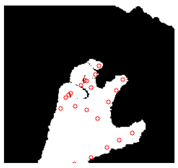
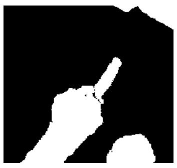
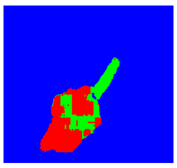
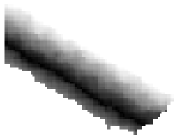
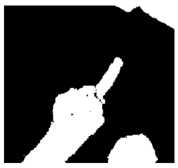
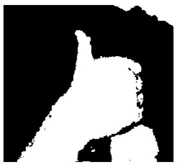
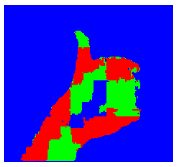
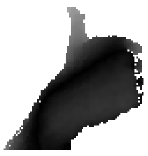
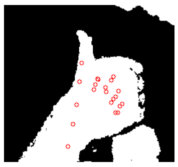
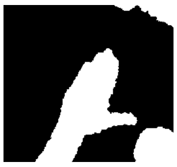
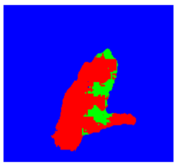
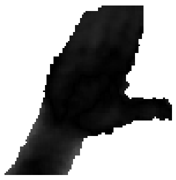
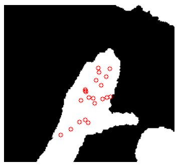


Figure 22: Demo Python application
It combines the Hand Segmentation model (refer to Section 3) with CrossInfoMobileNet to provide Hand Pose Estimation pipeline. There are 4 windows; “Segmentation map” depicts the raw segmentation map, “Depth” shows the raw depth map, “Cropped” shows the 96 × 96 px image that servers as input into CrossInfoMobileNet and finally “Joints” shows the locations of 3D joints predicted by CrossInfoMobileNet projected onto the depth image.

Table 4: Comparison of HPE pipeline results (using CrossInfoMobileNet).

Samples 1-5 show success cases of Hand Pose Estimation. The rest depicts failure cases.

#	Depth Image	Segmentation map	Cropped hand	Joint overlay
1				
2				
3				
4				
5				
6				
7				
8				

7 Conclusion

The goal of this thesis was to find a viable approach to 3D hand pose estimation from RGBD data and to implement a demonstrative application that would present the results. This task was divided into two independent parts; localization of hands inside the input RGBD image and estimation of 3D hand joints of the most dominant hand found in the input image.

Localization of the most dominant hand was solved by implementing a semantic segmentation neural network based on the HandSeg architecture [16] that classifies each pixel of depth image as belonging to either left hand, right hand or background class. It is a Fully-Convolutional architecture with bottleneck and skip connections in the style of U-Net [21]. By training the network on public hand segmentation datasets and then trying it on real-world data produced by a SL depth camera, it was found that semantic segmentation did not perform well enough to be used in this thesis. As the public datasets that I trained the network with were captured by a ToF depth sensor (which produces different kind of noise distribution compared to the SL depth camera), I decided to create a new hand segmentation dataset based on the methodology of [16]. Training the network on my own dataset with over 33 000 training samples indeed improved empirical results of semantic segmentation. The most dominant hand was selected as the biggest connected component in the segmented image. The requirement of being able to detect hands has therefore been fulfilled.

To accomplish the second part of the thesis — Hand Pose Estimation (HPE) from the detected hand image (i.e. obtaining 21 absolute 3D joint coordinates of the hand) — I decided to try two approaches; a traditional computer vision approach and a more modern one based on Deep Learning. To test the traditional way, I made an application that uses Particle Swarm Optimization algorithm to fit a Signed Distance Field model of a hand to the point cloud of the hand. Hand segmentation model described above was used to locate the most dominant hand in the input image, whose point cloud would be used for the fitting process. I found this method not to be suitable for the purposes of this thesis due to its below real-time performance, lack of robustness and precision.

By researching recent Deep Learning approaches to the HPE problem, I found and implemented three candidate neural network architectures and selected the one among them with the best potential for improvement — the CrossInfoNet architecture [5]. The authors' publicly available Tensorflow implementation allowed me to test the network's suitability and reimplement it into a more modern PyTorch framework in order to start experimenting with the network design. I tried 27 different experimental adjustments to the original architecture and ended up with the final architecture — dubbed *CrossInfoMobileNet* — that requires 2.37 times less multiply-add operations (FLOPs) and 2.22 times less parameters than my reimplementation of the CrossInfoNet network, while achieving the same error on the MSRA dataset [13]. This reduction of multiply-add operations resulted into 1.56 times faster real-world performance (0.94 vs. 0.60 milliseconds per one sample for my CrossInfoNet reimplementation vs. CrossInfoMobileNet).

Two applications were made; the first one implemented traditional computer vision approach to Hand Pose Estimation (see Section 4) and the second one used Deep Learning-based approach. It was deemed that the Deep Learning approach is vastly superior to the traditional one in terms of speed, robustness and precision, as can be seen from the final demo application that implements the whole pipeline (including hand segmentation) and visualizes the predicted 3D joint coordinates by projecting them onto the hand depth image.

In the future, this project could be improved by experimenting with the DeepLabV3+ neural network for semantic segmentation and trying to prune and quantize the CrossInfoMobileNet network. A smaller variant of CrossInfoMobileNet could be made, which would be trained by knowledge distillation.

References

[1] O. Wasenmüller and D. Stricker, "Comparison of kinect v1 and v2 depth images in terms of accuracy and precision," in *Computer Vision – ACCV 2016 Workshops*, C.-S. Chen, J. Lu, and K.-K. Ma, Eds. Cham: Springer International Publishing, 2017, pp. 34–45.

[2] M. Šimoník, "Signed distance field hand model." [Online]. Available: https://www.shadertoy.com/view/tsSGzK

[3] A. Spurr, J. Song, S. Park, and O. Hilliges, "Cross-modal deep variational hand pose estimation," *CoRR*, vol. abs/1803.11404, 2018. [Online]. Available: http://arxiv.org/abs/1803.11404

[4] S. Li and D. Lee, "Point-to-pose voting based hand pose estimation using residual permutation equivariant layer," *CoRR*, vol. abs/1812.02050, 2018. [Online]. Available: http://arxiv.org/abs/1812.02050

[5] K. Du, X. Lin, Y. Sun, and X. Ma, "Crossinfonet: Multi-task information sharing based hand pose estimation," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019, pp. 9888–9897.

[6] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[7] A. Howard, M. Sandler, G. Chu, L. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," *CoRR*, vol. abs/1905.02244, 2019. [Online]. Available: http://arxiv.org/abs/1905.02244

[8] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," *CoRR*, vol. abs/1709.01507, 2017. [Online]. Available: http://arxiv.org/abs/1709.01507

[9] I. Google, "Google soli." [Online]. Available: https://atap.google.com/soli/

[10] S. Tateno, Y. Zhu, and F. Meng, "Hand gesture recognition system for in-car device control based on infrared array sensor," in *2019 58th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, 2019, pp. 701–706.

[11] K.-Y. Chen, S. N. Patel, and S. Keller, "Finexus: Tracking precise motions of multiple fingertips using magnetic sensing," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1504–1514. [Online]. Available: https://doi.org/10.1145/2858036.2858125

[12] O. Glauser, S. Wu, D. Panozzo, O. Hilliges, and O. Sorkine-Hornung, “Interactive hand pose estimation using a stretch-sensing soft glove,” *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3306346.3322957

[13] X. Sun, Y. Wei, Shuang Liang, X. Tang, and J. Sun, “Cascaded hand pose regression,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 824–832.

[14] X. Chen, G. Wang, H. Guo, and C. Zhang, “Pose guided structured region ensemble network for cascaded hand pose estimation,” *CoRR*, vol. abs/1708.03416, 2017. [Online]. Available: http://arxiv.org/abs/1708.03416

[15] J. Taylor, V. Tankovich, D. Tang, C. Keskin, D. Kim, P. Davidson, A. Kowdle, and S. Izadi, “Articulated distance fields for ultra-fast tracking of hands interacting,” *ACM Trans. Graph.*, vol. 36, no. 6, pp. 244:1–244:12, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3130800.3130853

[16] S. R. Malireddi, F. Mueller, M. Oberweger, A. K. Bojja, V. Lepetit, C. Theobalt, and A. Tagliasacchi, “Handseg: A dataset for hand segmentation from depth images,” *CoRR*, vol. abs/1711.05944, 2017. [Online]. Available: http://arxiv.org/abs/1711.05944

[17] C. Qian, X. Sun, Y. Wei, X. Tang, and J. Sun, “Realtime and robust hand tracking from depth,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1106–1113.

[18] D. Tang, H. J. Chang, A. Tejani, and T. Kim, “Latent regression forest: Structured estimation of 3d articulated hand posture,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 3786–3793.

[19] H. Sarbolandi, D. Lefloch, and A. Kolb, “Kinect range sensing: Structured-light versus time-of-flight kinect,” *Computer Vision and Image Understanding*, vol. 139, pp. 1 – 20, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314215001071

[20] J. Tompson, M. Stein, Y. Lecun, and K. Perlin, “Real-time continuous pose recovery of human hands using convolutional networks,” *ACM Trans. Graph.*, vol. 33, no. 5, pp. 169:1–169:10, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629500

[21] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” *CoRR*, vol. abs/1505.04597, 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[22] L. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, “Encoder-decoder with atrous separable convolution for semantic image segmentation,” *CoRR*, vol. abs/1802.02611, 2018. [Online]. Available: http://arxiv.org/abs/1802.02611

[23] Íñigo Quílez, “distance functions.” [Online]. Available: https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

[24] B. Luff, A. Topalian, E. Wood, S. Khamis, P. Kohli, S. Izadi, R. Banks, A. Fitzgibbon, J. Shotton, L. Bordeaux, T. Cashman, B. Corish, C. Keskin, T. Sharp, E. Soto, D. Sweeney, and J. Valentin, “Efficient and precise interactive hand tracking through joint, continuous optimization of pose and correspondences,” *ACM Transactions on Graphics*, vol. 35, pp. 1–12, 07 2016.

[25] I. Zelinka, *SOMA — Self-Organizing Migrating Algorithm.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 167–217. [Online]. Available: https://doi.org/10.1007/978-3-540-39930-8_7

[26] S. Sridhar, A. Oulasvirta, and C. Theobalt, “Interactive markerless articulated hand motion tracking using rgb and depth data,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Dec. 2013. [Online]. Available: http://handtracker.mpi-inf.mpg.de/projects/handtracker_iccv2013/

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

[28] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, “Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation,” *CoRR*, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[29] D. Misra, “Mish: A self regularized non-monotonic neural activation function,” *ArXiv*, vol. abs/1908.08681, 2019.

[30] A. F. Agarap, “Deep learning using rectified linear units (relu),” *CoRR*, vol. abs/1803.08375, 2018. [Online]. Available: http://arxiv.org/abs/1803.08375

[31] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *CoRR*, vol. abs/1710.05941, 2017. [Online]. Available: http://arxiv.org/abs/1710.05941