BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS
ÚSTAV RADIOELEKTRONIKY

COMPUTER VISION AND HAND GESTURES DETECTION AND FINGERS TRACKING

POČÍTAČOVÉ VIDĚNÍ A DETEKCE GEST RUKOU A PRSTŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. Tomáš Bravenec
SUPERVISOR VEDOUCÍ PRÁCE	doc. Ing. Tomáš Frýza, Ph.D.

BRNO 2019

Diplomová práce

magisterský navazující studijní obor **Elektronika a sdělovací technika**
Ústav radioelektroniky

Student: Bc. Tomáš Bravenec	***ID:*** 173619
Ročník: 2	***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Počítačové vidění a detekce gest rukou a prstů

POKYNY PRO VYPRACOVÁNÍ:

Práce je zaměřena do oblasti detekce objektů v obraze/video sekvencích. Konkrétně se jedná o detekci a rozpoznávání gest rukou a trackování jednotlivých prstů. Prostudujte dostupné projekty, programy a funkce umožňující detekovat pohyby rukou a klasifikovat jejich gesta. Uvažujte dostupné knihovny pro počítačové vidění v jazycích Python, nebo C. Vytvořte snímací řetězec a ověřte fungování, příp. limity takovýchto řešení na PC.

Proveďte detailní testování vašeho snímacího systému. Aplikujte na dynamických video sekvencích s obtížně detekovatelnými pohyby, např. gesta raperů. Vytvořte popis navržených funkcí a zveřejněte je na GitHubu, či podobném úložišti open souce projektů. Dbejte na rošiřitelnost systému pro snadné vkládání nových detekujících gest.

DOPORUČENÁ LITERATURA:

[1] OpenCV [online]. 2018 [cit. 2018-05-16]. Dostupné z: http://opencv.org/

[2] Into Robotics. 9 OpenCV tutorials to detect and recognize hand gestures [online]. 2018 [cit. 2018-05-16]. Dostupné z: https://www.intorobotics.com/9-opencv-tutorials-hand-gesture-detection-recognition/

Termín zadání: 4.2.2019	***Termín odevzdání:*** 16.5.2019

Vedoucí práce: doc. Ing. Tomáš Frýza, Ph.D.
Konzultant:

prof. Ing. Tomáš Kratochvíl, Ph.D.
předseda oborové rady

ABSTRACT

This master's thesis is focused on hand gestures and finger detection in still images and video sequences. The thesis contains a summary of different approaches to hand gesture detections, advantages and disadvantages of each approach. The thesis also includes the realization of the platform independent application written in Python using OpenCV and PyTorch libraries, that can show a selected image or play a video sequence with highlighted recognized gestures.

KEYWORDS

Computer vision, hand detection, gesture recognition, image processing, video processing, OpenCV, PyTorch, Python, Deep Learning, convolutional neural networks, machine learning

ABSTRAKT

Diplomová práce je zaměřena na detekci a rozpoznání gest rukou a prstů ve statických obrazech i video sekvencích. Práce obsahuje shrnutí několika různých přístupů k samotné detekci a také jejich výhody i nevýhody. V práci je též obsažena realizace multiplatformní aplikace napsané v Pythonu s použitím knihoven OpenCV a PyTorch, která dokáže zobrazit vybraný obraz nebo přehrát video se zvýrazněním rozpoznaných gest.

KLÍČOVÁ SLOVA

Počítačové vidění, detekce rukou, rozpoznání gest, zpracování obrazu, zpracování videa, OpenCV, PyTorch, Python, Deep Learning, konvoluční neuronové sítě, strojové učení

ROZŠÍŘENÝ ABSTRAKT

Postupem času, tak jak to výpočetní výkon dovoluje oblast počítačového vidění nabírá na popularitě. A není se čemu divit, počítače i telefony jenž používáme každý den mají dostatek výpočetního výkonu pro analýzu obrazů a video sekvencí v reálném čase. Například naše mobilní telefony jsou ve většině případů rozpoznat tváře ve fotografiích a v některých případech pochopit i jednoduchá gesta rukou, jako například vzdálená spoušť pro pořízení fotografie, aniž bychom byli nuceni se dotknout telefonu. Počítače jsou velmi schopné těchto snadných detekcí, pokud vidí celé tváře nebo ruku, co ale v případě kdy ruce a prsty nejsou snadno viditelné? To je problém, pro který není snadné řešení.

Analýza gest rukou poskytuje další způsob pochopení lidského chování ve video sekvencích pro zrakově postižené, nebo způsob překladu znakové řeči na text. Dalšími příklady využití analýzy gest mohou být systémy ovládané pomocí gest v automobilovém průmyslu, nebo analýza neverbální komunikace mezi zločinci zachycenými na bezpečnostní kamery.

V této práci jsou představeny možné způsoby detekce rukou a samotných gest, které jsou následně popsány a porovnány včetně jejich kladů a záporů.

Hlavní zaměření práce je na tvorbu multiplatformní aplikace určené pro detekci rukou a rozpoznání gest. Základem této aplikace je programovací jazyk Python s knihovnami pro počítačové vidění.

Validace výsledků aplikace je provedena pomocí video sekvencí s rozdílnou obtížností viditelnosti rukou pořízenými za účelem testování aplikace a s využitím náhodných videí nalezených na internetu pro zjištění úspěšnosti detekce.

BRAVENEC, Tomáš. *Počítačové vidění a detekce gest rukou a prstů.* Brno, 2019. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/118437. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky. Vedoucí práce Tomáš Frýza.

DECLARATION

I declare that I have written the master's thesis titled "Computer vision and hand gestures detection and fingers tracking" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this master's thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation S11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Počítačové vidění a detekce gest rukou a prstů“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení S11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

ACKNOWLEDGEMENT

I would like to thank to the supervisor of this thesis doc. Ing. Tomáš Frýza, Ph.D. for professional guidance, consultations and work suggestions.

Brno ....

author's signature

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu doc. Ing. Tomáši Frýzovi, Ph.D. za odborné vedení, konzultace a podnětné návrhy k práci.

Brno ....

podpis autora

Contents

List of Figures

List of Tables

Listings

Introduction

For quite some time, the field of computer vision is rising in popularity. And there is no surprise, that computers and phones we use every day are powerful enough to analyze images and video sequences in real time. For example, our mobile phones are in most cases capable of recognizing our faces in photos and sometimes they can even understand some basic hand gestures for taking a picture without us even touching the phone. Computers are quite capable of these easy recognitions when they can see the whole face or hand, but what about cases when hands and fingers are not visible that well? This is quite an issue that is not that easy to handle.

Analysis of hand gestures is useful for providing another way of understanding what humans are doing in videos for visually impaired or blind people or translation of sign language to text. Another example could be gesture based controls of some systems in automotive industry, to analyze non-verbal communication of criminals caught on cameras etc.

This thesis will introduce multiple ways how hand gestures can be detected and the qualities and flaws of each approaches.

The main part of this thesis is focused on creating a hand gesture detection application and its functionality. The application is platform independent and written in Python with libraries for computer vision.

The evaluation of this application is done using video sequences with varying difficulty of visible hands captured specifically for testing this application and random videos found on the internet to see with how high success rates the application manages to do its job.

1 Hand and gesture recognition

To begin with gesture recognition, the hand must be detected first. There are a few approaches to finding the hand. These approaches differ from easier ones that are more susceptible to mistake and are not very robust, to the use of algorithms that are hard to confuse. Using these kinds of algorithms have a price: they may need significantly more processing power.

1.1 Contour analysis

The easiest method is analyzing a grayscale image (Figure 1.1 a)), as there is only one condition needed to use in order to get thresholded image (Figure 1.1 b)), if the pixel is part of a hand or if it is not. From a thresholded image, it is easy to get the contour of a hand and convex hull around the hand (Figure 1.1 c)). Then by the number of convex defects in the convex hull (Figure 1.1 d)) and the distances of these defects from each other, figure out what fingers are extended, and which are collapsed. [1]

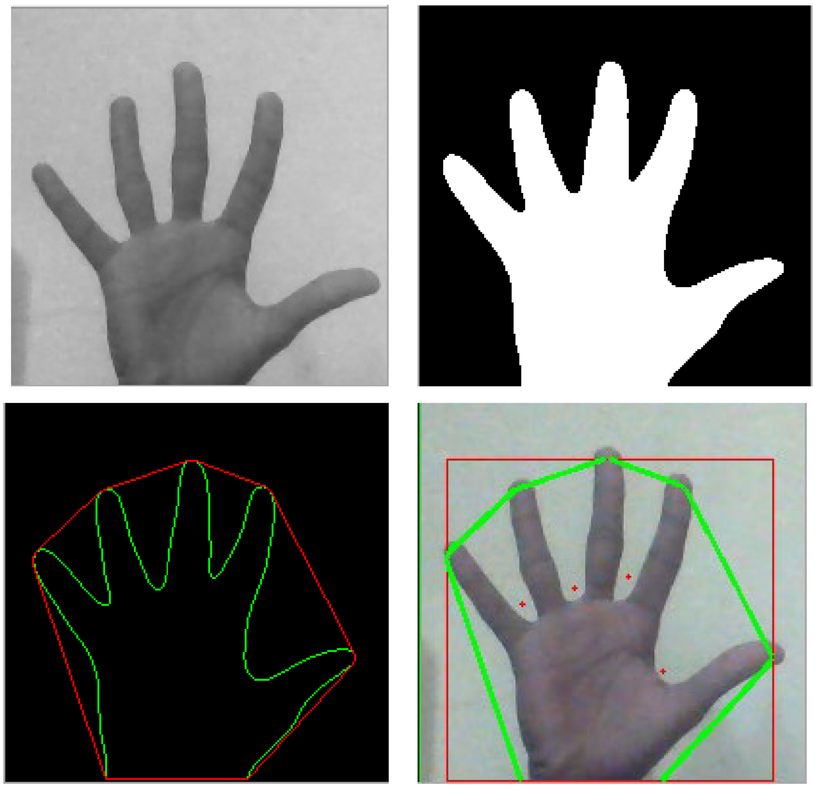


Figure 1.1 Contour analysis a) Detected hand b) Thresholded contour c) Detected edge and convex hull d) Detected convex hull defects [1]

One of the issues with this approach is the fact, that the hand must be easily separable from the background. This can be done either by thresholding, or other more complex approaches, such as separation by skin color, background subtraction etc. This also means that the algorithm can become easily confused when it comes to unusual background patterns or just a sharp change in lighting, causing a rapid change in skin color.

The biggest issue of this method is that it is incapable of recognizing gestures if the palm is not facing the camera directly, it also easily fails to identify collapsed fingers when the hand rotates. Because of these issues, this method is not suitable for detection of more complex gestures or during worse visibility of the hand.

1.2 Curve fitting

Curve fitting, also known as snakes, is one of the less usable methods for hand detection, as it is more suitable for hand tracking. This approach needs some initial guess or cooperation from the person using this method [2]. The initial guess could be made by the person matching the curve on screen with his hand presented in Figure 1.2. After a certain threshold of similarity is passed, tracking can start, and the curve is adjusted from the previous frame to match the outline of the hand. As it needs an initial guess, this method is not a very good choice for detection in video sequences, that are not prepared for detection using this approach.

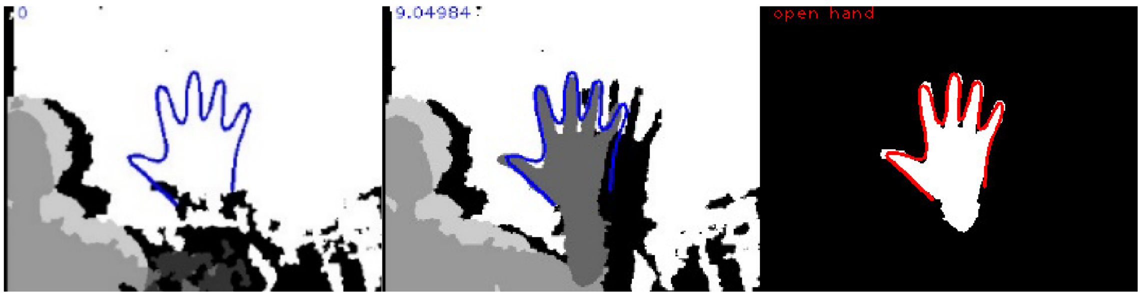


Figure 1.2 Curve fitting [2]

1.3 Model fitting

Another approach to gesture recognition is creating a virtual model of a hand, composed from “bones” and “joints” like in a real hand. At first, it uses contour analysis or depth image analysis if it is available in the source to detect fingertips, followed by connecting detected fingertips to the model, so the model joints can bend and recreate the gesture in a virtual environment [3]. An image of a virtual model is in Figure 1.3.

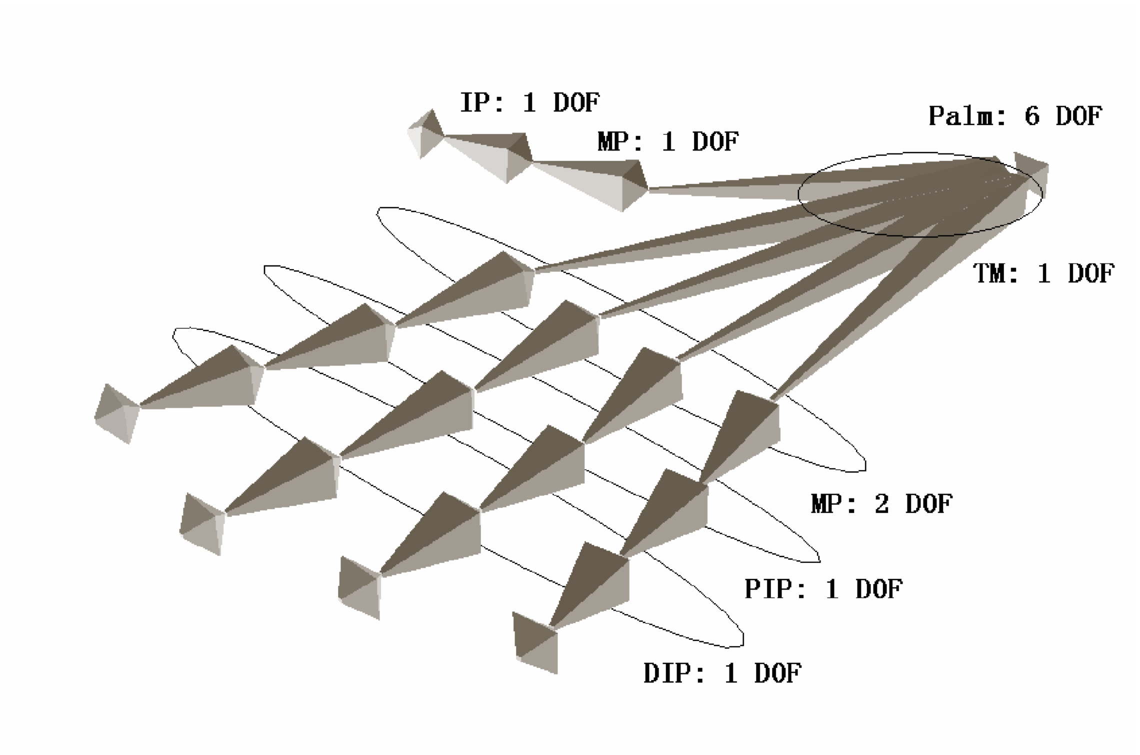


Figure 1.3 Skeleton of the hand model [3]

The problem here is again, that from certain angles, the model may not be properly connected to fingertips, which will cause unpredictable behavior, like guessing an incorrect gesture or losing focus on the hand itself [4]. On the other hand, systems like this could be easily modified to recognize more gestures by simply adding another configuration of the hand model with a description of how the fingers are bent.

1.3.1 Multiple angle model fitting

To make model fitting more accurate, more cameras can be used to capture hand movements from different angles, so the fingertips are always visible and can be connected to the model's end points at any moment.

This extension of model fitting is not usable for common video sequences, as they are not shot from different angles at the same time. This means that even though multiple angle model fitting can be more accurate, it is more useful in real time translation of sign language, where the person stands or sits in front of couple of cameras.

1.4 Depth based hand detection

Another way of detecting hands, not from existing video sequences but rather in real time is based around capturing not only color but also depth. This can be done using special capturing devices like Microsoft Kinect its capture of scene depth in the image of a human's arm is in Figure 1.4. From this image it is obvious that the depth levels have lower resolution than color, which means that objects like a hand will mostly be on one or two neighboring distance layers, making hand detection significantly easier than from a color image.

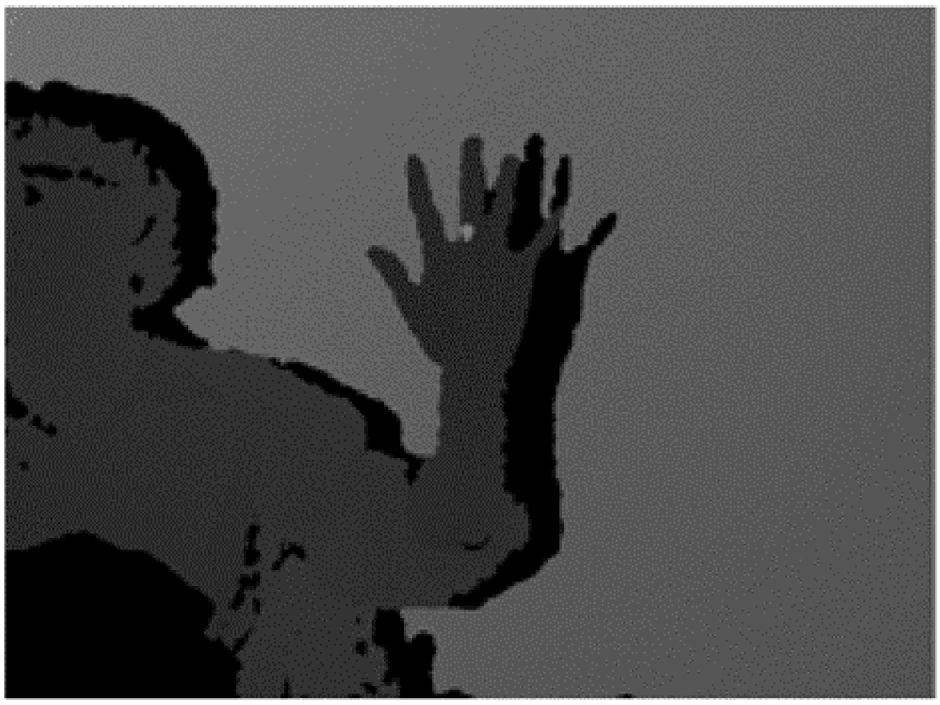


Figure 1.4 Microsoft Kinects depth capture [5]

The approach using depth to detect a hand has its positives, it does not care about background or lighting [5]. Its downside is if there is another object at the same distance as the hand in the depth map; the hand and object could blend together, though it is not that difficult to split it again by combining depth and color layers. This way the hand can be separated by selecting only the skin colored part of the depth layer on which the hand is located.

As is obvious from the fact that it needs data captured with a depth sensor, this approach is unusable when it comes to hand detection from normal images and video sequences.

1.5 Deep learning

In the last couple of years, the field of machine learning started to gain in popularity, as the main limitation in the past, the processing power, is more accessible than ever before. Considering the advancements in general purpose computing using graphics processors, the times needed to train artificial neural networks on a regular computer at home are comparable to times that were needed just a few years back on a supercomputer.

The deep learning itself is a subset of machine learning, that today makes use mostly of deep neural networks (these networks contain more than two layers of non-linear processing between input and output layers) to learn from huge amounts of data to solve problems without being explicitly programmed to do that. Typical representation of a deep learning neural network is in Figure 1.5. There are also other algorithms like recurrent neural networks, deep belief networks or deep Boltzmann machines that are part of deep learning, but they are not as widely used as deep learning neural networks. [6]

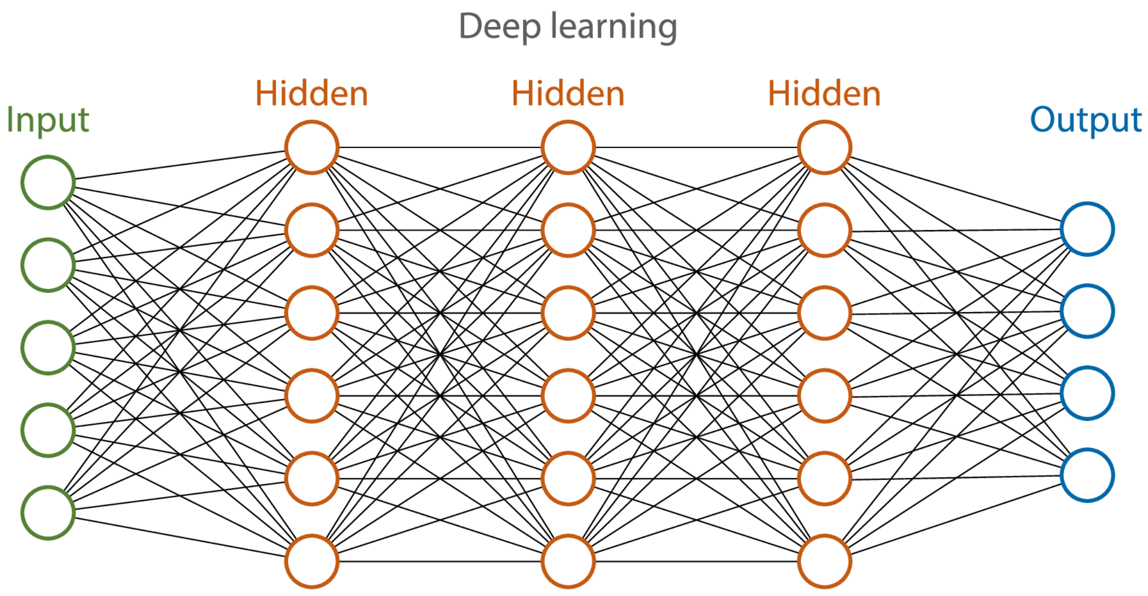


Figure 1.5 General deep learning neural network [7]

1.5.1 Convolutional neural networks

When it comes to image processing using deep learning, the most frequently used variant of a neural network is a convolutional neural network. This kind of network contains a couple of convolutional layers. The filters of these layers are acquired during the training process of the network and with appropriate training data. After the convolutional layers usually comes the pooling layer to reduce the amount of data for the next layer. The next layer is usually a single flattening layer followed by classic fully connected layers [8]. Architecture of the convolutional neural network is in Figure 1.6.

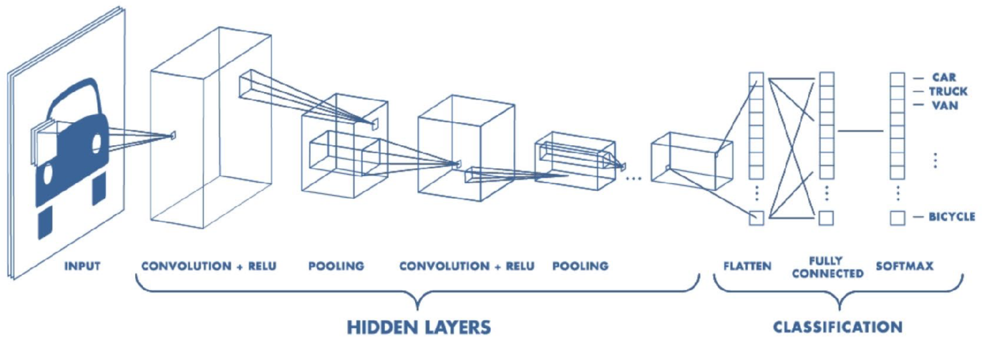


Figure 1.6 Architecture of convolutional neural network [9]

The reason for using convolutional neural networks for hand detection is simple: with correct training data, the network can learn itself what to look for, what is important and what is not. Of course, this is not based only on the training data, but on the model of the neural network. This means that every neural network is built differently, from a different number of layers and neurons in each layer.

Almost every convolutional neural network that works with images takes an input image in RGB color space. This is because after experimentations with other color spaces like HSV, LUV and other ones used in computer vision applications, the network which trained on images in RGB color space provided higher accuracy of predictions than the same network trained on images in different color space [10].

Convolutional neural networks could also be divided into two different groups, depending on the type of input and output. If the input image is a single object in the center of the image and the neural network is supposed to predict what kind of object it is, this category is called classifiers. On the other hand, if the image contains many different objects all over the place, and the neural networks output is supposed to be a prediction of bounding boxes and what kind of object is in each bounding box, these neural networks are called detectors.

2 Realization

The whole result of this thesis can be divided into parts. First, there is a need to detect a hand in the image before it can be recognized as a gesture. So, the second part is obviously rule based gesture classification, which should also be easily expandable. In the end the working application should be wrapped in multiplatform easy to use graphical user interface with easy gesture addition, an image and video viewer with detected gesture highlighting and logging of detected gestures into a file.

2.1 Hand detection

When it comes to detecting hands in various positions and different environments, it is quite difficult to assess some rules to detect hands accurately. For example, fist looks very different from open palm. For this very reason the approach of deep learning was selected. This process consists of creating or finding and adjusting existing datasets for this very purpose. With the prepared dataset the next step is creating a neural network from scratch or using an already existing and tested architecture and retraining it for the purpose of hand detection.

2.1.1 Oxford hands dataset

Before training a neural network, there must exist some data to train the network on. The first choice for a dataset was oxford hands dataset [11], as its already annotated. Its diverse images of hands in very different situations, poses etc. seemed like a great way to train a neural network that would generalize well and provide good results. This was not the case, even after training on this dataset for several days, the results were not good. This could have been because of the lower resolution of the images; a few of those images from the dataset are in Figure 2.1.

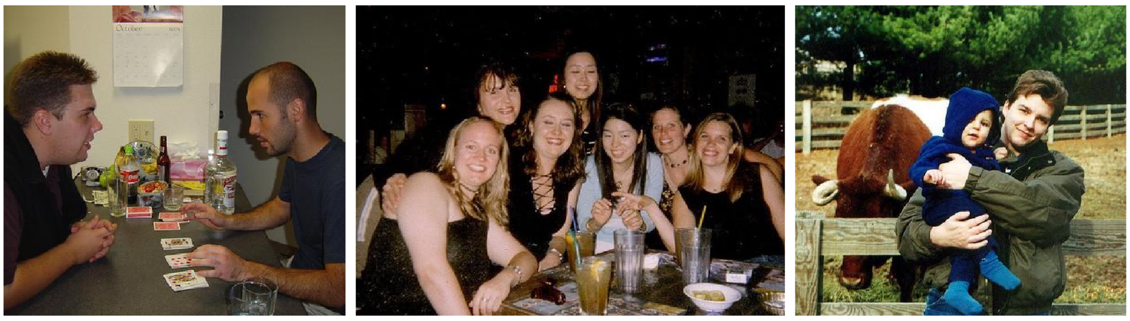


Figure 2.1 Images included in Oxford hands dataset [11]

2.1.2 EgoHands dataset

For the purpose of this thesis the EgoHands dataset [12] was chosen, due to its high-resolution images with already existing annotations and wide range of hand poses in a few different locations. This dataset contains 4400 training and 400 validation images as shown in Figure 2.2. Each picture includes at least one hand and maximally 4 hands, and these hands are not always clearly visible, as they can be obstructed from the camera by other hands or objects in the scene.

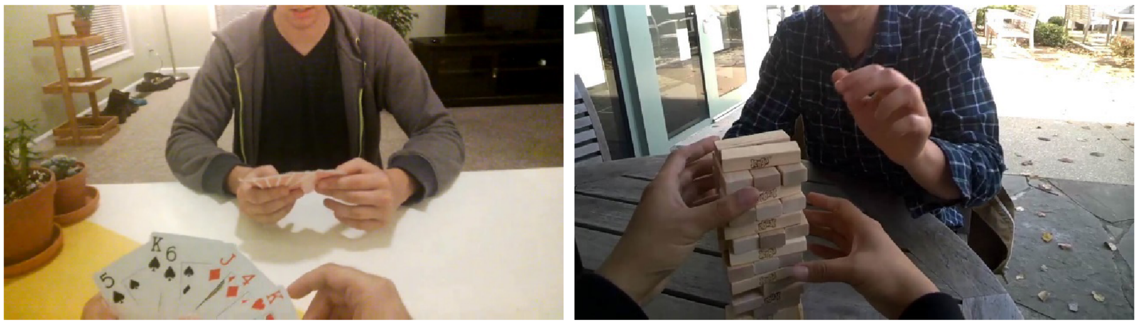


Figure 2.2 Images included in EgoHands dataset [12]

The annotations included in the dataset also had to be modified, as the original annotations were differentiating between left and right hand and even from which point of view is the hand captured. Annotations were also in different format than was needed for training the selected neural network. Conversion and annotation modification were done using simple python script. Modified annotations are shown in Figure 2.3 as yellow rectangles around hands.

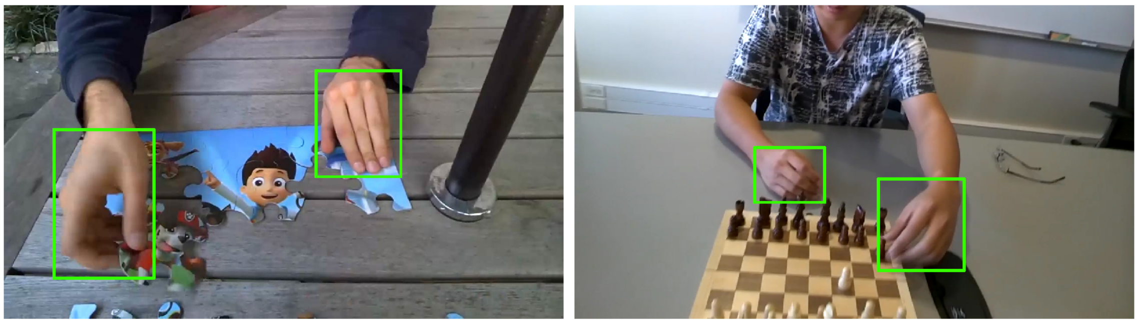


Figure 2.3 Modified annotations in EgoHands dataset [12]

2.1.3 New Zealand Sign Language Dictionary

The Ego Hands dataset is not completely universal, as it misses some hand gestures. Due to this reason the dataset was expanded with images from New Zealand Sign Language dictionary [13]. As the name suggests, this dictionary contains video sequences of humans presenting different signs of sign language. As there are a lot of videos in the dictionary, only 10 videos were used for dataset expansion. From these videos, a total of 515 frames were taken for training and 231 frames for validation. The videos provide quite a big range of different gestures. This diversity helps in recognition of hand poses that are not present in the EgoHands dataset. Unlike images from EgoHands dataset, these images have a very similar background, which means it is not a very good idea to train the neural network with images only from this dictionary. Some of the hand gestures contained in videos from New Zealand Sign Language dictionary are presented in Figure 2.4.

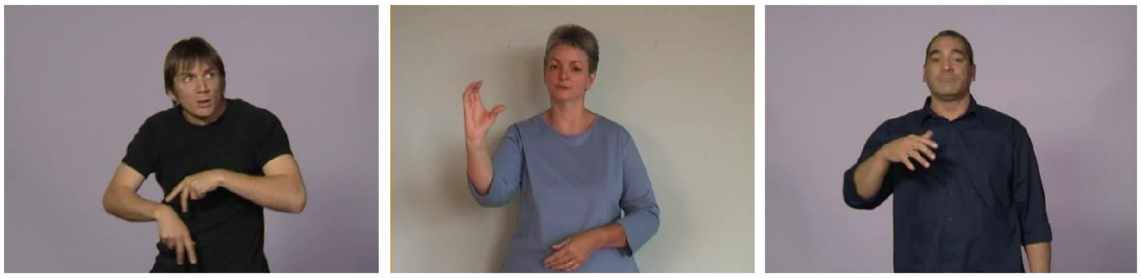


Figure 2.4 Images from videos in New Zealand Sign Language dictionary [13]

Unlike for images from EgoHands dataset, for frames from videos in this dataset the annotations did not exist. Because of that these had to be created from scratch. Few images with displayed bounding boxes are in Figure 2.5.

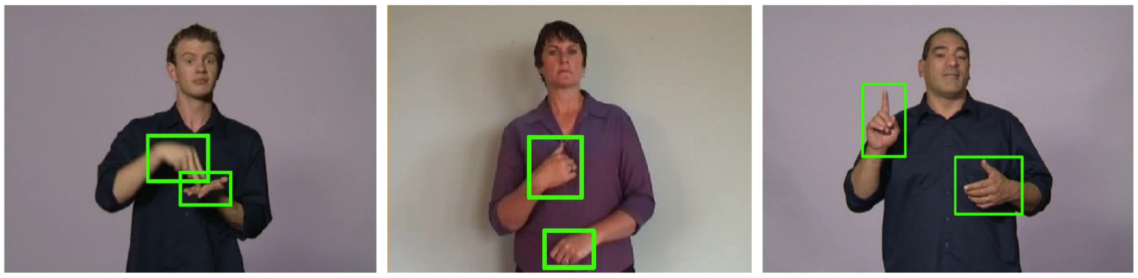


Figure 2.5 Annotated images from New Zealand Sign Language dictionary [13]

2.1.4 MPII Human pose estimation dataset

To make the training dataset even more robust, small part of the MPII Human pose estimation dataset [14], was used. Because the dataset is not focused on hands, they are not visible on many pictures from this dataset. For training 483 images were selected and 215 images for validation. On the used images, people are doing many different activities in a wide range of environments, from playing musical instruments or cooking in the kitchen to working with power tools. Some of the images even have people wearing gloves to farther improve chances of hand detection in difficult conditions as the detector cannot rely on skin color. Example of images from this dataset are in Figure 2.6.

Figure 2.6 Images contained in the MPII Human Pose dataset [14]

Just like with images from the New Zealand Sign Language dictionary, the images from the MPII Human Pose dataset do not have annotations of hands needed for training neural networks and these annotations had to be created. Images from this dataset with showed bounding boxes are in Figure 2.7.

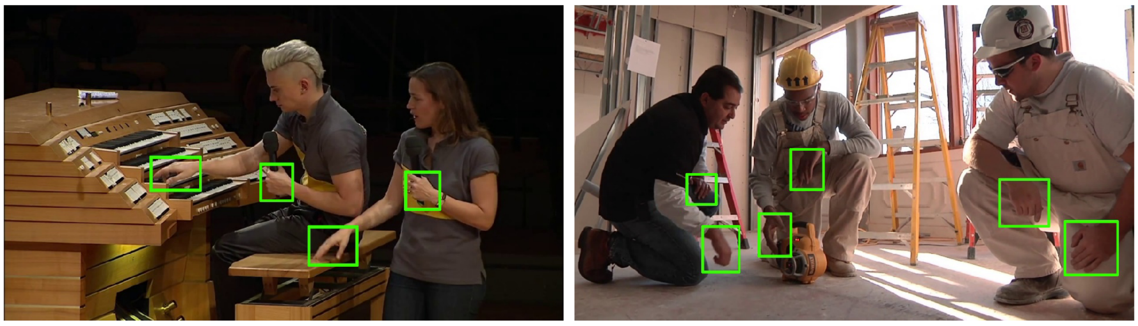


Figure 2.7 Images from MPII Human Pose dataset with annotations [14]

2.1.5 Neural network – YOLOv2

Instead of creating and testing new architectures of neural networks the architecture YOLOv2 [15] (You Only Look Once v2) was selected. This means, that instead of testing if the neural networks architecture is designed correctly for predicting bounding boxes, it just needed to be adjusted to only look for one class and then be retrained for detection using a previously selected dataset.

YOLOv2 is a fully convolutional neural network created out of 23 convolutional layers, 5 pooling layers, 2 routing layers, a reorganization layer and with a single detection layer. The relatively low depth of the network makes it work very well in real time processing.

The function of convolutional layers is obvious from their name. Pooling layers on the other hand might not be that obvious. These layers reduce the spatial dimensions of their input but keep their depth. Usually this reduction is done by a factor of 2. There are two different versions of pooling layers, YOLOv2 uses max pooling, which means that the input is divided into small squares, where dimensions of the square are the factor of the pooling layer. For YOLOv2, the input is divided into grids of 2 by 2 values and on the output of the layer is only the highest value from this grid. The other variant of this layer uses average pooling which, as its name suggests, the output produces an average value for each of the input grids.

Reorganization layers have a similar function to pooling layers, as these layers also change the dimensions of their input, but unlike pooling layers, they also change the depth and keep all the input values. As the name of the layers hint, the input is reorganized in a way that a single channel on the input will become more channels on the output depending on the settings of the layer. The reorganization layer in YOLOv2 uses a stride of 2, which means that the spatial dimensions in both directions would be halved and depth would grow four times.

Because the neural network uses routing layers, it means that the network actually does not run all of the layers sequentially, but rather works up to the routing layer sequentially and then takes the output of the layer to which the routing layer points to instead of taking output of the layer that preceeds it. Another option is if the routing layer points to multiple different layers, in that case all the outputs of layers pointed to are concatenated. The function of routing layers is clearly displayed in graphical representation of YOLOv2 with purple arrows in Figure 2.8. In this figure the horizontal numbers bellow layers signify how many convolutional filters are present in that layer, and angled numbers signify the spatial dimensions of all the layers since the last pooling or reorganization layer. Input dimensions for each convolutional layer are the dimensions of the previous layer times the amount of filters in that layer.

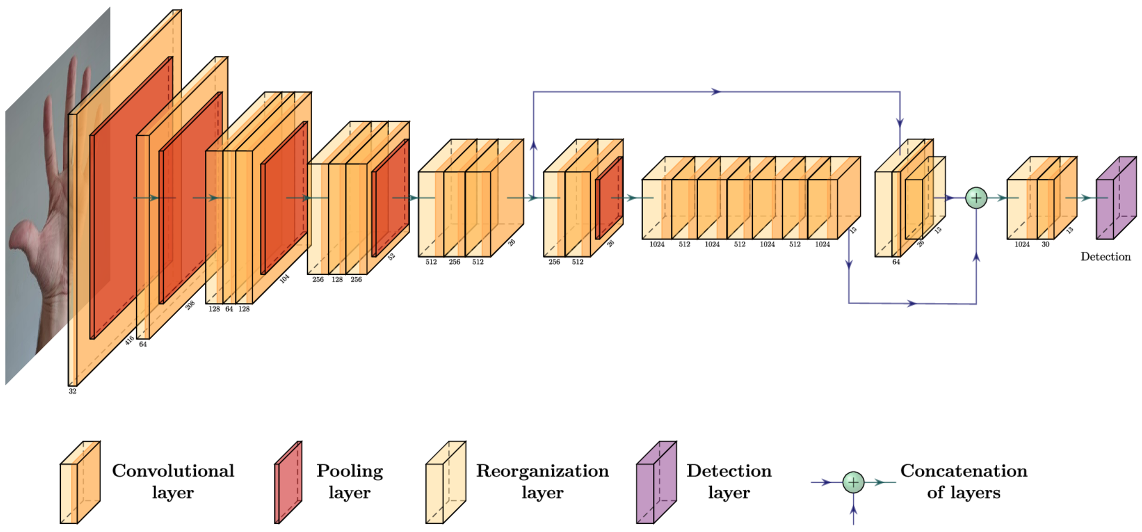


Figure 2.8 YOLOv2 network architecture

2.1.6 Modifying the YOLOv2 neural network

The actual network architecture is defined in the configuration file and the weights file. The original network's configuration file and its pretrained weights can be downloaded from the website [16]. The configuration file had to be slightly adjusted to look for only one class, which meant changing the number of classes in the detection layer and changing the filter count in the preceding convolutional layer to the appropriate amount for the number of classes in detection layers. These changes in the last two layers in the configuration file are in bold in Listing 2.1.

Listing 2.1 Modifications of YOLOv2 neural network's configuration

```
[convolutional]
size=1
stride=1
pad=1
filters=30
activation=linear

[region]
anchors=1.3221,1.73145, 3.19275,4.00944, 5.05587,8.09892, 9.47112,4.84053,
11.2364,10.0071
bias_match=1
classes=1
coords=4
num=5
softmax=1
jitter=.3
rescore=1
```

The filter count in the last convolutional layer before the detection layer is calculated from the count of *classes* that the neural network is supposed to be detecting by the equation [17]:

$$filters = (classes + coords\ +\ 1)\ \cdot\ num \tag{2.1}$$

Where *coords* represent the four attributes of bounding boxes (x, y, width and height), constant 1 is for confidence with which the object is detected and *num* stands for the number of anchor pairs in the region layer.

Anchor pairs represent initial sizes of bounding boxes in the detection layer, which is for YOLOv2 13x13 pixels, before the closest one to the detected object is resized. Anchors can be calculated from training data using K-Means clustering, but for detection of hands it is not necessary as hands can be in pretty much in any shape, just like all the different objects in the default networks configuration.

2.1.7 Training neural network

With the configuration file modified for the need of hand detection, the only thing remaining is training the network itself. For this purpose, the interpreter for YOLO based neural networks for deep learning framework PyTorch [18] was used [19]. As it is just an interpreter for training and detecting there was no need to change the code, because the changes to the neural networks were in their configuration files.

The neural network was trained for 150 epochs, where one epoch means a single pass through all the training data. The training was stopped after 150 epochs passed as the training loss and validation did not change much since epoch 100. Used weights were from training epoch 100 to prevent the issue of overfitting, that is a state of a neural network in which the network learned exact details of images and did not generalize very well or at all, even though the predictions are great on the training set, predictions on never before seen data are poor.

Training loss can be calculated in many ways. In the case of YOLOv2, it is divided into three components. The first is coordination loss, which represents how wrong the network was in detecting the location of the object in the image from the ground truth. The second component is confidence loss, which represents how much the neural network is sure about the detection. The last of the three errors, error in object classification, is calculated as binary cross entropy. As the modified network detects only a single object, this error is equal to zero. The final training loss is calculated as the sum of previously mentioned errors. Training loss of the neural network after each training epoch is in Figure 2.9.

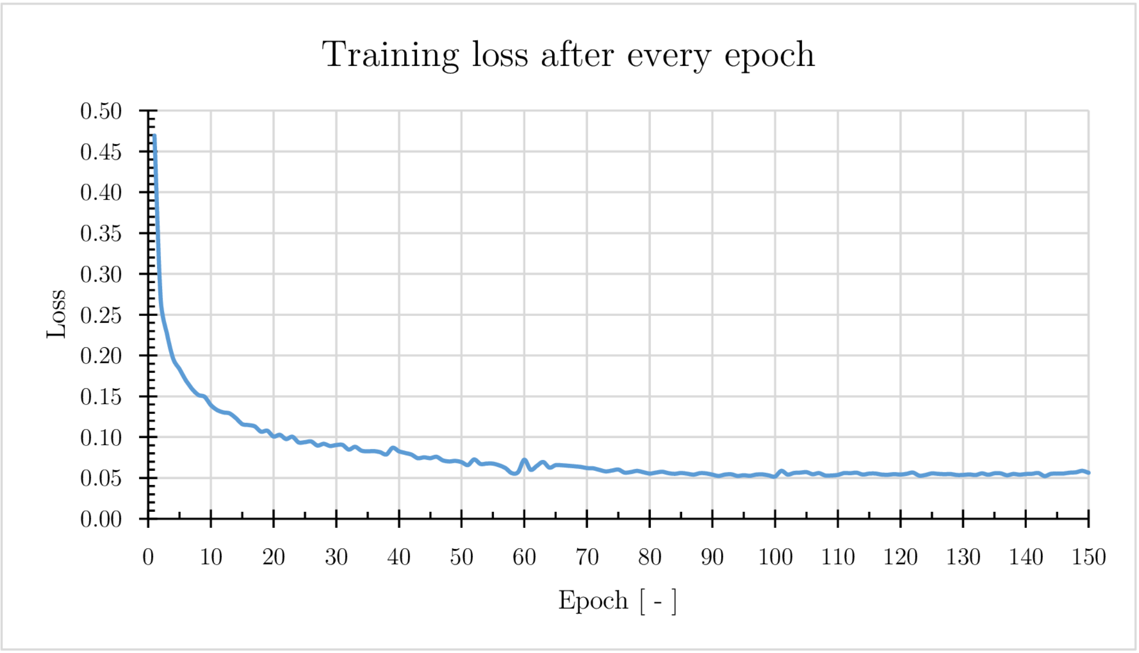


Figure 2.9 Training loss after each epoch

After a set number of epochs, the validation process was started. For the purpose of higher precision of displaying these data, the validation process was run after every epoch. This process consists of comparing ground truth bounding boxes and bounding boxes predicted by the network on images not used for training the network. The decision if the network predicted correctly is done using intersection over union (IoU) between prediction and ground truth calculated by equation (2.2). If the ratio of intersection over union is higher than the set threshold, in this case 0.5, then the network's predictions are accepted as correct.

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{2.2}$$

Figure 2.10 visually presents the area of union and area of overlap with red overlay in an image of detected hand (blue rectangle) and ground truth (green rectangle) for the same hand.

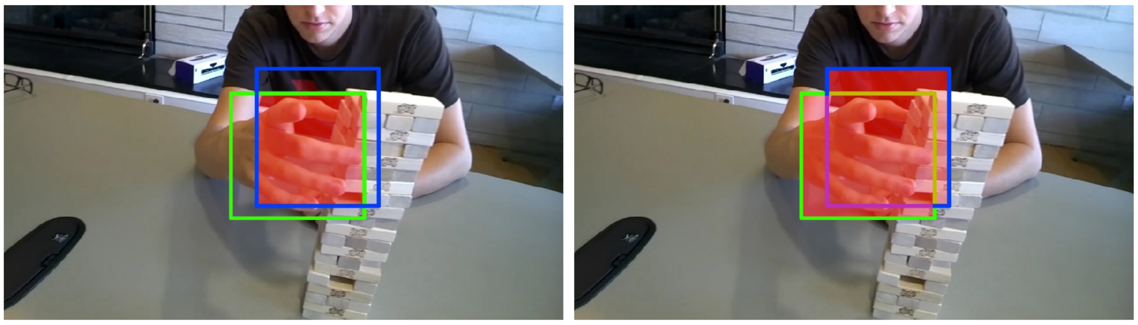


Figure 2.10 Area of Overlap and Area of Union

During validation, three values defining the accuracy of the network are calculated, two of these values are $Precision$ and $Recall$ [20], [21]. $Precision$ is defined as:

$$Precision = \frac{TP}{TP+FP} \tag{2.3}$$

Where TP means True Positive and corresponds to the amount of correct detections and FP stands for False Positive and matches the amount of incorrect predictions. If the neural network did not make incorrect predictions, the $Precision$ of the network would be 1.

$Recall$ is defined similarly to precision but instead of incorrect predictions, the number of undetected objects from the ground truth image is used in the equation:

$$Recall = \frac{TP}{TP+FN} \tag{2.4}$$

Where FN stands for False Negative and represents the mentioned undetected number of hands in an image. If $Recall$ equals 1, the neural network detected all the objects in the validation dataset. To better understand what $Precision$ and $Recall$ mean, Figure 2.11 explains it very well.

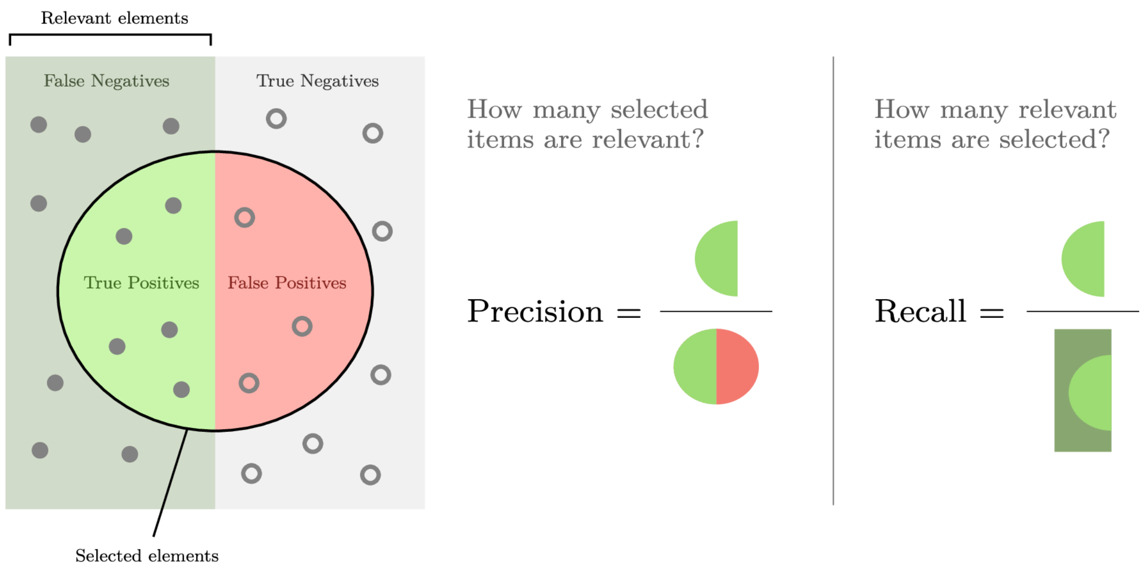


Figure 2.11 Graphical meaning of $Precision$ and $Recall$ [22]

The last of the three values describing the result of validation is $F\text{-}score$, it is used to measure validation accuracy as a whole and it is calculated as harmonic mean of $Precision$ and $Recall$ [23]:

$$F\text{-}score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.5}$$

From Figure 2.12 is clearly visible that training produced the biggest changes in precision in the first epochs. That is because at the beginning of training, the network's weights were trained for detecting multiple different objects and the changes had to be big to start detecting hands, this is done by used settings in the configuration file for training. At first, the network needs to learn more aggressively. A typical starting value of learning rate is 0.001 to quickly learn basic features of hands. After some time, in this case 40 000 steps, which equals to 66 epochs with training batch size of 8, the learning rate was lowered to 0.0001 to prevent too big of a jump in weights values and start learning finer features of hands. The learning rate was lowered again after epoch 95 to further decrease the difference between weights and slow down learning, which was not necessary as the validation results stayed pretty much the same since epoch 90.

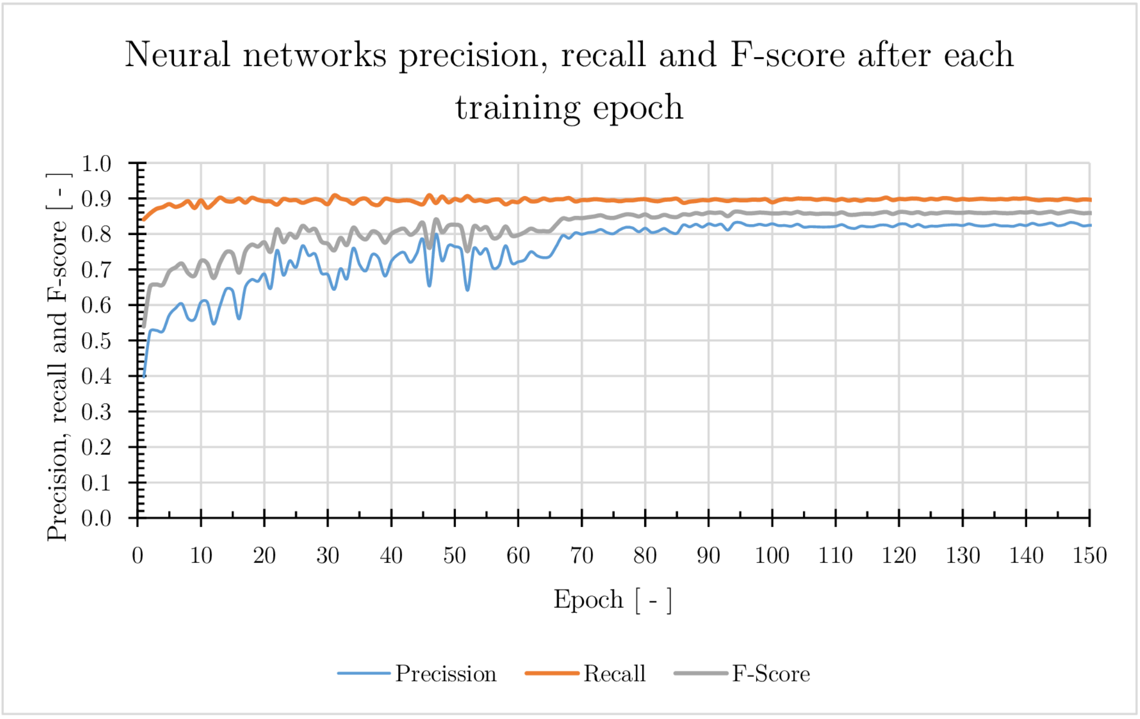


Figure 2.12 *Precision*, *recall* and *F-score* after each training epoch

2.2 Finger detection

For finger detection a pretrained neural network trained for detection of hand key points was used. This neural network is part of the project OpenPose [24]. OpenPose uses a neural network architecture called convolutional pose machines [25] to detect key points of specific parts of the human body. Out of the four networks, used convolutional pose machine is focused on hand key points detection [26]. This network takes as an input a single image, in which is supposed to be a single hand. From this image the network on the output generates 22 heatmaps. An example image from which these heatmaps are generated and heatmaps for each key point of index finger are in Figure 2.13.

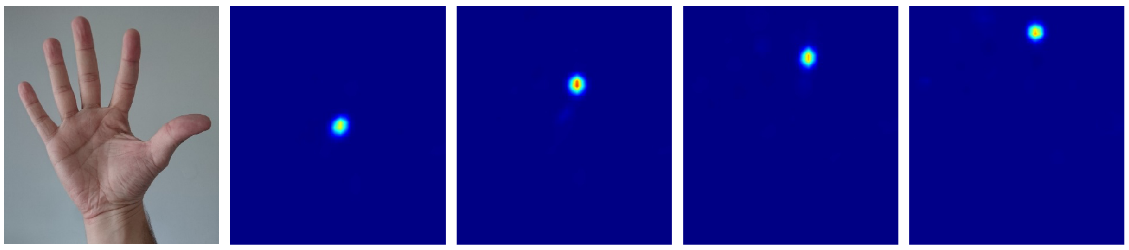


Figure 2.13 Heatmaps generated for index finger

Each of the first 21 heatmaps represent an approximate position of a single key point in the image and the last heatmap represents the background. From these heatmaps can be easily found the location of maximum, which represents the location of the found key point. After all the key points from all heatmaps are found, the gesture recognition process can be started. The detected key points are numbered and connected to create hand skeleton and all the heatmaps combined into one can be seen in Figure 2.14.

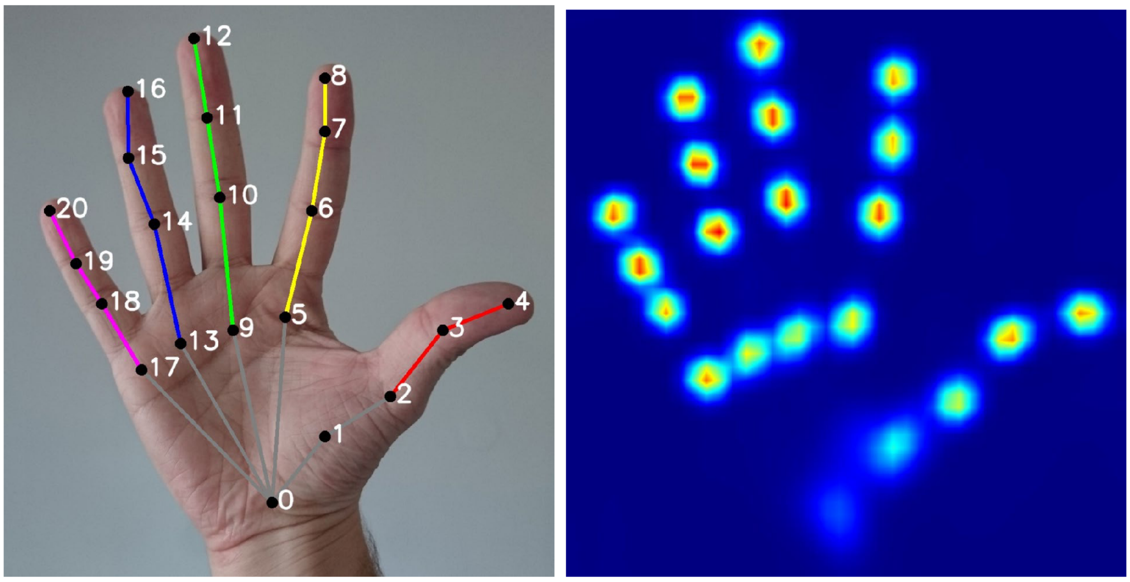


Figure 2.14 Hand key points and combined heatmap generated by CNN

One of the biggest downsides of OpenPose is the fact that it is trained to only detect key points of right hands. That means that the results of detection for the left hand are much worse than when using detection for the right hand. This issue can be avoided by flipping the image horizontally to effectively run the model over the right hand, followed up by flipping the detected key points around again to fit the left hand [26].

2.2.1 Left hand detection

As mentioned, the hand key point detector was not trained to detect key points of left hands [26]. If the hand is right or left, there is no issue in running the detection once with the only difference being the flipped input and, in the end, flipping the key points back. But if the hand could be either left or right, it is a different story. To battle this issue without doubling up the time needed for key point detection by running the detection again over the horizontally flipped input image if the results are not as expected, the flipped image is being concatenated to the original input image as is in Figure 2.15 and the detection is being run over this modified input. This way the time needed for detection stays approximately the same as the network runs only once.

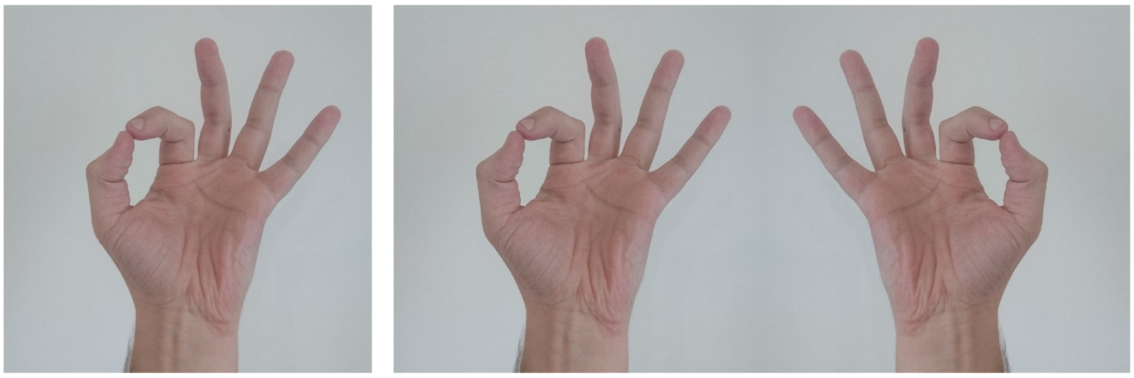


Figure 2.15 Normal and modified input for hand key point detection

While running the detection over input that contains two hands, more post processing is needed to correctly identify if the hand is left or right and to get the correct key points out of the detection output. For this is used the fact that the network generates heatmaps, instead of just coordinates, that helps in the fact that there are peaks in the heatmaps for finger detected in the original and in the flipped half of the network's input image. All the heatmaps of this modified input combined are in Figure 2.16. From the heatmap it is visible that in the right half of the image with flipped left hand, the values are generally higher and more pronounced. The reason for that is that the network is not as sure about the detection for a left hand. For the same reason, the detection for a right hand produces better results. It all comes down to the fact that the network was trained only over right hands.

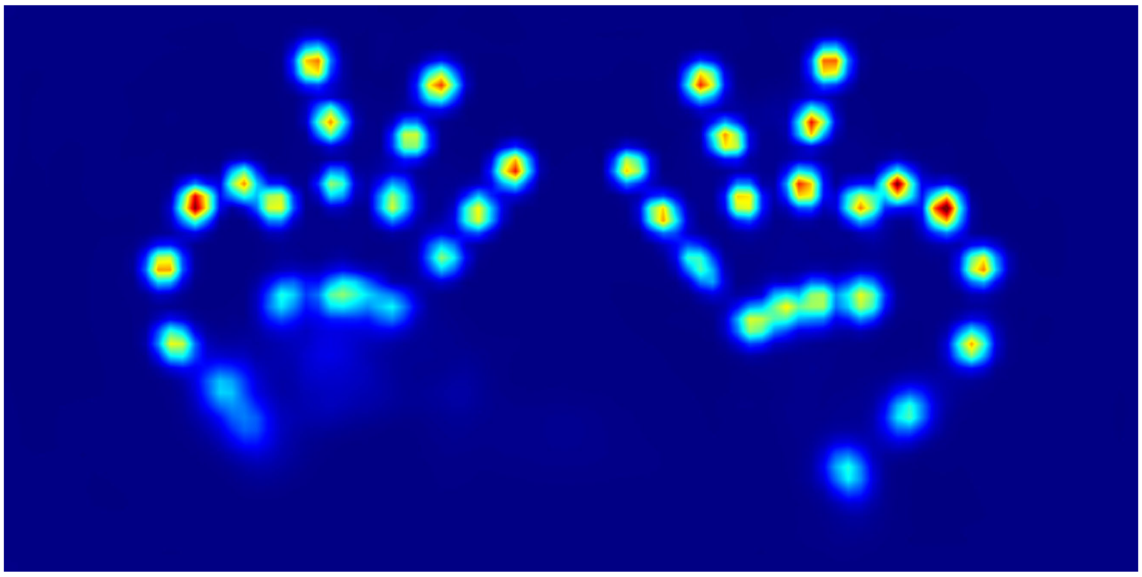


Figure 2.16 Combined heatmaps generated by CNN out of modified input

The fact that the values are generally higher in one half of the output heatmaps, it can then be used to deduce if the higher values are in the original half of the image or the flipped one. If there are more higher values in the original half, the hand is considered as right and similarly, if more of the higher values are in the flipped side of the heatmap, the hand is considered as left. This approach is also beneficial because it can help later in gesture recognition with limiting gestures only to one hand.

2.2.2 Model conversion

The hand detector uses the interpreter of the Darknet framework written on top of PyTorch, on the other hand the hand key point detector uses Caffe as it's backend framework. As PyTorch provides easy installation using python's package installer pip, while supporting all versions of python and CUDA, Caffe needs to be compiled before usage. Due to this, the hand key point detector was converted to use PyTorch. For the conversion, the utility MMdnn developed by Microsoft, was used for converting the model between different deep learning frameworks [27].

2.3 Gesture recognition

From the detected key points, it is possible to get all the information needed for classifying gestures. As the key points are in a list of vertical and horizontal coordinates the distances between key points can be used to determine the pose of each finger and the angle of the line given by these points and the horizontal axis indicates the direction the hand or finger is pointed to.

Distance d between the key points p and q can be calculated as a Euclidean distance for 2-dimensional space:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} \tag{2.6}$$

Where p_1 and q_1 are horizontal coordinates of both key points and p_2 and q_2 represent vertical coordinates. In implementation, for most of the distance comparisons, one of the compared distances is weighted by a constant C, to account for different lengths of finger sections. Except of distances, the angle the thumb or the hand is in relative to the horizontal axis is also used. All the poses the finger can be in or the direction the hand can point in, can then be written into gesture definition file.

All the states of fingers, that are analyzed have two options over those needed for complete gesture description. State *none* allows the finger not to be detected while still being able to correctly classify the gesture. State *any* then completely skips the category during the matching of detected hand pose to one of the predefined gestures. That means the gesture does not need the finger detected, or in any specific state.

If the hand pose does not match any of the predefined gestures, the output of pose to gesture matching returns *unknown*, instead of the gesture name. On the other hand, if the hand pose fit more than one gesture, the one that comes alphabetically first is returned.

2.3.1 Finger bends

The first thing needed for gesture classification is the state of bend for each finger. For all fingers except for thumb, which does not have a state of full bend, the bend state is defined in the gesture definition file by one of three states:

- straight
- partly_bent
- fully_bent

As is obvious, straight state means that the distance between the base and the tip of the finger is approximately the same as the sum of distances between all the four key points defining a finger. For the thumb, a partial bend is selected if it does not pass as straight. The condition that needs to be met for a finger to be considered straight is defined by equation (2.7) below. The distances that need to be of similar value for passing as straight are shown as blue lines in Figure 2.17 a):

$$C \cdot \sum_{i=n}^{n+2} d(k_i, k_{i+1}) \sim d(k_n, k_{n+3}) \tag{2.7}$$

Where n is the index of a key point at the base of a finger and can have a value of 5, 9, 13 or 17 and k_i is a key point with an index i. This is not possible to use for the thumb as it has only two sections in a finger. Due to this the equation needed to be modified:

$$C \cdot \sum_{i=2}^{3} d(k_i, k_{i+1}) \sim d(k_2, k_4) \tag{2.8}$$

Because the thumb is calculated only once and key points do not change indexes, the equation uses fixed index numbering for the distance between the base and the tip of the thumb.

If the finger does not pass the condition for being in the straight state, the type of bend is chosen with another distance comparison. Full bend uses a comparison of distances between the base and the tip of a finger and the length of the middle out of the three links defining a finger. If the base to the tip distance is smaller than the length of the middle link, the bend state is considered as fully bent, otherwise the partial bent state is selected. This can be described with inequation (2.9). Compared distances for a partial bent are shown with blue lines in Figure 2.17 b) and for full bent in Figure 2.17 c).

$$C \cdot d(k_n, k_{n+3}) < d(k_{n+1}, k_{n+2}) \tag{2.9}$$

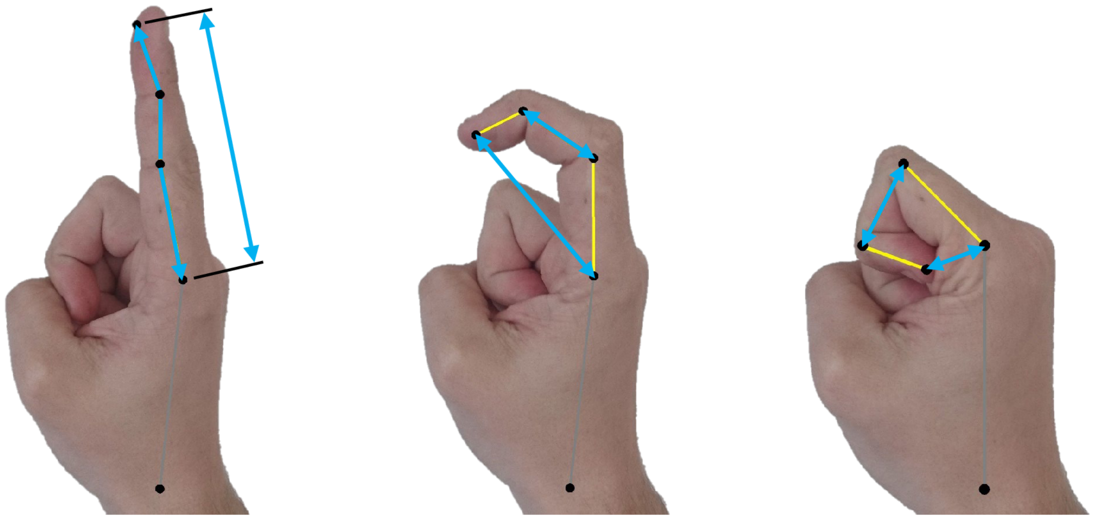


Figure 2.17 Finger bend states a) straight b) partly bent c) fully bent

2.3.2 Finger spread

Another important parameter needed for gesture recognition signifies how far the fingertips are spread from each other. This parameter is just like finger bend state calculated using the distance between two points and is always defined for two neighboring fingers. In the gesture definition file, the spread of fingers is defined with one out of two possible states:

- far
- close

The state is decided from comparing distances between bases and the tips of fingers, except for the gap between the little finger and the ring finger, as the tip of the little finger is approximately one section of a finger lower than the tip of ring finger. Due to this, for calculating the distance between these fingers, the bottom part of the last link of the ring finger is used, instead of its tip. The state is then decided depending on the result of inequation:

$$C \cdot d(k_n, k_{n+4}) < d(k_{n+3}, k_{n+7}) \tag{2.10}$$

Where k_n stands for the key point at the base of the index, middle or ring finger. In the case where the distance between the fingertips is bigger than the distance between their bases, as is visible in Figure 2.18 a), the far state is selected. If the tips of neighboring fingers are nearly the same or a shorter distance from each other than bases as can be seen in Figure 2.18 b), the close state is selected.

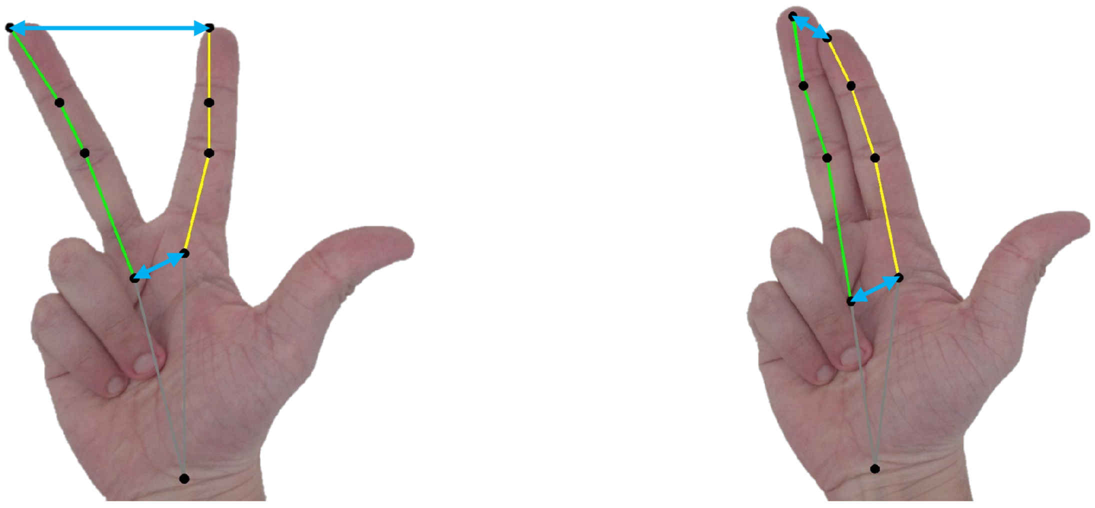


Figure 2.18 Finger spread states a) far b) close

2.3.3 Thumb position

Next on the list of finger poses is the position of the thumb. The thumb in general can be in three different poses against the palm, these poses are in gesture definition named as:

- over
- close
- far

At first, the option if the thumb is placed over the palm is tested with comparison of two distances. The distances between the tip of the thumb and bases of the middle finger and the index are compared. In the case where the distance to the base of the middle finger is smaller than to the base of index finger, the thumb is considered as over the palm, as can be seen in Figure 2.19 a) or defined by inequation:

$$C \cdot d(k_4, k_5) < d(k_4, k_9) \tag{2.11}$$

If the thumb does not fit the rules to be over the palm, the decision about its closeness to the palm is made. For this decision are needed the distances between the base of the index finger and key points at the tip of the thumb and in the middle of the thumb. Unless the distance to the tip of the thumb is approximately the same or shorter than to the middle of the thumb the position of the thumb is close to the palm as in Figure 2.19 b), otherwise if it is like in Figure 2.19 c) the thumb position is considered as far from the palm.

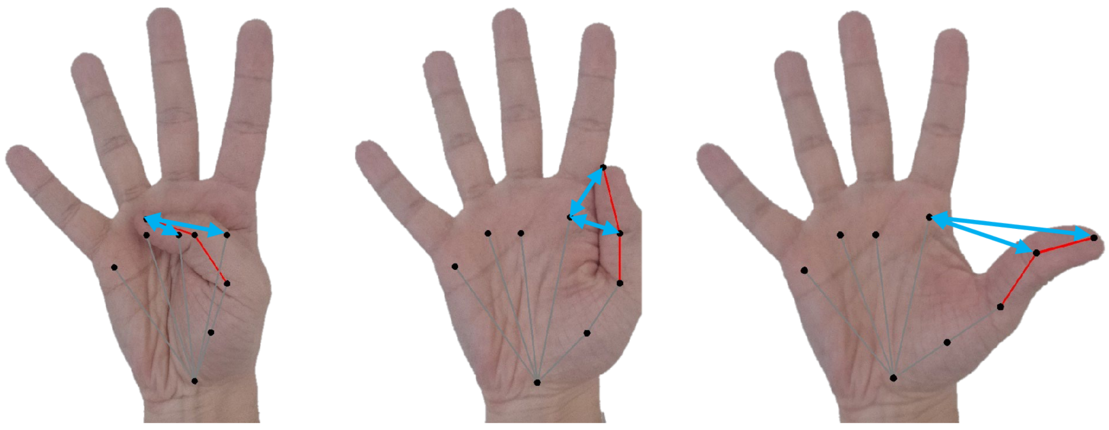


Figure 2.19 Thumb position states a) over b) close c) far

2.3.4 Thumb tip position

Some hand gestures depend on the distance between the tip of the thumb and tip of other fingers. It may be very important to determine whether the fingertips are very close to each other and even touching. In the gesture definition this state is defined by an array of all the fingers the thumb is touching or with options for detection:

- index
- middle
- ring
- little

Just like with previous poses, the decision is based on the comparison of distances; in this case three, instead of two, for a higher chance of correct pose estimation. One of the distances lays between the tips of the thumb and the finger that the thumb might be touching. If this distance is shorter than the length of the last link of the thumb, as is described by equation (2.9) or in Figure 2.20 a), the fingers are considered close enough and the finger name is added into an array of all other fingers the thumb is close to. Otherwise the situation might resemble the pose the hand is in Figure 2.20 b).

$$d(k_4, k_{n+3}) < d(k_3, k_4) \tag{2.12}$$

Where k stands for key point with specific index. Key points are in absolute states, because the same index always represent the same key point. Key point k_n stands for the base of the finger the thumb might be touching.

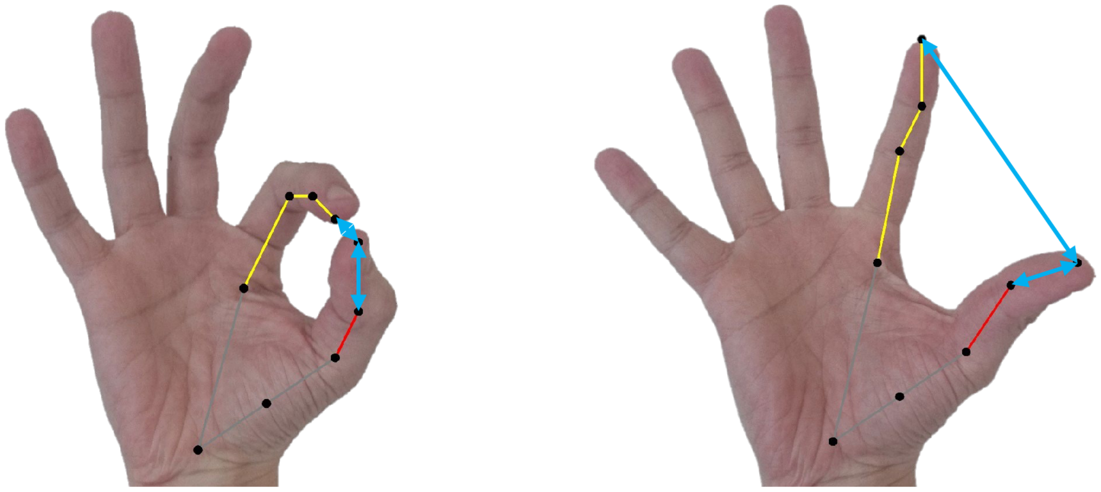


Figure 2.20 Thumb tip position a) index finger b) none

2.3.5 Thumb direction

In some cases, the gesture may also need the direction of the thumb, be it pointing downwards, upwards, left or right. Just like all other states of fingers, this is defined in the gesture definition file with states:

- up
- down
- left
- right

Unlike with previous finger states, the direction of the thumb is calculated as an angle between the line defined by two points and positive direction of horizontal axis. From the coordinates of the two points, the following can be calculated: the sides of a right triangle. These sides can then be used to calculate the angle with the trigonometry function arc tangent. This function returns value in range from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$, which is not enough to determine direction. Because of this, the conditional function for arc tangent called ***arctan2*** which takes in consideration the sign of both input arguments and its output range is from $-\pi$ to π in radians or from -180 to 180 degrees after conversion, which covers all the directions possible in 2-dimensional space, is used:

$$\mathrm{arctan2}(y, x) = \begin{cases} \tan^{-1}\left(\frac{y}{x}\right), & \text{if } x > 0 \\ \tan^{-1}\left(\frac{y}{x}\right) + \pi, & \text{if } x < 0 \text{ and } y \geq 0 \\ \tan^{-1}\left(\frac{y}{x}\right) - \pi, & \text{if } x < 0 \text{ and } y < 0 \\ +\frac{\pi}{2}, & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2}, & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined}, & \text{if } x = 0 \text{ and } y = 0 \end{cases} \quad (2.13)$$

Where output of the $arctan2$ function is the calculated angle, and arguments x and y are points defining the hypotenuse of the right triangle. For the calculation, points at the tip of the thumb and point at the base of the thumb are used. From the angle, it is then easy to decide which direction the thumb is pointing to. Graphical representation of the arctan2 function is in Figure 2.21.

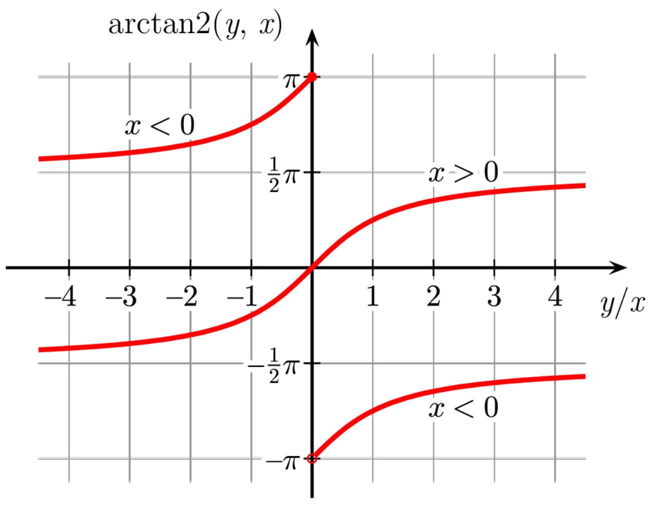


Figure 2.21 The output of function arctan2 [28]

The thumb can be directed up as is in Figure 2.22 a) which happens if the calculated angle is within the range of 45 to 135 degrees. Similarly, if the thumb is pointing to the right, which can be seen in Figure 2.22 b), the range the angle must fit into is from −45 to 45 degrees. In the case where the thumb is facing downwards, which is shown in Figure 2.22 c), the needed range of degrees is from −45 to −135. If the thumb is pointing to the left like in Figure 2.22 d), the range is the remainder to fulfill the whole 360 degrees from −135 to −180 and from 135 to 180 degrees.

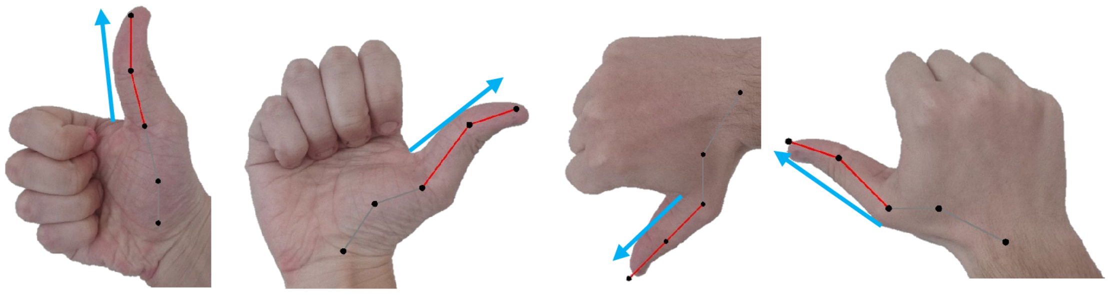


Figure 2.22 Thumb directions a) up b) right c) down d) left

2.3.6 Hand orientation

The last of the parameters defining hand pose focuses on the direction the hand is pointing to. It is calculated very similarly to the thumb direction and uses the same group of five states for the definition of gesture:

- up
- down
- left
- right

To determine the direction the hand is pointing to, the angle between the base of the hand and averaged coordinates of the bases of fingers is needed. The angle is then calculated the same way as with thumb direction using equation (2.13). Similarly, the direction is then determined by the same rules as the direction of the thumb. Figure 2.23 represents all the recognized hand directions.

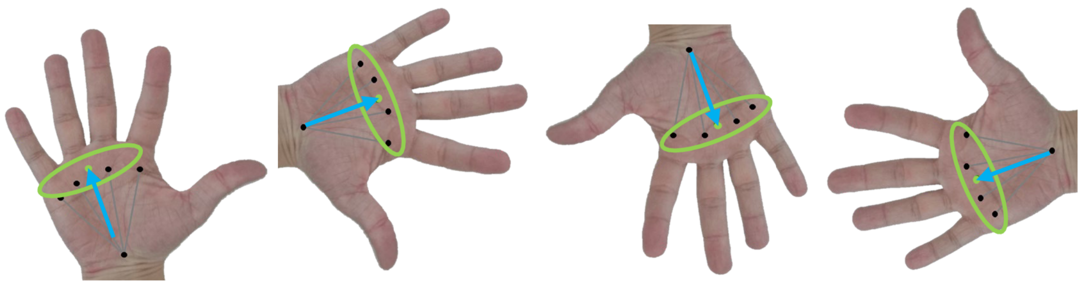


Figure 2.23 Hand directions a) up b) right c) down d) left

2.3.7 Gesture definition format

Each gesture is defined by a single JSON file in the gestures folder. These files contain a readable and easily modifiable structure with the name of the gesture and the definition of poses the finger can be in for the one gesture. The general structure of the gesture definition file is in Listing 2.2 with all the main level categories.

Listing 2.2 Gesture definition format – main categories

```
{
    "name": "…",
    "hand": [
        …
    ]
    "finger_bends": {
        …
    },
    "finger_spreads": {
        …
    },
    "thumb_position": [
        …
    ],
    "thumb-tip": [
        …
    ],
    "thumb-direction": [
        …
    ],
    "hand-direction": [
        …
    ]
}
```

Except for the gesture name, all the possible states are always in the square brackets, which are in JSON format used to encapsulate array members. The states are then string names of enumeration types used in the application code, to make readability by the human eye easier.

Curly brackets are used to represent objects; members of an object are then represented by a string name and the corresponding value. In this case the values are mostly arrays and other objects. Objects are used for finger bend states where the name of the member value is necessary to differentiate between fingers and for finger spreads to distinguish for which fingers the spread is in between. The final bend state and spread of fingers is again defined as an array of string names.

To allow the hand or pose of finger to be in any state possible and to skip the category in gesture matching, the keyword “any” can be used to indicate that the category does not matter. For example, keyword “any” can be used to identify that the gesture can work for both the left and right hand. Two definitions, one using a list of string names and the second one with only the keyword “any” but with the same outcome are in Listing 2.3.

Listing 2.3 Gesture definition format – keyword “any”

```
"hand": [
    "left",
    "right",
    "unknown"
]
```

```
"hand": [
    "any"
]
```

2.4 Graphical user interface

To ensure ease of use, a graphical user interface for the application was created. For this purpose, the open-source cross platform Python library Kivy [29] was used. Kivy uses event-based programming, resulting in the application running in super loop and just responding to event callbacks from the user interface. The UI can be designed either directly from Python, which can become very confusing with bigger projects, or by using Kivy proprietary language called KV, in which a tree structure of widgets with rule-based properties can be defined. Hello World type of application UI using KV language is in Listing 2.4.

Listing 2.4 Hello world in KV language

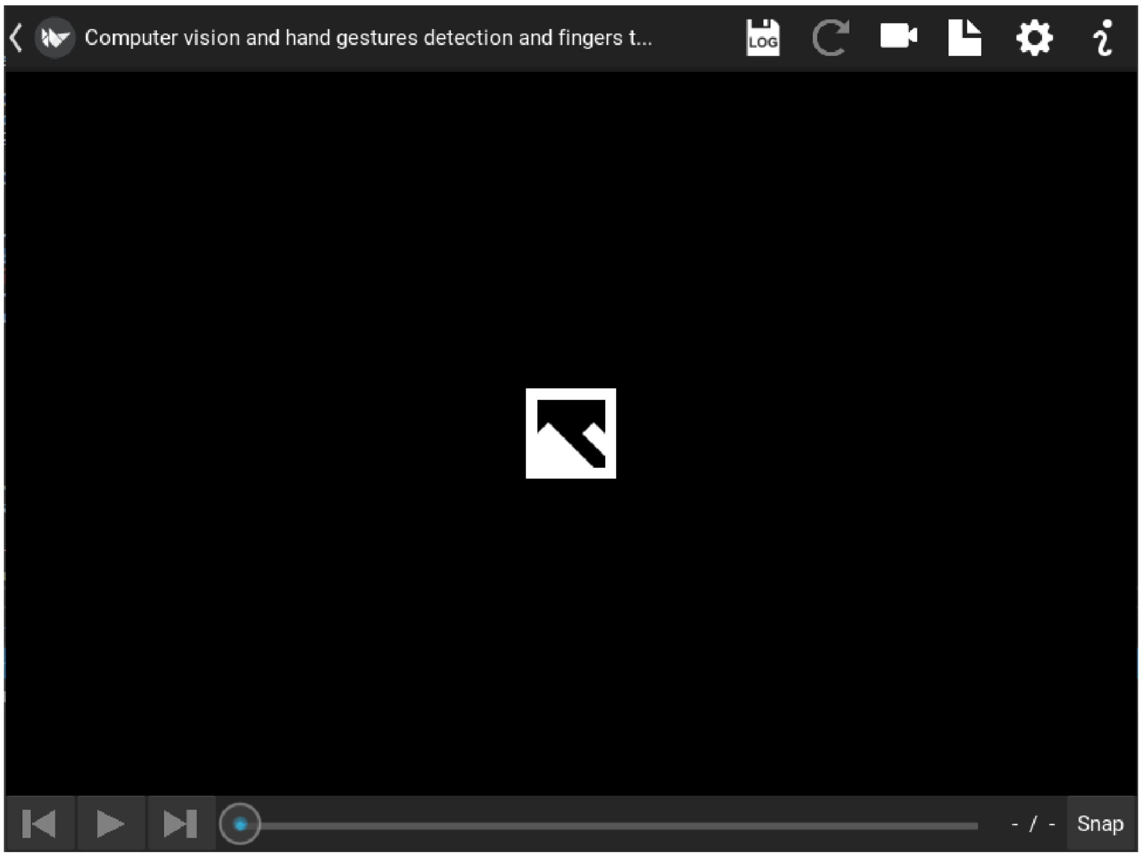
```
#:kivy 1.10.1

BoxLayout:
    orientation: ‘vertical’

    Label:
        text: ‘Hello ’ + ‘World!’

    Button:
        text: ‘Close’
        on_press: exit()
```

This simple application will be composed of two widgets, label occupying the top half and button in the lower half of the window. KV language can also use simple python commands and conditions in its properties. This is presented with the concatenation of strings in the text property of the label. Python function calls can be also used, like exit() function call in the on_press event callback.

2.4.1 Main screen

The application is designed to use a single screen, which contains a video player with a hide-able control panel at the bottom of the screen, which contains the typical play/pause button, option to go through the video frame by frame in either direction, position slider for easy navigation in the video, current time in the video and the length of the input sequence followed up by a quick screenshot button. At the top of the application window is located an action bar containing function buttons at the top.

Each button launches a callback function which results in opening a popup overlay with file browser, setting the video source to camera, sliding a panel with more options into the screen, button that reloads the source file or enables log saving. The base screen of the application is in Figure 2.24.

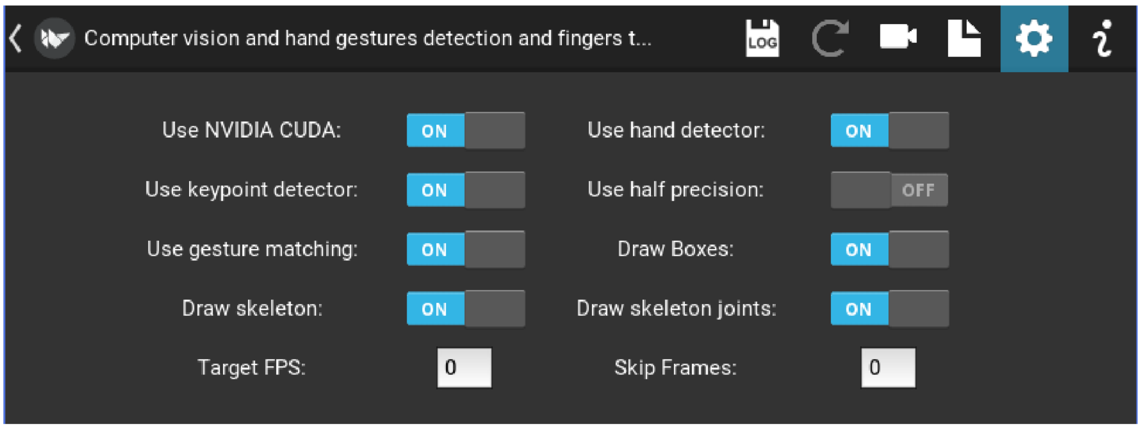

Computer vision and hand gestures detection and fingers t...
LOG
- / -
Snap
Figure 2.24 Created application window

In the design of the application, most of the used icons belong to the open-source icon pack called Open Iconic [30] with the colors changed to fit the dark interface, and some icons were made from scratch, such as the icon for save image button because Open Iconic did not contain an icon usable for this function.

2.4.2 Settings

The settings panel contains options for configuring the detection itself. One of the options is the usage of NVIDIA CUDA which is by default turned on if the system contains a supported GPU. Other options allow the hand detector, finger detector or gesture matching to be disabled, which can be done if the GPU does not have enough video memory to fit both neural networks into it or the functionality just is not needed. To be able to use this application with GPUs with less amount of memory, the option to use only half precision floating point models is available. This may also produce faster processing on supported GPUs but for the cost of possibly lower accuracy. Half precision is also not available for use on CPUs, so the option is automatically disabled if the NVIDIA CUDA is not used.

More cosmetic options include the drawing of hand skeletons, joints or detected bounding boxes into the image or the frame of a video sequence. The last two options represent the frame rate the detection should try to get if the hardware is powerful enough; the base frame rate of video or camera is set when the source is selected, and how many frames should be skipped between detections, which is by default 0. The settings panel is shown in Figure 2.25.


Computer vision and hand gestures detection and fingers t...
LOG

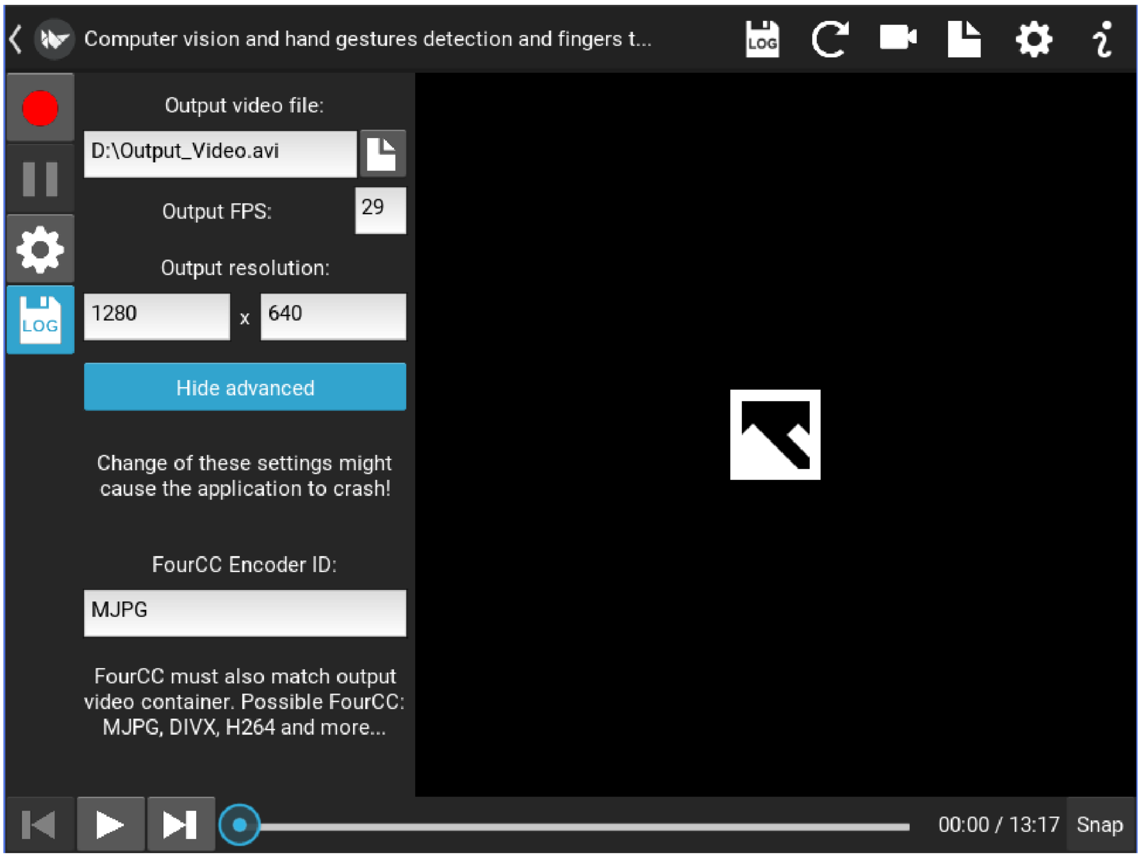
Use NVIDIA CUDA: ON
Use hand detector: ON
Use keypoint detector: ON
Use half precision: OFF
Use gesture matching: ON
Draw Boxes: ON
Draw skeleton: ON
Draw skeleton joints: ON
Target FPS: 0
Skip Frames: 0

Figure 2.25 Detection settings available in application

Except for the settings that change the processing unit and precision of calculation, this is because big data transfers of models between GPU and system memory or the need to reload weights, all of the settings can be toggled during detection and the new setup is used for detection over the next frame.

2.4.3 Record panel and settings

Saving the video is designed in a slightly different way than is usual for applications working with video sequences. Instead of saving the whole video and waiting for the whole video to be processed, the saving is done using a recorder. The recorder can be launched at any moment in the video and can be paused to allow skipping of certain parts of the video. The record panel also includes optional toggle to save log during the output video saving. Both expanded record panel and record settings panel are shown in Figure 2.26.



Computer vision and hand gestures detection and fingers t...

Output video file: D:\Output_Video.avi

Output FPS: 29

Output resolution: 1280 x 640

Change of these settings might cause the application to crash!

FourCC Encoder ID: MJPG

FourCC must also match output video container. Possible FourCC: MJPG, DIVX, H264 and more...

00:00 / 13:17

Figure 2.26 Record panel and record settings

All the settings are set automatically after opening a video or loading up a camera to match the resolution and frame rate of the input. This does not mean that the settings cannot be different, as the frame rate and resolution can be changed to almost anything and the frames will then be resized and written with the frame rate set before the recording is started.

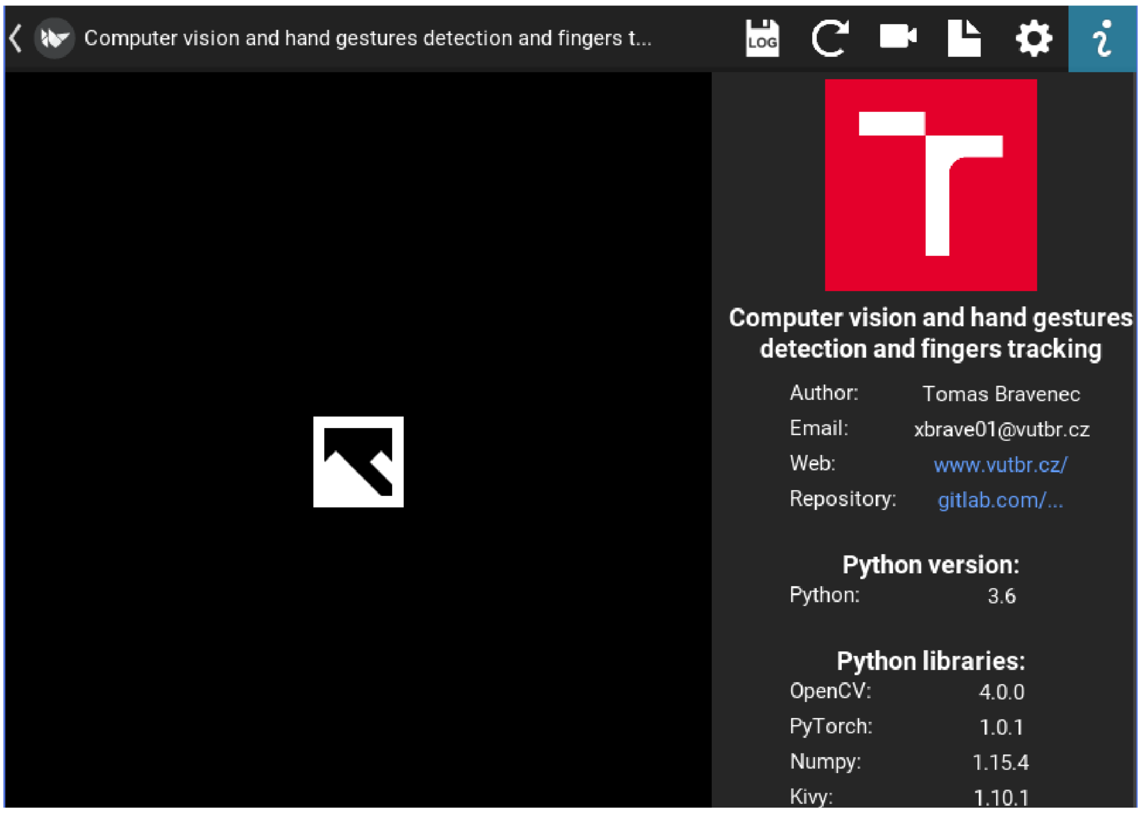
The advanced record settings contain a single text box, in which can be written FourCC (four-character code) defining an encoder that is to be used while generating an output video sequence. There are many FourCC sequences [31], but it is impossible to say upfront which will work on specific systems. Because this is the only place for the user to input an incorrect setting, a warning about possible application crash is also included. Most common FourCC codes are MJPG, DIVX, H264 but there are many more. By default, FourCC used by the application is

MJPG, which stands for the codec Motion JPG and should be available with every OpenCV build, but the produced output file is very large compared to more advanced codecs like DIVX or H264.

The approach of using a recorder also allows changes in detection settings during the saving process, like disabling hand detector, changing target frame rate or just turning off the rendering of hand skeletons.

2.4.4 Info panel

Last of the UI elements is panel containing information about the application, version of python and versions of used libraries, hyperlinks to project repository and university website and contact email. Expanded info panel is in Figure 2.27.


Computer vision and hand gestures detection and fingers t...
LOG

Computer vision and hand gestures detection and fingers tracking

Author:	Tomas Bravenec
Email:	xbrave01@vutbr.cz
Web:	www.vutbr.cz/
Repository:	gitlab.com/...

Python version:

Python:	3.6

Python libraries:

OpenCV:	4.0.0
PyTorch:	1.0.1
Numpy:	1.15.4
Kivy:	1.10.1

Figure 2.27 Application's info panel

2.5 Application development

The whole application can be divided into four parts. Three of these parts represent the whole process from hand detection to the final gesture recognition. The last part of the application is the user interface, which is then by callbacks and rule-based functions connected with the computational part of the application.

2.5.1 Models download

Because of the size of the weights for both neural networks, these are not included in the source code repository. There are two options on how to download these. Either the weights will be downloaded at first launch of the application, showing the progress of the weights download in the terminal window, before the UI loads or downloading them manually before launching the main application by running the script get_models.py from the repository [32]. The result is the same as the application calls the functions from get_models.py on every launch to check if the models are present and downloads them if they are not.

2.5.2 Background logic

The logic behind the complete gesture detection is divided into three python modules. Hand key point detection and gesture matching are in their own modules. Hand detection is included in the module with the wrapper class.

The hand detection neural network contains all the logic in the Darknet interpreter python package, and the only preprocessing needed is conversion of color format from BGR to RGB. That means the function call for prediction over an input image is included in the wrapper class without any postprocessing.

The situation is different when it comes to the hand key point detector from the project OpenPose, because the weights have been converted to PyTorch, the whole preprocessing and postprocessing had to be written from scratch. Even if that was not the case, because of the decision making in the case of the left or right hand, the implementation would be very different anyway.

Gesture recognition, unlike either of the detections is tied to the hand key points detector and subsequently cannot be launched on its own. Gesture matching consists of getting the pose out of the key points, which is then followed up by gesture matching. The main idea behind the algorithm is to check if the current parameter of the pose can be in the defined gesture. If any of the pose parameters cannot be in the tested gesture, that gesture is removed from the list and the next definition is tested. The best-case scenario would end up in either an empty list or a list with a single remaining gesture definition left for an unrecognized gesture and a recognized gesture respectively. In the case the list contains more than one gesture definition, the one first loaded into memory is used as the resulting gesture.

The whole detection logic is wrapped in the hand tracker class that provides a single function to provide complete detection from a static image or a single frame of a video sequence. The class also contains functions and state variables to allow for changing detection settings.

These functions are useful for easy binding to the switches in the application's settings, that allow to easily change the output from the prediction function. Some of these changes though cannot be done during detection and can only be made while the detection is not running.

2.5.3 Multi-threaded processing

To make the UI of the application responsive without freezing the UI during tasks that need a lot of time for calculation, the processing logic of the neural networks was moved to a separate thread. This means that the application's UI stays responsive even during time consuming detection. To keep full control of this thread, the control is done using simple semaphores. One that keeps the thread running and the second, that allows the processing to start. If the thread is running, but processing is stopped, the thread is put to sleep for half a second, before it checks if processing is required or not. The main reason for this delay is to make the application use less processing power at the time of not doing anything. The processing thread is also created only at the time of changing the source for detection, and only after the previously running thread stops. This approach makes threading relatively easy, because there are always at most two threads running.

2.5.4 Logging

The application includes two logging systems, one runs only during recording and the second one runs all the time. Even though logging is always running, the log is saved only if the option is selected.

If the option to save log is used and the source is a video, the log will be saved on a change or reload of the source file. This approach ensures that the log contains information about every processed frame and is the only option, because of the atypical saving system. The log for an image is saved only if the processed image is saved and log for a camera feed is not taken at all as there is no video information to compare it to after the frames from the camera are processed.

The situation is different if the recording is running. At the start of the recording, a second logging object is created, which records every information about every frame saved into the output video file. This also means the logging object starts indexing the frames from zero and uses the output framerate to calculate the timestamp in the log to match the output video.

The logs always contain a header, which for images contains the output image path and the dimensions of the image. The log then contains a single line for each of the hands in the image. These records contain the information about the bounding box in normalized format and string representation of the detected hand and gesture. In the case the hand detector is not used, the bounding box location is put in the center of the picture with a width and height matching the dimensions of the image. If the hand key point detector or gesture matching is not used, the word “unknown” is used in the log. If the gesture does not match any of the defined gestures, the word “unknown” is used instead of the gesture name. The format of the log with header and record is in Listing 2.5

Listing 2.5 Log header and record format for static images

```
<filename>, Shape: <width>x<height>
<x> <y> <width> <height> <hand> <gesture>
```

The format of the log header for video sequences compared to the log for a single image also includes information about the video framerate. The records of the log then also include frame index and calculated timestamp from the frame index and the framerate. After these two new values, the format is the same, with the bounding box information in normalized format and the string representation of the hand and gesture. Format of the log for video sequence with all the information is in Listing 2.6.

Listing 2.6 Log header and record format for video sequences

```
<filename>, FPS: <framerate>, Shape: <width>x<height>
<frame> <time> <x> <y> <width> <height> <hand> <gesture>
```

2.6 Application prerequisites

The created application uses very memory and computationally intensive methods of image processing, so the hardware of the system must be appropriate. There are prerequisites when it comes to both the hardware and the software of the system.

2.6.1 Source code repository

The application’s source code, including full change history and step by step installation guide for both Microsoft Windows and Linux based systems, is stored completely in a git repository [32].

2.6.2 Hardware requirements

When it comes to processing using a CPU, which is possible but not recommended because of long processing times, the system should have at least 8 GB of system memory, but 16 GB or more is recommended, as the system itself normally uses at least 2 GB on its own, and 6 GB is then easily filled with the weights of both neural networks and processed image, and in the worst case scenario the system might start moving data into swap. This can then result in a very unresponsive system, not just the application behavior.

If the system contains a supported NVIDIA GPU, the system memory can be just 8 GB, but the GPU memory should have at least 6 GB be for smooth functionality. With either the hand detector or the hand key point detector disabled and with half precision processing enabled, the application can run even on cards with 2 GB of video memory.

There is no requirement on the CPU performance, but for the best performance, the model of CPU should not bottleneck the GPU available in the system and vice versa. The result of bottlenecking is lower frame rate in the example as a result of the CPU not serving the images for processing quickly enough or the GPU waiting for the commands from the CPU, so the load of the GPU is nowhere near the load it could be with adequate CPU.

2.6.3 Software requirements – Microsoft Windows

The application on the operating system Microsoft Windows, only needs installation of 64-bit Python version 3.6 with pip package manager installed. Although the Python version can be higher, the link to PyTorch package wheel in `requirements.txt` would have to be changed according to PyTorch Get Started guide [33]. The python installation should also be added into the system path during installation to ensure there would be no issues during package installation process. The NVIDIA video driver corresponding to the version of the CUDA toolkit, the PyTorch package has been compiled with, installed is also necessary, if the accelerated GPU computing is to be used. The minimal video driver version can be found in NVIDIA CUDA documentation [34]. The full installation of the NVIDIA CUDA Toolkit is not necessary as PyTorch already comes with prebuild binaries needed for GPU accelerated computation.

All the Python packages needed are in file in the root of the repository requirements.txt and can all be installed with the command in Listing 2.7. To allow usage of the same `requirements.txt` on Microsoft Windows and on Linux based systems, the OS specific packages are marked inside the file with environmental markers.

Listing 2.7 Python packages installation command on Microsoft Windows

```
python -m pip install -r requirements.txt
```

2.6.4 Software requirements – Linux

Just like with Microsoft Windows, the recommended Python version is 3.6 in 64-bit version, higher versions of Python can be used, but the links to the PyTorch wheel in `requirements.txt` needs to be changed for the same reasons as for installation in Microsoft Windows and the new link can be found using Get Started guide on PyTorch website [33]. Unlike with Microsoft Windows, Linux based systems need a few packages installed through the system package manager. The main package in question is the framework for graphical user interface Kivy and its dependencies, as mentioned in the Kivy installation guide for Linux [35]. The Python installation also needs packages:

- cython
- setuptools
- wheels

These packages are needed to build the Kivy wheel before the installation itself. Just like with Microsoft Windows, the appropriate NVIDIA video driver [34] is necessary for PyTorch to allow NVIDA CUDA in application settings. After that the setup is like the setup process in Microsoft Windows. All the python packages are in the same file `requirements.txt` and on Linux can be installed with command in Listing 2.8.

Listing 2.8 Python packages installation command on Linux

```
pip3 install -r requirements.txt
```

3 Dataset creation

Using only the EgoHands dataset [12], the application provided good detection results, when it came to hands doing stuff on a table, like playing chess or Jenga. Which makes sense as it is created out of multiple video scenes where people play cards, chess and Jenga. This also means, that the results when it comes to hand detection in various poses, considerably different to the actions that the dataset was created on, were not exactly good. The same thing applies for MPII Human Pose dataset, as it contains people in various poses, training neural networks for detection of hands using only this dataset provides disappointing results. Similarly, for the New Zealand Sign Language dictionary, which contains images in similar settings without more widely varied conditions. That meant that the datasets had to be expanded to create a more varied set of images.

3.1 Obtaining images

To be sure that the dataset contains most of the gestures humans can do with their hands, it is vital to use images that contain these gestures in different environments, lighting, poses and so on. For this very reason, it is a good idea to combine existing datasets or use frames from videos that are on the internet and create own data by recording what the network should train on.

3.1.1 Combining existing datasets

The easiest way of expanding datasets is to combine two of them together. Especially if both datasets include annotations of the object needed for training. In that case, combining datasets is just about converting annotations into the required format. If the dataset does not have desired annotations, then these must be created manually using one of the labeling tools.

This is the method used in expanding the EgoHands dataset with images from the New Zealand Sign Language dictionary and the MPII Human Pose dataset. As mentioned before, neither of these had required annotations, so these were created manually.

3.1.2 Recording own data

For the purpose of recording own data, Python script saving a frame from a webcam after a set interval or selecting frames from a video sequence recorded on a camera can be used. Either of these approaches is usable in this case, although the recording device should have high quality image capturing, as the neural network provides better results if trained on images with higher resolutions, rather than lower resolution images. An example of such images with ground truth annotations are in Figure 3.1. This approach was used to create evaluation dataset for left hand recognition and gesture classification part of the application.

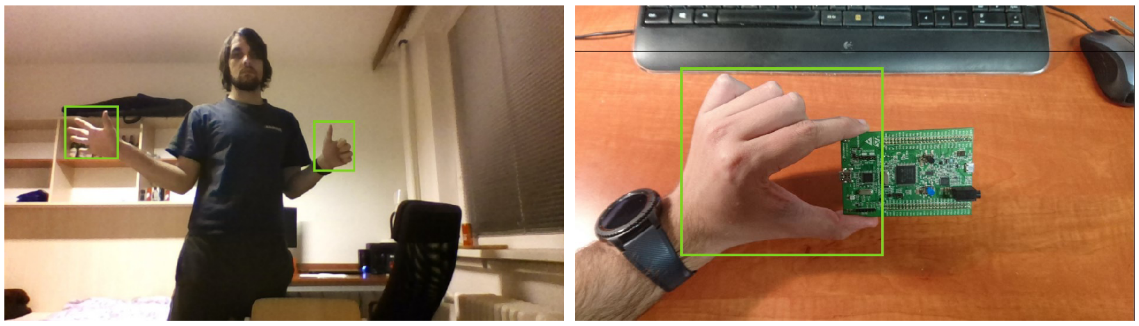


Figure 3.1 Custom images for dataset extension

3.1.3 Getting videos from the internet

Another way of modifying the existing dataset is using frames from videos with license that is allowing reuse of the content, that can be found on the internet. A couple of frames from videos from YouTube can be seen in Figure 3.2. Scraping videos from the internet is a viable option, but this approach to dataset extension was not necessary as the three combined sets of images provided a high variety of training data.

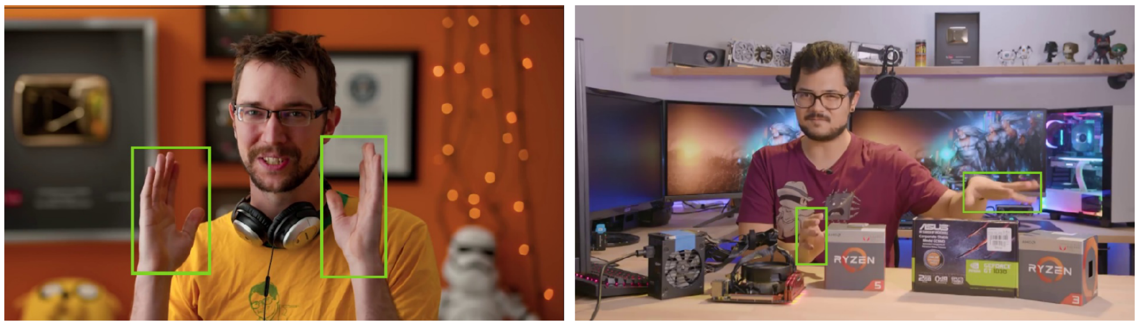


Figure 3.2 Images from YouTube videos [36], [37]

3.2 Annotation of images

Annotation can be done in two ways. One way is by manually creating files and measuring distances from the corner of an image followed by width and height of the object. The data must then be manually inserted into a file. This approach is not ideal as doing this for thousands of images would be extremely ineffective.

Because of this, annotation tools are much more effective. Their purpose is to create annotations in the format that would work for a specific network or in some format, which could be easily transformed into another. The process of creating the annotation is mostly automatic, the only thing that must be done is manually selecting the object that needs to be annotated and the tools take care of the rest.

3.2.1 Tool LabelImg

The YOLOv3 neural network needs specifically annotated images for training. The annotation consists of a single txt file for each image, that contains the ground truth information about objects for which the network should train and their bounding boxes; each object in the image is placed on a new line. The locations and dimensions of bounding boxes are in normalized format independent of the image resolution and object class is represented by its index. The format of these annotation files is in Listing 3.1.

Listing 3.1 Format of image annotations for YOLO based networks

```
<class> <x> <y> <width> <height>
```

For annotating all the additional images the multiplatform application LabelImg [38] was used, which can generate annotation files in the correct format from the bounding boxes drawn into the image. User interface of LabelImg is shown in Figure 3.3.

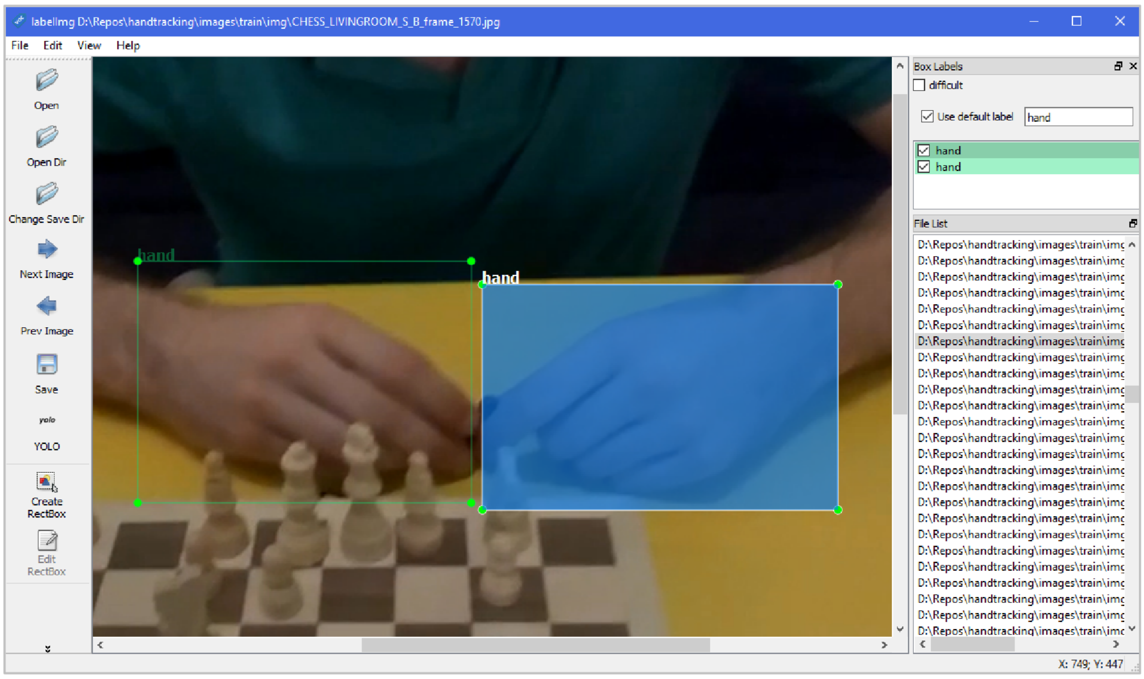


Figure 3.3 Annotation tool LabelImg

These annotations are usable only with neural networks based on YOLO architecture, but can be easily converted for use with any other neural network with simple scripts written in python. After annotating, the dataset was ready for training the neural network.

3.2.2 Website Supervise.ly

Another very useful tool for dataset annotations is the website called Supervisely [39]. This online based tool is very capable as it can create the annotations in its editor, which is shown in Figure 3.4. It can also show statistics like how many images are left to annotate, and even the percentage of image space, that is occupied by the objects.

Even though it seems like it can do a lot already, the functionality does not end here, Supervisely has also implemented data transformation language, that can easily divide the dataset into multiple smaller datasets, which is useful for creating subsets of the dataset for training and evaluation. Another interesting feature is data augmentation, which extends the dataset by color shifting, flipping and rotating existing images.

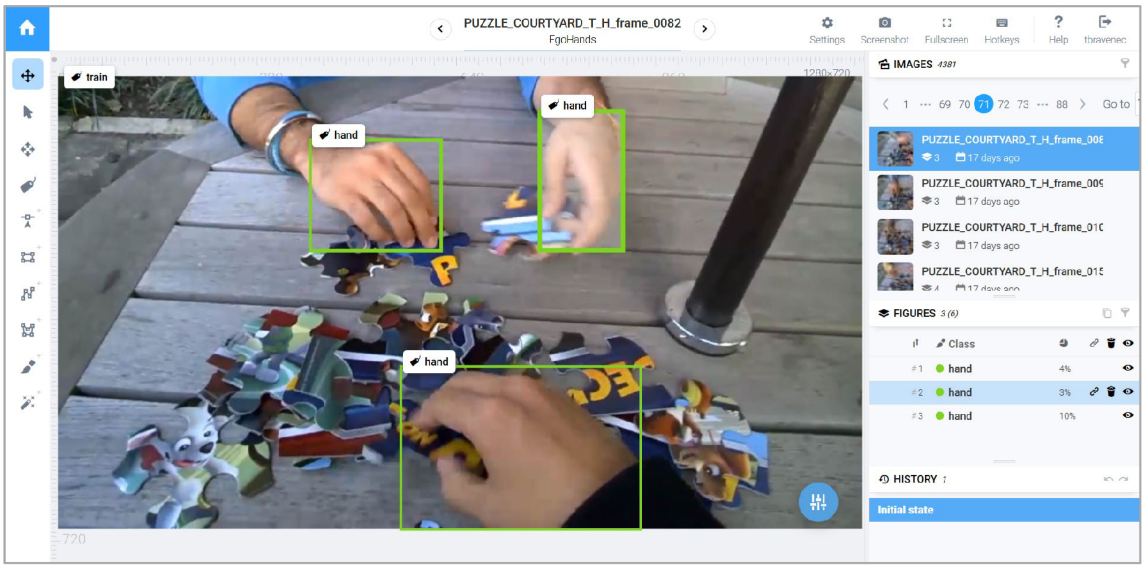


Figure 3.4 Annotation tool Supervisely

On top of that, supervisely also contains the means to train a few predefined networks using the annotated and augmented datasets. This feature though is not implemented on the website itself but needs a separate Linux based system with NVIDIA CUDA support, as the website will only control the machine that will do the computing.

4 Evaluation and testing

As with any deep learning applications, it is highly recommended to use GPUs to speed up the forward pass of input data through the neural network and out of the output. Most of the deep learning frameworks like TensorFlow, PyTorch and others use NVIDIA CUDA for this purpose. The framework used for training and then testing was PyTorch, because of this, the GPUs used to evaluate the performance of the application are only from NVIDIA. As there is pretty much no point in creating a hand detection solution with hardware that almost no one has, one of the GPUs used is a few generations behind current ones and targeted on laptops, which means it is less powerful than its desktop counterpart. The second tested GPU is targeted at desktops and is a single generation ahead of the laptop one. Even then, the desktop GPU belonged to the mid-range and was nowhere near as powerful as the most expensive GPUs from the same generation. Parameters of both tested graphics cards are listed in Table 4.1.

Table 4.1 GPU used for evaluation and testing

	NVIDIA GeForce GTX 850M	**NVIDIA GeForce GTX 1060**
CUDA cores	640	1280
Base core clock	0.901 GHz	1.607 GHz
Max boost clock	1.084 GHz	1.835 GHz
Memory	2 GB	6 GB
Memory bandwidth	80.0 GB/s	192.2 GB/s
Memory type	DDR3	GDDR5
Manufacturing process	28 nm	16 nm
Architecture	Maxwell	Pascal
Target system	Laptop	Desktop
Launch date	March 2014	July 2016

There are some options to get even better performance out of the graphics cards, if they support it. Because the architecture of the GPUs changes a lot in between generations, these usually do not differ just in the performance improvements but also in the features of the GPUs. For example, compared to the Maxwell architecture, graphics cards based on the Pascal architecture support mixed precision processing [40]. In practice, this means that lowering the precision of the data types from 32-bit floating point to the floating point represented with only 16 bits, not just the memory requirements but also the time needed for processing will be cut in half. Another improvement in performance could be gained with GPUs based on the Volta architectures, some GPUs based on the Turing architecture and possibly architectures released in the future. Graphics cards based on these architectures may contain not just CUDA cores, but also tensor cores [41]. Tensor cores are specifically optimized computing cores for matrix operations which are used at the core of deep learning applications.

Just for good measure, testing was also done on CPUs available in systems with tested GPUs, to show the performance loss on systems without a GPU from NVIDIA. Same as for the tested GPUs, testing was done on a few years old laptop processor to show how quickly the neural network can detect hands on lower end hardware and on much newer desktop CPU to show the difference in performance achievable with a more modern CPU. Tested CPU parameters are in Table 4.2.

Table 4.2 CPUs used for evaluation and testing

	Intel Core i7 4700HQ	**Intel Core i5 8400**
Cores	4	6
Threads	8	6
Base core clock	2.4 GHz	2.8 GHz
Max boost clock	3.4 GHz	4.0 GHz
Memory	16 GB	8 GB
Memory type	DDR3	DDR4
Manufacturing process	22 nm	14 nm
Architecture	Haswell	Coffee Lake
Target system	Laptop	Desktop
Launch date	June 2013	October 2017

4.1 Hand detection

On the hand detection testing can be looked at from two angles, accuracy and speed. Accuracy of the network is calculated during training, so it can be evaluated subjectively. On the other hand, the speed of detection can be easily measured.

4.1.1 Detection accuracy

From predictions made on the validation dataset, most of the time the hands are detected correctly, and predictions of bounding boxes are very close to actual ground truths. The comparison of ground truth and predictions made by the neural network can be seen in Figure 4.1.

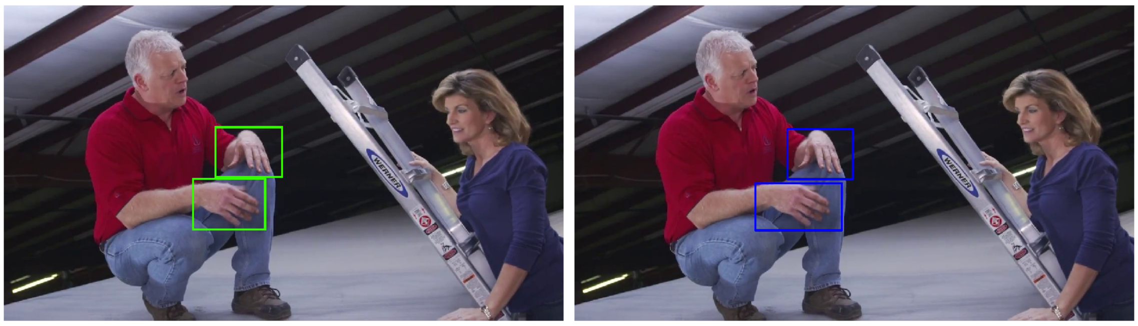


Figure 4.1 Comparison of ground truth and CNNs prediction [14] a) ground truth b) predictions by neural network

Even though most of the time the trained neural network manages to detect hands correctly there are times it can get confused and show incorrect detections. This can happen when the image is blurry, hands are obscured from full view by other objects, the objects look from a certain angle as human hands or just make incorrect predictions without a reason. These situations can be seen in Figure 4.2 where in one image two hands are detected as one, just like an ear and a tool in a belt, or in the second image where the design on the jersey of one of the basketball players is recognized as a hand. On the other hand, from this second image, it can be taken as fact that hand detection works on various skin colors.

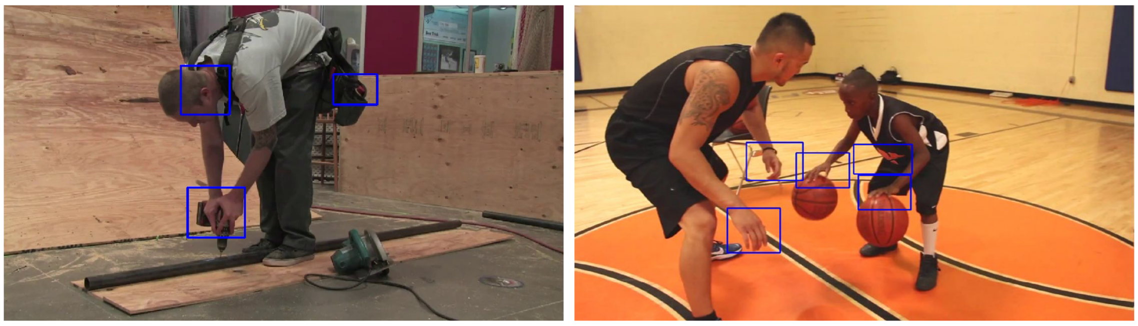


Figure 4.2 Incorrect detections [14]

Because the hands from the first-person point of view included in the EgoHands dataset were not used for training the neural network, hands from this perspective are mostly not being recognized either. In Figure 4.3 are shown examples of missed detections from the first-person point of view.

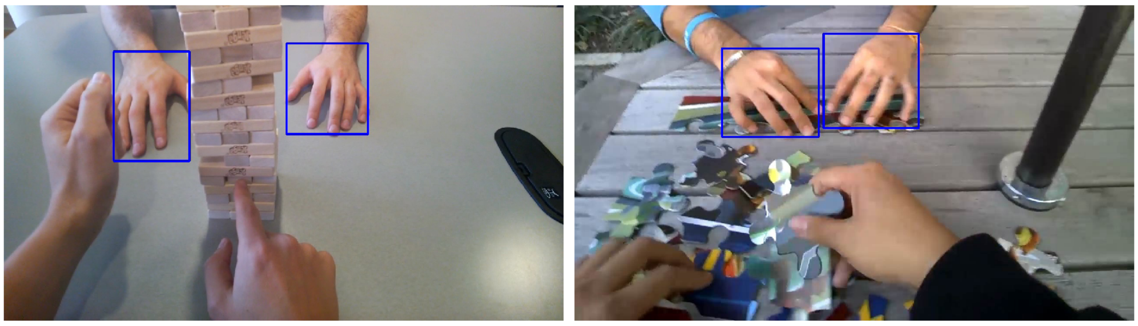


Figure 4.3 Undetected hands [12]

4.1.2 Detection speed

To make things clear, the neural network has a fixed input size, which means that all input frames are resized to the resolution set in the configuration file of the neural network before it was trained. In this case, the resolution is 416x416 pixels. As expected, the times needed for running the detector on any of the available CPUs provided unexceptional results as low as 0.61 frames per second using a mobile processor, the desktop CPU even though four generations newer and with higher clock speed, managed to be only three times as quick. Using the GPUs provided much better results, even the older mobile graphics card easily outperformed both tested CPUs by a big margin and managed to get to more than three times higher framerate than the desktop CPU. Although the mobile GPU did beat both CPUs by a big difference, using a newer desktop GPU provided a massive performance increase, although not as high as it could be, due to the bottlenecking of the graphics card by the CPU in the system. All the measured frame rates and times needed to process a single frame of a video sequence are listed in Table 4.3.

Table 4.3 Detection speed comparison between GPU and CPU

	Frames / second	**ms / frame**
Intel Core i7 4700HQ	0.61	1639.34
Intel Core i5 8400	1.83	546.45
NVIDIA GeForce GTX 850M	6.81	146.84
NVIDIA GeForce GTX 1060 6GB	28.91	34.59

4.2 Hand key points detection

As the key point detector belongs to the project OpenPose, the evaluation of the neural network model was already done in paper [26]. What can be tested and does not belong to the original paper, is the decision if the hand is left or right.

4.2.1 Left hand recognition

Since OpenPose uses another neural network, specifically for human pose estimation, the decision for which hand the pass through the network should be is quite straight forward. The implementation is vastly different from the one used. Because of that, the accuracy of the current implementation had to be evaluated.

For the evaluation, 147 hands were used in a set of 101 testing images that contain people showing hands with different gestures using either one or both of their hands. This set of images was also taken with varying lighting conditions and in different environments. Two of the pictures from this testing set with annotations of left and right hand are in Figure 4.4.

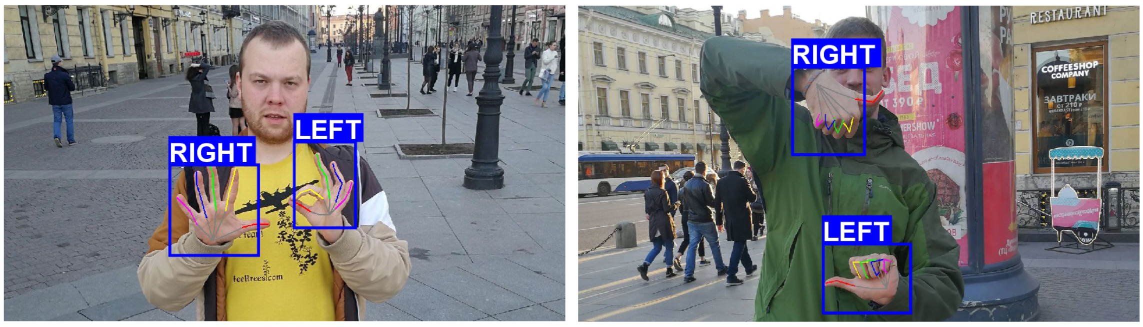


Figure 4.4 Images from the evaluation dataset with hand annotations

The implementation of the hand classifier managed to correctly classify the hand in 94.5% of all tested cases. Exceptions to the correct detections are mostly the cases when the hand is positioned in a way that can resemble the other hand. This situation usually occurs at moments when it is not clear from the image cutout containing the hand, which hand it is. A situation like this, where the neural network can be mistaken by the very similar outlines and pretty much the same distribution of key points, is depicted in Figure 4.5, and even in this situation the incorrect detection is not certain and depends on the quality of the input image.

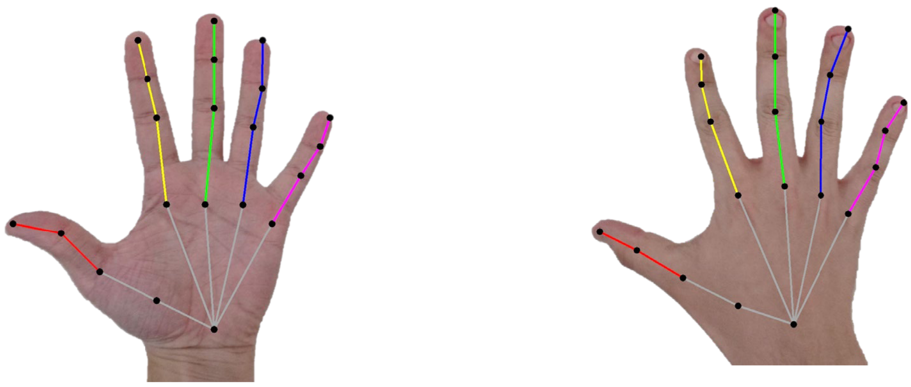


Figure 4.5 Left and right hand with similar key point distribution

To represent both correct and incorrect classifications of hands, the confusion matrix in Figure 4.6 was created. From this matrix it is visible that only in a few instances the classification was not accurate, and the number of incorrect classifications is insignificant when compared to the number of correct ones. From the confusion matrix it is clearly visible that the logic behind the hand classifier is accurate in most cases.

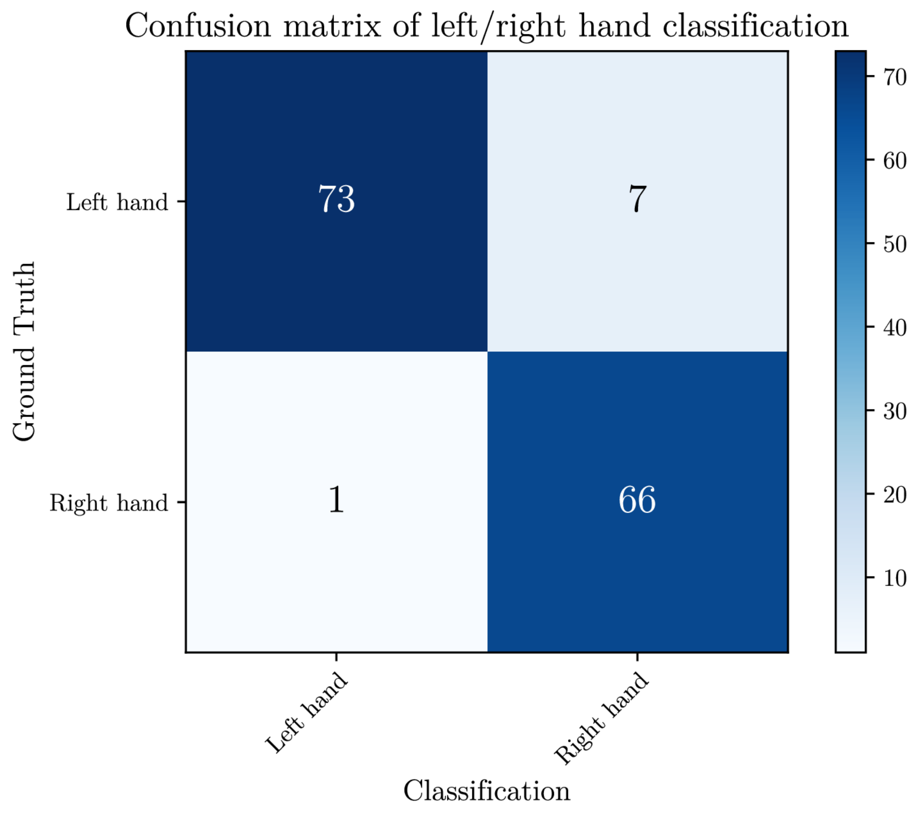


Figure 4.6 Confusion matrix for recognition of left and right hand

4.2.2 Half Precision calculation

Even though the full evaluation of the hand key point detector was not necessary, due to the option of using half precision floating points, the accuracy can be lower than expected. This issue was encountered when working with the neural network only once using an image found on the internet, where the usage of half precision made a difference. While using single precision, the hands and hand key points were detected correctly and the gesture classificator predicted both gestures correctly as is in Figure 4.7 a) even though the second key point of the ring finger on the left hand is slightly off the correct position. On the other hand while using half precision, the second key point of the index finger on the right hand was not detected correctly which is in Figure 4.7 b), which might have been due to overflow in one or more of the layers in the neural network. This resulted in incorrect pose estimation and wrong gesture classification. Even though there was a difference in the predicted key points of the right hand, key points of the left hand stayed the same, no matter the precision used for calculation.

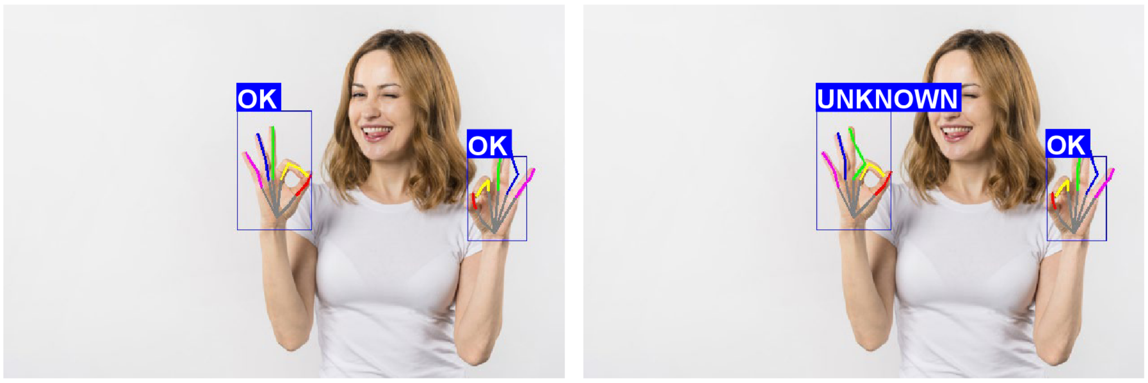


Figure 4.7 Effect of half precision on hand key points detection [42]
a) single precision b) half precision

4.3 Gesture classification

For the purpose of testing the gesture recognition system, eight gestures were predefined. These gestures were also included in the creation of the dataset for evaluating the hand and gesture classifier. All the predefined gestures are presented in Figure 4.8.

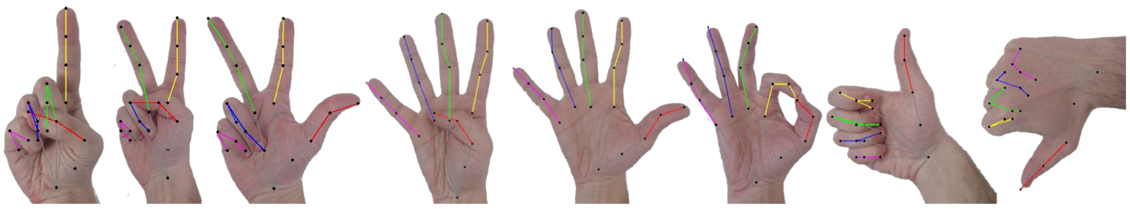


Figure 4.8 Predefined gestures a) One b) Two c) Three d) Four e) Five
f) OK g) Thumbs up h) Thumbs down

Correct classification of a gesture is ultimately dependent on the predicted hand key points positions. This makes evaluation of gesture classification quite difficult, because the error might not be in the gesture classification, but in the output of the key point detector.

To evaluate, the same set of 101 images used to evaluate the classification of the left or right hand was used. These images were manually labeled with the gestures shown in them. If the label matched the predicted gesture or was unknown because the gesture was not in the predefined set, the prediction was taken as correct. In the case the hand key point detector produced a result that was obviously wrong, like in Figure 4.9, that detection was not used in the calculation of the success rate of the gesture classificator as it does not objectively represent the error in the gesture classification.

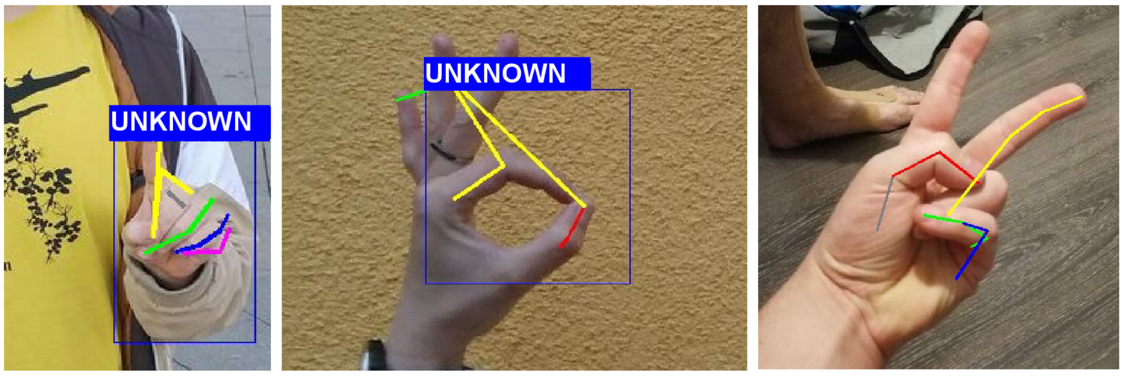


Figure 4.9 Incorrect hand key points detection

The images from the evaluation dataset used for classification contained either a single hand, where hand detection was not necessary, or with multiple hands in various poses with the need for the hand detector. Both cases of the evaluated images are displayed in Figure 4.10.

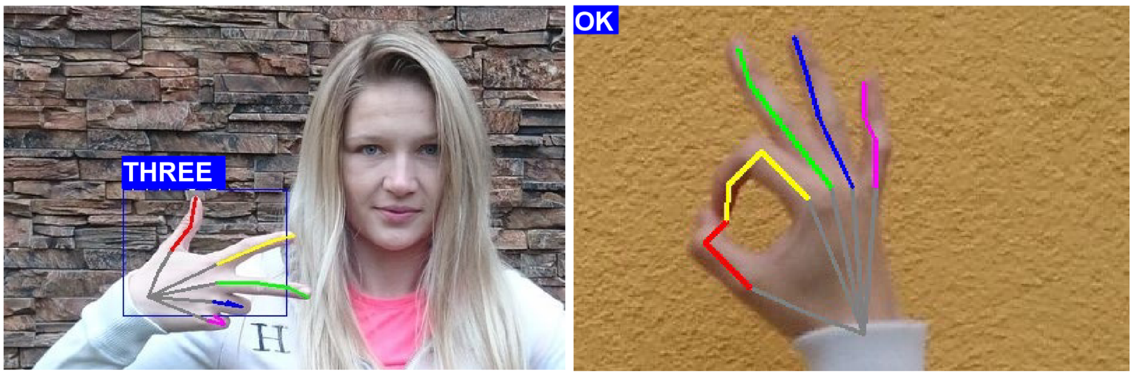


Figure 4.10 Gesture recognition evaluation images

The success rate was then calculated as a simple ratio between the correct classifications and total hand gestures used for testing. This produced a success rate of 79.8%. The success rate had to be from the beginning lower than the success rate of the left- or right-hand detection, because if the key points were meant for the other hand, they are most of the time not very usable for further gesture matching. The reason for slightly lower success rate of gesture matching is the hand key point detection neural network. Because the fingers can be hidden from the view, the neural network must guess the pose, the finger might be in. The key points of a finger can be predicted in a position that does not match the gesture rules. Because of this, even though subjectively the hand pose fits the gesture, it does not actually pass the rules.

The evaluation dataset contains 119 hands, on which the hand key point detector subjectively predicts the locations of key points correctly. Out of these 119 hands, each of the tested gestures was represented with approximately 10 to 18 occurrences in the dataset. On top of the tested gestures, 16 hand poses that did not match any of the predefined gestures were included to also test if the gesture matching logic understands the unknown hand poses correctly. From the results, a confusion matrix in Figure 4.11 was also created for visual representation of the accuracy. As is visible, most of the classifications are on the main diagonal, which means these classifications were correct, and if they were not, in most cases the gesture was classified as an unknown gesture.

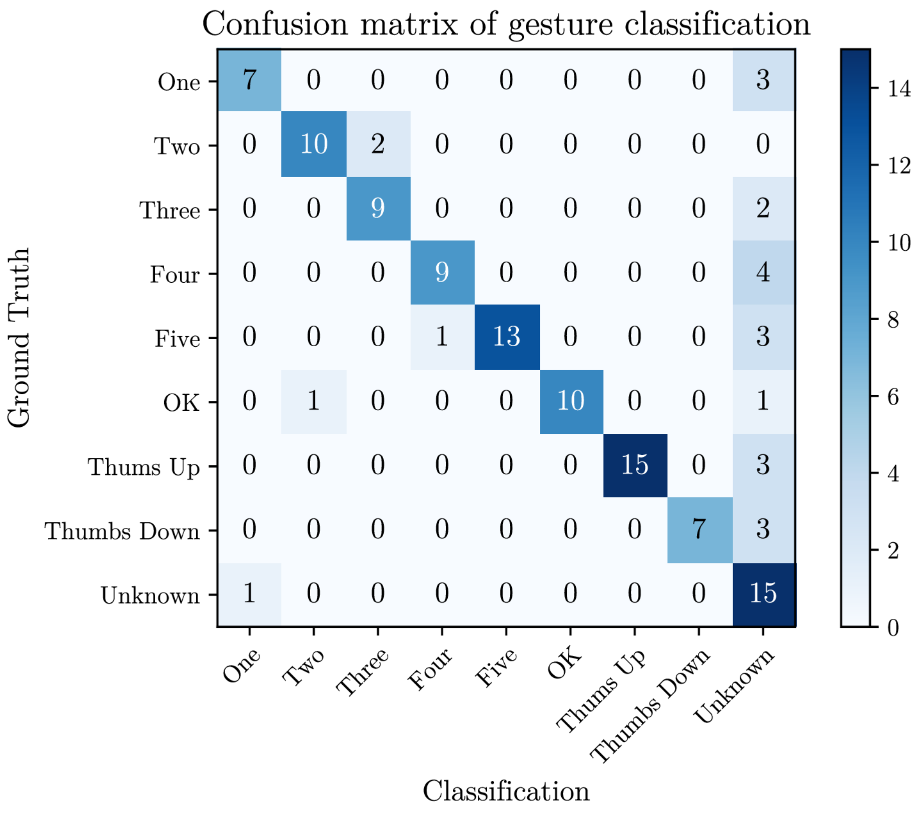


Figure 4.11 Confusion matrix for predicted gestures

5 Expansion possibilities

The computational backend of the application could be expanded in the future to provide better detection results and additional functionality.

5.1 Detector modifications

As the neural network used for hand detection does make mistakes, the result could be further improved without modifying the code by retraining YOLOv2 with an even more diverse dataset of hands, or by replacing the neural network by a different YOLO based architecture. The successor of YOLOv2, YOLOv3 [43] already exists and should produce much better results because it uses not just one but three detection layers, each in different scale of the input image, but due to much higher performance requirements, the older version was used. Changing the detector can be easily done by just swapping the configuration file and the corresponding weights for another YOLO based neural network.

5.2 Key points post processing

The OpenPose key point detector produces output with a lot of jitter that can be expected as the detection is always done on a single frame, but it could also be improved by performing detection over a time window, and smoothing out the detection in the frame at the center of the window. The result would then provide a smoother less jumpy movement of the drawn hand skeletons. That would be easier to match the gesture to.

This detection over a time window could also be used to fix incorrect detection between a couple of frames. While in the majority of the frames the finger would be detected in a similar place and in the middle of the window there would occur a sudden jump in the location of the key points, followed up by returning to a location very similar to the previously detected sequence, the incorrect location of key point could be replaced by approximating the key points position.

Both approaches would result in smoother, higher quality detection, although for the price of losing real time processing, depending on the size of the detection window.

Conclusion

The goal of this master's thesis was to study and analyze possible approaches to hand detection, gesture recognition and finger tracking, select one of the possible approaches to the issue and create a multiplatform application capable of processing images, video sequences and a camera stream.

Most of the approaches to hand detection expect at least some kind of cooperation with the person in front of the camera, be it wearing colored gloves to easily detect important points of the hand, or just expecting the hand to be in a pre-defined part of the image. Other approaches might require specialized hardware for video capture with depth channel and so on. These issues made most of the generally used approaches unusable.

To create a system capable of hand detection that is not dependent on lighting or the environment, it is almost impossible to use a rule-based system. That might result in confusion during detection in situations not thought about during the creation of the system. To avoid these issues and to create much more robust hand detection the approach using a neural network was chosen.

Because neural networks need a lot of training data to produce usable results, the training dataset was created with a combination of the EgoHands dataset, MPII Human Pose estimation dataset and a couple of videos from the New Zealand Sign Language dictionary. The combined dataset provided a high variety of hands in different environments, lighting and poses, and contained over 6000 images, usually with more than one hand in each. After training, the neural network YOLOv2 resulted in very good detection results with 89.2% of all relevant objects, in this case hands, detected. Out of all the detections, the network managed to find the hands properly in 85.7% of all cases.

The hand detection specific neural network was necessary due to the usage of another neural network, for predicting hand key points, which needs an image with only a single hand. This network is one of the networks used in the project OpenPose and can predict the position of fingers, even if the fingers are hidden from the camera view.

Even though the network is from the project OpenPose, the processing of the image before and after the forward pass through the neural network is completely different from the OpenPose implementation. The additional processing adds the capability to recognize whether the hand sent through the network was left or right correctly in almost 95% of all cases, which adds the possibility to implement gesture recognition with gestures specific for either left or right hand.

The biggest downside of using the neural network from the project OpenPose is quite slow processing due to the sheer size of the network. But with more and more powerful hardware available every couple of years, this issue will cease to be a problem in the future.

The expandable gesture recognition part of the whole system works with comparisons of Euclidean distances between important key points. Depending on those distances, the system predicts the pose the finger or fingers are in. Since the gesture recognition highly depends on the output of the neural network for the hand key point detection, the success rate cannot be as high as it could be, if the locations of hand key points were not predicted but known for certain. Even then, the tested gestures were recognized correctly in 79.8% of all the cases in the evaluation set of images.

The gestures the system tries to detect are defined in the gesture definition files and new gestures can be easily added by creating a new gesture definition file in the gestures folder. These definition files should contain the new description of the poses the fingers of the hand can be in.

When it comes to detection performance, there is no comparison between the CPU and the GPU, even when using a modern desktop CPU; an old laptop GPU is a much better choice for running neural networks, due to their parallel nature. That said, the CPU in the system should not be bad either, as the application might run into performance issues due to bottlenecking.

The whole detection logic is then connected to a graphical user interface, that makes the interaction with the logic easy and user friendly while providing additional functionality compared to using just the script included in the module with hand tracking class.

The UI of the application is composed out of a video player, which allows for easy presentation of the processing output, stepping through a video frame by frame in both directions and using a slider for skipping parts of the video completely. Through the UI it is also possible to get to the recording system that allows to save video easily, with the option to change the detection settings during the detection itself. The application also allows to save all the detected bounding boxes, and gestures into a log.

The logic behind the detection of the hands could be improved in the future by further expanding the training dataset and retraining the neural network, or even replacing it with another YOLO based network. Improvements could also be done on the hand key point detection network, by smoothing out the key points locations and fixing incorrect key point locations with position approximations.

References

[1] S. Vipul, "Gesture Recognition using OpenCV + Python," [Online]. Available: http://vipulsharma20.blogspot.com/2015/03/gesture-recognition-using-opencv-python.html.

[2] More Than Technical, "Extending the hand tracker with snakes and optimizations," 26 May 2013. [Online]. Available: http://www.morethantechnical.com/2013/05/26/extending-the-hand-tracker-with-snakes-and-optimizations-w-code-opencv/.

[3] H. Du and E. Charbon, "3D Hand Model Fitting for Virtual Keyboard System," in *2007 IEEE Workshop on Applications of Computer Vision (WACV '07)*, Austin, TX, February 2007. [Online]. Available: https://ieeexplore.ieee.org/document/4118760.

[4] More Than Technical, "Hand gesture recognition via model fitting in energy minimization w/OpenCV," 28 December 2010. [Online]. Available: http://www.morethantechnical.com/2010/12/28/hand-gesture-recognition-via-model-fitting-in-energy-minimization-wopencv/.

[5] T. Q. Vinh and N. T. Tri, "Hand gesture recognition based on depth image using kinect sensor," in *2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, Ho Chi Minh, September 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7302218.

[6] Wikipedia contributors, "Deep Learning," Wikipedia, The Free Encyclopedia, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Deep_learning.

[7] AltexSoft, "Fraud Detection: How Machine Learning Systems Help Reveal Scams in Fintech, Healthcare, and eCommerce," [Online]. Available: https://www.altexsoft.com/whitepapers/fraud-detection-how-machine-learning-systems-help-reveal-scams-in-fintech-healthcare-and-ecommerce/.

[8] V. Gupta, "Learn OpenCV: Image Classification using Convolutional Neural Networks in Keras," 29 November 2017. [Online]. Available: https://www.learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras/.

[9] S. Patel and J. Pingel, "Introduction to Deep Learning: What Are Convolutional Neural Networks?," MathWorks, [Online]. Available: https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html.

[10] K. S. Reddy, U. Singh and P. K. Uttam, "Effect of image colourspace on performance of convolution neural networks," in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, Bangalore, May 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8256949.

[11] A. Mittal, A. Zisserman and P. Torr, "Hand detection using multiple proposals," in *British Machine Vision Conference*, 2011. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/data/hands/.

[12] S. Bambach, S. Lee, D. J. Crandall and C. Yu, "Lending A Hand: Detecting Hands and Recognizing Activities in Complex Egocentric Interactions," in *The IEEE International Conference on Computer Vision (ICCV)*, Santiago, December 2015. [Online]. Available: https://ieeexplore.ieee.org/document /7410583.

[13] "New Zealand Sign Language Dictionary," [Online]. Available: https://www.nzsl.nz/.

[14] M. Andriluka, L. Pishchulin, P. Gehler and S. Bernt, "2D Human Pose Estimation: New Benchmark and State of the Art Analysis," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* June 2014.

[15] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *arXiv preprint arXiv:1612.08242,* 2016.

[16] "YOLO: Real-Time Object Detection," 2016. [Online]. Available: https://pjreddie.com/darknet/yolov2/.

[17] AlexeyAB, "Darknet," GitHub, 26 July 2016. [Online]. Available: https://github.com/AlexeyAB/darknet.

[18] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga and A. Lerer, "Automatic differentiation in PyTorch," *NIPS-W,* 2017.

[19] Y.-S. Yun, "pytorch-0.4-yolov3," GitHub, 2018. [Online]. Available: https://github.com/andy-yun/pytorch-0.4-yolov3.

[20] Google, "Precision and Recall," 1 October 2018. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.

[21] R. Padilla, "Metrics for object detection," GitHub, 2018. [Online]. Available: https://github.com/rafaelpadilla/Object-Detection-Metrics.

[22] Wikipedia contributors, "Precision and recall," Wikipedia, The Free Encyclopedia, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Precision_and_recall.

[23] Y. Sasaki, "The truth of the F-measure," *Teach Tutor Mater,* 2007.

[24] CMU Perceptual Computing Lab, "OpenPose: Real-time multi-person keypoint detection library for body, face, hands, and foot estimation," GitHub, [Online]. Available: https://github.com/CMU-Perceptual-Computing-Lab/openpose.

[25] S.-E. Wei, V. Ramakrishna , T. Kanade and Y. Sheikh, "Convolutional pose machines," *CVPR,* 2016.

[26] T. Simon, H. Joo, I. Matthews and Y. Sheikh, "Hand Keypoint Detection in Single Images using Multiview Bootstrapping," *CVPR,* 2017.

[27] Microsoft, "MMdnn," GitHub, 2017. [Online]. Available: https://github.com/Microsoft/MMdnn.

[28] Wikipedia contributors, "Atan2," Wikipedia, The Free Encyclopedia, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Atan2.

[29] "Kivy: Cross-Platform Python Framework for NUI Development," Kivy, [Online]. Available: https://kivy.org/.

[30] Iconic, "Open Iconic," GitHub, 2014. [Online]. Available: https://github.com/iconic/open-iconic.

[31] "Video Codecs and Pixel Format," 2011. [Online]. Available: https://www.fourcc.org/.

[32] T. Bravenec, "Computer vision and hand gestures detection and fingers tracking," GitLab, 2019. [Online]. Available: https://gitlab.com/tbravenec/computer-vision-and-hand-gestures-detection-and-fingers-tracking.

[33] PyTorch, "Get Started," [Online]. Available: https://pytorch.org/get-started/locally/.

[34] NVIDIA Corporation, "CUDA Compatibility," [Online]. Available: https://docs.nvidia.com/deploy/cuda-compatibility/index.html#binary-compatibility___table-toolkit-driver.

[35] "Kivy: Installation on Linux - Kivy," Kivy, [Online]. Available: https://kivy.org/doc/stable/installation/installation-linux.html.

[36] T. V. Hemert, "Taran uncut-ish interview," 3 April 2017. [Online]. Available: https://www.youtube.com/watch?v=Fues_3ZarpE.

[37] Bitwit, "Are Ryzen APUs a GOOD alternative to overpriced GPUs?," 12 February 2018. [Online]. Available: https://www.youtube.com /watch?v=N1DgTvGxmAQ.

[38] Tzutalin, "LabelImg," Git code, 2015. [Online]. Available: https://github.com /tzutalin/labelImg.

[39] Deep Systems LLC, "Supervisely," 2017. [Online]. Available: https://supervise.ly/.

[40] M. Harris, "Inside Pascal: NVIDIA's Newest Computing Platform," 5 April 2016. [Online]. Available: https://devblogs.nvidia.com/inside-pascal/.

[41] E. Kilgariff, H. Moreton, N. Stam and Bell Brandon, "NVIDIA Turing Architecture In-Depth," 14 September 2018. [Online]. Available: https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/.

[42] Freepik, "Graphic resources for everyone," [Online]. Available: https://www.freepik.com.

[43] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv preprint arXiv:1804.02767,* 2018.

[44] HarisIqbal88, "PlotNeuralNet," GitHub, 2018. [Online]. Available: https://github.com/HarisIqbal88/PlotNeuralNet.

List of Symbols and Abbreviations

k_i	Hand key point with index i
C	Weighting constant

GPU	Graphics Processing Unit
CPU	Central Processing Unit
CNN	Convolutional Neural Network
YOLO	You Only Look Once
OS	Operating System
IoU	Intersection over Union
TP	True Positive
FP	False Positive
FN	False Negative

Attachments

Complete structure of the git repository [32] containing the source code, icons and images used in the development of the application:

```
\ ........................................root of the repository
├─.git
├─cfg ...............................configuration files for CNNs
│ ├─keypoints.py
│ └─yolov2.cfg
├─darknet ............................darknet interpreter package
│ ├─__init__.py
│ ├─cfg.py
│ ├─darknet.py
│ ├─region_layer.py
│ ├─utils.py
│ └─yolo_layer.py
├─data ................................images used in UI elements
│ ├─icons
│ │ ├─app.ico
│ │ ├─cog.png
│ │ ├─file.png
│ │ ├─floppy.png
│ │ ├─image.png
│ │ ├─info.png
│ │ ├─log.png
│ │ ├─media-pause.png
│ │ ├─media-pause-disabled.png
│ │ ├─media-play.png
│ │ ├─media-play-disabled.png
│ │ ├─media-record.png
│ │ ├─media-record-disabled.png
│ │ ├─media-step-backward.png
│ │ ├─media-step-backward-disabled.png
│ │ ├─media-step-forward.png
│ │ ├─media-step-forward-disabled.png
│ │ ├─media-stop.png
│ │ ├─reload.png
│ │ ├─reload-disabled.png
│ │ └─video.png
│ └─logos
│   └─BUT_symbol_RGB_EN.png
├─enums ..........................package with enumeration modules
│ ├─__init__.py
│ ├─finger_bends.py
│ ├─finger_spread.py
│ ├─hand_directions.py
│ ├─hands.py
│ ├─input_types.py
│ ├─thumb_directions.py
│ ├─thumb_positions.py
│ └─thumb_tip_positions.py
│
```

```
├─gestures ............................gesture definition files
│ ├─five.json
│ ├─four.json
│ ├─ok.json
│ ├─one.json
│ ├─three.json
│ ├─thumbs_down.json
│ ├─thumbs_up.json
│ └─two.json
├─.gitignore
├─constants.py
├─get_models.py
├─hand_gestures.py
├─hand_keypoints.py
├─hand_tracking.py
├─LICENSE
├─logger.py
├─main.kv
├─main_app.py ..........................................main script
├─README.md
├─requirements.txt ......................list of required packages
└─utils.py
```