Department of Informatics

Technische Universität München

Bachelor's Thesis in Informatics

Data Preprocessing for Sign Language Detection with Machine Learning Models

Christian Kellinger

Department of Informatics

Technische Universität München

Bachelor's Thesis in Informatics

Data Preprocessing for Sign Language Detection with Machine Learning Models

Datenvorverarbeitung für Gebrdensprachenerkennung mit Maschinellem Lernen

Author:	Christian Kellinger
Examiner:	Univ.-Prof. Dr. Christian Mendl
Assistant advisor:	Dr. Felix Dietrich
Submission Date:	October 15th, 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

October 15th, 2021 Christian Kellinger

Acknowledgments

Firstly, I want to thank my advisor Felix Dietrich for his regular and helpful advice on my thesis, its appropriate scientific style, and the complete process around it. Secondly, I want to thank my fellow students in the pose estimation group for interesting input and general advice, especially Ashish Khanal, who helped me getting started on the computer vision part of this thesis. And thirdly, I want to thank my friend Leonardo, who was always available for proofreading my work and commenting my English style.

Abstract

With the general progress in artificial neural networks, sign language detection from live video feeds has become a popular research field in recent years. Right now, sign language detection is where spoken language detection from live audio feeds was 10 years ago, being restricted in context, vocabulary, grammatical diversity and user-friendliness. But with the rise of efficient and light-weight applications for real-time human pose estimation on mobile devices, a major improvement in quality and usability of sign language detection is possible in the next years.
In this thesis I talk about why preprocessing is necessary for sign language detection, and how the structure of sign language itself sets special requirements for the preprocessing of sign language detection. I talk about general computer vision algorithms used for preprocessing, and give a quick overview over the history of sign language detection and its development.
For the main part of my thesis, I examine MediaPipe Holistic, a real-time human pose estimation application using machine learning that reaches unprecedented accuracy on mobile devices. I conclude that it has the possibility to make sign language detection with machine learning viable for real-time applications. By analysis of its code and experiments I conclude that it reaches its accuracy and low latency by intelligently re-using computation results and not queuing up frames which can not be analyzed in real-time anymore. Additionally I conclude that preprocessing based on human skin colour is very helpful for sign language detection, but MediaPipe Holistic's own human skin colour based preprocessing actually degrades its accuracy when handling sign language. In summary, MediaPipe Holistic shows the benefits human pose estimation can offer to make sign language detection real-time viable even on low-end hardware by only analyzing frames if the model is not busy computing prior frames. But it cannot be deployed as-is as preprocessing, because it fails to accurately track typical poses used in sign language.

Contents

1. Introduction

In the modern era real-time sign language video-to-speech translation would be an important tool for a barrier-free and inclusive society. For this purpose applications using supervised machine learning concepts and algorithms seem fitting, because they have proven to be powerful in dealing with classification problems. Those concepts are already successfully deployed in the translation of spoken languages, for example in Google's free Google Translate[1], which utilizes a 16-layer long-short-term-memory recurrent neural network [18].

Unlike spoken language translation, the problem of bidirectional sign language translation consists of two separate sub-problems: Firstly, detecting the spoken language, and generating or rendering sign language from it (*Sign Language Generation*, audio-to-video). Secondly, detecting the sign language and translating it into spoken language or text (*Sign Language Detection*, video-to-audio or video-to-text). In this thesis we will only consider topics regarding the sign language detection sub-problem.

Looking at the state-of-the-art of sign language detection, we can see that it is right now roughly where spoken language translation was 10 years ago: existing solutions, like SignAll[2] and SLAIT[3], are limited in real-life usability because they lack in vocabulary, only work in specific scenarios, cannot recognize advanced grammatical structures, or require complex physical setups to function, like special gloves or multiple cameras.

Only in recent years we have seen substantial improvement in sign language detection with artificial neural networks [2] [3] [10], while earlier approaches reliant on other models showed less accuracy [14] [13] [16].

But with the advancement of light, accurate and customizable machine learning solutions for human pose estimation, like MediaPipe Holistic, we could see an even greater improvement in the next years, as indicated by [5]. Pose estimation software provides real-time human body tracking, in recent years even on mobile hardware. Using a variety of computer vision algorithms as well as machine learning, they extract a set of 2D or 3D landmark coordinates of the joints and other important parts of the human body from an image. It still logically keeps all necessary information for sign language detection, only in

[1] https://translate.google.com/
[2] https://www.signall.us/
[3] https://www.slait.ai/

a structured way, while it simultaneously reduces the amount of input values for the main detection model drastically.

The simplified pipeline of a sign language detection model using pose estimation as preprocessing could look like in figure 1.1

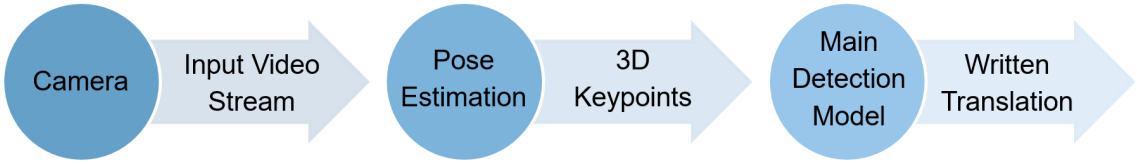


Figure 1.1.: Simplified Sign Language Detection Pipeline with Pose Estimation

With the continuous improvement we see in accessible real-time human pose estimation right now, it is reasonable to assume that optimizing the preprocessing of the raw video input data is critical in improving convenient real-time machine learning solutions for sign language detection further. This is not limited to pose estimation as a preprocessing step: pose estimation itself needs to utilize computer vision algorithms for preprocessing to run as efficient as possible.

This thesis is structured as follows: In chapter 2, I will talk about background theory useful for understanding of sign language detection, including the necessities for sign language detection preprocessing resulting from the structure of sign language itself in 2.1, a general explanation of why we need preprocessing in 2.2, an overview over basic and common computer vision algorithms used for preprocessing in 2.3, an overview of high-quality data sets for sign language detection in 2.4, and a brief explanation of the history and origin of sign language detection in 2.5.
In chapter 3, the main part of my thesis, I give an introduction to MediaPipe in 3.1, examine human pose estimation as preprocessing for sign language detection with the example of MediaPipe in 3.2, and also investigate MediaPipe's own preprocessing and estimate its suitability for sign language detection in 3.3.
In chapter 4, I will give an summary of my findings and an outlook over the topic.

2. Background Theory

In this chapter I will give an overview of background information which is important for this thesis regarding sign language, general computer vision preprocessing, and the progression of sign language detection.

2.1. Structure of Sign Language

An understanding for the structure and characteristics of sign language is as important for sign language detection as an understanding of spoken language is e.g. for natural language detection. Since sign languages were constructed by deaf people with little to no input by hearing people, the differences between spoken language and its respective sign language counterpart (e.g. American English and American Sign Language) run deeper than the obviously different transmission medium.

The important aspects for sign language detection shall be explained here on the example of ASL (American Sign Language), but it can be generalized to all variations of sign language, for example DGS (Deutsche Gebrdensprache, German Sign Language).

2.1.1. Basic Sign Language Components

The key components of sign language are the eponymous signs made with the hands. According to Aarons [1, p. 6], each sign consists of four components:

- hand shape
- hand location
- hand movement
- palm orientation

For the most cases, one word corresponds exactly to one sign, which are formed sequentially in the standard word order subject-verb-object.

But there are some words whose signs can not be differentiated by hand sign alone. For example, according to [17], the words ”brother” and ”sister” in DGS have the exact same hand gesture, and can only be distinguished because the signer makes different shapes with the mouth for both words.

2.1.2. Advanced Sign Language Components

Apart from the hand signs, sign language consists of three additional components which are described with the term *non-manual marking*.

According to [1], these three components and their uses are:

- **Head/Body pose**: The combined head and body position conveys temporal information, while the head expresses grammatical constructs like questions, affirmations, denials and conditional clauses.
- **Facial Expression**: In combination with other non-manual markings, the facial expression makes up a big part of sign-language grammar. For example, it can express subjunctives.
- **lip pattern**: The lip movement supports the meaning of the hand signs and are often borrowed from the spoken version of the respective word. This way they clarify the meaning and allow differentiation between similar or identical hand signs.

As we can see, sign language communication in its total is conveyed via four channels. Hand signs and lip patterns mostly form the words, while head/body pose and facial expressions encapsulate most of the grammar of sign language. This is in contrast to most spoken languages, where grammatical constructs are indicated with adverbials, for example in English.

2.1.3. Consequences for Sign Language Detection

With this knowledge about the structures of sign language we can evaluate different steps of the preprocessing pipeline: As we can see, we need to analyze all four channels simultaneously and then combine the results for an accurate detection of natural sign language. Otherwise we are presented a hierarchy of sign language complexity if we do not utilize all channels:

- If we just use the hand signs to train a machine learning model, we are theoretically still able to detect basic main clauses.
- If we also analyze the lip patterns, we do not gain new possibilities, but instead increase the accuracy of our model, as shown by [17]
- If we now additionally analyze head position and facial expression, we should be able distinguish between different grammatical constructs.

Ideally we want to realize our detection with only one camera for simplicity and user-friendliness. But with each additional channel this becomes increasingly complicated, as

the different channels vary greatly in dimension, position and amount of movement and subtlety of movement. Therefore, it makes sense to split the detection among multiple sub-models, analyzing body pose, face and hands separately, each deploying its own pipeline suiting the needs of this sub-model. Extra care needs to be taken here with preprocessing, to ensure that it is catered towards the special needs of the singular channels.

2.2. Why Preprocess?

In machine learning, data preprocessing is the task of preparing and refining the raw input data before feeding it into the main machine learning model. Data preprocessing has two motivations: Reducing the computational effort, and improving the accuracy of the inference.

2.2.1. Reducing Computational Effort

For machine learning models to be applicable in everyday life they need to function in uncontrolled environments on mobile hardware in real time. As the environment cannot be exactly predicted, this means that there is a lot of input data which the model has to consume to be able to fulfill its task. Without any preparation, it is increasingly unlikely for a machine learning model on mobile hardware to run in real time with this amount of data to process.

That is where preprocessing is used in the pipeline: Using human understanding of the system and the task we automatically reduce the raw input data to the parts that are relevant, removing unnecessary information. As the main model now does not have to differentiate between important and unimportant aspects itself, its computational effort is reduced compared to when it had to handle the raw data.

Of course for the preprocessing to be viable, three criteria must be met:

- The reduction of computation in the main model has to surpass the additional computation added by the preprocessing step, so that we come out with a significant overall reduction in computational effort.
- The performance of the model must not be significantly reduced by the preprocessing (e.g. by wrongly identifying important aspects of the data as unimportant)
- The preprocessing should not require expensive or difficult-to-use hardware. For example a solution relying on multiple camera angles for a 3D reconstruction of the person might be very accurate, but is not practical in the use-case we strive for, which is sign language detection "in the wild"

2.2.2. Improving Accuracy

In the training phase, there may be correlation without causality between important and unimportant information in the training data, which worsen the models performance on the test and validation data if the model learned to rely on this pattern. Or worse, if there is correlation in training, test and validation data, the model is tested correctly, but is unusable on real use-cases.

By preprocessing the input data we can reduce the "distraction" of the model by making it focus on data we as humans identified as important for the task at hand. This reduces the risk of correlated information in the training data set.

2.3. Important Computer Vision Algorithms

As sign language detection software naturally has to use cameras as input devices, this makes computer vision algorithms an important part of the implementation. For detection solutions with machine learning they typically make up a large part of the preprocessing pipeline by preparing and refining the input video data to make it more digestible for the main model.

The computer vision algorithms deployed in a sign language detection pipeline should therefore have a net positive outcome on the accuracy/effort ratio. Meaning that they either should have low computational effort and simultaneously save the main model computational load, or they should improve the quality of the main model inference enough to make up for the additional effort. This is especially true if the solution is supposed to operate in real time, as this requires low latency.

An extensive overview of computer vision algorithms is given by Davies [4] and Forsyth [7], while an evaluation of different approaches for sign language is given by Oudah et al. [15]. While no where near complete, important ones for sing language detection in the scope of this thesis are:

1. Image Scaling
2. Noise Reduction
3. Image Segmentation
4. Skin Color Detection
5. Bounding Boxes

2.3.1. Image Scaling

Image scaling is the task of resizing a digital image to a desired number of pixels in width and length, without removing parts from the image. This can either be done to a higher resolution, also known as upscaling or upsampling, or to a lower resolution, also called downscaling or downsampling.

Image scaling is utilized in any application using neural networks, because they are designed to have a specific number of input neurons. For example, if we want to input 32x32 grayscale images, we need $32 \cdot 32 = 1024$ input nodes. If we instead want to input 16x16 RGB images, we need $16 \cdot 16 \cdot 3 = 768$ input nodes, considering that each pixel is a triple of information in RGB. To match the architecture of the neural network, the image needs to be re-scaled to the specified resolution and converted to the specified color dimension.

There exists a multitude of algorithms for this task, with varying quality regarding different criteria, like edge preservation etc. But as can be expected, higher quality algorithms take longer to complete and therefore introduce higher latency, which can be a problem for like sign language detection.

Common algorithms are:

- Nearest Neighbour Interpolation
- Biliniar Sampling
- Bicubic Sampling
- Sinc and Lanczos resampling
- Specially trained Convolutional Neural Networks

2.3.2. Image Noise Reduction

When filming with any digital camera, the image can never be perfectly captured. This is a problem of every physical measuring process: The measured value y is never the exact value x, it always diverges by a random error ϵ. Mathematically this can be modeled as $y = x + \epsilon$, where ϵ is called the *random gaussian measurement noise*.

Although the noise can vary in intensity depending on the complexity and sensitivity of the measurement device, it is always desirable or even necessary to remove or reduce the noise before the measurement is further processed.

In the situation of sign language detection, our measurement device is a digital camera measuring different color values of the single pixels. Because of random noise, these values can be randomly off, resulting in a noisy image. This can be a problem for machine learning models, as important details, like edges, can be lost this way.

But the problem of denoising is also difficult, as "[...] noise, edge, and texture are high frequency components, it is difficult to distinguish them in the process of denoising and the denoised images could inevitably lose some details." (Fan et al., [6]).
There are many algorithms for denoising, as can be seen in [6]. They can be roughly categorized into four classes:

- Spatial domain methods, which work on the normal image space
- Transform domain methods, which work in the frequency space with the Fourier-transformation of the image
- Variational methods, which use probabilistic models to find the most likely original image
- Convoluted neural networks, which are trained to denoise images.

2.3.3. Image Segmentation

Image segmentation is the task of recognizing related pixels in an image. For example separating an object from the background, or separating multiple objects from another.

In sign language detection, image segmentation can be used to separate the signer from the background to eliminate the backgrounds impact on the main model, thus making detection more robust to real life use cases "in the wild".

According to [15], there are many different image segmentation algorithms based on many different principles, including:

- Tresholding
- Motion Detection
- Clustering Methods
- Edge Detection Methods
- Neural Networks
- Variational Methods

2.3.4. Skin Color Detection

Skin color detection is a subcategory of image segmentation specifically aimed at detecting humans, and therefore very suitable for sign language detection. It aims to find accumulations of pixels within a certain color range in the image.

Skin color detection is very dependent on the color space of the image for the quality and robustness of its inference. Certain color spaces, like RGB, are less suited, because the color value of skin can change drastically in this spaces with the lighting. According to [15], saturation- or luminance-based color spaces, like HSV and YUV, are better suited for skin color detection, because lighting conditions can be separated better from the base skin color. This is usually no problem, as conversion from standard image spaces like RGB or BGR to HSV or YUV is fairly easy in both directions.

2.3.5. Bounding Boxes

The term ”bounding boxes” refers to a technique which combines other preprocessing steps with human knowledge about the task we want to solve. It aims at identifying, extracting and annotating so-called *regions of interest* (”ROIs”) which are the main interest for the subsequent parts of the model for solving the task at hand. For sign language detection, important ROIs are the hands and the head.

For human readability the ROIs are often visually enclosed by rectangular bounding boxes. To remove unrelated information from the whole image, only the content of the bounding boxes is cropped out and delivered to the detection model. For example, a model for detecting hand shapes only needs to be fed the hands. Therefore it makes sense to first invoke bounding box detection for hands on the whole image. Only in a second step the main model with the task of identifying the exact shape of the hand gets the content of this bounding boxes as input, reducing distractions from the background.

This step seems to be especially helpful for sign language detection, as indicated by Brumm and Grigat [12].

2.4. Available Sign Language Data Sets

For a machine learning application to learn sign language detection, we need an a data set with a great variety of people signing in natural sign language, as well as annotations of what they are signing and which markings and signs the use. Optional, but beneficial for an approach deploying pose estimation, would be an additional full-body analysis with pose estimation software, like OpenPose or MediaPipe.

Due to a recent surge in interest on the linguistic research of sign language, many high-quality sign language data sets from all around the world are available. For this thesis, I focused on the public DGS-Corpus[1], a long-term project from the Academy of Sciences in Hamburg, as its material depicts real deaf people holding natural sign language conversations in pairs of two.

An extensive overview over high-quality data sets for european sign languages [11] is given by EASIER[2], a project funded by the European Union for the research of a sign language translation system using machine learning.

2.5. A Brief History of Sign Language Detection

Sign language detection evolved from general gesture detection, which can be considered as a subset of sign language detection. It extends the focus in two dimensions: from static hand gestures to dynamic hand signs, and from only the hands to the face and upper body as well. These changes altered the requirements of the main model, from memoryless models to models with memory, as well as the requirements for the preprocessing pipelines.

The following section on gesture detection are a compilation of information from Oudah et al. [15].

2.5.1. Gesture Detection: The Predecessor of Sign Language Detection

Hand gesture detection has always been an interesting and popular research field as it offers an intuitive way of human-machine-interaction for many possible platforms and use-cases. Generally, approaches to gesture-detection can be classified into two types: The hand-sensor type and the computer vision type.

Hand-Attached Sensors

This approach uses glove-like devices with attached sensors which the user has to wear on his hands. The sensors physically measure position and orientation of the palm, as well as the relative position of the fingers to the palm to generate a 3D image of hand-keypoints, which can then be recognized and used to control a computer system. The glove itself is connected to the computer either by wires or wireless.

A great variety of sensors can be used for this task, such as gyroscopes, acceleration sensors, angular sensors etc., or a combination of those.

[1] https://www.sign-lang.uni-hamburg.de/dgs-korpus/index.php/welcome.html
[2] https://www.project-easier.eu/de/

This approach has been proven to accurately and consistently track the hands of the wearer. They are however not intuitive and accessible, as there are the following problems for the use case we strive for:

1. The complex technology comes with high acquisition costs.
2. The gloves may not be suitable for people with certain illnesses or injuries, as well as elderly people.
3. The wiring and the need to first put the gloves on may be restrictive in certain situations.

These drawbacks prevented the hand-sensor approach from spreading to the general population.

Computer Vision Recognition

This approach relies on detection of the hands via computer cameras and computer vision algorithms. Different approaches rely on different types of cameras, like RGB, infrared [9], depth or kinect cameras. There are two sub-types in this approach:

Marked Hands Similar to the sensor-glove, this approach also utilizes gloves for hand detection. But instead of relying on sensors attached to the gloves, markings are painted or physically attached on the glove in certain patterns, which allows the computer to create a 3D image of the hand just by perceiving the markings trough the camera.

This approach is a compromise between the sensor-glove and the naked hand camera, because it still needs extra equipment which comes with drawbacks for people who might not be able to wear them. On the contrary, the acquisition cost is not nearly as high as the sensor glove, because no complex technology is built into it.

Naked Hands This is the most unobtrusive approach, as no physical contact is required with any equipment. This approach however had to fight with low detection accuracy for a long time, making it useless for commercial or practical use. Only with the rise of deep neural networks and their powerful classification capabilities this approach reached sufficient accuracy for real use-cases.

2.5.2. From Gestures to Signs

Sign language detection evolved from a general use-case of gesture detection to a specialized field. As already said, sign language can only be captured with all its characteristics if we include the movement and positioning of the upper body, head and lips additionally

to the hands. Therefore, basic gesture detection models can be used to detect very simplified sign language, but complete sign language need more complex systems which also track the upper body and head. This can be solved similarly to hand gesture detection either with sensors or with cameras, only that here the sensors are even more intrusive and therefore impractical. Hence the camera approach is the only viable option.

This option has been greatly climbing in popularity with the rise of neural networks and machine learning frameworks which make the process of setting up different inference pipelines and combining their information increasingly easier.

3. Human Pose Estimation as Preprocessing for Sign Language Detection

To absolutely estimate the quality of different preprocessing approaches for sign language detection with machine learning, one would have to implement those preprocessing steps and then train a neural network for sign language detection from scratch. But this is out of scope for a bachelor's thesis, because the training, labeling, feeding and hyperparameter-tuning would take a lot of time, effort and computational resources.

SO Instead I tried to exemplarily examine an existing, free, and available solution which can be used for sign language detection in real time, and try to explore which preprocessing they use, as they apparently found a good approach.

Although there are many solutions for translation of single signs, there is no fully functional and convenient commercial or free software for sign language translation available now; it's development is roughly 10 years behind that of written speech translation. Nevertheless there are some projects and companies which aim for natural sign language translation, as written in the introduction.

But during my research I found another similar class of applications, which can be examined as both a surrogate for sign language detection, as well as effective preprocessing for sign language detection itself: human pose estimation applications working with machine learning.

Human pose estimation applications take a video stream as an input, and deliver as output for every frame a set of 2D or 3D keypoints which track important joints of the human body. These points are sufficient for creating a 2D or 3D stick-man model of the perceived human body. A very known free-to-use solution is OpenPose[1], which focuses on real-time multi-person full pose estimation with 135 2D keypoints.

But OpenPose itself does not deploy machine learning, but instead relies on **part affinity fields**, which makes it unsuited as a surrogate. So instead I decided to examine a different solution which utilizes machine learning to offer real-time human pose estimation with 3D keypoints on mobile devices: MediaPipe Holistic.

[1] https://github.com/CMU-Perceptual-Computing-Lab/openpose

3.1. Introduction to MediaPipe

MediaPipe[2] is a framework by Google which offers organization and modularization for data flow pipelines of machine learning software. If an application os designed to take a constant stream of information as input, like video, audio or other data streams, MediaPipe handles the data processing pipeline, offers synchronization for multiple information sources, and organizes computation among multiple CPU cores. It is currently in alpha at version 0.7, and is made public since 2019

As a demonstration of the framework's capabilities, MediaPipe offers out-of-the-box, yet customizable solutions for real-time human pose estimation and object detection on CPU and GPU. These solutions can be accessed either via Python, C++ and JavaScript APIs, or by compiling the code with Bazel and executing it directly on the command line. The demo applications, which run on both desktop and mobile hardware, include object tracking, object recognition, selfie segmentation, hand tracking, face tracking, pose tracking, and holistic tracking which combines hand, face and pose tracking. For these solutions MediaPipe utilizes completely trained TensorFlow inference models.

The MediaPipe framework and demo solutions are freely available on GitHub[3], as MediaPipe is licensed under the Apache License 2.0.

The strength of MediaPipe is its flexibility and readability. It organizes the data flow in a graph, as can be seen in figure 3.1, which makes the pipeline comprehensible for humans, especially for complex pipelines. In this graph the nodes consume input packages, process them by performing specified computations, and produce output packages which the nodes further down the graph will consume. Every package has a timestamps, meaning there is no global clock synchronizing the data flow. Instead, synchronization is handled locally by the nodes according to an input policy, which itself is completely adjustable.

Talking about MediaPipe in the context of sign language detection preprocessing, especially the ready-to-use hand or holistic tracking are interesting to us. As explained in section 2.1, the former contains all information to detect simple sign language, the latter to detect complete sign language.

I will examine at MediaPipe preprocessing in two ways:

1. MediaPipe Holistic as a whole as a preprocessing step in a sign language detection application pipeline, because it condenses a RGB video with hundreds of thousands of pixels into a stream of few landmark coordinates.

[2] https://mediapipe.dev/
[3] https://github.com/google/mediapipe

Figure 3.1.: Basic MediaPipe Graph

2. The preprocessing of MediaPipe Holistic itself, to see which preprocessing steps it deploys for its own inference.

3.2. MediaPipe Holistic as Preprocessing

The human body tracking of MediaPipe Holistic has the potential to be used as an effective preprocessing step for a sign language detection pipeline. If we assume a RGB webcam input with a SD resolution of 640x360 pixels, we have $640 \cdot 360 = 230.400$ pixels per frame. As each pixel carries a triple of information, there are 691.200 integer values of information per frame. Filming with 30 frames per second, this results in $20.736.000$ floats of information per second.

If we were to first preprocess this frame with MediaPipe Holistic, it would extract from these 691.200 values:

- 33 pose landmarks
- 21 hand landmarks per hand
- 468 face landmarks

meaning a total of 543 landmarks. With 3D coordinates for each landmark, this means a total of $543 \cdot 3 = 1629$ float values of information per frame, meaning 48.870 per second.

Let us now assume that we want to run sign language detection on one frame with an artificial neural network of fixed architecture first without, then with preprocessing. Because

every information of this frame has to to be fed into the neural network at once, this means that we need $n_1 = 691.200$ input neurons in the first case, and $n_2 = 1629$ input neurons in the second case for our input layer. Let the number of neurons in the first hidden layer be fix with m, and assume full connection between the input layer and the first hidden layer.

Neglecting the computational effort of the activation function as a constant offset, we can model the computations from one layer to the other with the following matrix-vector-multiplication:

$$h_1 = A \cdot x \tag{3.1}$$

where $x \in \mathbf{R}^n$ is a vector containing the input values, $h_1 \in \mathbf{R}^m$ is a vector containing the values of the first hidden layer, and $A \in \mathbf{R}^{n \times m}$ is the weight matrix connecting the input layer and first hidden layer. As matrix-vector-multiplication for a matrix $A \in \mathbf{R}^{n \times m}$ can be done in $O(n \cdot m)$ operations, we can say that for fixed m, the computational effort is linear in the number of input neurons n.

The computations of the subsequent hidden layers are independent from the input layer, so it is the same for both version of the artificial neural network and can be neglected. Therefore, as we have a reduction in input values from $n_1 = 691.200$ to $n_2 = 1629$ by 99,76%, this means that the computational effort only in the main sign language detection model is also reduced by 99,76% per frame from the case without preprocessing to the case with preprocessing. By using MediaPipe Holistic as preprocessing, we theoretically reach an enormous reduction in computational effort for the main model, while simultaneously also already filtering out unnecessary information and structuring the important information for an imaginary artificial neural network with the task of sign language detection.

Logic and our human intuitive understanding about sing language detection tells us that we still have all information necessary for sign language detection after using human pose estimation as preprocessing. Therefore we can say that MediaPipe Holistic can be used as an efficient and effective preprocessing method for real-time sign language detection by offering division of labour: Instead of tasking the main model to filter out unimportant information and structure the important information, we outsource this effort to a dedicated human pose estimation model, so the the main model can focus only on the translation.

Of course human pose estimation is only viable as preprocessing **if** it 1.) proves to be sufficiently accurate, and 2.) is less heavy in computation than what was reduced in the main model's computations, so that we have a net reduction in computational effort by deploying it.

3.3. Preprocessing of MediaPipe Holistic

Human pose estimation with MediaPipe Holistic is in many ways similar to real-time sign language detection with machine learning. Both analyze input video streams of humans moving their body, face, arms and hands. Both need to deploy computer vision algorithms to reduce and structure the input information so they can function in real-time with low latency.

Therefore, assuming MediaPipe Holistic as a substitute for real sign language detection, it is justified to also assume that good preprocessing for MediaPipe Holistic also constitutes good preprocessing for sign language detection.

As MediaPipe Holistic's real-time tracking works visually very good on mobile hardware, as I tested on my own notebook, my presumption is that it does utilize an efficient and effective way of video preprocessing for their detection and tracking solutions. This leads to my hypothesis: by finding out which preprocessing steps MediaPipe Holistic deploys, I can assume these exact steps to also be viable in a real-time sign language detection application.

Because MediaPipe is a framework, the offered demo-solutions, like MediaPipe Holistic, are customizable by changing the properties of the graph, adding self-written nodes, or setting new input policies. But the elemental inference nodes, which do the actual tracking with TensorFlow, are only available fully trained, with no further information about their architecture or the training data set they used. So we can't carry out experiments based on swapping or changing essential parts of the preprocessing pipeline which we think work better, because without any possibility to train the changed pipeline in the exact same way, the new changes are bound to perform worse, because the model was trained with the original preprocessing.

Therefore, my attempt to find out which preprocessing MediaPipe Holistic uses relies on two approaches:

- **The Inner View:** I looked at the code of MediaPipe Holistic's graph to determine important preprocessing steps in the data flow.

- **The Outer View:** I made a series of experiments where I input a suitable video to a MediaPipe model and measured the time and accuracy of the tracking. Then I transformed the video and let MediaPipe analyze it again. By comparing runtime and accuracy of the analysis to the original case, I want to figure out important properties of the video needed for good inference.

3.3.1. Inner View of MediaPipe

The complete code of MediaPipe is maintained via GitHub. Anyone can freely download and execute it, and repositories with contributions by the community are listed on the MediaPipe website. Installing it and running the ready-to-use demo solutions can be easily done, either by Python, C++ or JavaScript APIs, or by compiling othem via Bazil and running them in the terminal.

Every application designed with the MediaPipe framework can be separated into two parts:

- The *graph*, which specifies the data flow of this particular application with its topology, as well as the interface of the computation nodes. It defines their names, inputs, outputs, and connections among each other.
  For modularity and reusability, graphs can also be imported as subgraphs into the current graph and treated like nodes.

- The *implementation*, which defines for every node how the outputs are computed from the inputs based on the interface. MediaPipe uses C++ to write the implementation.

The graphs are defined in .pbtxt files, which stands for *protocol buffer text*. They are humanly readable, but graphs with more than a few nodes are not really easily understandable. Because of this, MediaPipe offers the MediaPipe Visualizer[4] to visually display the graph with it's nodes and connections, but at the current version it only seems to work in the chrome browser.

For example, the graph we can see in A.1 is created by the .pbtxt code in 3.1.

I will examine the graphs of MediaPipe Holistic, because it is the only useful solution as a sign language detection surrogate. As the graphs and the navigation trough them can be quite complex, visual representations of each graph I talk about can be found in appendix A and are referenced individually at the respective place.

Starting the Analysis

The topmost graph of MediaPipe Holistic executed on CPU is **holistic_tracking_cpu.pbtxt**, shown in A.2. The most important nodes here are:

- The *FlowLimiter* node

[4] https://viz.mediapipe.dev/

- The *HolisticLandmarkCpu* node, which executes the tracking on CPU
- The renderer and annotator, which render the output video with the landmark annotations overlaid

The *FlowLimiter* plays a big part in enabling MediaPipe to run on weaker mobile hardware in real time. If frames come in while the subsequent nodes are still busy processing a configurable maximum limit of prior frames, these incoming frames are dropped at the flow limiter, before any further computation time is wasted on them. Only if the subsequent nodes finished their processing, a new frame is allowed to pass trough. Although this flow limiter reduces the accuracy of the tracking, at the same time it reduces latency of the whole model as it prevents it from queuing up data which is not real-time any more. For this, the limit is often set to 1.

To understand the tracking we need to look at the tracking node. This is actually a subgraph implemented in **holistic_landmark_cpu.pbtxt**, shown in A.3. Here we can see separate nodes for the three components of holistic tracking, namely pose, hand and face landmarking. Important to notice here is that pose landmarking is carried out before hand and face landmarking, as the latter two receive the pose landmarks as part of their input. That is because they use the information of the pose landmarks to calculate ROIs for the face and hands, which avoids the invocation of dedicated models for face and hand ROI estimation and therefore saves computation.

Navigating even deeper we take a look at the *PoseLandmarkCpu* node, which is actually another subgraph implemented in **pose_landmark_cpu.pbtxt**, shown in A.4. Here we can see a complex network of nodes, whose purpose can be simplified to the following: The very first incoming frame invokes the *PoseDetectionCpu* node in the lower right, which is used to determine a rough ROI for pose landmarks. This ROI is then delivered to the *PoseLandmarkByRoiCpu* node in the upper left, which computes the pose landmarks for this frame. These landmarks in turn are returned by this sub-graph via the*pose_landmarks* output node, and additionally delivered to the *PoseLandmarksToRoi* node, which calculates a ROI for the pose based on the landmarks. This ROI is then used as an ROI for the next frame because of the assumption that the body does not move around too much in between two frames. The *PoseDetectionCpu* node is only invoked again if the landmarking fails, which saves computation time because calculating a ROI from just an image is harder than calculating it from landmarks.

Looking even deeper into the subgraph of the *PoseLandmarkByRoiCpu* node does not reveal new information regarding preprocessing. The inference on the trained model happens here, so only actions related to that are preformed, like scaling the image to the resolution the inference model requires. Going back up the graphs and looking into the face and hand landmark subgraphs instead of the pose landmark subgraph also does not re-

veal new techniques.

Inner View: Summary

In conclusion, from looking at the structure and components of the MediaPipe Holistic data flow represented by the graph we found multiple important steps in MediaPipe's standard data preprocessing:

- The flow limiter, which ensures low latency and real-time computations even on weak machines, at the potential cost accuracy.
- The results from pose estimation are re-used to determine ROIs for face and hands, which avoids wasting computation to determine ROIs inside the face and hand landmark nodes.
- For pose estimation, the landmarks from the previous frame are used to calculate a ROI for the next frame, which avoids expensive ROI estimation on the raw new frame.

In general, computation is reduced by logically reusing previous results from previous inference.

3.3.2. Outer View on MediaPipe

To further examine which properties of a video stream are important to MediaPipe Holistic's landmarking and which are not, I decided to make an experiment where I let MediaPipe Holistic analyze something and then try to draw conclusions from its outputs. One important notice here is that the out-of-the-box MediaPipe demos have two modes of operation, depending on the type of input:

- **Camera Input:**
  If a camera is set as the input to the tracking model, MediaPipe Holistic operates in real-time mode. To meet the time restraints and ensure low latency, the flow limiter node we saw in subsection 3.3.1 is activated at the beginning of the graph, which drops incoming frames if the subsequent graph is still busy with calculations. The dropped frames are not analyzed, but only sent to the rendering node so they still appear in the output video, if a video was specified as an output.
- **Video Input:**
  If a video file is set as the input, MediaPipe Holistic operates in deterministic mode. This means the flow limiter is deactivated and inference is run on every single frame. On weaker computers this means that the analysis can take significantly longer than the duration of the video, especially for complex inference like the holistic tracking with many keypoints.

To emulate the real use-case of a sign language detection software, but still be able to make consistent comparisons, I decided for a video of a person speaking sign language frontal to the camera, instead me of trying to perform the exact same actions in multiple live camera sessions. I then want to modify the video slightly to take away properties which might be important for MediaPipe landmarking. Then I analyze the original video and the modified videos with MediaPipe Hands and MediaPipe Holistic, and determine the quality of the inferred landmarks visually as well as the total inference time with the **/usr/bin/time** command on the terminal. By comparing the inference time and quality on the original video vs. the modified ones I want to roughly estimate properties of the videos which are used in the preprocessing to benefit the inference process, and therefore make assumptions which preprocessing techniques MediaPipe Holistic uses.

Experimental Setup

To find a suitable video I looked at the public DGS-Korpus[5], a long-term project from the Academy of Sciences in Hamburg to aid research of DGS ("Deutsche Gebrdensprache", engl: german sign language). Their videos come very close to the real use-case I am looking for, apart from the fact that they purposely used monotonous backgrounds. With kind approval of the DGS-Korpus administrators I chose the video [8] as the material for this experiment.

My preliminary assumptions on important video properties which influence the inference quality were firstly the skin color of the signer, and secondly a sufficient resolution to detect the subtle movements of the face, as well as edges to separate the face from the hands

To examine the first point, I removed the color from the test video. Using ffmpeg and the command "**ffmpeg -i input -vf hue=s=0 output**", I completely desaturated the video the same way grayscale conversion would, but still kept the video in RGB mode. I also converted the video to grayscale mode with the command "**ffmpeg -i input -vf format=gray output**". Additionally I wanted to check if there is a noticeable difference if MediaPipe analyses a video with a light-skinned person signing versus a dark-skinned person signing. But sadly I could not find a video of a dark-skinned person signing on the DGS-Korpus. This is probably because their objective is linguistic research on DGS, which is why they cared more about diverse places of origin in Germany for the signers than their diverse outer appearances. As no such video with similar circumstances as the first one was available to me, I dropped the idea of comparing the inference quality based on different skin types.

For the second point I reduced the resolution of the original video from 640x360 to

[5]https://www.sign-lang.uni-hamburg.de/dgs-korpus/index.php/dgs-korpus.html

320x180 with the command "**ffmpeg -i input -vf scale=320:180 output**". In the end, I had 4 versions of the video: original, desaturated, grayscale and reduced.

Experimental Execution

I compiled and ran MediaPipe Holistic and MediaPipe Hands on Ubuntu 20.04 directly in the terminal. I chose CPU mode over GPU mode, because I only had an integrated GPU available. In a first step, I analyzed every version of the video with MediaPipe Holistic and MediaPipe Hands to have an output video I can examine. I a second step, I ran MediaPipe Holistic and MediaPipe Hands again with the **/usr/bin/time** command to measure the time, but without specifying an output file to reduce the influence of I/O on the results. Every combination was executed 4 times in total, every time without any other active processes scheduled on the computer, as MediaPipe utilizes all available CPU cores of the system. The averaged results can be found in table 3.1, the full results in table 3.2.

Table 3.1.: Averaged Results

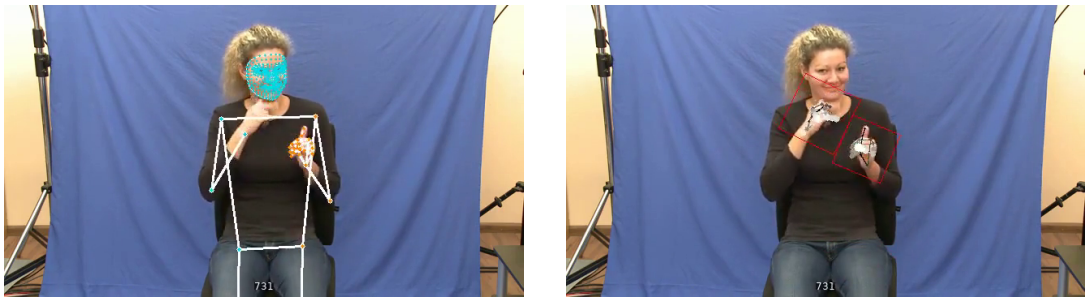
Average Inference Times from **/usr/bin/time** Command in Seconds, Rounded to two Decimals				
Experiment		Times		
		user	system	real
Holistic	Normal	6299,76	335,28	2948,42
	Desaturated	6187,58	305,61	2971,74
	Gray	6324,36	347,30	3025,66
	Reduced	4510,37	51,52	2716,57
Hands	Normal	2732,32	285,07	1153,12
	Desaturated	2832,12	281,60	1366,81
	Gray	2974,78	311,71	1412,77
	Reduced	1282,09	22,05	1023,92

It is important to note that we operate in MediaPipe's deterministic mode now: The original video is 1161 seconds long. If the deterministic analysis takes longer than that, logically frames must be dropped if we were to run the exact same scenario of the video in real-time mode. The longer the analysis takes, the more frames would have to be dropped. Meaning higher analysis time in deterministic mode equals worse accuracy in real-time mode when comparing two videos with each other.

Evaluation of the Experiments

Combining the times with a visual estimation of the quality we can make the following observations:

Comparing the inference times of Holistic Normal to Hands Normal we can see a reduction of user time by 57%, of system time by 15%, and of real time of 61%, meaning Holistic takes much longer both computation and real time wise. This makes sense, because Holistic has to track 543 keypoints total, while Hands only has to track 41 total. But a 92% reduction in keypoints with only a 61% reduction in real time indicates that not all keypoints are equally hard to track.
Visually comparing Holistic Normal and Hands Normal we can see Holistic having problems tracking the hands correctly if they are in front of the face, if the hands are in front of each other, or if the plane of the palm is orthogonal to the camera. Hands also has the same problems, but is more accurate in detecting hands in front of the face, as can be seen in figure 3.2. This is not to say that Holistic's tracking is bad; I would estimate it at 90% accuracy for this video. But it has the mentioned weaknesses.

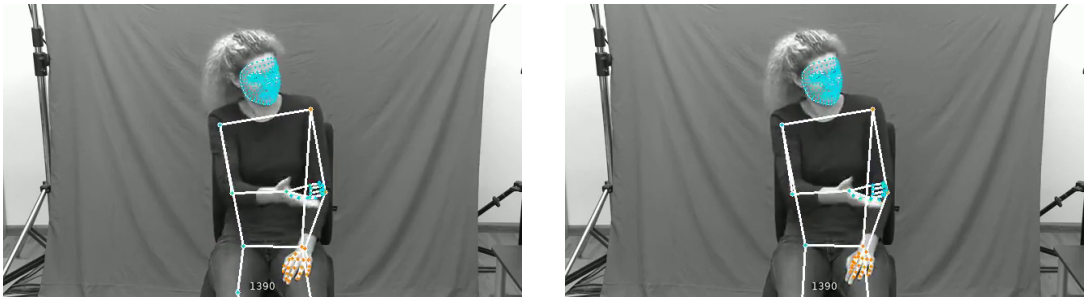


(a) Holistic Normal (b) Hands Normal

Figure 3.2.: Two Different Inferences of the Normal Video

Comparing the inference times of Holistic Desaturated and Holistic Gray we can see an increase of user time by 2%, of system time by 14%, and of real time by 2%, meaning inference on the grayscale version takes a little longer than on the desaturated version. Comparing the inference times of Hands Desaturated and Hands Gray, we see that they have similar increases, but slightly larger percent-wise. Looking at the absolute difference of time, we can see in both Holistic and Hands an absolute increase from Desaturated to Gray of ca. 130s user time, ca. 30s system time, and ca. 50s real time. This indicates that this time differences are a constant offset due to the automatic conversion from grayscale to RGB mode MediaPipe Holistic makes because its inference model's architecture requires it.
Visually comparing Holistic Desaturated and Holistic Gray to each other, one would ex-

pect their inference to be the same, as Holistic and Hands only operate on RGB videos and therefore convert every grayscale frame internally to RGB. This should be identical to the desaturated version of each frame. But looking at the inferences, we can see very minor differences in the two versions. In figure 3.3, we see a typical characteristic of these differences: The dominant hand is always detected exactly the same in both versions, in this frame they even have the exact same glitch. But on very few occasions, the non-dominant, resting hand is detected slightly more accurate in the desaturated version than in the gray version This behaviour is consistent over the whole video.
Surprised by this behaviour, I decided to revisit my assumption that Desaturated and Gray were identical if they were converted into the other color mode. In an image editing program I took the same frame from both versions and subtracted them to spot differences. And indeed: ca. 10% of the pixels were not identical on the two versions. This happened exclusively at the edge between the person and the background, and explains the difference in inference. The Gray experiments therefore need to be redone for reliable visual comparison.

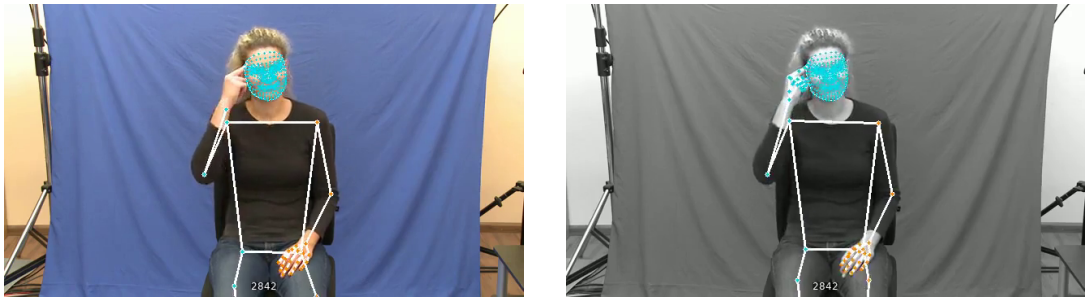


(a) Holistic Desaturated (RGB) (b) Holistic Gray (grayscale)

Figure 3.3.: The Dominant Hand is Tracked Equally, but the Non-Dominant Hand Differently

Comparing the inference times of Holistic Normal and Holistic Desaturated we can see an reduction of user and system time by 2% and 9% respectively, but an increase of real time by 1%.
Visually comparing Holistic Normal and Holistic Desaturated the first thing to notice is that the hand keypoints in the desaturated video are constantly slightly trembling around their place, in contrast to the hand keypoints in the normal video, which remain consistently smooth and steady. This indicates that the hand sub-model uses the skin colour for the landmarking. The model either utilizes it in its own preprocessing, or the machine-learning node itself learned to recognize skin colour during training. As stated, I sadly could not examine the impact of different skin colours on the inference, nor do I know the training data set of MediaPipe Holistic.
The second thing to notice is that the inference on the desaturated video showed an un-

expected improvement over the inference on the normal one: On a few occasions where Holistic failed to correctly landmark a hand in front of the person's face it would succeed in the desaturated version, as can be seen in figure 3.4. As it is highly unlikely that the model was trained with desaturated pictures, the best explanation to this behaviour is that hand detection indeed relies on skin colour for its preprocessing, and therefore is not able to identify a hand correctly if it visually overlaps with the face. The preprocessing for the hand detection sub-pipeline seemingly cannot separate the hand from the face here. On the other hand, the face tracking is still successful, probably because the face tracking sub-pipeline uses different preprocessing methods. The exact difference between these two sub-pipelines needs to be further investigated.



(a) Holistic Normal (b) Holistic Desaturated

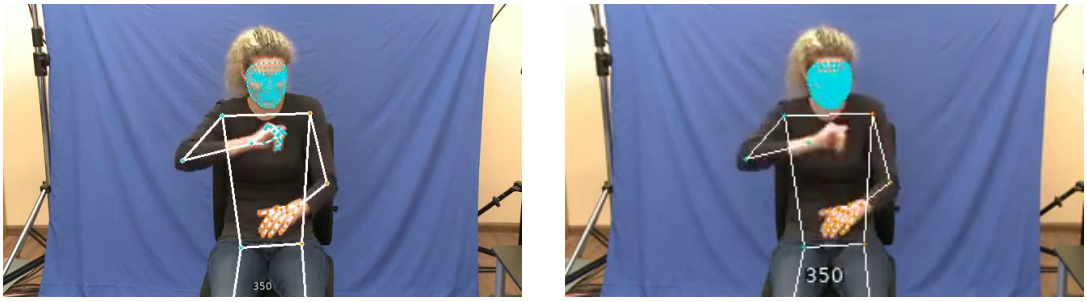
Figure 3.4.: Holistic's Hand Tracking is Successful in the Desaturated Version, but Not in the Normal One

Comparing the inference times of Holistic Normal to Holistic Reduced we see a reduction of user time by 29%, of system time by 85%, and of real time by 8%. This indicates the inference having problems, being either:

- The process is I/O bound. As output is disabled, it could only be that the input is for some reason slower for the reduced version than for the normal one. Although I don't have an explanation to why this could be the case, therefore this possibility seems rather unlikely.
- The process is CPU bound and experiences more context swaps. As no other active processes were running on the machine at the same time, this could be explained with MediaPipe Holistic itself creating additional threads compared to the inference on the normal version.

Either way, these disproportional decreases in time indicate that we made the inference for MediaPipe Holistic more difficult overall. One obvious reason could be that edge detection was not as possible anymore, thus negatively impacting many other preprocessing steps, like the bounding boxes.

Visually comparing the Holistic Normal to Holistic Reduced, the tracking on the reduced video performs clearly worse than on the normal version. Also, the detection rate for overlapping hands or hands over faces is worse in the reduced version, further indicating at MediaPipe Holistic's issues in distinguishing them due to preprocessing methods utilizing skin color. Additionally, inference on the reduced video seems to struggle with fast hand movements: On singular frames the hand tracking is sometimes not executed, as can be seen in figure 3.5, even though the previous and subsequent frames are landmarked correctly.



(a) Holistic Normal (b) Holistic Reduced

Figure 3.5.: Holistic Experiences Dropouts if the Signer Moves Fast in the Reduced Version

```
input_stream: "input_video"
output_stream: "output_video"

# Node A
node{
        calculator: "A"
        input_stream: "IMAGE:input_video"
        output_stream: "OUTPUT_1:output_A1"
        output_stream: "OUTPUT_2:output_A2"
}

# Node B
node{
        calculator: "B"
        input_stream: "INPUT_1:output_A1"
        input_stream: "INPUT_2:output_A2"
        output_stream: "OUTPUT:output_B"
}

# Node C
node{
        calculator: "C"
        input_stream: "INPUT_1:input_video"
        input_stream: "INPUT_2:output_A2"
        input_stream: "INPUT_3:output_B"
        output_stream: "OUTPUT:output_video"
}
```

Source Code 3.1.: Proto Code for the Graph A.1

Table 3.2.: Complete Results

Complete Inference Times from **/usr/bin/time** Command in Seconds, Rounded to two Decimals				
Experiment		Times		
		user	system	real
Holistic	Normal	6338,91	330,71	2948,91
		6345,20	331.07	2950,23
		6252,92	339,50	2947,38
		6261,99	339,85	2947,17
	Desaturated	6218,26	329,99	2974,53
		6215,59	329,70	2971,90
		6147,85	334,65	2969,77
		6168,62	228,10	2970,78
	Gray	6370,91	343,50	3027,81
		6358,81	341,93	3026,87
		6280,80	350,82	3023,52
		6286,92	352,95	3024,45
	Reduced	4489,56	52,15	2720,78
		4510,22	51,13	2714,45
		4517,00	51,19	2714,23
		4524,70	51,50	2716,83
Hands	Normal	2750,48	282,08	1154,58
		2744,62	280,41	1154,16
		2731,60	288,81	1152,27
		2702,59	288,98	1151,47
	Desaturated	2853,89	279,22	1373,17
		2844,69	276,74	1370,32
		2816,31	287,14	1362,82
		2813,60	283,29	1360,93
	Gray	2997,01	303,73	1418,30
		2985,28	313,28	1416,26
		2962,93	315,92	1408,79
		2953,91	313,91	1407,74
	Reduced	1289,45	21,78	1026,81
		1289,56	22,68	1027,58
		1273,20	22,18	1019,94
		1276,16	21,55	1021,35

4. Conclusion

4.1. Summary

In this thesis I discussed various topics regarding preprocessing for sign language detection with machine learning models.
In chapter 2, I talked about background theory useful for understanding of sign language detection. In section 2.1 I talked about the necessities for sign language detection preprocessing resulting from the structure of sign language itself, in 2.2 I gave a general explanation of why we need preprocessing, in 2.3 I gave an overview over common computer vision algorithms used for preprocessing in 2.4 I explained where to find high-quality data sets for research on sign language detection, and in 2.5 I outlined the origins of sign language detection.
In chapter 3, the main part of my thesis, I gave an introduction to MediaPipe in 3.1. I examined MediaPipe Holistic as an example of human pose estimation as preprocessing for sign language detection in 3.2, and also investigated MediaPipe's own preprocessing and estimated its suitability for sign language detection in 3.3.

As we have seen, real-time mobile sign language detection is a complex but promising topic, which we are only just beginning to fully explore. The challenges of sign language detection with machine learning partially come from the complex nature of sign language itself: Other than spoken languages, the conveyed information are split to three separate channels, namely the hands, the face, and the upper body, such that only the combination of all their information carries all information. In this construct, the hands tend to convey the words themselves, while the face and upper body tend to carry grammatical and temporal information. These special circumstances require special handling of all three channels, as every single channel differs in detail, scale and scope of movement from the others.

Only with efficient and effective preprocessing and computer vision algorithms, the main model can be relieved enough to ensure real-time viability, appropriate accuracy and simple hardware setup of the sign language detection. A promising approach for such a preprocessing method is pose estimation, which tracks certain points of the human body as 3D landmarks, allowing full stick-figure reconstruction of the movement. Real-time pose estimation without complex multi-camera setups or physical body markers is a very new and recent field in itself and is still in early development. But recently, MediaPipe with its built-in tracking solutions has shown unprecedented low latency and high track-

ing accuracy in real scenarios.

Therefore, I decided to take a closer look at MediaPipe, especially MediaPipe Holistic, which combines pose, hand and face tracking with adequate levels of detail. MediaPipe Holistic is interesting for sign language detection preprocessing in two ways: Firstly as an outlook on how pose estimation as preprocessing could benefit sign language detection, and secondly as a case study on which preprocessing methods MediaPipe Holistic itself uses in order to reach its good performance.

The reduction in information by pose estimation was obvious, as we condensed an image with hundreds of thousands of pixels into ca. 500 3D landmarks. The inner workings of MediaPipe Holistic were less obvious, and extensive understanding of its structure and code as well as experiments were necessary to uncover only some of its preprocessing mechanisms, like skin colour detection and the flow limiting. In this process I showed that MediaPipe Holistic is not suitable as-is as preprocessing for sign language detection, because it fails to landmark certain poses typical for sign language with sufficient accuracy, like overlapping hands or hands over or next to the face. This is probably a consequence of two factors:

1. The training data was not designed and collected with sing language in mind, therefore the model could not learn to reliably separate hands and face if they visually overlap.

2. The preprocessing was seemingly not designed with overlapping body parts in mind, as the reliance on skin colour detection sometimes leads the preprocessing to view overlapping skin-coloured limbs as one single structure. This structure then has no resemblance to any known limb for the main model, which makes he landmarking fail.

Nevertheless, the concept is promising and there are already start-ups trying to utilize the possibilities of MediaPipe for sign language detection, like SignAll[1].

4.2. Outlook

There are still many topics to explore regarding preprocessing for sign language detection, pose estimation as preprocessing, and MediaPipe as a case study for pose estimation:

- Advanced examination of the MediaPipe graphs and the code could unveil more methods used by its solutions.

[1] https://www.signall.us/

- A full estimation of MediaPipe's possibilities could be made by building solutions solely dedicated to sing language detection, while adjusting preprocessing, hyper-parameters and the training data set.
- Finding a sign language data set with a dark-skinned person signing and comparing the accuracy of MediaPipe Holistic between different skin colours to show if it is usable for all people.
- Running MediaPipe Holistic on different hardware to estimate its viability on low-end to high-end mobile devices, as well as running it on GPU instead of CPU.
- Examining different solutions for human pose estimation, like OpenPose, to uncover new preprocessing methods, even if OpenPose does not use machine learning.

Generally, sign language detection is an interesting, but underappreciated topic that has only gained momentum in the past few years. Due to artificial neural networks and an unprecedented accuracy in new human pose estimation applications, development in this field has the potential to progress significantly in the next years. This is not only exciting from an computer science point of view, but also help for more inclusion and understanding of deaf people in society.

Bibliography

[1] Debra Aarons. *Aspects of the Syntax of American Sign Language*. Dissertation, Boston University, 1994.

[2] Necati Camgoz, Simon Hadfield, Oscar Koller, Hermann Ney, and Richard Bowden. Neural sign language translation. *2018 IEEE conference on Computer Vision and Pattern Recognition*, 03 2018.

[3] Runpeng Cui, Hu Liu, and Changshui Zhang. Recurrent convolutional neural networks for continuous sign language recognition by staged optimization. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1610–1618, 2017.

[4] E. R. Davies. *Computer vision: Principles, aldagorithms, applications, learning*. Academic Press, London and San Diego, CA and Cambridge, MA and Kidlington, fifth edition edition, 2018.

[5] Georgios Evangelidis, Gurkirt Singh, and Radu Horaud. Continuous gesture recognition from articulated poses. *2014 European Conference on Computer Vision*, 09 2014.

[6] Linwei Fan, Fan Zhang, Hui Fan, and Caiming Zhang. Brief review of image denoising techniques. *Visual Computing for Industry, Biomedicine, and Art*, 2(1):7, 2019.

[7] David Forsyth and Jean Ponce. *Computer Vision International Edition PDF EBook : A Modern Approach*. Pearson Education, Limited, Harlow, UNITED KINGDOM, 2015.

[8] Thomas Hanke, Susanne König, Reiner Konrad, Gabriele Langer, Patricia Barbeito Rey-Geißler, Dolly Blanck, Stefan Goldschmidt, Ilona Hofmann, Sung-Eun Hong, Olga Jeziorski, Thimo Kleyboldt, Lutz König, Silke Matthes, Rie Nishio, Christian Rathmann, Uta Salden, Sven Wagner, and Satu Worseck. Diskussion – münchen (bayern-süd) (meine dgs. öffentliches korpus der deutschen gebärdensprache, 3. release), 2020.

[9] Seo Yul Kim, Hong Gul Han, Jin Woo Kim, Sanghoon Lee, and Tae Wook Kim. A hand gesture recognition sensor using reflected impulses. *IEEE Sensors Journal*, 17(10):2975–2976, 2017.

[10] Oscar Koller, Sepehr Zargaran, and Hermann Ney. Re-sign: Re-aligned end-to-end sequence modelling with deep recurrent cnn-hmms. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3416–3424, 2017.

[11] Maria Kopf, Marc Schulder, and Thomas Hanke. Overview of datasets for the sign languages of europe. doi.org/10.25592/UHHFDM.9561, 2021.

[12] Maren Brumm and Rolf-Rainer Grigat. Optimised preprocessing for automatic mouth gesture classification. *Proceedings of the LREC2020 9th Workshop on the Representation and Processing of Sign Languages: Sign Language Resources in the Service of the Language Community, Technological Challenges and Application Perspectives*, pages 27–32, 2020.

[13] Pavlo Molchanov, Xiaodong Yang, Shalini Gupta, Kihwan Kim, Stephen Tyree, and Jan Kautz. Online detection and classification of dynamic hand gestures with recurrent 3d convolutional neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 4207–4215, 06 2016.

[14] Oscar Koller, Jens Forster, and Hermann Ney. Continuous sign language recognition: Towards large vocabulary statistical recognition systems handling multiple signers. *Computer Vision and Image Understanding*, 141:108–125, 2015.

[15] Munir Oudah, Ali Al-Naji, and Javaan Chahl. Hand gesture recognition based on computer vision: A review of techniques. *Journal of Imaging*, 6(8):73, 2020.

[16] Lionel Pigou, Aron Oord, Sander Dieleman, Mieke Van Herreweghe, and J. Dambre. Beyond temporal pooling: Recurrence and temporal convolutions for gesture recognition in video. *International Journal of Computer Vision 126(2-4)*, 126, 04 2018.

[17] Ulrich von Agris, Moritz Knorr, and Karl-Friedrich Kraiss. The significance of facial features for automatic sign language recognition. *2008 8th IEEE International Conference on Automatic Face & Gesture Recognition*, pages 1–6, 2008.

[18] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc Le V, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation.

Appendix

A. MediaPipe Graphs

The following graph visualizations are taken from the MediaPipe Visualizer which is created and shared by Google at **https://viz.mediapipe.dev/**

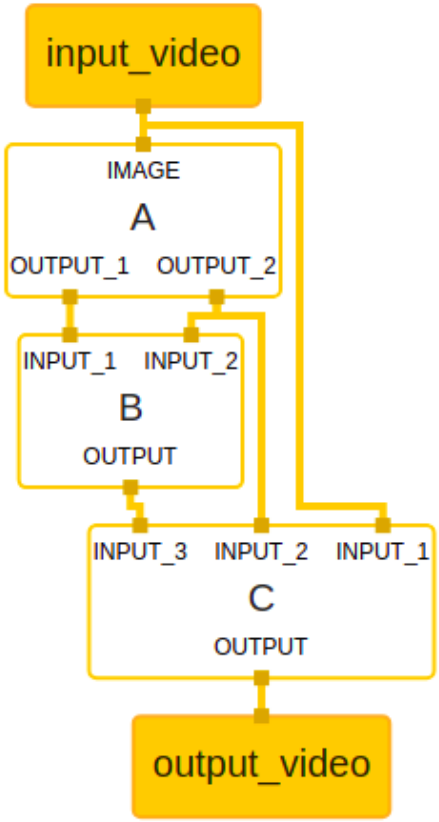


Figure A.1.: Example Graph created by Source Code 3.1

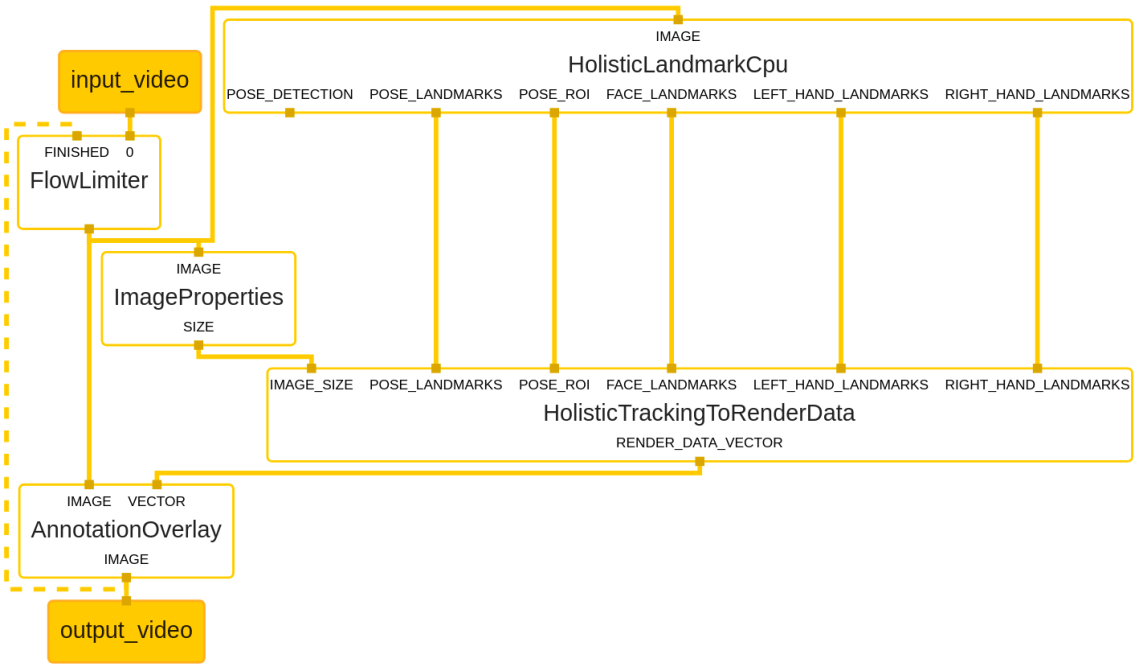


Figure A.2.: The Graph of MediaPipe Holistic: holistic_tracking_cpu.pbtxt

Discussed in section 3.3.1

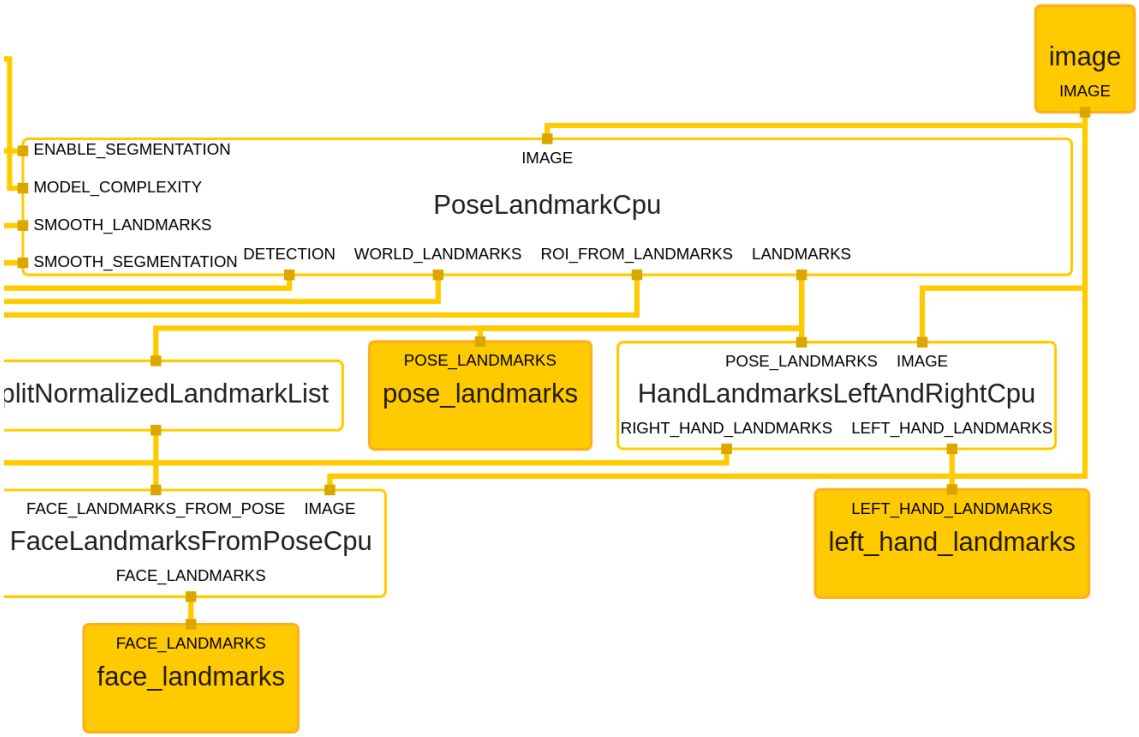


Figure A.3.: The HolisticLandmarkCpu subgraph from A.2: holistic_landmark_cpu.pbtxt

Discussed in section 3.3.1, Unimportant parts cut out

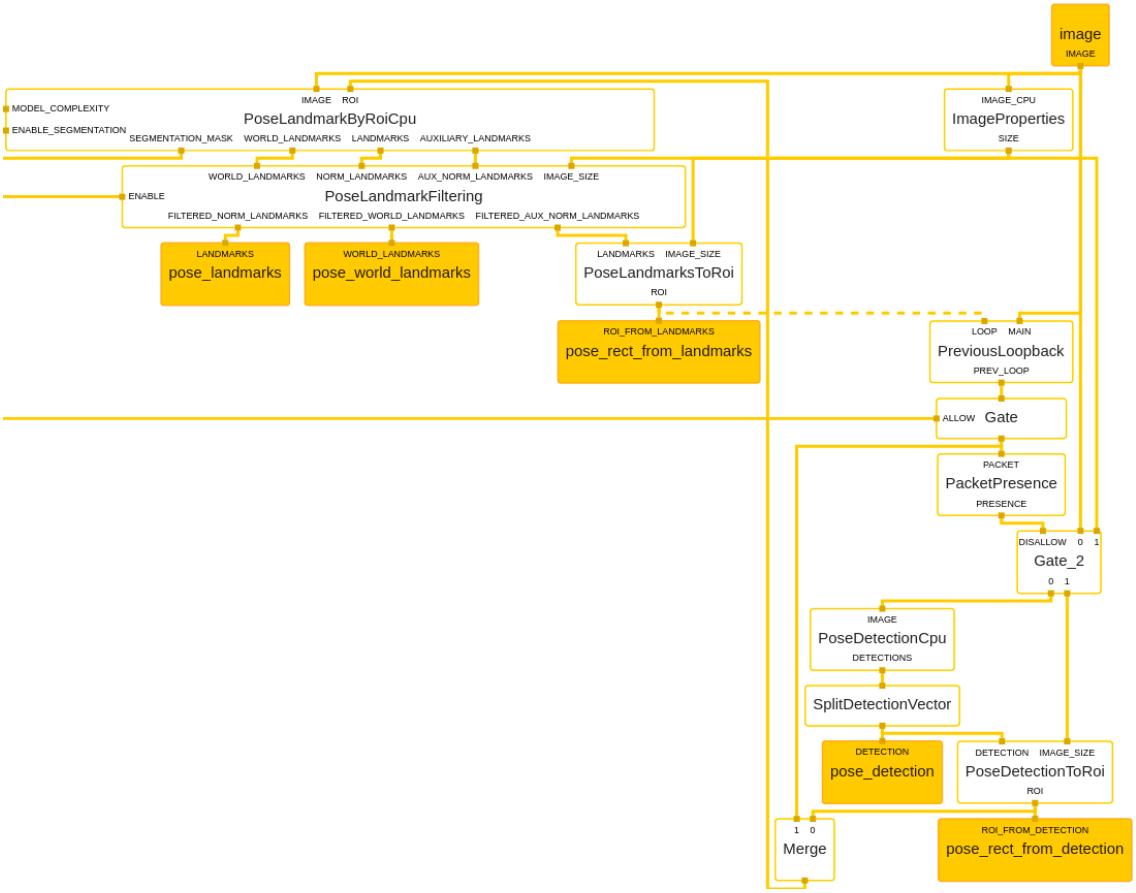


Figure A.4.: The PoseLandmarkCpu subgraph from A.3: pose_landmark_cpu.pbtxt

Discussed in section 3.3.1, Unimportant parts cut out