DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Sign Language Translation on Mobile Devices

Maximilian Karpfinger

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Sign Language Translation on Mobile Devices

Übersetzung von Gebärdensprachen auf mobilen Endgeräten

Author:	Maximilian Karpfinger
Supervisor:	Prof. Dr. Christian Mendl
Advisor:	Dr. Felix Dietrich
Submission Date:	15.04.2022

Acknowledgments

I would like to thank Prof. Mendl and Dr. Dietrich for giving me the opportunity to write this thesis. As my advisor Dr. Dietrich always gave great insights in our meetings and was always fast to answer my questions to problems that occurred during this project. I also want to thank my girlfriend for taking the pictures of me doing sign language to test the app and supporting me emotionally during the whole study.

Abstract

It is very important for the people of the deaf community to communicate with each other and also with people that are not able to speak sign languages, but the language barrier makes this difficult. However, automated translation of sign language to spoken language is a difficult task. For the time being, there is not one application which does sign language translation in this way. There are some approaches which can detect and translate alphabetical characters but not full sentences. While there are many tools that support people who need seeing or reading assistance like the Google Assistant, the number of tools available for the deaf community is still very small. Though, with new technologies and smartphones more opportunities open up. This thesis is about translating sign language on mobile devices. In particular an app for Android smartphones was created that can translate sign language to text in real-time. The framework behind this technology is MediaPipe by Google. By using holistic tracking, hand, facial and pose landmarks are extracted from a camera stream and put into a model that maps these landmarks to translated German text which will then be displayed on the smartphone's display. Three different models were trained with the data provided by the DGS-Korpus which is a long-term project by the Academy of Sciences and Humanities in Hamburg on sign languages. As a result, an Android app that uses a neural network trained on the 100 most appearing signs in the available videos of the DGS-Korpus to translate pose landmarks to words was successfully implemented. The network has approximately 30% accuracy, so a lot of improvements are still possible.

Contents

1. Introduction

Mobile phones assist humans in everyday life. More generally speaking, many assisting tools on mobile phones were introduced in the last decade. While most of these technologies focus on making the usability of the device easier for humans, the amount of assisting tools that deal with sign language is low.

Nevertheless, automated sign language translation and the topic of sign languages have been a hot topic in research areas in recent years. This is mostly due to the fact that automated translation of sign language is a more complex task than the translation from one spoken language to another spoken language and also in comparison to text to voice problems or vice versa. Research on a machine translation of sign language is where it has been years ago for other problems like generating text from voice or reading a given text. For the latter many famous tools like the Google Assistant or Siri were created which both assist humans in everyday life.

There are many situations in which an app that is able to translate sign language could be helpful. While deaf people can communicate with other deaf people, it can be challenging for a deaf and a non-deaf person to talk to each other if the non-deaf person cannot speak sign language. There is a lot of use for an app that can translate sign language. For example, when a deaf person needs to see a doctor, or wants to have their hair cut at the barber. More generally speaking, any person offering a service could use the app to communicate with deaf people. Expressing their needs is crucial for every human being.

With recent advancements in machine learning, new ideas for solving some of these problems have come up. For example with the help of neural networks, image analysis has made huge improvements. By applying these techniques it is possible to make human pose estimation and thereby finding key points, for instance of the hands and face. In this master's thesis the creation of an application for mobile devices running on Android is discussed. The goal is to create a translation pipeline and for that matter use existing pose estimation modules from the MediaPipe framework. In the main part of this thesis the implementation of such a pipeline is elaborated. However, in addition to the pipeline a model that is able to translate key points from the pose estimation models to words is also required. In order to train neural networks data is needed. Thus, the main part of this thesis also discusses the data acquisition, where videos from an online source which show people speaking sign language are analysed. These videos are annotated with the meaning of each signed gesture. By using these annotations, a data set is established which combines key points from pose estimation with a given label representing a word. Through that models can be trained that classify gestures.

For the translation model, three different types of architectures are analyzed and tested on the data set created from the previously mentioned data. As a result, an application was built that can translate gestures from German sign language to text as can be seen in Figure 1.1. The Figure shows a screenshot from the app translating German sign language in real-time.

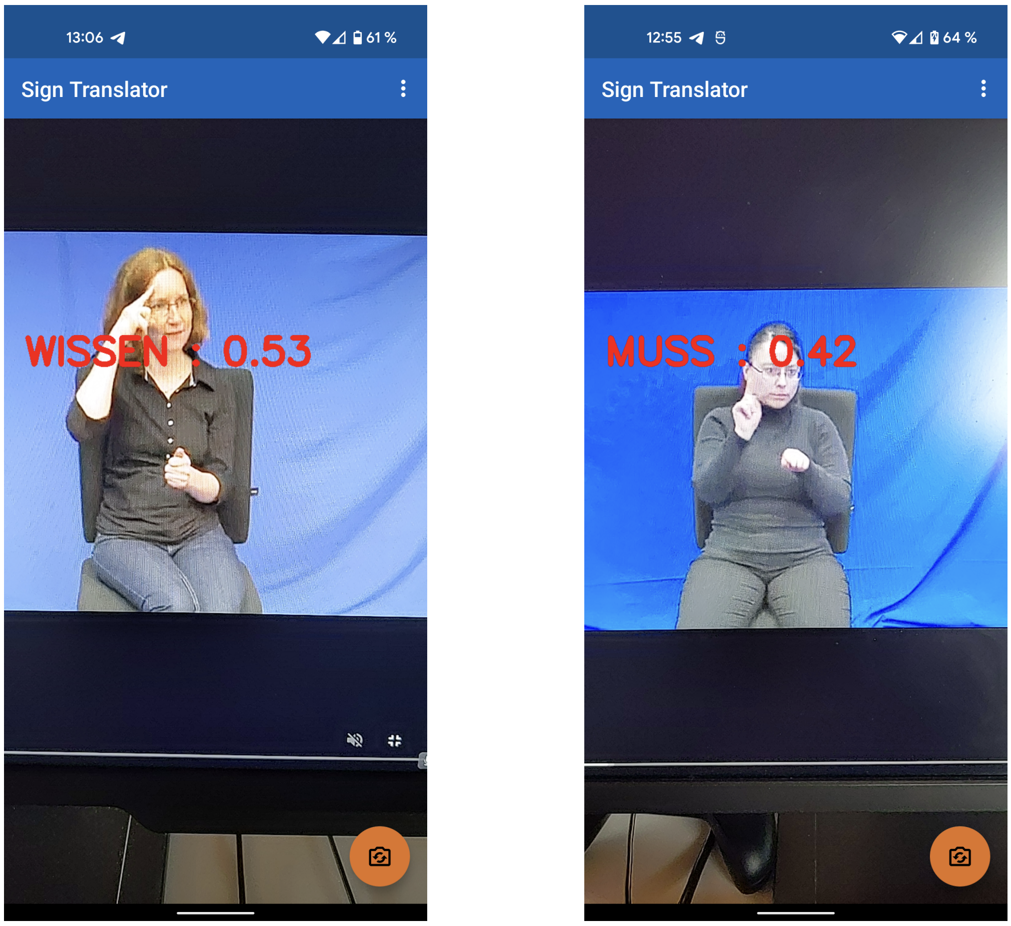


Figure 1.1.: Screenshots of gestures being translated by the app. The app displays the translated sign and also the score of the classification network for the prediction.

Finally, this thesis is structured in the following way. There are three sections, whereas the first section *State of the Art* is about putting the thesis into its context by citing relevant literature. Here, sign language is discussed at first. There will be explanations on how sign language works according to literature. This is followed by a section on tools on mobile phones that assist humans. After that a section on neural networks introduces the network architectures, which have been mentioned previously and are deployed in this thesis. Subsequently, some previous work that focuses on sign language recognition is referred to. Continuing with the main part of this thesis *Sign Language Translation on Mobile Devices*, which is about the implementation of an app that is able to do sign language translation. Starting with an overview of the data source and how they have collected their data. Then the framework used to create the pipeline for the app is introduced. Here, the focus lies on the framework's key concepts and how it can be used to create apps for mobile devices. Also, an example module from the framework is introduced. Afterwards a section on ideas and requirements for a sign language translation model follows. Here, the architectures, which are used in the training phase, are presented. Then follows a section on how the data sets, used to train the translation models, were created and what evaluation metrics are used to evaluate their performance. The training of the translation models is showcased. After that, the pipeline's different parts are explained and how they are implemented. In the next section, screenshots from the apps are presented that show the app being used on videos showing people speaking sign language. In the last section, the translation models are evaluated on a test set. The app will also be used in different scenarios to investigate how different scenarios might impact the app's possibility to translate certain gestures. Additionally, the pipeline will be examined on its performance in terms of computational costs. Finally, in the third and last section *Conclusions* the work of this thesis is summarized, discussed and an outlook is given on thoughts and ideas about possible improvements or future additions to the pipeline and app.

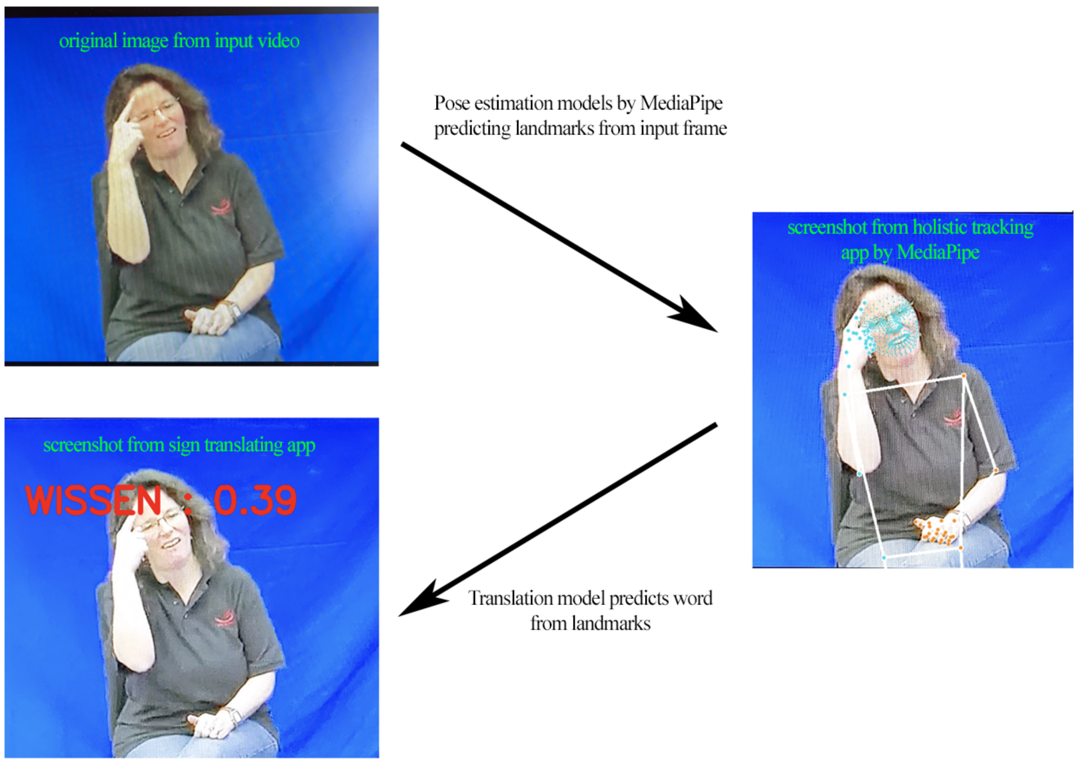

original image from input video
Pose estimation models by MediaPipe predicting landmarks from input frame
screenshot from holistic tracking app by MediaPipe
Translation model predicts word from landmarks
screenshot from sign translating app
WISSEN : 0.39

Figure 1.2.: Dataflow from input image to landmarks to translated sign.

2. State of the Art

This section gives an introduction to sign language, especially on German sign language. The differences of sign language to spoken language are elaborated and the key concepts of German sign language are explained superficially. Then some tools for mobile devices which assist humans in everyday life are mentioned and also some apps which touch the topic of sign language are brought up. This is followed by a section about neural networks architectures which are used in this thesis. In the final section of this chapter related works on sign language detection and recognition are noted.

2.1. Sign Language

Sign languages are a form of communication which is based on visual aspects in contrast to spoken language which depends on vocal aspects. Because different sign languages exist, just like multiple other spoken languages exist, the focus in this thesis will be on German sign language. Similar to spoken German, dialects exist in the German sign language, whereas the idioms may have different words. By using hands, mouth and pose a signer can create or use existing signs to communicate. Sign language itself is a complex language with rules that are unlike the rules used in spoken languages [1]. The language is used mostly by deaf people. As [2] states, there are approximately 80.000 deaf people in Germany and around 16 million hearing-impaired people. The website also states that most of the deaf people turned deaf during their life while only 15% inherited their deafness.

Coming back to German sign language, it is not a word by word translation from spoken German, but rather uses a different grammar and vocabulary [3]. Transmitting information in sign language can be done with two key components which are manual and non-manual means of expression. Manual expressions are done by moving the hands or arms, whereas manual means of expression can be divided into four categories. These are the shape of the hand, the orientation, the position in terms of space and the movement itself [4]. [3] elaborates further on these and found that there are around 30 different shapes of the hand which express a different meaning. In addition, the orientation is determined by either the palm of the hand or by the beginning of the fingers. Both could point in any direction of the 3D space around the signer. Next on, the position of the hand is also important. Again, the position of the hand can be placed in any place around the signing person, however, most of the signs are placed in front of the chest [3]. This 3D space is called the signing space. The final property of manual expressions is the movement of the hand which also has several attributes. Moving the hand in a certain path, moving the fingers and how often the movement is repeated all determine what a signer is expressing. The speed of the movement is also important. [3] makes an example where a person signs the gesture WORK, but does so in a slow movement. This could mean that the person expresses WORK-dull. The sign can be supported by non-manual means of expression.

These expressions also can be divided into four categories. Beginning with the facial expression, [5] differentiates between non linguistic and linguistic sign language facial expressions. Here, non linguistic expressions are the basic emotions and the other ones are expressions with a function in sign language. These can be adverbs, marking certain words, changing roles and more [3]. The most important areas for facial expressions are around the eyes and the mouth [3]. Proceeding with the next aspect of non-manual means of expressions which is the eye's expressions and the viewing direction. Similar to spoken language, the eyes can be used to see how a person is feeling, e.g. if they are paying attention or are distracted. Further utilization is looking into the signing space, looking at the partner of conversation, view of a role, and looking at the signing hand [3]. Signers can change roles by imitating the person's pose and facial expression, which is helpful for story telling. Looking to a place in the signing space can be used to reference some object which was indexed to be placed there beforehand and looking at the hand can be used to draw the conversational partner's attention to it [3]. Following with the next component which is the posture of body and head. By moving the body and head in a certain way, a signer can imitate someone else as mentioned before. It is also used for direct and indirect speech. However, not all of the body is important because the legs are not used in German sign language [3]. Finally, the last aspect of non-manual expression is the mouthings. These are mostly used for words which do not have a sign in sign language. One can either move the lips to imitate saying the word or say the word non-vocally. Another use of the mouthings is to give further information to a sign. As an example [3] states that both BROTHER and SISTER share the same sign, but can be distinguished by the mouthing. Not all sign languages utilize the mouth as German sign language does. For instance, in American sign language mouthings are replaced by finger signs because in the US, they are socially more accepted for being more subtle [3]. When a signing person uses both hands, there are two rules that must be considered. The signing has to be either symmetric or one hand has to be the dominant one, which is the active one and the other hand has to stay passive [3]. Considering all this information about German sign language, this thesis describes how a pipeline is created for application on mobile devices that tries to translate signs by using a model that gets pose estimation information as input. [1] claims that there are no apps or computer programs that translate sign language as of now.

2.2. Existing Assistance Tools on Mobile Devices Using Machine Learning

While there may not be existing tools for sign language translation, there are other apps that use image processing. Furthermore, tools which support a user by reading certain texts on their phone to them or write text from what the user is saying have been around for years. In the following, some of these will be mentioned. Beginning with the framework MediaPipe [6] by Google, which offers a variety of solutions which all use object detection or pose estimation, for example doing face detection, object detection and tracking or selfie segmentation. These solutions can be downloaded, as pre-built versions for Android devices already exist. Moving on to tools that support a user with reading or by voice, famous examples are the Google Assistant [7], or Siri [8] by Apple. Both of them can support a user by listening to their voice commands, for example doing an online lookup, calling contacts, writing messages or reading texts to them. Although these tools provide a variety of functionality, none of them have features which focus on translating sign language. Some apps exist which handle the topic of sign languages. However, most of them are dictionaries or for instance the EiS app [9] which has its focus on teaching sign language.

2.3. Neural Networks

All the tools mentioned in the section before which use pose estimation utilize machine learning techniques, like the object detection app by MediaPipe. Some sort of model for detecting and translating sign language needs to be implemented in the sign language translation application as well. In machine learning various approaches exist for solving problems of different kinds. The most basic tasks are regression and classification. For classification tasks of images neural networks can be used [10].

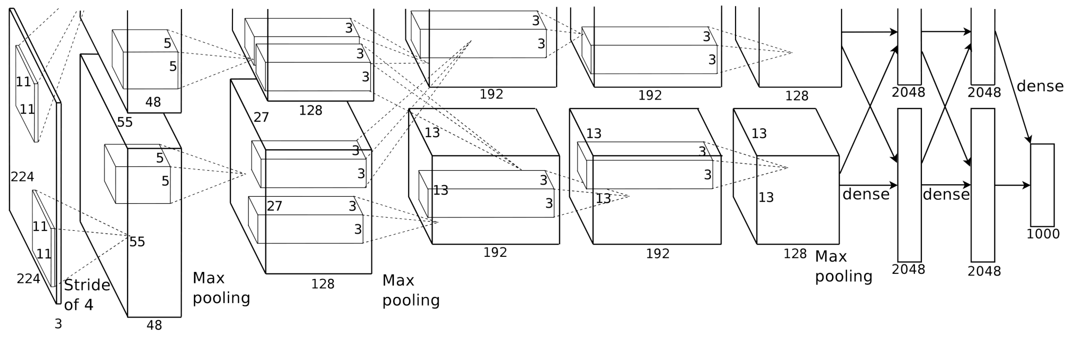


Figure 2.1.: An illustration of the architecture of AlexNet. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264– 4096–4096–1000.
Figure from [10], including the cut-off on the top.

As seen in the architecture of AlexNet [10] which is visualized in Figure 2.1, the network consists of multiple convolutional layers whereas some have a max pooling layer in between. With these layers the network tries to find features in the given images by applying kernels which have been learned by the model.

A more sophisticated approach, which also has a more adequate input format for the problem case of this project, is **PointNet** [11]. PointNet originally was developed for point clouds in three dimensional space. However, as shown in [11] it can also be used for two dimensional point clouds. This is important for this study, because the training data will be two-dimensional.

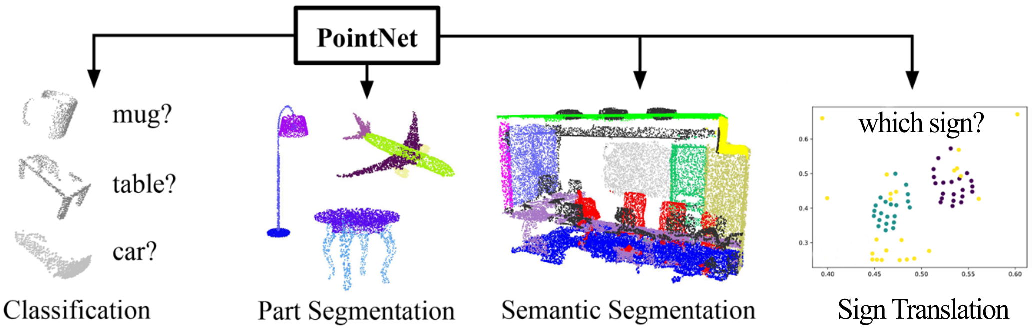


Figure 2.2.: Original applications of PointNet as in [11]. In addition, application on landmarks for gesture translation.
Figure adapted from [11].

As seen in Figure 2.2 the network is capable of doing classifications from a set of data points with n dimensions. In addition, it can do semantic segmentation and part segmentation when the architecture is adapted. In this project, landmarks of hands, face and pose will be used to translate sign language to spoken language. These landmarks will be available in sets. Because the input for PointNet is a set of data points, it fits the use case in this project a lot more than ordinary convolutional neural networks which are usually used on two or three dimensional images. The authors of [11] state that the input data points have three properties. First, the data points are not ordered. This means that regardless of the order of points, or in the case of this project landmarks, the network should be invariant to any permutation. While landmarks will actually be provided in order, the landmarks themselves can be arbitrarily permuted. Second, the points can be in relation with each other. Such a relation can be for example the distance between two points. This also holds for landmarks since they are structured locally and therefore build meaningful subsets (e.g. left hand). Third, the points are invariant to numerous transformations like rotation and translation. This also holds for landmarks. Together, these three properties need to be handled by a model so the network's architecture was designed appropriately.

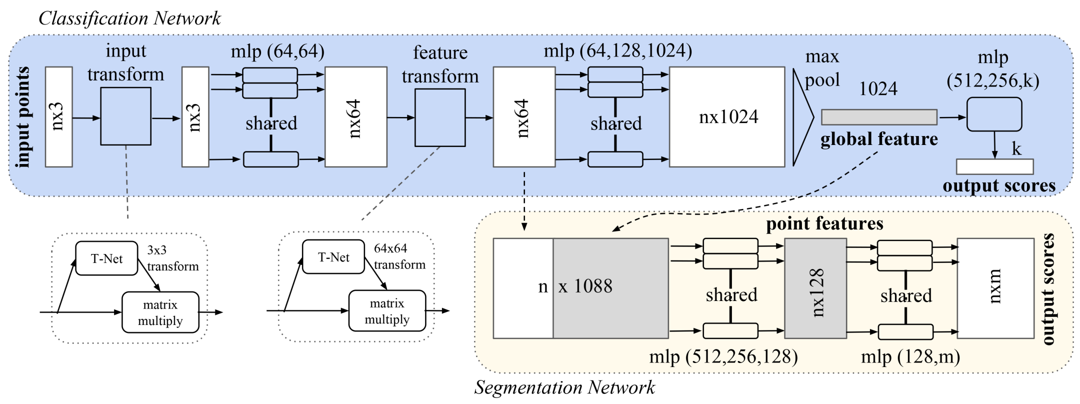


Figure 2.3.: PointNet Architecture. The classification network takes n points as input, applies input and feature transformations, and then aggregates point features by max pooling. The output is classification scores for k classes. The segmentation network is an extension to the classification net. It concatenates global and local features and outputs per point scores. “mlp” stands for multi-layer perceptron, numbers in bracket are layer sizes. Batchnorm is used for all layers with ReLU. Dropout layers are used for the last mlp in classification net.
Figure from [11].

In Figure 2.3 the architecture of PointNet is visualized. However, only the blue part matters for this project as the lower part is the architecture for a segmentation network.

Another useful class of network architectures are graph neural networks [12]. These networks can operate on graphs instead of images or data points like the models discussed previously. GNNs can be used for either graph classification or classification of nodes. Graphs are a different form of data structure in which information can be saved in nodes and relationships between the nodes are stored as edges. In the nodes the coordinates of the landmarks will be stored and the edges are a binary value, marking if two nodes are connected to each other or not. The landmarks from hands and pose can be seen as a graph by connecting them in a reasonable way. Landmarks will be represented as nodes, whereas the edges connect landmarks. For example, nodes which represent the fingers should have edges to each other and all fingers should be connected with the wrist of the hand. Additionally, the hands need to be connected to the pose nodes as well.

2.4. Previous Work on Sign Language Recognition

Automatic sign language translation has been a hot topic in researches regarding computer vision in the recent years. In the thesis [13] the author explores how sign languages can be translated in a two-step process. First, landmarks from pose and hands are detected to recognize a gesture. In the second step the gesture is mapped to a word.

The author focuses on American sign language (ASL) which is different to German sign language and uses videos from the data set [14]. In the first step they create a model that detects alphabet gestures from images and also recognizes each letter of the alphabet. Then they also test their model on images captured from a webcam. For the landmarks from the hands MediaPipe is used which is also used in this thesis. Detecting landmarks for the pose is done by utilizing OpenPose [15].

In another thesis [16] which also focuses on detection of gestures by using hand pose estimation, the author also uses the framework MediaPipe and compares it to a classical approach which utilizes computer vision techniques. The author describes the classical approach in a seven step algorithm, whereas in the first steps the image is collected from the webcam and a color filter is applied where the face is already removed by blacking it out. In the next step, the image is reduced to the hand only, by applying a skin color filter. This is followed by applying the Sobel [17] filter to detect edges of the hand. In the last steps the number of fingers are calculated and thereby detecting the gesture. As a result the author finds that MediaPipe is more accurate than the classical approach.

Another thesis [18] discusses how data preprocessing is important for sign language detection. Because instead of using whole images for translation models, a lot of computing complexity can be saved by using only the important features of the image which are the landmarks of a signing person. The author's research concentrates on how MediaPipe does these preprocessing steps.

3. Sign Language Translation on Mobile Devices

This chapter gives an introduction to the DGS-Korpus which provides videos of people speaking sign language. These videos can be used for machine learning models that focus on sign languages. There is also a section which explains the framework MediaPipe. This framework is used in this thesis to create a pipeline for sign language gesture translation. The framework's concepts will be elaborated and how a graph, that represents a pipeline for the translation of gestures, is created. The app's user interface is also shown and the app is evaluated on its performance. It is important that the pipeline used in the app is not too expensive in terms of computing costs, as mobile devices tend to have hardware which cannot deal with huge complex models.

3.1. DGS-Korpus Data

Training neural networks requires a lot of data. Therefore, for this project data from the DGS-Korpus will be used and in the following a brief overview of the DGS-Korpus and their data acquisition conventions is given. The DGS-Korpus is a long term project by the Academy of Sciences in Hamburg with focus on documenting and researching German sign language. It aims to collect and share data of sign language and also to provide a dictionary of German sign language. Starting the data acquisition was done by finding participants who fulfill the requirements mentioned in [19]. These carefully selected participants needed to speak German sign language fluently, live among different regions in Germany, be of different age, gender and have different occupations. The participants were sat in-front of each other in a studio with blue screens as background. Each informant had a camera facing them. A moderator then started to ask questions to begin the discussion and kept the conversation going when the informants had nothing more to say, by asking new questions or adding comments. The topic of conversation varied among many subjects like sports, politics and everyday life. A full list of all topics can be seen in [20] and the studio setup is visualized in Figure 3.1.

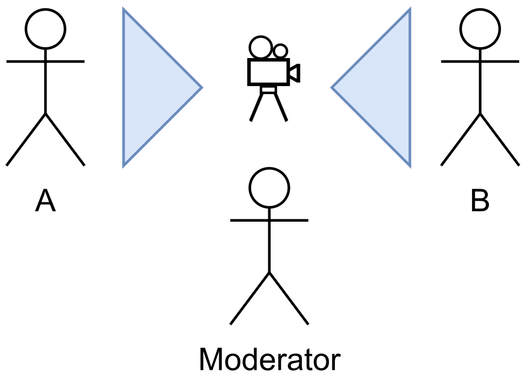


Figure 3.1.: Own visualization of the studio setup. Participant A and B are placed in front of a screen with a camera facing them.

After the recording of the discussions was done, all the video-material needed to be annotated. This was done in several steps. In the first step, the videos were segmented by the occurring signs into tokens. Each token will be annotated by a type later on. As a rule it was chosen that the beginning of one gesture is not necessarily the end of the previous gesture. Therefore, the transition between gestures is not part of the gestures themselves. An advantage of this rule is that tokens of one type do not vary as much as when the transition is included. Another way could be to have the beginning of a token as the end of the previous token. In addition to the gesture, mouthings are also taken into consideration, because they help finding the correct type of a token. However, segmentation does not depend on mouthings, but they need to be annotated too. These and more conventions of the segmentation are further elaborated in [21]. In a next step, all tokens needed to be annotated, which is called the lemmatization step. In this project double glossing was used. For this, a type hierarchy was used with a parent-children relation. Each type is the parent of multiple subtypes and is further described by a citation form, whereas subtypes are usually described by a combination of gesture and mouthing. All types and subtypes can have lexical and phonological variants. These need to be labeled as such to distinguish between the variants. Signs can be of different nature such as lexical, productive, pointing or name. Name signs are either anonymized or in the case of famous people, they are named after said person. Pointing signs are where either the whole hand or some fingers are used to point at a local position or person. Productive signs, which are created in the context by the signer, help provide information. Lexical signs tend to be stable among different contexts and represent actual words of a language. Signs can be either signed by one or two hands. For simplicity it was decided to have only one hand annotated for each token, even if both hands sign a different gesture. Whenever this happens, the dominant hand gets the annotation. Other signs and more about naming conventions can be found in [22].

3.2. MediaPipe

In the following section the framework MediaPipe is elaborated. At first an overview of its concepts will be given which is followed by a subsection about how MediaPipe is applicable on mobile Android devices. Finally, the existing Holistic Tracking module is examined.

3.2.1. MediaPipe Framework's Concepts

MediaPipe is a framework by Google which provides functionality to create or use existing machine learning software with focus on image, video and audio sources for several platforms [6]. With MediaPipe it is possible to create a complete pipeline from taking camera input and using this as an input for a neural network to produce an output that can be viewed on the screen. The framework also takes care of inference on the translation model. MediaPipe also provides ready to use solutions for problems like face detection, object detection and many more [23]. In the following the framework's concepts such as graphs, calculators, packets and streams are elaborated. Beginning with graphs, which represent a complete pipeline from input to output, they consist of several nodes. A graph has its specification file where all nodes and the connections between them are listed. Nodes either represent a single calculator or a subgraph. Subgraphs contain several nodes again and were introduced to simplify re-usability of certain workflows. Continuing with calculators, they can have an arbitrary amount of input and output streams. These streams are used to transmit packages. Inside the calculators all the data processing work is done, so the typical workflow inside one calculator is to retrieve one package from the previous calculator, process it and send it to the next calculator. All calculators provide four methods which are explained in the following. The method *GetContract()* checks if all input and output streams are available and have the correct type. It is basically used in a verification step in the initialization of a graph which checks the specification file of a graph. The method *Open()* is called when the graph starts and takes care of all necessary work which needs to be done before processing packets. For all calculators with input streams the method *Process()* is called whenever at least one input stream has a packet ready. Inside this method all computing should be done and as well as sending packets to the next calculators by output streams. The last method *Close()* is called when an error occurs in either *Open()* or *Process()* or when all of the calculator's input streams close. Another way for a calculator to acquire data is via input side-packets or options which can be provided in the specification file. Options are implemented as C++ protocol buffers [24]. As a last concept of this framework, packets are explained. Packets contain data and are sent between calculators. They also provide a timestamp which is needed for synchronisation [25].

3.2.2. MediaPipe on Android Based Devices

One of the platforms which can be targeted by MediaPipe is Android, so it is possible to bring existing machine learning solutions to android devices and develop on their bases. Android apps can be built using the tool Bazel [26]. For this, some prerequisites need to be met, for example the Android SDK version must be at least 30 and the Android NDK version needs to be between 18 and 21. Also, Bazel is set to version 3.7.2. An Android App needs a directory for its source files called *res*, an AndroidManifest, a BUILD file and a MainActivity. In the resource directory all resource files like drawables or layout files for the activities are stored. For a small project only a layout file for the MainActivity is needed. Furthermore, values for strings, colors and styles are stored in this directory. In the AndroidManifest meta information about the app are stored such as all activities. Here, atleast the MainActivity needs to be announced. In addition, there are some meta data values set from MediaPipe. In the BUILD file everything needed for building the app is declared. Starting with all necessary Android libraries, the AndroidManifest, dependencies and linking information are provided. Finally, all Tflite models and further files which should be used are declared in the assets, as well as the graph as binary which will be used in the app. Here the values which get set in the AndroidManifest are maintained. In the MainActivity libraries are loaded for example the needed OpenCV [27] and MediaPipe libraries. In the method *onCreate()* the layout of the MainActivity is inflated and then the processor with the provided graph is initialized. Beforehand, permissions for camera access is requested. For this project the graph for holistic tracking by MediaPipe is used by the processor in the MainActivity, but it is adapted to this thesis' work.

3.2.3. MediaPipe Holistic Tracking

The MediaPipe Holistic Tracking is an existing solution in the MediaPipe framework [28]. This solution was created to track human pose movements. It consists of several modules of the framework and combines the face landmarks, human pose and hand tracking. Therefore it can be used for many use cases such as sport analysis, but more important for this project sign language translation as it also provides landmarks for hands and pose in addition to the already mentioned face landmarks. Altogether, this pipeline creates 543 landmarks where 468 are from the face, 33 from the pose and 21 for both hands. For that, it uses existing models from the framework. Additionally, a hand re-crop model is uFsed which supports finding the region of interest in case the pose model predicts too inaccurate results. As output, the solutions display the found landmarks and connect them accordingly.

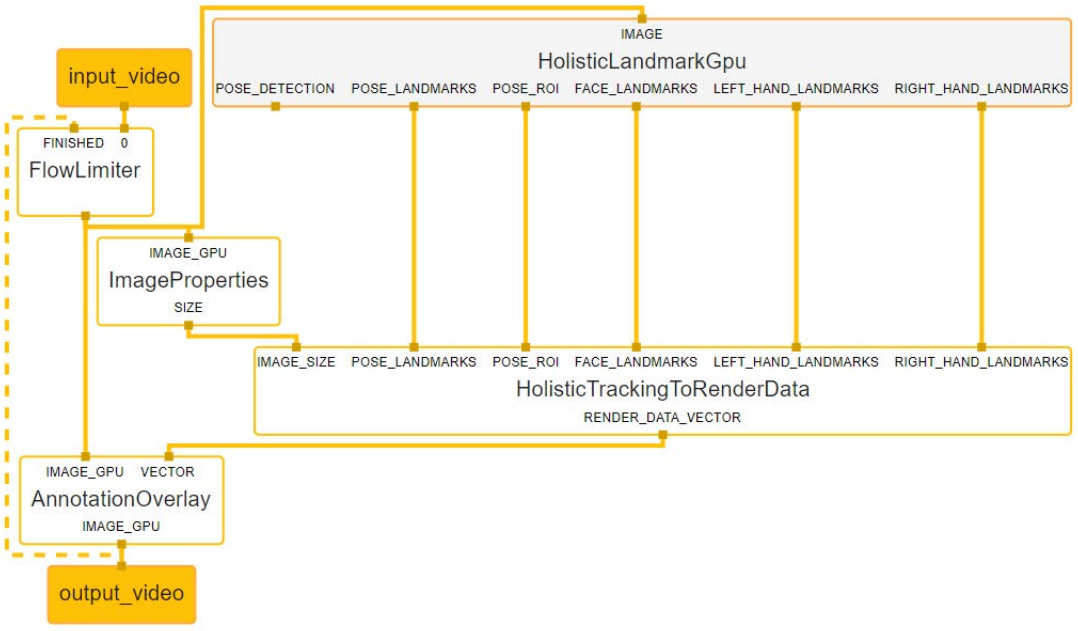

input_video
FINISHED 0
FlowLimiter
IMAGE
HolisticLandmarkGpu
POSE_DETECTION POSE_LANDMARKS POSE_ROI FACE_LANDMARKS LEFT_HAND_LANDMARKS RIGHT_HAND_LANDMARKS
IMAGE_GPU
ImageProperties
SIZE
IMAGE_SIZE POSE_LANDMARKS POSE_ROI FACE_LANDMARKS LEFT_HAND_LANDMARKS RIGHT_HAND_LANDMARKS
HolisticTrackingToRenderData
RENDER_DATA_VECTOR
IMAGE_GPU VECTOR
AnnotationOverlay
IMAGE_GPU
output_video

Figure 3.2.: Original graph topology of holistic tracking module by MediaPipe.

In Figure 3.2 the topology from the holistic tracking graph can be seen. The image was created using the visualization tool of MediaPipe by uploading the specification file of the graph. As it can be seen, the graph has as the the input video as the topmost node which is taken as input by the FlowLimiter calculator. This calculator drops input frames when necessary and has a back edge as another input from the AnnotationOverlay subgraph. The FlowLimiter outputs unaltered frames from the input video stream from the camera. These frames then get fed into the HolisticLandmarkGpu subgraph and the ImageProperties calculator. The later processes the frames and outputs the size of the frames. The HolisticLandmarkGpu is a subgraph where all the important tracking work is done, which will be explained in the following.

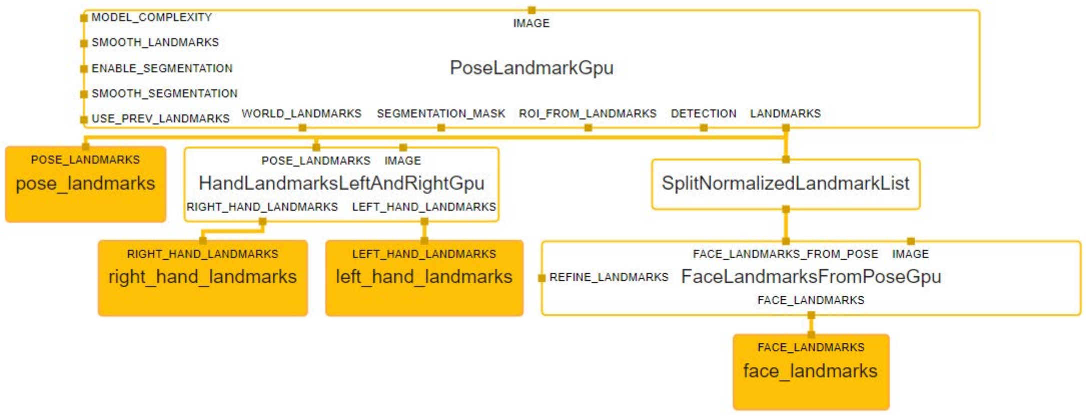
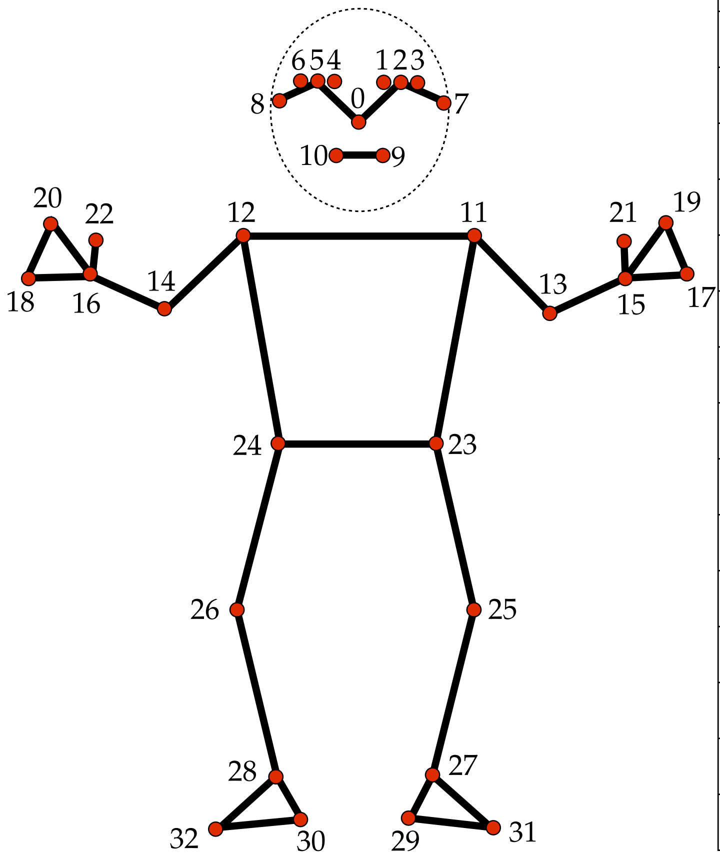


Figure 3.3.: Original HolisticLandmark subgraph.

In Figure 3.3 the subgraph topology of HolisticLandmarkGpu can be seen, where the input packets from PoseLandMarkGpu and some of the output streams, which are not important for this thesis, are omitted for visibility reasons. On top is PoseLandMarkGpu which is another subgraph, where the pose landmarks are calculated at first. These landmarks are then used in the following subgraphs HandLandmarkLeftAndRightGpu and FaceLandmarksFromPoseGpu which extract face and hand landmarks from a frame. In addition, the FaceLandmarksFromPoseGpu takes in a side packet which tells the subgraph whether to refine the face landmarks, meaning that the landmarks around the lips should be refined, resulting in ten extra face landmarks.

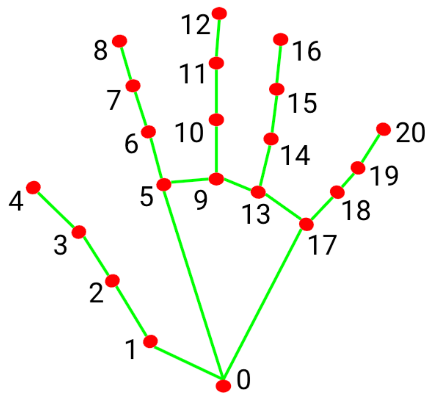
As it can be seen in Figure 3.2, all the landmarks are collected in another subgraph called HolisticTrackingToRenderData. As the name suggests, in this subgraph all the landmarks are combined and converted into a rendering vector. In a last step this rendering vector is used by the AnnotationOverlay calculator which then produces an output video. The AnnotationOverlay calculator also has a back edge to the FlowLimiter calculator, to inform it how many frames are actually created for the output. The limiter uses this information to adapt the flow of frames from the camera and throttles its output when needed.

Actually, the subgraph PoseLandmarkGpu is an existing module from the framework called Pose [29]. This module was made to detect and track human poses. Figure 3.4 visualizes the 33 detected landmarks. Furthermore, the subgraph HandLandmarksLeftAndRightGpu also uses existing models from [30]. A visualization of the hand's landmarks can be found in Figure 3.5. A last module which is also utilized is Face Mesh [31] from MediaPipe. With this model, 468 face landmarks can be detected. Interestingly, the models used in this module are lightweight, so they can also be used on mobile devices.



0. nose	17. left_pinky
1. left_eye_inner	18. right_pinky
2. left_eye	19. left_index
3. left_eye_outer	20. right_index
4. right_eye_inner	21. left_thumb
5. right_eye	22. right_thumb
6. right_eye_outer	23. left_hip
7. left_ear	24. right_hip
8. right_ear	25. left_knee
9. mouth_left	26. right_knee
10. mouth_right	27. left_ankle
11. left_shoulder	28. right_ankle
12. right_shoulder	29. left_heel
13. left_elbow	30. right_heel
14. right_elbow	31. left_foot_index
15. left_wrist	32. right_foot_index
16. right_wrist	

Figure 3.4.: All 33 pose landmarks.
Figure from [29].



0. WRIST	11. MIDDLE_FINGER_DIP
1. THUMB_CMC	12. MIDDLE_FINGER_TIP
2. THUMB_MCP	13. RING_FINGER_MCP
3. THUMB_IP	14. RING_FINGER_PIP
4. THUMB_TIP	15. RING_FINGER_DIP
5. INDEX_FINGER_MCP	16. RING_FINGER_TIP
6. INDEX_FINGER_PIP	17. PINKY_MCP
7. INDEX_FINGER_DIP	18. PINKY_PIP
8. INDEX_FINGER_TIP	19. PINKY_DIP
9. MIDDLE_FINGER_MCP	20. PINKY_TIP
10. MIDDLE_FINGER_PIP	

Figure 3.5.: 21 hand landmarks.
Figure from [30].

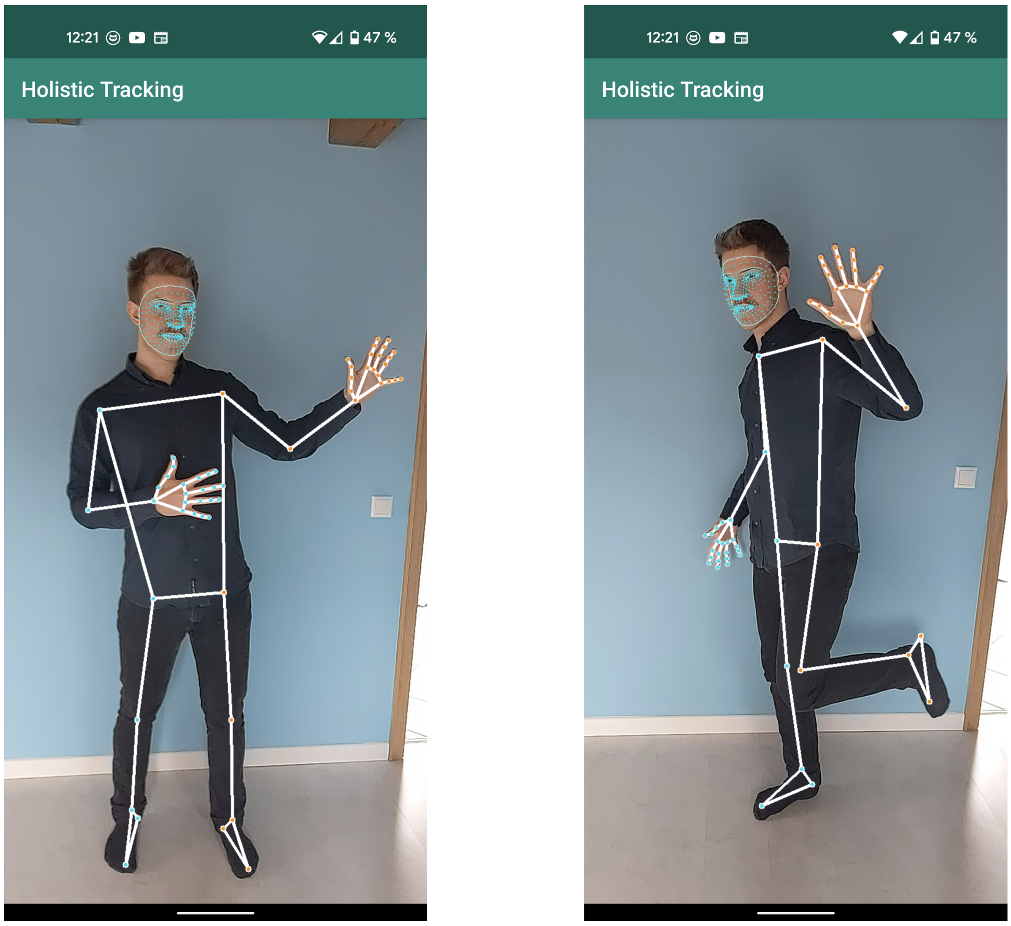


Figure 3.6.: Screenshot from MediaPipe's Holistic Tracking app.

In Figure 3.6 two screenshots from the prebuilt Holistic Tracking app for Android devices by MediaPipe can be seen. As visualized, MediaPipe does a good job precisely tracking the landmarks. However, there are cases when the framework has problems. For example, the author of [18] states that the models have difficulties when hands are overlapping or when one hand is placed in front of the face.

3.3. Sign Language Gesture Translation Model

Before a new pipeline can be created which is adapted to this thesis' problem, there are a few assumptions about the model, which is responsible for translating sign language, that need to be made. As previously mentioned, German sign language has two key concepts which are manual and non-manual means of expressions. A sophisticated translation model needs to consider all the aspects of the two key concepts to translate a single gesture. However, a perfect translation model would be able to translate gestures to the corresponding types and also form sentences in spoken German language from the types. An approach to this could be to split the tasks into several models that work together. For example one model to classify mouthings and one for the pose including both hands. While the meaning of gestures depends on various things like position of the hands, MediaPipe is not able to give meaningful results on 3D coordinates for pose estimation at this point [28]. However, this is important in sign language because signers do not only use a 2D space around them but also put information into gestures by signing close or far away from their bodies. Another problem is that signers often use their hand to point at objects or people around them. These signs can be detected as indexing signs, but to create complete sentences the model would need more information about the surroundings of the signer. One more problem is that signers sometimes create new gestures which depend on the context. These signs are known as productive signs and it would be very difficult for a network to translate them as they are most probably unknown to it.

MediaPipe can provide information about landmarks of the face, pose and both hands. In this thesis only models which take these landmarks as input will be used. This means that the pipeline needs to provide functionality to combine the landmarks and convert them into tensors so that the translation model can use them as input. The pipeline also has to take care of the model's output and transform it into rendering data. In this thesis three different types of networks will be used and evaluated. All of them perform classification tasks of single input frames.

Beginning with the convolutional network, the network has two convolutional layers with ReLU activation and as output one dense layer with Softmax activation as visualized in Figure 3.7. The first convolutional layer has a kernel of size two and also strides of size two. This makes the kernel consider each landmark individually. In the second convolutional layer, the kernel has size 16. Because of that, the layer will consider the 65 input landmarks as groups of 16 landmarks. This output is then flattened and forwarded into two fully connected layers that decrease the size of the input down to 128. As a final layer another dense layer is used with an output size of 10 or 100, depending on the amount of classes.

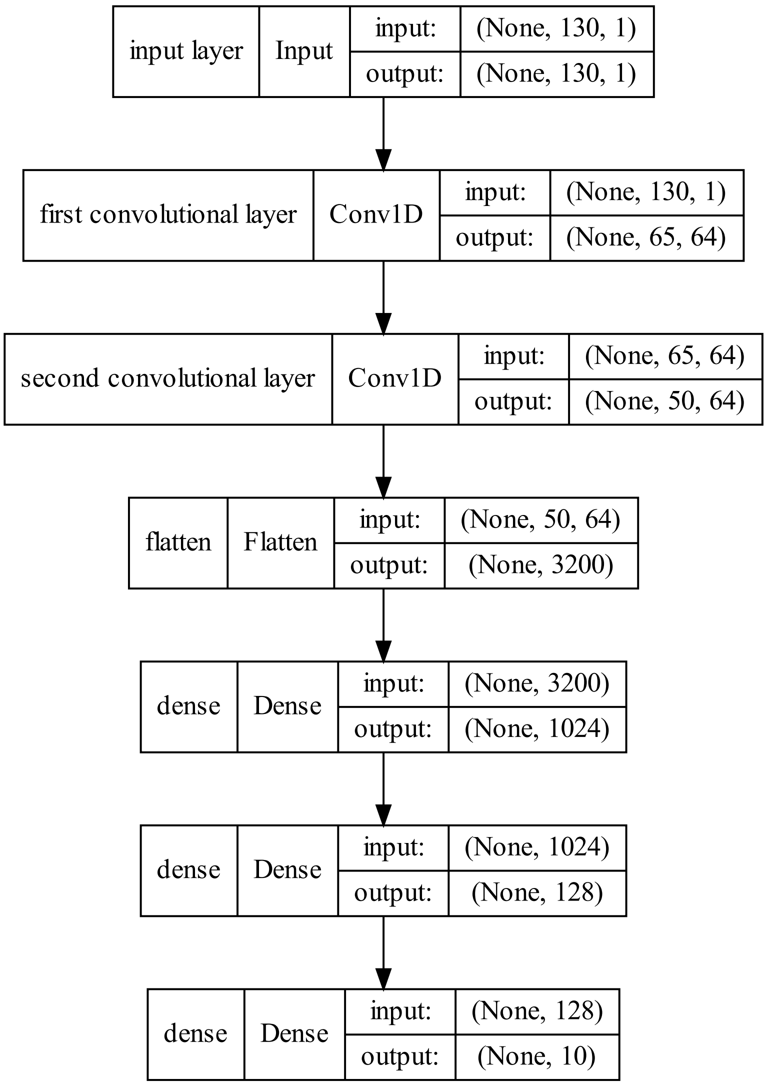


Figure 3.7.: Architecture of the CNN model.

The second network is PointNet which was discussed before. Its architecture can be seen in Figure 2.3. The third and final model is a graph neural network. This model consists of graph convolutional and pooling layers from Spektral [32] as it can be seen in Figure 3.8. Graph convolutional layers take a whole batch of graphs as input and forward the same nodes and adjacency matrix. However, the last dimension of the node's features is changed to a a desired size. They also apply ReLU activation. After each convolutional layer a pooling layer takes the batch as input and changes the shape for each graph by applying soft clustering [32]. After these layers the number of nodes is reduced to the provided size and also the adjacency matrix is adapted to the new size.

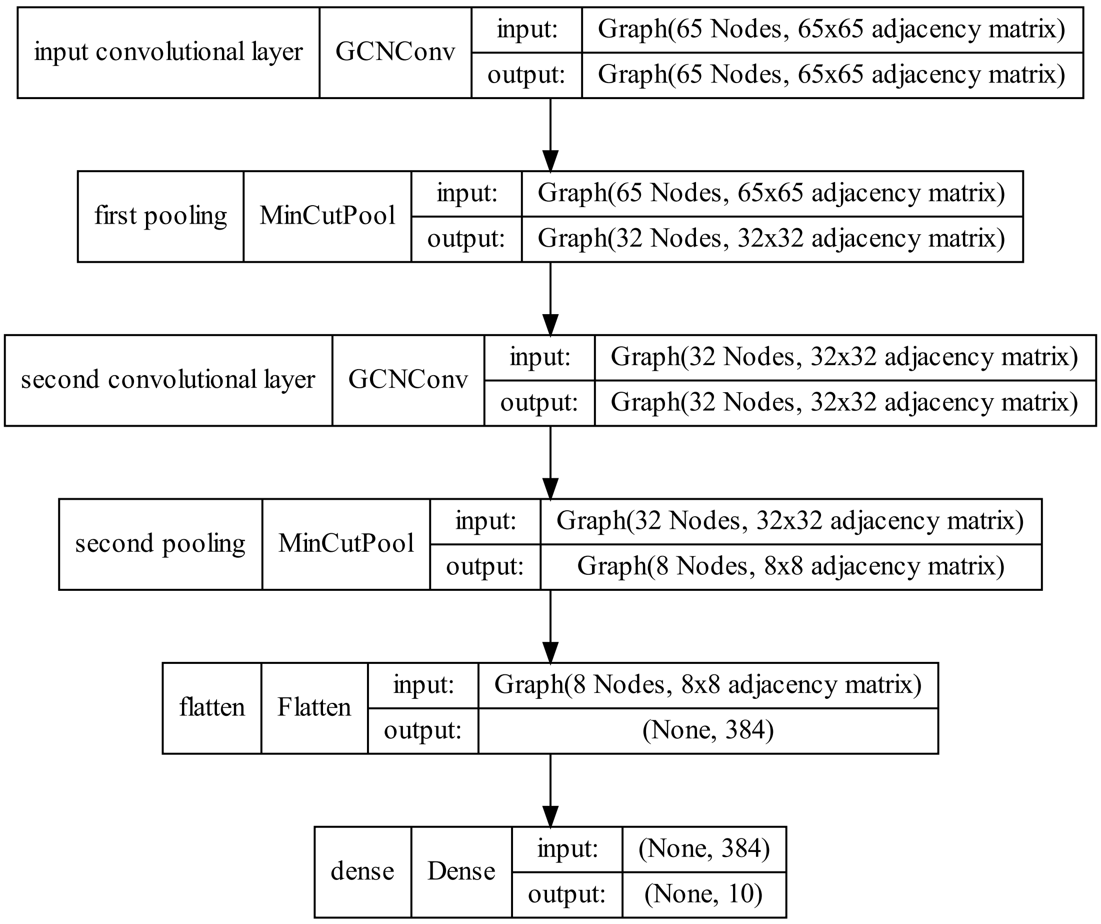

input convolutional layer
GCNConv
input:
Graph(65 Nodes, 65x65 adjacency matrix)
output:
Graph(65 Nodes, 65x65 adjacency matrix)
first pooling
MinCutPool
input:
Graph(65 Nodes, 65x65 adjacency matrix)
output:
Graph(32 Nodes, 32x32 adjacency matrix)
second convolutional layer
GCNConv
input:
Graph(32 Nodes, 32x32 adjacency matrix)
output:
Graph(32 Nodes, 32x32 adjacency matrix)
second pooling
MinCutPool
input:
Graph(32 Nodes, 32x32 adjacency matrix)
output:
Graph(8 Nodes, 8x8 adjacency matrix)
flatten
Flatten
input:
Graph(8 Nodes, 8x8 adjacency matrix)
output:
(None, 384)
dense
Dense
input:
(None, 384)
output:
(None, 10)

Figure 3.8.: Architecture of the GNN used in this thesis.

3.4. Translation Pipeline

In this section, the main part of this thesis is described which is the sign language translation pipeline. In this section all steps of the implementation of a pipeline, which does sign gesture translation on mobile devices, are explained. Beginning with the data preprocessing step which is necessary for creating a data set on which a translation model can be trained on. Then the evaluation metrics for the trained models are listed and also metrics for the performance of the app. This is followed by a section on how the models were trained and which results were achieved. In the final subsection of this section the implemented pipeline is explained in detail.

3.4.1. Data Preprocessing

For training the translation model, it is mandatory to have data in an adequate format which can be put into the network. Most preferably the data should be in form of a set of features and a target value, whereas the features will be the landmarks of a frame and the target value is the corresponding meaning of the gesture seen in the frame. For each transcript in the DGS-Korpus, a video for participant A and B and a ELAN file which has an XML-like structure is provided. After acquiring all videos and ELAN files, in a first filtering step all ELAN files, which are called meta files in the following, with no time-step data are removed. This occurs in all transcripts that are of the *Joke* category as they are not annotated by the DGS-Korpus.

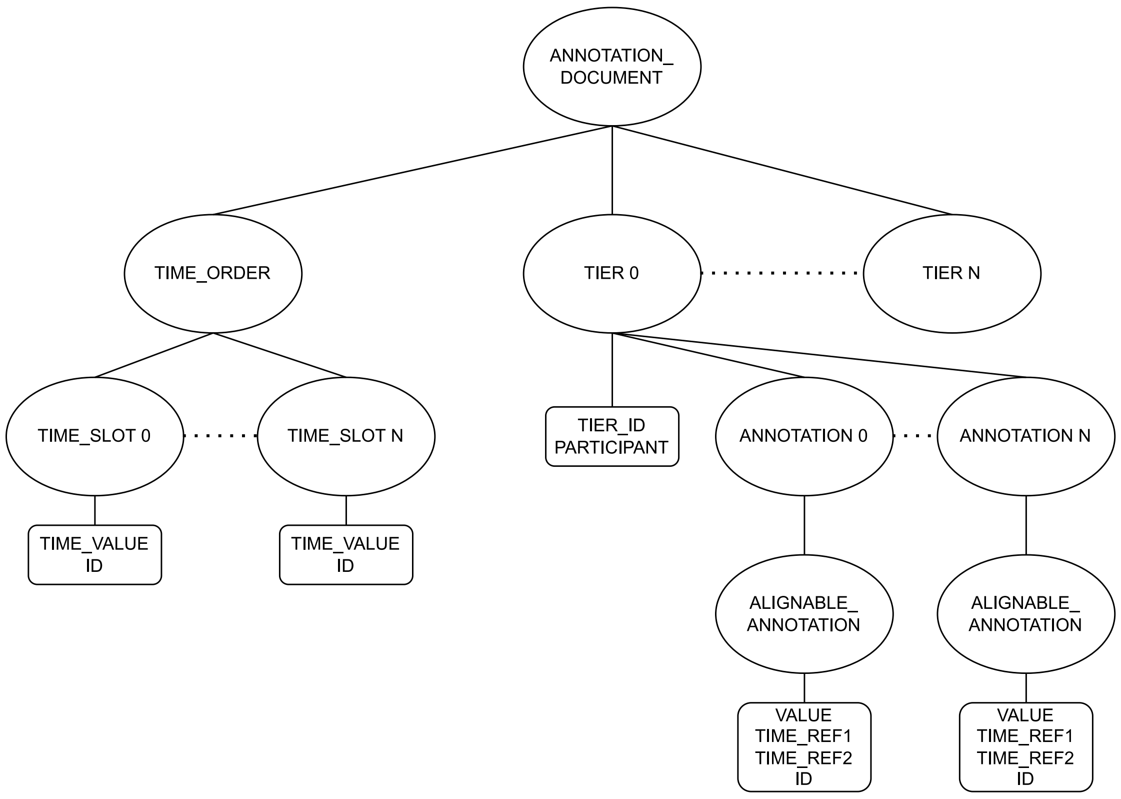


Figure 3.9.: Structure of a meta file. It has an XML-like structure with the node ANNOTATION_DOCUMENT as the root element.

Continuing with the format of the meta files as seen in Figure 3.9, where the names are slightly altered for readability reasons. These files have a root element which is the ANNOTATION_DOCUMENT node. The root's children are one TIME_ORDER node and a variable amount of TIER nodes, depending on whether translation for English or mouthings are annotated. Another reason for this variation is when only one participant is annotated. On one hand is the TIME_ORDER node which has all TIME_SLOT elements as children that store the TIME_VALUE and an ID as attributes. On the other hand, the TIER nodes represent annotations for a participant.

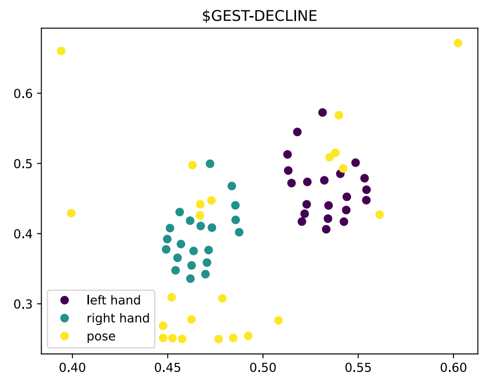
TIER ID	REFERENCE
Deutsche Übersetzung A	-
Translation into English A	-
Lexem Gebärde r A	-
Lexem Sign r A	Lexem Gebärde r A
Gebärde r A	Lexem Gebärde r A
Sign r A	Lexem Gebärde r A
Lexem Gebärde l A	-
Lexem Sign l A	Lexem Gebärde l A
Gebärde l A	Lexem Gebärde l A
Sign l A	Lexem Gebärde l A
Mundbild Mundgestik A	-
Deutsche Übersetzung B	-
Translation into English B	-
Lexem Gebärde B A	-
Lexem Sign r B	Lexem Gebärde r B
Gebärde r B	Lexem Gebärde r B
Sign r B	Lexem Gebärde r B
Lexem Gebärde l B	-
Lexem Sign l B	Lexem Gebärde l B
Gebärde l B	Lexem Gebärde l B
Sign l B	Lexem Gebärde l B
Mundbild Mundgestik B	-
Moderator	-
Translation into English Mod	-

Table 3.1.: All annotation tiers.

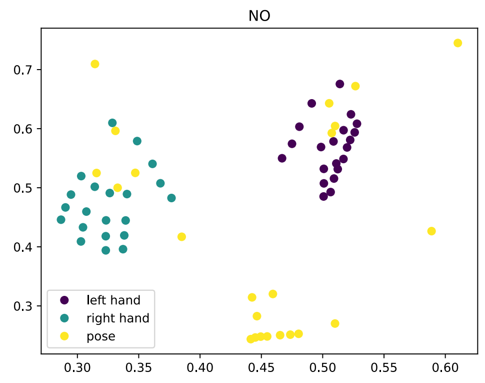
All different annotation tiers can be found in table 3.1. In this project the tiers of Gebärde of the right and left hands for both participants A and B are used. Thus, the training will use data from the types which got referenced from the subtypes stored in the corresponding Lexem sign tiers. The decision to use types instead of subtypes was made to decrease the number of possible classes in the training of the network. This also decreases the difficulty of the training.

As mentioned before, the data needs to be fed into a neural network. Thus, it is desired to have features and a label for every frame from the given TIME_SLOTS child of the TIME_ORDER element. Because the model will be trained on single frames and not sequences, it is necessary to get the final frame of a gesture. This is done in several steps. For all meta files left after the first filtering a csv file for each of the participants A and B with the desired values is created, as the annotations are stored in a shared meta file. In a first step the file is parsed into a tree object by an XML parser. Then, all time slot values of the tree are filled into a list. A dictionary is created that stores the list of time slots and all the annotations of the tree for both participants and both hands in German language.

At this point, it is required to create a merged list of annotations of the left and right hand keeping the time order in place. Now, for every final time value of an annotation a label is annotated and only the according feature values need to be created. As these feature values are represented by the landmarks found by the MediaPipe holistic tracking graph, every final frame from an annotation is processed by the graph. The resulting landmarks for face, hands and pose are stored in a dictionary. These landmark values and the label get written to a csv file, whereas all csv files are merged into a large csv file. But, before these values can be used as training data, a numpy array from the large csv file is created since it is easier to transform the data in that form. This numpy array also gets stored in a file, which can be accessed as a data set used for training networks.



(a) Normalized landmarks for the gesture $GEST-DECLINE.



(b) Normalized landmarks for the gesture NO.

Figure 3.10.: Normalized landmarks plotted in a coordinate system with different colors for different parts.

In Figure 3.10 two training samples can be seen. The landmarks in the Figure are colored by the group they belong to. It is important to note that the holistic tracking model normalizes these landmarks by the given image size. The landmarks are also flipped horizontally. Furthermore, the lexical and phonological variant information are already removed from the labels, but this will be explained in the next section.

Additionally, for the graph neural networks the data needs to be transformed first. Instead of simply using landmarks as input, this model needs these landmarks to be stored as a graph. For that, a graph for each sample needs to be created. For all landmarks, a node is created which stores the x and y coordinate from the landmark. As the graph will be topological the same for all samples, the adjacency matrix is also the same for all samples. The nodes are connected as visualized in Figure 3.4 and Figure 3.5. Furthermore, the hands are connected to the pose subgraph by connecting the wrist of the hands to the wrist of the pose landmarks. Because all nodes should be reachable from all other nodes, the landmarks from the pose which describe the points of the face also need to be connected to the landmarks from the pose which represent the body. This is done by connecting the mouth with the shoulders as seen in Figure 3.11.

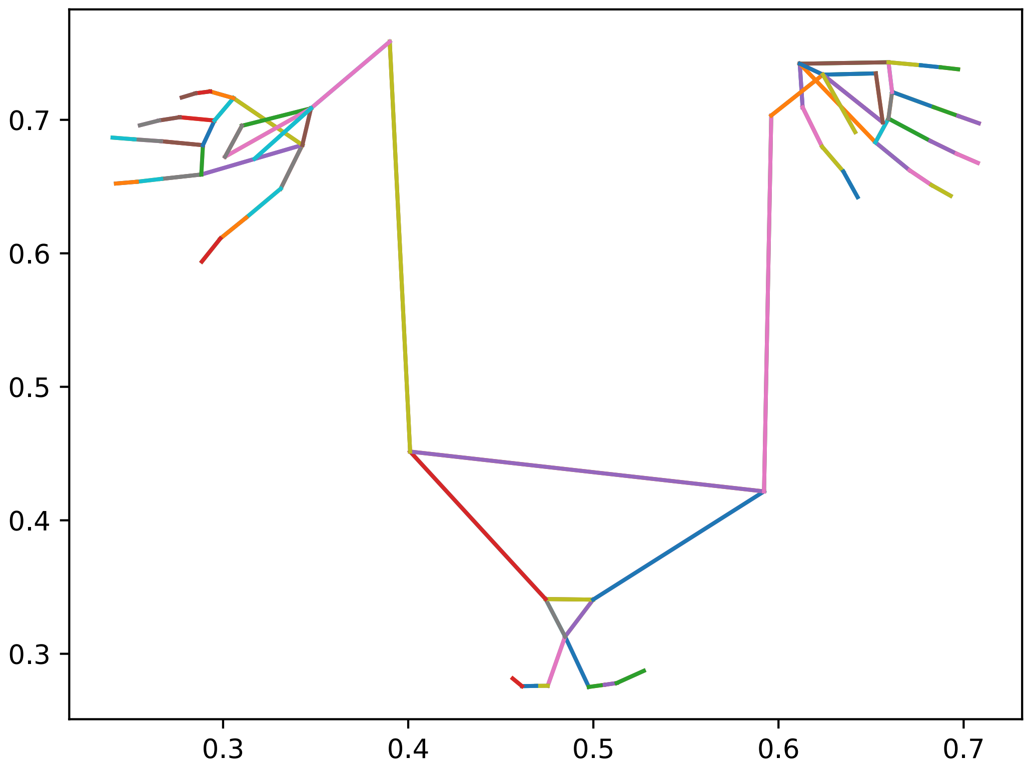


Figure 3.11.: Own visualization of how the nodes are connected to each other nodes. Each node represents a landmark from either pose or hand. The landmarks are horizontally flipped by the tracking module.

3.4.2. Evaluation Metric

For all models the ADAM [33] optimizer is used and sparse categorical crossentropy as a metric for the loss. Sparse categorical crossentropy represents the crossentropy loss between the model's predictions and the true labels. The loss is defined as

$$H(p, q) = -\sum_{\forall x} p(x) log(q(x)).$$

Here, p is the true distribution of a label and q represents the models predicted distribution for each class. For measuring scores of a model more metrics will be utilized. First, the precision which is defined as

$$precision = \frac{TP}{TP + FP}.$$

TP is the number of true positive predictions and incorrect predictions for each class. Second, the recall is another important metric and defined as

$$recall = \frac{TP}{TP + FN}.$$

In this metric the number of correctly predicted samples is divided by the sum of correctly predicted samples and the number of samples which were not predicted as the classes true label. Another metric which is utilized is the f_1 score. It represents the accuracy of a model. It is a combination of the two previous metrics precision and recall and defined as

$$f_1 = \frac{TP}{TP + \frac{1}{2}(FP + FN)}.$$

The f_1 score is an important metric because it looks at both the precision and recall and thus not only the proportion of true positives to false positives or true positives to false negatives. As a final metric the accuracy of the model is studied. The accuracy is defined as

$$accuracy = \frac{TP + TN}{|N|}.$$

In words this means the number of correctly classified samples divided by the number of total samples N. In general, the accuracy metric is used when there are no class imbalances. The f_1 score is better when there are class imbalances, because when the model only learns to correctly classify a dominant class, it will not be useful in a real world application. A theoretical example could be the following. If in a language the most often appearing word is GOOD, with an occurrence of 80% among all other words. Then a model that only learns to predict the class GOOD will achieve an accuracy of 80% which would be a decent score. In comparison to that the f_1 score also considers the distribution of the classes. However, the f_1 is harder to interpret as it is a combination of precision and recall.

As performance of apps for mobile devices is a crucial topic, the pipeline will be measured in terms of time spent per calculator, frames per second and idle times. Therefore, metrics for measuring the performance of the pipeline are needed as it is important to keep computational costs as low as possible. For the profiling of the app MediaPipe's built-in tracer and profiler are used [34]. With the help of these tools, it is possible to measure how much time calculators need to process incoming packages and send new packages as output. Among the most important metrics, which these tools keep track of, are the frames per second which each calculator can produce, the latency for each calculator, the time spent within each calculator and the number of packages which were received and either completed or dropped. The latency represents the time a calculator needs from receiving a package until actually processing it. Enabling the profiling is done by adding the statement seen in Listing 3.1 to the main graph's file. It is also required to add a flag while building the app by setting MEDIAPIPE_PROFILING to 1.

```
# profiler's values
profiler_config {
  trace_enabled: true
  enable_profiler: true
  trace_log_interval_count: 200
  trace_log_path: "/sdcard/Download/"
}
```

Listing 3.1.: The profiling configuration.

3.4.3. Training the Translation Model

Like in all scenarios where supervised machine learning is used, the data first needs to be split into a training and a test set, whereas the test set is left untouched, until a decision for a final model is made. A requirement for the test data is that it has to be unseen, so it is necessary to select participants from the set of all participants and exclude them. Another requirement is that the set should have a similar distribution as the training set. Participants can be distinguished by a combination of the geographical region value and a unique id for that region, whereupon 13 regions exist.

ber	fra	goe	hb	hh	koe	lei	mst	mue	mvp	nue	sh	stu
28	30	20	16	16	42	28	30	26	16	17	16	36

Table 3.2.: Amount of participants per region.

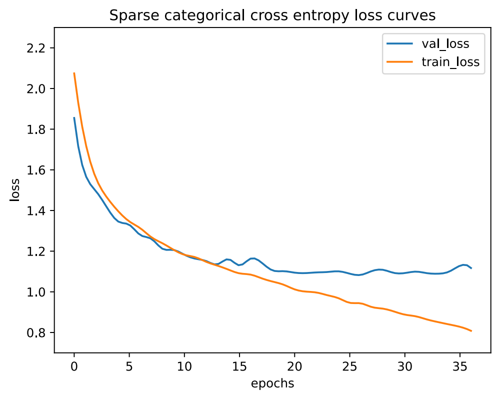
In Table 3.2 the amount of participants per region can be seen, altogether 321 participants exist. For this project, two randomly selected participants per region defined by the DGS-Korpus were put into the test set. The sampling was done by creating lists of participants for all regions and then sampling two random numbers in the range of 0 to the length of the list. These random numbers are used as indexes and the participants at the according index of the list are chosen for the test set. All the videos, where a participant of the test set is shown, are moved into a separate directory. Altogether, the test set contains 49 videos. In comparison to the training set with 527 videos, the test set has roughly 8.5% of the total available annotated videos. However, this does not guarantee that the actual size of the test set is 8.5% because not all videos have an equal amount of annotated frames. This is because of the different length of videos and in some cases participants are less communicative and rather listen to the other participant. Yet, it is approximately in the range of 8.5% since the number of videos in the test set is high enough to have a roughly averaged amount of annotations. By doing this sampling of participants, there will be no bias of German sign language in the test set in terms of regional differences.
The aforementioned data set stored as a numpy array is used to create sub data sets. As a first experiment a top-10 classes data set is created where the ten most used signs are

taken into consideration. But before that, the labels of each sample were changed. A label consists of several information such as the actual name of the type, the phonological variant and the lexical variant. Finally, a label can have an asterisk at the end to show that the according sign differs from its normal form and a circumflex to show that it is a type and not a sub-type. An example for a label is HOUSE1Aˆ. For this project all phonological and lexical variant information are removed. The information about the lexical variant is represented by the number after the types name and the phonological variant is represented by the letter after the number. Additionally, the asterisk and circumflex are also removed. Altogether, this groups labels into their type's content. An advantage of this grouping is that there will be fewer classes and more samples per class. However, with this approach information is lost which would be needed for a real world sign language translation model.

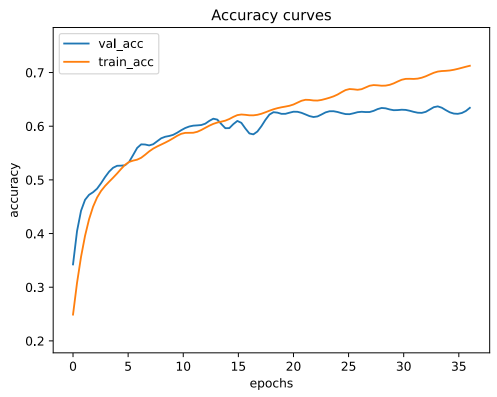
I	$INDEX	$GEST-OFF	$PROD	$GEST	NO	$NUM-ONE-TO-TEN	GOOD	TO-KNOW-OR-KNOWLEDGE	MUST
22702	22129	12433	5588	5391	4626	4623	3334	2999	2965

Table 3.3.: Top 10 classes of signs in the available DGS-Korpus' videos.

After transforming the labels, the ten most often appearing classes are seen in 3.3. For training the first network another data set is created with an equal amount of samples per class. The amount was set to 2000 samples per class. After the preprocessing steps, the models can be trained. Beginning with the CNN, it was trained with a randomly split training and validation set whereas the training set has 80% of the available samples. The batch size used for the training was set to 128 and as an optimizer ADAM with learning rate set to 0.001 was utilized. In addition to that, early stopping depending on the validation loss with patience 10 and sparse categorical cross entropy as a loss function was used. After 37 epochs the training was stopped by the early stopping mechanism. In Figure 3.12 the loss curve for both training and validation set can be seen as well as the accuracy of both set. Table 3.4 shows the classes' individual precision and recall scores. Overall, the network achieves a validation set accuracy of 63%. It is noticeable that the model has better scores for classes that represent a type that is actually a real word unlike classes like $GEST, $PROD and $INDEX. $INDEX is used by a signer to point at something locally and $PROD are gestures which are made up in the moment by the signer and are no existing gestures.



(a) Categorical cross entropy loss over epochs.



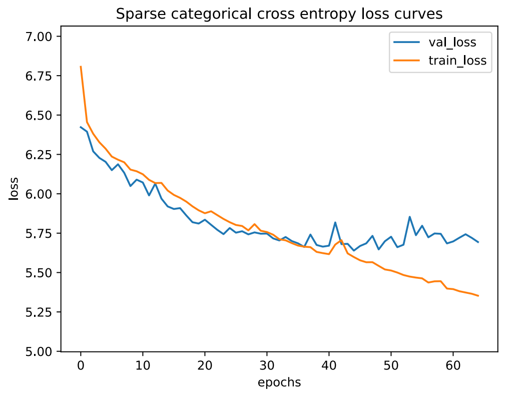
(b) Accuracy over epochs

Figure 3.12.: Training curves for the CNN on 10 classes.

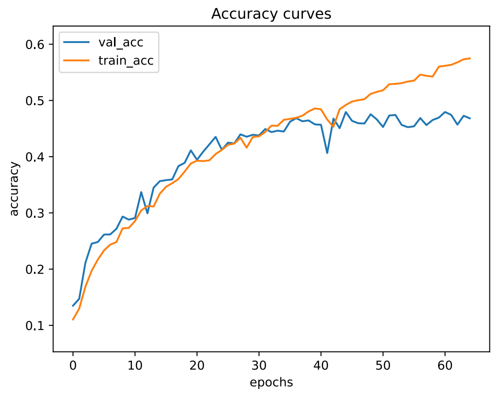
Simple CNN scores on each class				
Label	precision	recall	f1-score	support
TO-KNOW-OR-KNOWLEDGE	0.81	0.78	0.79	428
$INDEX	0.47	0.54	0.51	409
$GEST	0.29	0.33	0.31	355
MUST	0.59	0.68	0.63	391
I	0.73	0.85	0.78	409
$NUM-ONE-TO-TEN	0.66	0.60	0.63	382
$PROD	0.51	0.48	0.49	398
NO	0.72	0.68	0.70	421
GOOD	0.9	0.81	0.85	391
$GEST-OFF	0.63	0.48	0.54	416

Table 3.4.: CNN's scores for each class.

The same was done for the training of the PointNet. While its architecture has a better fit to the problem, it could not surpass the results of the CNN model. Overall, it reaches an accuracy of 47.5% on a validation set. Noticeably is that in comparison to the simple CNN all scores are shifted by more or less the same amount, besides for the classes $INDEX and $PROD which are far off.



(a) Categorical cross entropy loss over epochs.



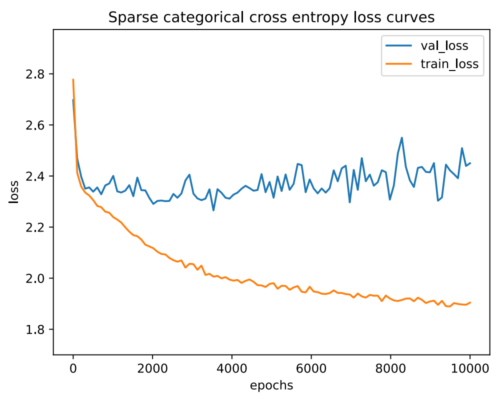
(b) Accuracy over epochs.

Figure 3.13.: Training curves of PointNet on 10 classes.

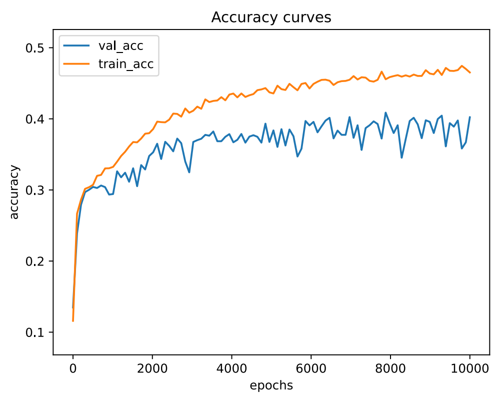
PointNet scores on each class				
Label	precision	recall	f1-score	support
TO-KNOW-OR-KNOWLEDGE	0.75	0.66	0.71	428
$INDEX	0.33	0.28	0.30	409
$GEST	0.23	0.08	0.12	355
MUST	0.49	0.57	0.53	391
I	0.52	0.68	0.59	409
$NUM-ONE-TO-TEN	0.37	0.43	0.40	382
$PROD	0.35	0.42	0.38	398
NO	0.56	0.51	0.53	421
GOOD	0.53	0.49	0.51	391
$GEST-OFF	0.42	0.50	0.46	416

Table 3.5.: PointNet's scores for each class.

As the third network architecture, the graph neural network is also trained on the same data set.



(a) Categorical cross entropy loss over epochs.



(b) Accuracy over epochs.

Figure 3.14.: Training curves of the GNN on 10 classes.

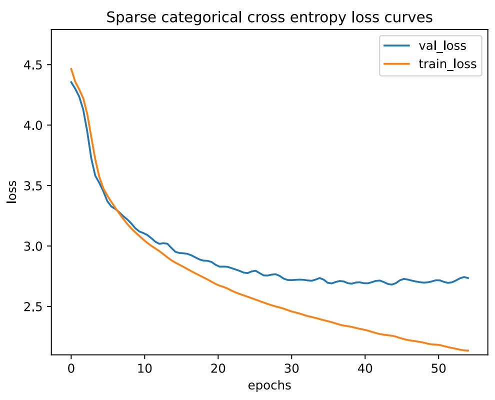
Most notably, the training for this network takes a lot more epochs to converge. After approximately 10000 epochs the validation accuracy stops increasing and peaks at 41% as can be seen in Figure 3.14. Note, that the curves are smoothed.

GNN scores on each class				
Label	precision	recall	f1-score	support
TO-KNOW-OR-KNOWLEDGE	0.53	0.75	0.62	400
$INDEX	0.25	0.22	0.24	404
$GEST	0.21	0.23	0.22	411
MUST	0.4	0.2	0.26	382
I	0.34	0.78	0.48	426
$NUM-ONE-TO-TEN	0.3	0.31	0.31	426
$PROD	0.35	0.36	0.35	363
NO	0.57	0.28	0.38	404
GOOD	0.43	0.18	0.25	383
$GEST-OFF	0.41	0.32	0.36	401

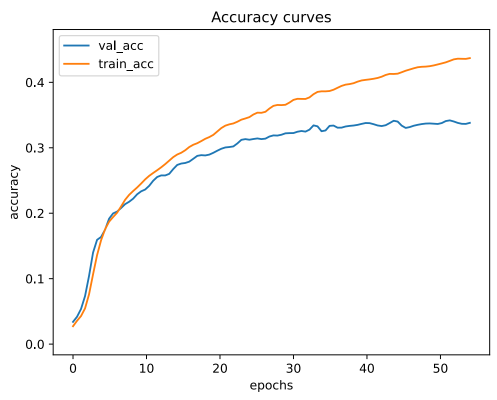
Table 3.6.: GNN's scores for each class.

Overall, the GNN has similar precision scores for each class compared to PointNet. However, the f_1 is lower for all classes. The best f_1 score is achieved by the class TO-KNOW-OR-KNOWLEDGE, followed by the class I.

In the next experiment, the networks were trained on a data set with the top 100 classes. For this, all three architectures were kept the same as before, but with a change in the output layer to adapt to 100 classes instead of 10 classes. All classes can be found in Table A.3. For that a new sub data set is created in the same way it was done for the top 10 class data set. However, as there are fewer samples per classes, the amount per class is set to 500. Doing so, an equalized data set is created.



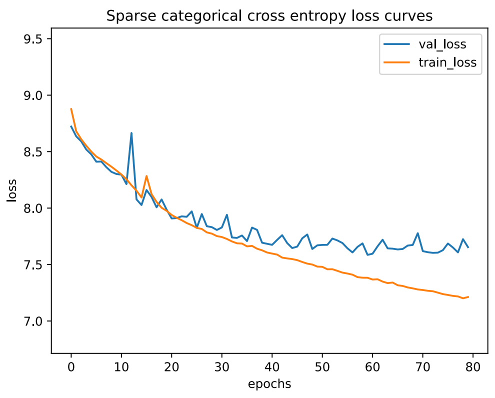
(a) Categorical cross entropy loss over epochs.



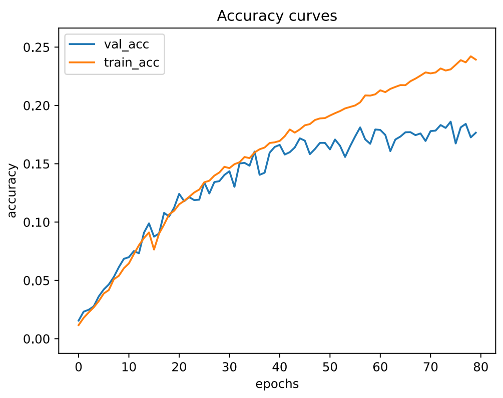
(b) Accuracy over epochs.

Figure 3.15.: CNN's performance on the top 100 classes dataset.

Figure 3.15 shows that the CNN model still is able to learn to distinguish the classes, however the accuracy dropped from 63% to approximately 33% on a validation set going from 10 to 100 classes.



(a) Categorical cross entropy loss over epochs.

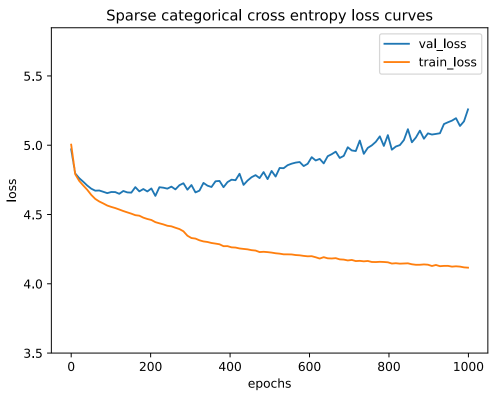


(b) Accuracy over epochs

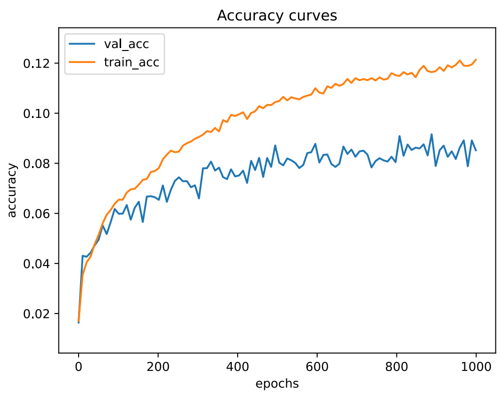
Figure 3.16.: PointNet's performance on the top 100 classes data set.

Again, PointNet is also trained on the data set with the top 100 most appearing classes. As seen in Figure 3.16 the model peaks at 19% validation accuracy. Therefore, it achieves a lower accuracy than the convolutional network.
Finally, the GNN is trained on the same data set as the other two networks. After 1000 epochs the results yields, as seen in Figure 3.17. It is again the worst performing model among the three networks, despite the fact that it has been trained for a lot more epochs. Another noticeable fact is, that the validation loss already is increasing after 200 epochs,

but the validation accuracy is still increasing after that. Again, the curves are smoothed to reduce jitter.



(a) Categorical cross entropy loss over epochs.



(b) Accuracy over epochs.

Figure 3.17.: GNN's performance on the top 100 classes data set.

3.4.4. Adapting Holistic Tracking Graph

Instead of creating a new pipeline from scratch, existing modules and calculators from MediaPipe are used. As a base for the graph, used in the sign language translation app, the holistic tracking graph seen in Figure 3.2 will be used and adapted such that it still does all the landmark tracking. However, it will be adapted in terms of what can be seen on the display and the translation model also needs to be implemented.

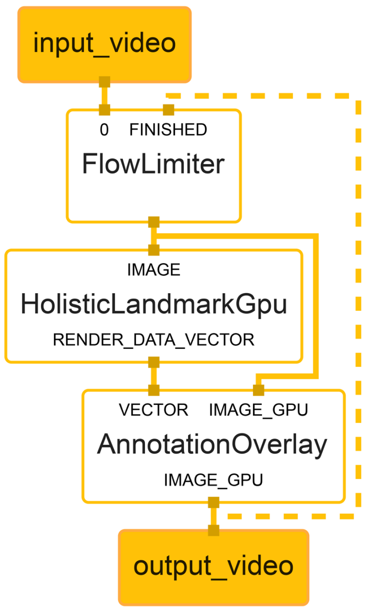


Figure 3.18.: Graph topology of sign translating app.

In Figure 3.18 it can be seen that the graph has changed in the mentioned regions. First of all, the subgraph HolisticTrackingToRenderData is removed completely, as all the visualization, like pose, hand and face landmarks displayed by it, is not necessary in the translation app. On top of that the HolisticLandmarkGpu subgraph now only outputs the render data stored in a vector as packets to the AnnotationOverlay calculator. Nevertheless, all the major changes happen inside the HolisticLandmarkGpu subgraph.

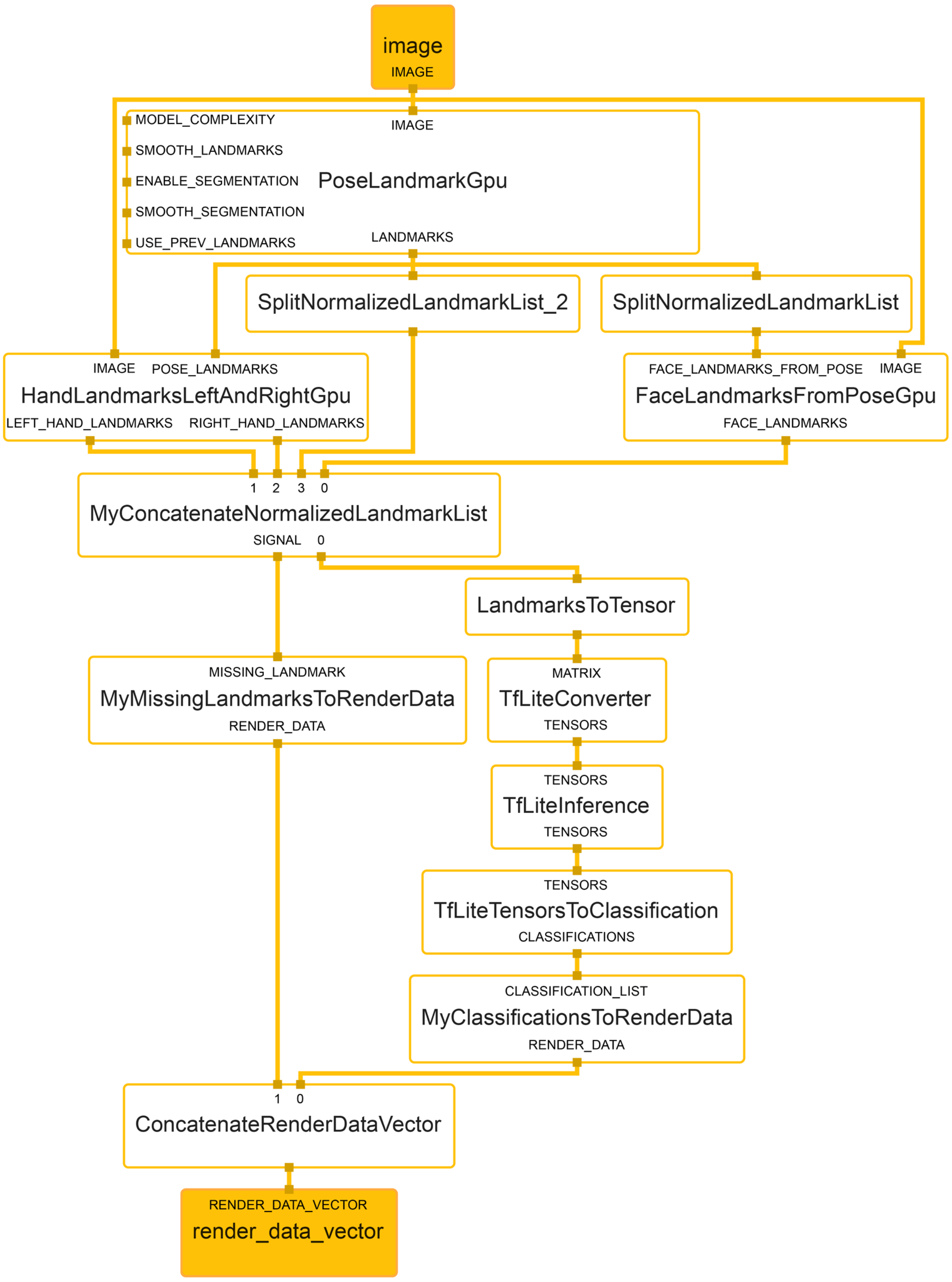


Figure 3.19.: Adapted HolisticLandmark subgraph to do sign translation from landmarks.

Figure 3.19 shows the transformed subgraph HolisticLandmark. As before, input side packets from the subgraph PoseLandmarkGpu are omitted for visibility reasons. Beginning with the first side packet which is MODEL_COMPLEXITY, it determines the complexity of the model used for detecting pose landmarks. Higher complexity means higher accuracy but also higher computational costs, thus it is set to 1 which produces more accurate landmarks than when using complexity 0, but the computational costs are kept reasonable in comparison to complexity 2. Continuing with SMOOTH_LANDMARKS which is the next side packet that is set to true as default. This helps reduce jitter across different frames for pose landmarks [28]. ENABLE_SEGMENTATION is set to false since a segmentation mask is not needed, thus SMOOTH_SEGMENTATION is also set to false. The changed subgraph HolisticLandmark still takes the images provided by the FlowLimiter calculator and forwards them to three further subgraphs which are PoseLandmarkGpu, FaceLandmarksFromPoseGpu and HandLandmarksLeftAndRightGpu. The first one also provides inputs for the later two. Altogether, these subgraphs create landmarks for pose, face and hands.

```
# Predicts pose landmarks.
node {
  calculator: "PoseLandmarkGpu"
  input_stream: "IMAGE:image"
  input_side_packet: "MODEL_COMPLEXITY:model_complexity"
  input_side_packet: "SMOOTH_LANDMARKS:smooth_landmarks"
  input_side_packet: "ENABLE_SEGMENTATION:enable_segmentation"
  input_side_packet: "SMOOTH_SEGMENTATION:smooth_segmentation"
  input_side_packet: "USE_PREV_LANDMARKS:use_prev_landmarks"
  output_stream: "LANDMARKS:pose_landmarks"
}

# Predicts left and right hand landmarks based on the initial
# pose landmarks.
node {
  calculator: "HandLandmarksLeftAndRightGpu"
  input_stream: "IMAGE:image"
  input_stream: "POSE_LANDMARKS:pose_landmarks"
  output_stream: "LEFT_HAND_LANDMARKS:left_hand_landmarks"
  output_stream: "RIGHT_HAND_LANDMARKS:right_hand_landmarks"
}

# Extracts face-related pose landmarks.
node {
  calculator: "SplitNormalizedLandmarkListCalculator"
  input_stream: "pose_landmarks"
  output_stream: "face_landmarks_from_pose"
  options: {
    [mediapipe.SplitVectorCalculatorOptions.ext] {
      ranges: { begin: 0 end: 11 }
    }
  }
}

# Predicts face landmarks based on the initial pose landmarks.
node {
  calculator: "FaceLandmarksFromPoseGpu"
  input_stream: "IMAGE:image"
  input_stream: "FACE_LANDMARKS_FROM_POSE:face_landmarks_from_pose"
  output_stream: "FACE_LANDMARKS:face_landmarks"
}
```

Listing 3.2.: First nodes handling the detection of landmarks.

3.4.4.1. Combining Landmarks

As these landmarks need to be interpreted by the translation model, they first need to be concatenated and transformed into a tensor. But before that, the landmarks from the pose are stripped from lower body information because they are not needed in the translation of German sign language as mentioned in 3.4.1 . This is done in the SplitNormalizedLandmarkList calculator which can be parameterized with the desired range of landmarks of a given input landmark list stream. Beginning with the concatenation step, which is done in the MyConcatenateNormalizedLandmarkList calculator. As the name suggests, it takes lists of landmarks as input and combines them into a list of normalized landmarks. The landmarks are already normalized by the subgraphs beforehand, so all it needs to do is merge the provided lists into a bigger list. The calculator does so by iterating over its input streams from the three previous subgraphs, whereas the subgraph responsible for the hand landmarks has two outputs, one for each hand. In the implementation the calculator actually also considers the landmarks from the face subgraph. However, it is told to skip these by setting its options for skip_face_landmarks to true. Furthermore, this calculator has an option only_emmit_if_all_present which changes the output behaviour in such a manner that when it is set to True, the calculator will only output a package to the next calculator when all the input streams have a package ready to be consumed. This is a crucial decision to be made for the inference on the model, because sometimes there are not packages from all input streams ready. There are several reasons for this, for example when the hands are not present in a given frame or the face landmarks could not be dealt with. Hence, when this option is set to True, the graph will have less outputs, for example when a hand is hidden behind the body. However, when the option is set to False, it is necessary to take care of the missing packets from the input streams. In this implementation this is done by creating dummy landmarks which can be set to a desired valued e.g. 0 or -1. While this is possible and maybe useful in other projects with different goals, this approach will not be used in this thesis as training a model already is a challenging problem, so making the model learn to take care of these dummy landmarks would be even more difficult. Providing these two boolean values is done by adding an option to the calculator. The MyConcatenateNormalizedLandmarkList calculator has a second output which is a signal stored in a Classification proto. The signal is used to provide information when some of the landmarks could not be acquired. In Table A.2 all different signal codes are listed. The code represents the sum of missing inputs, whereas each input's value is calculated by two to the power of its index in the input stream list. Handling the signal is done in a calculator explained later.

```
# Removes lower body from pose landmarks.
node {
  calculator: "SplitNormalizedLandmarkListCalculator"
  input_stream: "pose_landmarks"
  output_stream: "no_lower_body_pose_landmarks"
  options: {
    [mediapipe.SplitVectorCalculatorOptions.ext] {
      ranges: { begin: 0 end: 23 }
    }
  }
}

# Combines pose landmarks all together. Piped to translation model
node {
  calculator: "MyConcatenateNormalizedLandmarkListCalculator"
  input_stream: "face_landmarks"
  input_stream: "left_hand_landmarks"
  input_stream: "right_hand_landmarks"
  input_stream: "no_lower_body_pose_landmarks"
  output_stream: "landmarks_merged"
  output_stream: "SIGNAL:missing_landmarks"
  node_options: {
    [type.googleapis.com/mediapipe.
    MyConcatenateVectorCalculatorOptions] {
      only_emit_if_all_present: true
      skip_face_landmarks: true
    }
  }
}
```

Listing 3.3.: Next nodes, combining the present landmarks.

3.4.4.2. Converting Landmarks to Tensors

After this first step, the combined list of normalized landmarks is send to the next calculator which is LandmarksToTensor. Inside this, the list of landmarks is taken and stored in a buffer. Here, only the x and y values of the landmarks are kept, as the model is not trained on z, visibility and presence. From the buffer a Matrix of the MediaPipe Matrix format is created and send to the next calculator.
The following three calculators are provided by the MediaPipe framework. Beginning with TfLiteConverter calculator. This calculator can have either a Matrix proto, Tensor proto or an Image proto as input. This is the reason why the landmarks were transformed from a list into a Matrix in the first place. The main work this calculator does, is transform its input into TfLite tensors and can be used with tensors on CPU or GPU.

```
# Converts merged landmarks to Matrix format
node {
  calculator: "LandmarksToTensorCalculator"
  input_stream: "landmarks_merged"
  output_stream: "matrix_from_landmarks"
  node_options: {
    [type.googleapis.com/mediapipe
    .LandmarksToTensorCalculatorOptions]{
    attributes: [X, Y]
    }
  }
}

# Converts Matrix to Tensors
node {
  calculator: "TfLiteConverterCalculator"
  input_stream: "MATRIX:matrix_from_landmarks"
  output_stream: "TENSORS:tensors_from_matrix"
  options: {
    [mediapipe.TfLiteConverterCalculatorOptions.ext] {
      zero_center: false
    }
  }
}
```

Listing 3.4.: Next nodes, converting landmarks to tensors which can be put into the translation model.

3.4.4.3. Inference on Translation Model

Followed by the TfLiteInference calculator which takes the TfLite tensors from its input stream and inferences them on the model provided in the calculator's options. The output of this calculator is the model's output tensors.

```
# Predicts type class.
node {
  calculator: "TfLiteInferenceCalculator"
  input_stream: "TENSORS:tensors_from_matrix"
  output_stream: "TENSORS:tensors"
  options: {
    [mediapipe.TfLiteInferenceCalculatorOptions.ext] {
      model_path: "mediapipe/modules/holistic_landmark/model.tflite"
    }
  }
}
```

Listing 3.5.: Node that takes care of inference on the translation model.

3.4.4.4. Converting Tensors to Classifications

Since a classification model is used, the output tensors from the translation model need to be transformed into Classifications. This is done by the TfLiteTensorsToClassification calculator. Classification is another format of the MediaPipe framework which is represented by its index in a given label-map, the probability score calculated by the model and a label name. Classifications are represented in C++ protocol buffers. The calculator has three options it can be set to. First of, the path to the label-map. Second of, a minimum score threshold which needs to be exceeded for a gesture to be qualified as a classification and a top k value which determines how many classifications will be added to the output in descending order depending on their scores. Each packet received by this calculator goes through the following processing steps. In the beginning the calculator checks if it is set to binary classification or multiple classes classification. For this thesis only the later is needed. It then proceeds to check all scores for each class and creates a Classification for all that surpass the threshold. These Classifications are then stored into a list, which gets sorted and cut in a final step, where the last elements are cut depending on the value set of top k. In this project k will be set to 1. The top k classifications are outputted as a Classification list.

```
# Tensors to classification proto
node {
  calculator: "TfLiteTensorsToClassificationCalculator"
  input_stream: "TENSORS:tensors"
  output_stream: "CLASSIFICATIONS:classifications"
  options: {
    [mediapipe.TfLiteTensorsToClassificationCalculatorOptions.ext] {
      top_k : 1
      min_score_threshold: 0.1
      label_map_path: "labelmap.txt"
    }
  }
}
```

Listing 3.6.: Node converting tensors from model to classification C++ proto.

3.4.4.5. Converting Results to Render Data

Classifications now need to be displayed which is done in the continuing two calculators. MyClassificationToRenderData takes the list and transforms it to render data. It does so by iterating over the list of Classifications and adding render data for each of the Classifications. The render data contains values of the color given in the calculators options, a text with properties like the string to be displayed, font height and thickness. Another calculator to generate render data is the MyMissingLandmarksToRenderData calculator. It takes a Classification as input where the signal of the MyConcatenateNormalizedLandmarkList is stored. Render data from this calculator is set to be shown at a lower position than the output of the translation model. Also, for better visibility a filled rectangle which is placed behind the signal's text is added to the render data. Altogether, the render data gets piped to the last node of the HolisticLandmark subgraph, which is the ConcatenateRenderData calculator. In this last calculator the render data is converted into a vector of render data. This calculator also has the functionality to concatenate multiple render data input streams into one output stream, similar to the ConcatenateNormalizedLandmarkList calculator.
Coming back to the graph seen in Figure 3.18, in a final step the output from the HolisticLandmarkGpu subgraph and images from the FlowLimiter are taken as input by the AnnotationOverlay calculator. Here output images, that show the information stored in the render data vector, are created which are displayed on the screen of the device.

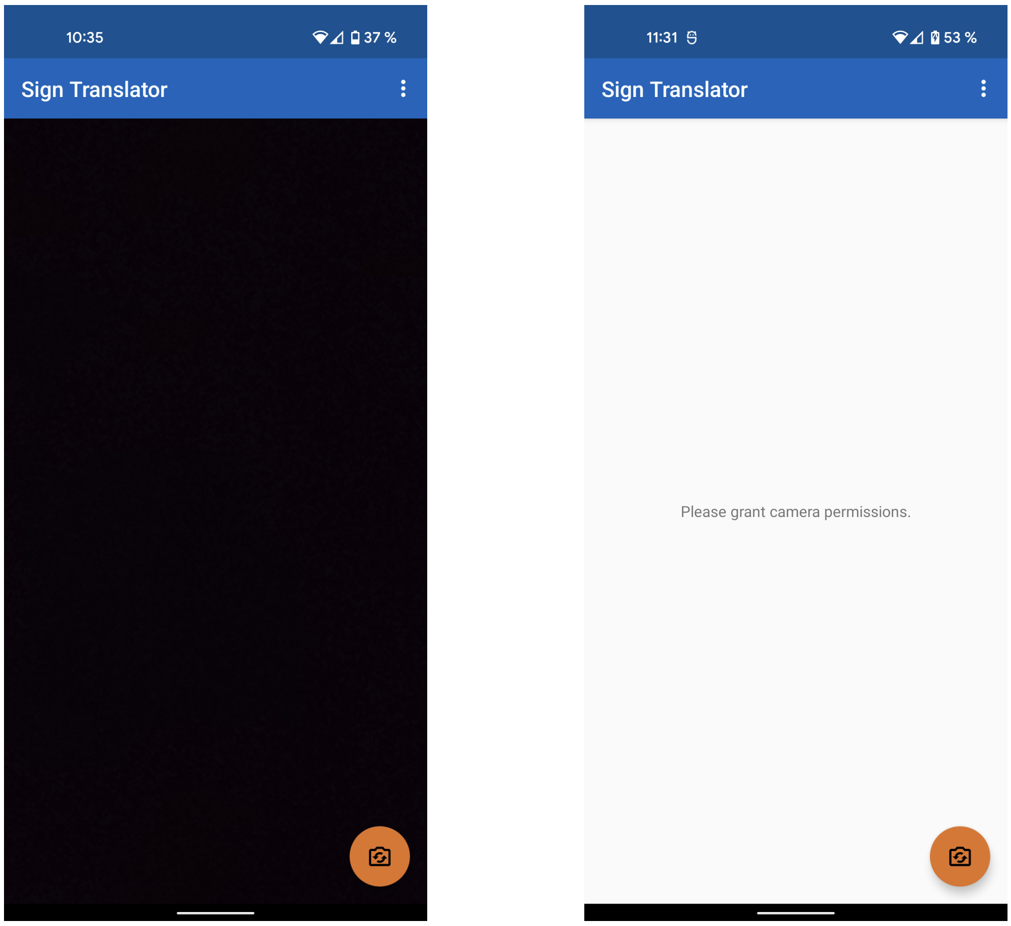
```
# classification proto to render data
node {
  calculator: "MyClassificationsToRenderDataCalculator"
  input_stream: "CLASSIFICATION_LIST:classifications"
  output_stream: "RENDER_DATA:classification_render_data"
  options {
    [mediapipe.MyClassificationsToRenderDataCalculatorOptions.ext] {
      produce_empty_packet : false
      color { r: 255 g: 0 b: 0 }
    }
  }
}

# missing landmarks info to render data
node{
  calculator: "MyMissingLandmarksToRenderDataCalculator"
  input_stream: "MISSING_LANDMARK:missing_landmarks"
  output_stream: "RENDER_DATA:missing_info_render_data"
  options {
    [mediapipe.MyMissingLandmarksToRenderDataCalculatorOptions.ext] {
      produce_empty_packet : false
      color { r: 0 g: 0 b: 0 }
      fill_color { r: 255 g: 255 b: 255}
    }
  }
}

# Concatenates all render data.
node {
  calculator: "ConcatenateRenderDataVectorCalculator"
  input_stream: "classification_render_data"
  input_stream: "missing_info_render_data"
  output_stream: "render_data_vector"
}
```

Listing 3.7.: Nodes taking care of creating rendering data.

All python code for creating the data set and training the models can be found in a git repository stored in https://gitlab.lrz.de/ga94mor/ma.
Furthermore, the adapted MediaPipe directory including all new calculators is also stored there. Also, a bash file which builds and deploys the app via adb can be found in the repository. The code from MediaPipe is licensed under Apache 2.0 [35].

3.5. User Interface

In this chapter the app's user interface is documented. For that, the design of the app is elicited. Beginning with the two existing activities, the app contains a MainActivity as all apps do. In general, activities represent pages in Android which usually fill up the whole screen. The MainActivity is launched when starting the app and its layout is a ConstraintLayout in this case. ConstraintLayouts are an easy way to structure user interface as desired.

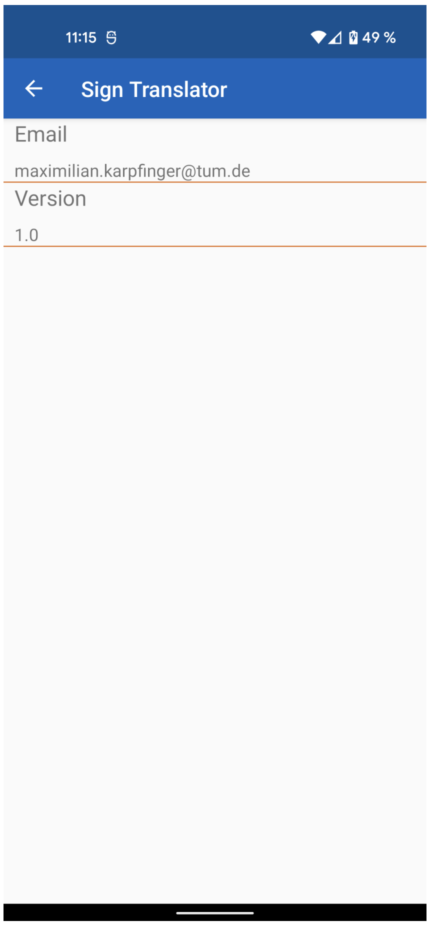


Please grant camera permissions.

Figure 3.20.: Screenshot of the MainActvitiy of the app.

As the left screenshot of Figure 3.20 shows, there are four different parts in this activity. Beginning with the top bar, it contains the app's name on the left and an action menu on the right represented by three dots below one another. From there the second activity can be launched which is the SettingsActivity. The third part is the orange FloatingActionButton with a camera icon that illustrates a switch on it. With this button the user is able to change between back and front camera, however, the back camera is the default one. The FloatingActionButton is utilized, because it can live over another view, which in this case is the FrameLayout in which the output video from the pipeline is visualized. Finally, the black part of the screenshot is the FrameLayout which shows the pipeline's output as already mentioned. Here, the translated gestures are visualized in addition to what the camera is seeing. On the right screenshot, a message can be seen which asks the user to grant camera permissions. However, this screen can only be seen if the user does not grant the permission the first time the app is started. If denied, the app is not able to use the camera at all. After restarting, the user can choose again to grant this essential permission or by editing the permissions of the app in their phone's setting permission can be given. The previously mentioned SettingsActivity can be viewed in Figure 3.21. For this activity's layout the LinearLayout was chosen because it only needs to represent a simple list of items and LinearLayouts are preferable for simple layouts.

As this activity has the MainActivity as its parent, there is a button with a left arrow which lets the user navigate back to the main page. The content of this activity is a list of items which contain information about the app. Additionally, there could be items which allow the user to parameterize the app, but more on that in the last chapter of this thesis. The items of the list have a LinearLayout as their layout with two TextView objects that represent the title and the description of the item and are separated by a line. More items can be appended by adding values to the arrays storing the information in the SettingsActivity.



Email

maximilian.karpfinger@tum.de

Version

1.0

Figure 3.21.: The SettingsActivity's user interface.

3.6. Testing & Results

At first, this section reviews the performance of the app. Furthermore, the trained networks are evaluated on a test set to check if they meet their achieved accuracy. At last, the app is tested in different scenarios and checked if it is able to perform sign language translation.

3.6.1. Performance evaluation

On mobile devices, real-time applications need to be efficient because the hardware usually is slow. In order to measure the performance of the calculators used in the translation pipeline, MediaPipe's profiling and tracing tools are used. For a first experiment a Google Pixel 4a is used on a 60 seconds long video from the DGS-Korpus.

name	counter	completed	dropped	fps	time	latency
AnnotationOverlay	648	647	1	4.856	3.568	202.372
ConcatenateRenderDataVector	648	648	0	3.113	0.021	321.178
FlowLimiter	2439	648	1791	4.806	89.795	118.261
LandmarksToTensor	376	376	0	2.952	0.021	338.724
MyClassificationsToRenderData	376	376	0	2.948	0.026	339.19
MyConcatenateNormalizedLandmarkList	648	648	0	3.117	0.125	320.669
MyMissingLandmarksToRenderData	272	272	0	3.376	0.037	296.135
SplitNormalizedLandmarkList	648	648	0	3.796	0.044	263.389
TfLiteConverter	376	376	0	2.951	0.026	338.8
TfLiteInference	376	376	0	2.949	0.233	338.859
TfLiteTensorsToClassification	376	376	0	2.948	0.018	339.144

Table 3.7.: App profiling statistics. Mobile device used is Google Pixel 4a and the translation model is the CNN designed on classifying the top 10 classes of gestures.

From Table 3.7 there are a few interesting insights on how the calculators of the translation part of the app are utilized. The total number of packages that went into the FlowLimiter calculator from the video's input stream is 2439. Out of that, 1791 were dropped. As explained previously, this calculator only forwards packages when the next calculator is ready and does not create waiting queue or similar buffering strategies. In total, approximately 73.4 % of all packages were dropped. Another remarkable fact is that the FlowLimiter has the highest average time spent within the calculator in milliseconds and thereby having a total of 38.3% total time spent in the whole pipeline. After the FlowLimiter calculator, numerous calculators from the landmark detection subgraphs of hands, face and pose are actually used. However, these are left out from the table for visibility reasons. After these three subgraphs the SplitNormalized calculator is next in the pipeline. As it can be seen, it receives all the packages which went from the FlowLimiter through the detection subgraphs and also forwards the same number of packages. Followed by the calculators for combining and converting landmarks and also by applying inference on the translation model and converting the results back to render data, all these calculators share a small average time spent within themselves and also do not drop any packages. None of them have a noticeable amount of time spent. However, as the TfLiteInference calculator uses a small model in this experiment it would be important to check if the pipeline behaves the same for a bigger and more complex model. The results from the profiler also show that all calculators are run on seven threads besides the AnnotationOverlay calculator.

3.6.2. Test Set Evaluation on all Translation Models

For the test set evaluation at first a test set is created. As previously elaborated, the videos for the test set are selected by randomly sampling participants and excluding them from the other data. Then, in the same way as for the training data, a test set is created. Additionally, all samples which do not have a label of the top 100 classes training set are removed. For the GNN the data also is transformed to graphs.

Model	Training	Validation	Test
CNN	0.44	0.34	0.32
PointNet	0.23	0.19	0.18
GNN	0.12	0.08	0.07

Table 3.8.: Accuracy of all models on all sets. The low accuracy can be explained because of multiple reasons. For example, using only 65 landmarks from pose and hands of the final frame of a sign is not enough and more data is needed like the previous frames and considering landmarks from the face. Furthermore, for the amount of classes more sophisticated models need to be used. More reasons are discussed in the Conclusions section.

Resulting test set accuracy is lower than the validation accuracy for all three models. However, this is expected, because the data is completely unknown to the model. Again, the performance compared in between the models does not change as the convolutional network still achieves the highest accuracy, followed by PointNet and then the graph neural network.

3.6.3. Different scenarios

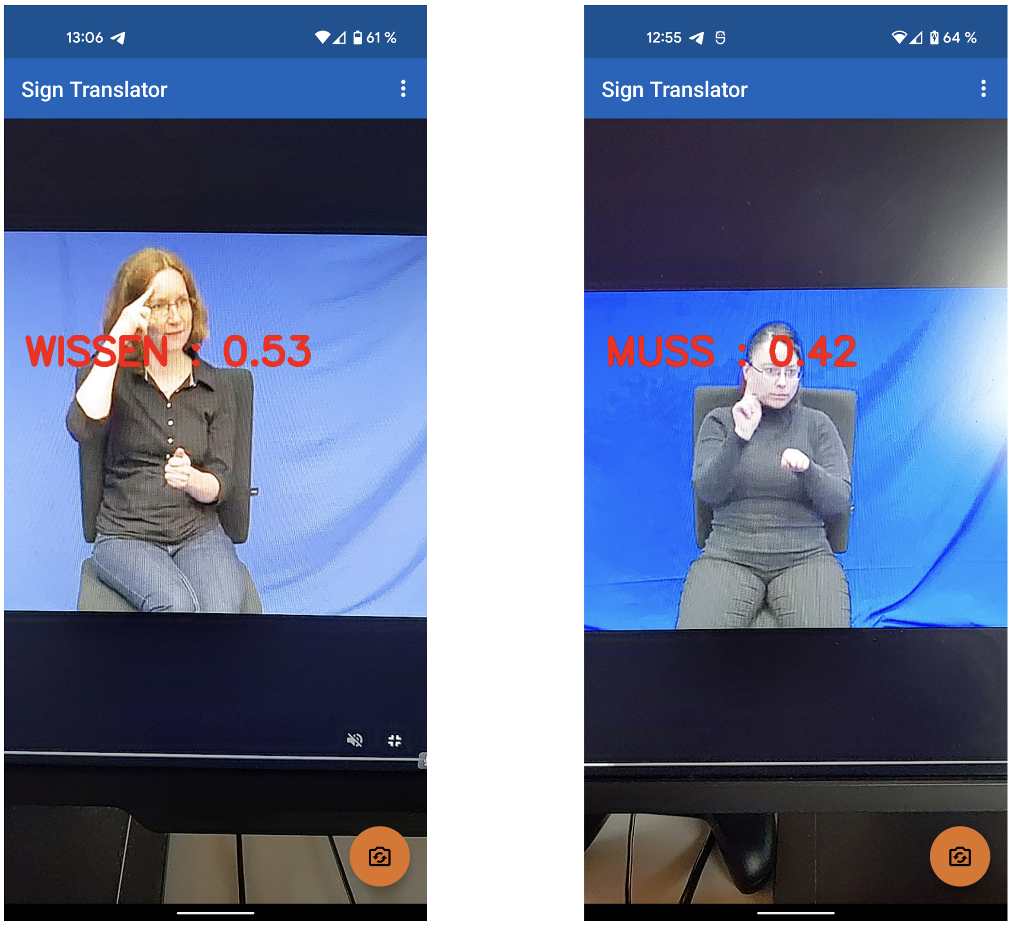
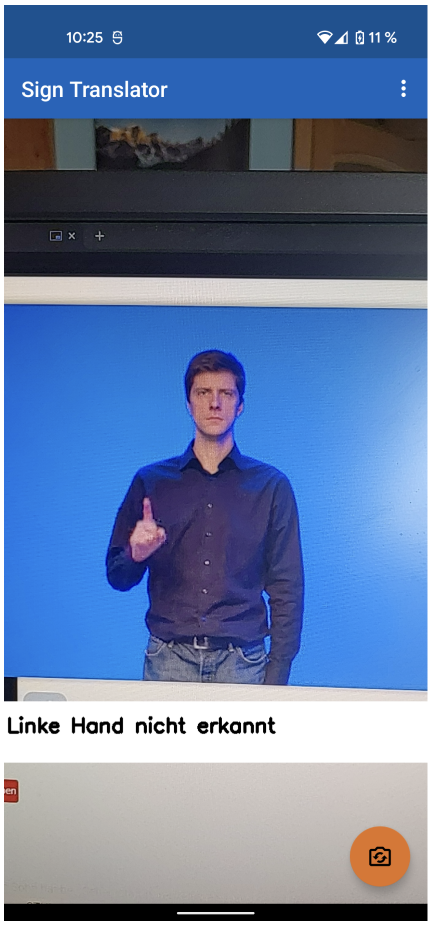


Figure 3.22.: Screenshot of gestures from DGS-Korpus videos being translated.

Figure 3.22 shows screenshots of the app translating gestures from videos of the DGS Korpus. In this case, the convolutional network is used which is trained on the top 10 classes. In the left frame the person is signing the gesture for TO-KNOW-OR-KNOWLEDGE which is WISSEN in German. This gesture's characteristic is the index finger pointing to the head. The app successfully recognizes this and annotates the output video with the string. Additionally, the network's score for the prediction is also seen next to the translation. On the right frame a different sign can be seen, this time the sign MUST which is MUSS in German. Again, both the word and the network's prediction score are rendered on top of the output video.



Linke Hand nicht erkannt

Figure 3.23.: Info message displayed by the app when landmarks are missing from a frame.

As previously discussed, the app needs to handle the case when the pose estimators detect landmarks, but not all do in a certain frame. For example, when one hand is not visible. This case can be seen in Figure 3.23 where a person is actually signing the gesture for MUST, however, their left hand is cut off in the image. Thus, the app does not try to translate the gesture, but rather informs the user that the landmarks from the left hand cannot be detected. This would be the same, if for instance the right hand is hidden or even both hands are not visible.

In another test scenario the app was tested outside at night. The experiment showed that the app is not able to detect the necessary landmarks when there is not enough light for the camera. Furthermore, the holistic tracking app by MediaPipe was tested at the same time to confirm that the pose estimation models do not detect the landmarks.

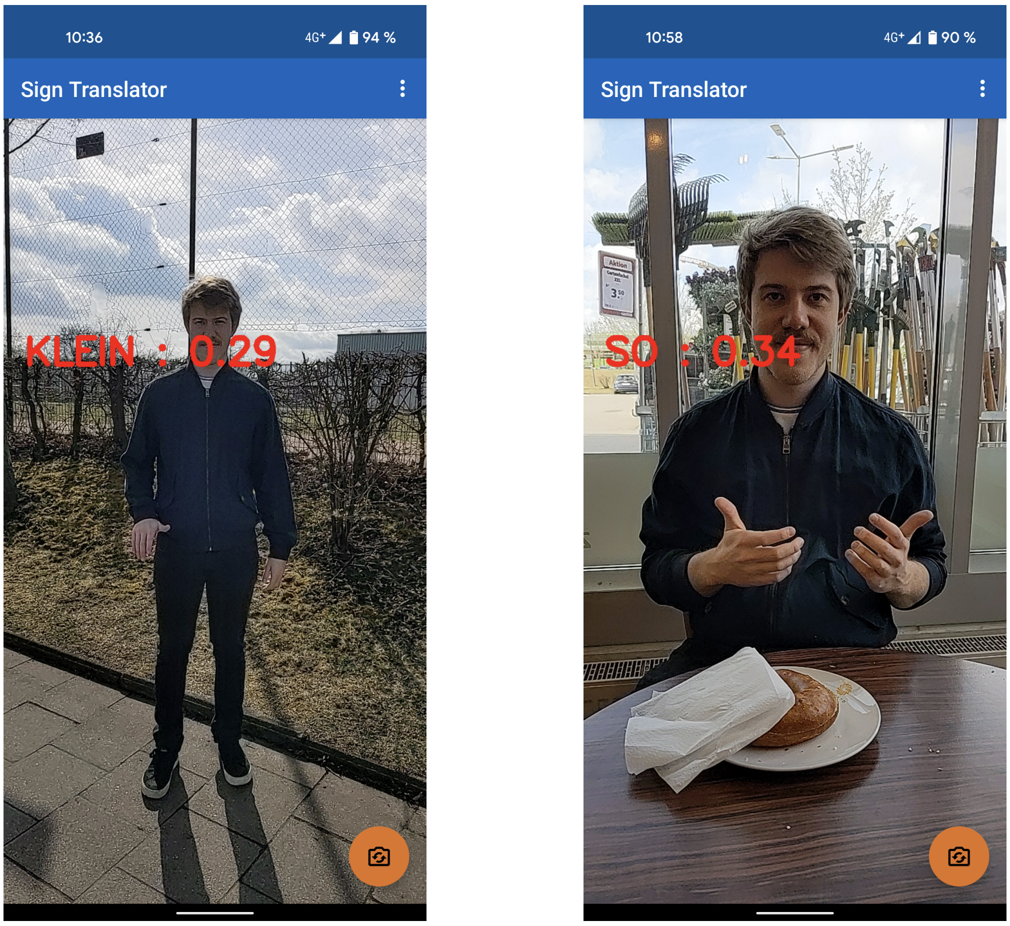


Figure 3.24.: Screenshot of gestures being translated in a real world scenario.

Figure 3.24 shows that the app can work with more difficult backgrounds than a blue wall. The left image was taken outside during sunny weather and the app detects the sign KLEIN which is German for small. On the right image, the app was used in a coffee shop. Here, the app detects the sign SO which is *like this* in English. Additionally, the CNN which was trained on the top 100 most used signs was used for these pictures.

4. Conclusions

In this final chapter of the thesis, the study on sign language translation on mobile devices is first summarized, followed by a discussion on the findings and then an outlook on future work which could extend on the implemented pipeline is given.

As a first step, the functionality and rules of sign language were examined. Certain classes of signs are elicited, for example indexing or productive signs in comparison to normal signs. The differences between manual and non-manual means of expression are explained. On one hand are the possibilities of using hands and arms and on the other hand a signer can support their message by adding non-manual means of expression, for example by using facial expressions. In the second step, architectures of neural networks that can be used for the translation of sign language gestures were investigated. After reading into the literature of the mentioned topics, the translation pipeline had to be implemented. For that the decision to settle with the MediaPipe framework was made. With MediaPipe it is necessary to create a pipeline of calculators from an input video stream to an output video stream. Because MediaPipe already has a complete module that does pose estimation and provides landmarks of face, hands and pose. The module was used and a new pipeline was created from it. Summarizing, the new pipeline had to deal with combining the provided landmarks and then converting these to tensors, so that they can be fed into a TensorflowLite model. Finally, the results from the translation model had to be transformed into render data which can be displayed and seen on the mobile phone. For the training of the models, data from the DGS-Korpus was used. They provide video material of people speaking German sign language, which is annotated with the meaning of each gesture. Three different architectures were tried. First, a small convolutional network which surprisingly achieved the best results. Second, PointNet was also able to learn to recognize signs. Third, a graph neural network was trained which achieved the lowest accuracy of the networks. The pipeline also was evaluated in terms of computational costs for each calculator to show that the app is able to perform gesture translation in real time. Concluding, the implementation of an app, that is able to do sign language translation, was done successfully, however, the road to achieve results that are applicable in the real world is long.

Now, the study of this thesis is discussed. Starting with the pipeline of the app, it was decided to use the framework MediaPipe by Google. The main reason for that was that the framework offers a broad variety of modules for pose estimation and is also able to build apps for mobile devices which is important for this thesis. Another possibility is OpenPose, however, the support for Android devices is scarce in comparison to MediaPipe. Building the pose estimation modules from scratch and thereby not needing a framework to begin with is not a reasonable approach for such a thesis, due to the limited time available. Going on to the implementation of the pipeline, it can be said that the task was done successfully. Every required part works as intended, from combining existing landmarks to converting them to tensors and applying them on a translation model. Resulting classification is rendered onto the display of the device as desired. Making use of existing calculators for converting landmarks instead of creating an own implementation seems reasonable to reduce implementation costs. An improvement to the pipeline could be to create a subgraph which includes all the introduced calculators. By doing so, the subgraph could be used in other models easily, if desired. Again, the pipeline itself was not created from scratch but rather is an adaption from the existing holistic tracking module by MediaPipe whereas all the visualization parts for landmarks and joints are removed and instead the part for translating gestures from landmarks is added. While the app technically works, it is not yet applicable in the real world. This is because of the lack of a sophisticated translating model which is able to correctly translate gestures. Coming back to the key concepts of sign languages, it can be said that the models used in this study only looked at three of the four categories from the manual means of expression. These are the shape, orientation and the position of the hand. Still, these features lack some information as they are only provided in 2D. The missing aspect of the manual means of expression is the movement of the hand. Sequences of landmarks would need to be considered for this instead of only the final frame of a gesture. From the non-manual means of expression none of the four categories are considered. This is because the landmarks from the face are not included. Even if such a model would be implemented, there is still a lot of work which needs to be done because translating single signs is not enough. As sign languages are different to spoken languages another model which maps a sequence of translated signs to spoken language sentences is required. With that, one could speak of real time sign language translation. In this study, three models with different architectures were tested. Beginning with the convolutional network, which surprisingly had the best results on the validation set. It achieved an accuracy up to 33% on the top 100 classes set. Continuing with PointNet, which at first seems like the superior model as it focuses on data points instead of images which is the normal use case of convolutional networks. However, as the training showed, it was not quite able to achieve the same results, peaking at approximately 19% on the validation set. Reasons for that could be that PointNet was designed to work on 3D points instead of the available 2D landmarks. In the future, these landmarks are likely to be present in 3D as well, so it might be interesting to see how PointNet performs in that case. Another cause for the lower accuracy is that PointNet actually was meant to work on data points from point clouds. This means that the points could be sampled for example from a plain. Landmarks, in contrast to that, are not from 3D objects where one just cannot sample points from. Finally, the worst performing model was the graph neural network. While it might not achieve the same results as the other two networks, it probably fits the problem case the best. Because the landmarks can form graph structures, for example fingers from the hand, it seems intuitive to work with GNNs. A more thought-out model is needed. For example, techniques to prevent over-fitting such as dropout could be useful. More and different kinds of layers can help as well. An idea here would be to have a hierarchical structure which at first looks at all landmarks and then reduces the amount of nodes in the later layers. Not surprisingly, the results from the test set reflect the findings from the training of the network, as all models almost achieve their validation accuracy. This is desirable, because then the models prove to generalize and thus being able to correctly predict on unseen data. Reviewing the results from the profiling and tracing of the app's performance, the app does not require too high computational costs. All introduced calculators have a low average time spent per package in comparison to the existing calculators from the holistic tracking module. The TfLiteInference calculator has the most time spent of all the new calculators. Because the profiling was done with a small translation network it would be interesting to see how much the time spent increases when using a larger model. The app was tested in different scenarios. Beginning with the first scenarios which were videos from the DGS-Korpus, it can be said that the app is able to detect landmarks and thereby also use the landmarks to predict a translation. However, when there is not enough light, the app does not work as the landmarks from the pose cannot be found. Because of that, it would be important to have functionality to turn the flashlight on. As seen in Figure 3.24 the app also works in more realistic scenarios. Instead of using a blue wall as background, the app is tested in the real world. With good lighting the app is able to do its job outside in nature or even in a coffee shop. However, when the lighting was very bright due to sunny weather, the app sometimes struggles to detect all landmarks and therefore shows the message that some of the landmarks are missing.

There is a lot of work which can build up on this project. One huge topic is improving the translation of sign language. The approach used in this thesis for the translation of signs is not sophisticated enough for an application used in the real world. For example, for an appropriate translation, one would need three dimensional landmarks coordinates instead of the existing two dimensional ones. With these the usage of the space around a signing person could be understood more clearly. Furthermore, the network would need to be trained on subtypes instead of types. Another approach to this could be to train another network that translates from types to subtypes. As real world signs do not only depend on their final frame, it is also required to utilize some sort of network that can take sequences of frames as input. Using more frames per sign will most probably enhance the quality of the translation. A future project should also take the landmarks from the face into consideration and especially have a look at mouthings. For instance, these could be separately detected by a network that was trained on mouthings. Together with another model that takes the rest of the landmarks from pose and hands as input, the translation could be improved. Changing the pipeline would be necessary for that. However, it can be done without great effort, as the pipeline can be easily adapted and more calculators can be added to it. After all, this would still not be enough for a complete translation of sign languages as only single signs will be translated. Another model which translates sequences of detected signs to spoken German language would be required. This seems to be a very complex task on its own, as the German sign language differs from German spoken language in terms of grammar and vocabulary and cannot be translated word by word, as it is done in other NLP tasks. There are some features which could be added to the current app. For example, an activity could be added in which the user first selects a translation model which then will be loaded by the graph. Thereby, it would allow users to switch between multiple sign languages, if a trained network exists for that given language. The settings menu could be improved by offering more parameterization possibilities. Here, parameters for displaying text could be set or additional options for calculators, like a minimum threshold for classifications to be displayed. One feature which would improve usability could be a button which allows the user to switch off/on the flashlight. This can help when the app cannot detect landmarks due to bad lightning. For example, the implemented FloatingActionButton could be expanded and then offer the functionality for switching between front and back camera and additionally turning off/on the flashlight.

List of Figures

List of Tables

List of Listings

Bibliography

[1] D. Gehörlosen-Bund. (2022). "Deutsche gebärdensprache," [Online]. Available: https://www.gehoerlosen-bund.de/faq/deutsche%5C%20geb%C3%A4rdensprache%5C%20(dgs) (visited on 02/02/2022).

[2] D. Gehörlosen-Bund. (2022). "Deutsche gebärdensprache," [Online]. Available: https://www.gehoerlosen-bund.de/faq/geh%C3%B6rlosigkeit (visited on 02/02/2022).

[3] L. Marcel, "Avatare zur darstellung von gebärdensprache," Masterarbeit, Brandenburgische Technische Universität Cottbus, 2012.

[4] J. Stokoe William C., "Sign Language Structure: An Outline of the Visual Communication Systems of the American Deaf," *The Journal of Deaf Studies and Deaf Education*, vol. 10, no. 1, pp. 3–37, Jan. 2005, ISSN: 1081-4159. DOI: 10.1093/deafed/eni001. eprint: https://academic.oup.com/jdsde/article-pdf/10/1/3/1034248/eni001.pdf.

[5] C. Papaspyrou, *Grammatik der Deutschen Gebärdensprache aus der Sicht gehörloser Fachleute*. Signum Verlag, 2008.

[6] MediaPipe. (2022). "Live ml anywhere," [Online]. Available: https://mediapipe.dev/ (visited on 12/25/2021).

[7] Google. (2022). "Google assistant," [Online]. Available: https://assistant.google.com/intl/de_de/ (visited on 02/09/2022).

[8] Apple. (2022). "Siri," [Online]. Available: https://www.apple.com/de/siri/ (visited on 02/09/2022).

[9] W. für Unterstützte Kommunikation UG. (2022). "Eis - eine inklusive sprachlernapp," [Online]. Available: https://www.eis-app.de/ (visited on 02/09/2022).

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012.

[11] R. Q. Charles, H. Su, M. Kaichun, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 77–85. DOI: 10.1109/CVPR.2017.16.

[12] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," *CoRR*, vol. abs/1812.08434, 2018. arXiv: 1812.08434.

[13] R. Khan, "Sign language recognition from a webcam video stream," Masterarbeit, Technische Universität München, 2022.

[14] A. C. Duarte, S. Palaskar, D. Ghadiyaram, K. DeHaan, F. Metze, J. Torres, and X. Giró-i-Nieto, "How2sign: A large-scale multimodal dataset for continuous american sign language," *CoRR*, vol. abs/2008.08143, 2020. arXiv: `2008.08143`.

[15] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh, "Openpose: Realtime multi-person 2d pose estimation using part affinity fields," *CoRR*, vol. abs/1812.08008, 2018. arXiv: `1812.08008`.

[16] A. Khanal, "Hand pose estimation and gesture detection from webcam images," Masterarbeit, Technische Universität München, 2021.

[17] N. Kanopoulos, N. Vasanthavada, and R. L. Baker, "Design of an image edge detection filter using the sobel operator," *IEEE Journal of solid-state circuits*, vol. 23, no. 2, pp. 358–367, 1988.

[18] C. Kellinger, "Data preprocessing for sign language detection with machine learning models," Bachelorarbeit, Technische Universität München, 2021.

[19] S. König, G. Langer, T. Hanke, R. Konrad, D. Blanck, S. Goldschmidt, I. Hofmann, S.-E. Hong, O. Jeziorski, L. König, R. Nishio, C. Rathmann, S. Matthes, and S. Worseck, *Handbuch für Kontaktpersonen Teil I: Projekt, Werbung, Informantensuche, Raumsuche*, Oct. 2020. DOI: `10.25592/uhhfdm.1893`.

[20] G. Langer, S. König, T. Hanke, R. Konrad, D. Blanck, S. Goldschmidt, I. Hofmann, S.-E. Hong, O. Jeziorski, L. König, R. Nishio, C. Rathmann, S. Matthes, and S. Worseck, *Handbuch für Kontaktpersonen Teil II: Erhebung, Einverständniserklärung*, Oct. 2020. DOI: `10.25592/uhhfdm.1895`.

[21] T. Hanke, S.-E. Hong, S. König, R. Konrad, G. Langer, S. Matthes, R. Nishio, and A. Regen, "Segmentierung / segmentation," German and English, DGS-Korpus project, IDGS, Hamburg University, Hamburg, Germany, Project Note AP03-2010-01, version 3, Mar. 2019. DOI: `10.25592/uhhfdm.817`.

[22] R. Konrad, T. Hanke, G. Langer, S. König, L. König, R. Nishio, and A. Regen, "Öffentliches DGS-Korpus: Annotationskonventionen / Public DGS Corpus: Annotation conventions," German and English, DGS-Korpus project, IDGS, Hamburg University, Hamburg, Germany, Project Note AP03-2018-01, version 3, Sep. 2020. DOI: `10.25592/uhhfdm.822`.

[23] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C.-L. Chang, M. Yong, J. Lee, W.-T. Chang, W. Hua, M. Georg, and M. Grundmann, "Mediapipe: A framework for perceiving and processing reality," in *Third Workshop on Computer Vision for AR/VR at IEEE Computer Vision and Pattern Recognition (CVPR) 2019*, 2019.

[24] G. Developers. (2022). "Protocol buffers," [Online]. Available: `https://developers.google.com/protocol-buffers/docs/cpptutorial` (visited on 02/07/2022).

[25] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C.-L. Chang, M. G. Yong, J. Lee, W.-T. Chang, W. Hua, M. Georg, and M. Grundmann, *Mediapipe: A framework for building perception pipelines*, 2019. arXiv: `1906.08172 [cs.DC]`.

[26] Bazel. (2022). "Build and test software of any size, quickly and reliably," [Online]. Available: `https://bazel.build/` (visited on 12/25/2021).

[27] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[28] MediaPipe. (2022). "Mediapipe holistic — simultaneous face, hand and pose prediction, on device," [Online]. Available: `https://ai.googleblog.com/2020/12/mediapipe-holistic-simultaneous-face.html` (visited on 12/25/2021).

[29] MediaPipe. (2022). "Mediapipe pose," [Online]. Available: `https://google.github.io/mediapipe/solutions/pose.html` (visited on 02/15/2022).

[30] MediaPipe. (2022). "Mediapipe hands," [Online]. Available: `https://google.github.io/mediapipe/solutions/hands.html` (visited on 02/15/2022).

[31] MediaPipe. (2022). "Mediapipe face mesh," [Online]. Available: `https://google.github.io/mediapipe/solutions/face_mesh.html` (visited on 02/16/2022).

[32] Spektral. (2022). "Spektral," [Online]. Available: `https://graphneural.network/` (visited on 02/28/2022).

[33] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: `1412.6980 [cs.LG]`.

[34] MediaPipe. (2022). "Tracing and profiling," [Online]. Available: `https://google.github.io/mediapipe/tools/tracing_and_profiling.html` (visited on 03/08/2022).

[35] Apache. (2022). "Apache license, version 2.0," [Online]. Available: `https://www.apache.org/licenses/LICENSE-2.0` (visited on 03/19/2022).

A. Appendix

ber-14
ber-54
fra-43
fra-63
goe-06
goe-35
hb-26
hb-10
hh-04
hh-13
koe-06
koe-55
lei-41
lei-72
mst-12
mst-19
mue-36
mue-11
mvp-17
mvp-29
nue-08
nue-01
sh-04
sh-05
stu-61
stu-04

Table A.1.: Randomly sampled participants for the test set.

Code	Landmarks not detected
0	None
1	Face
2	Left hand
3	Face and left hand
4	Right hand
5	Face and right hand
6	Both hands
7	Face and both hands
8	Pose
9	Pose and face
10	Pose and left hand
11	Pose, left hand and face
12	Pose and right hand
13	Pose, right hand and face
14	Pose, both hands
15	All

Table A.2.: Signal codes of missing landmarks.

$GEST-AUFMERKSAMKEIT	BEISPIEL	VERGANGENHEIT	ZUSAMMEN	KLAR
WIE	SEHEN-AUF	$INDEX	$GEST-NM	$GEST
TITEL-ÜBERSCHRIFT	ZEIT	MUSS	GEHÖREN	ICH
GEBÄRDEN	STIMMT	GLEICH	HAUPT	$NUM-EINER
HÖREND	TAUB	DANACH	KANN	DA
FÜR	$PROD	ABLAUF	VIEL	EIGEN
GEFÜHL	$GEST-ABWINKEN	ARBEITEN	ANDERS	AUF-PERSON
GEBEN	AB	RICHTUNG	MEHR	KÖRPER
UNGEFÄHR	NASAL	JAHR	FREI	RUND
$GEST-RUHIG-BLEIBEN	BEREICH	INTERESSE	NEIN	ACHTUNG
GUT	SPRACHE	PERSON	SAGEN	KEIN
WAS	WIEDER	JA	KLEIN	GRUPPE
SCHREIBEN	BIS	$NUM-ORD	ORT	AUSSEHEN-GESICHT
DU	UNTER	KOMMEN	HIER-JETZT	$NUM-ZEHNER
BEDEUTUNG	ENDE	ALLE	$ORAL	AUSSEN
VIERECK	SO	$LIST1	FALTE-WANGE	SCHNEIDEN
WEG-VERLIEREN	KREUZ	$GEST-OFF	UNTERSCHIED	$GEST-ÜBERLEGEN
$GEST-NM-KOPFSCHÜTTELN	OFT	SEHR	LASSEN	SEHEN
ABER	$ALPHA	PICKELHAUBE	UM	GELD
$GEST-ICH-WEISS-NICHT	KOMMA	$GEST-NM-KOPFNICKEN	WISSEN	BESCHEID

Table A.3.: Classes of the top 100 classes data set.