UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

2D to 3D body pose estimation for sign language with Deep Learning

Degree Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Pol Pérez Granero

In partial fulfillment
of the requirements for the degree in
Enginyeria de Tecnologies i Serveis de Telecomunicació **ENGINEERING**

Advisors: Xavier Giró i Nieto (UPC), Kevin McGuinness (DCU)
Barcelona, June 2020

Acknowledgements

I would like to express my deep appreciation to my supervisors Dr Kevin McGuinness and Dr Xavier Giró i Nieto for their guidance, enthusiasm and commitment to this project. Thanks are also to PhD student Amanda Duarte for supporting this work to date and providing the dataset which made possible to carry out the project. Finally, I would like to express my gratitude to PhD student at DCU Enric Moreu for helping out with some technical difficulties.

Contents

List of Figures

List of Tables

Abstract

This project aims at leveraging the challenge of using 3D poses for Sign Language translation or animation by transforming 2D pose datasets into 3D ones. The goal is, using a 3D dataset of American Sign Language, to train a deep neural network that will predict the depth coordinates of the skeleton keypoints from 2D coordinates. Specifically, it will be explored a Long Short-Term Memory network, an architecture broadly used for sequence to sequence tasks.

The conclusions extracted on this report are that despite some of the results being good enough to be used for actual 3D SL annotation, the majority of them lack the precision to do so, and they are too variant with respect to the dataset split. It is also concluded that the solutions approached here could be improved by adding some regularization methods, more powerful hardware to run better experiments, and new input features such as keypoint visibility.

Revision history and approval record

Revision	Date	Purpose
0	17/06/2020	Document creation
1	27/06/2020	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Pol Pérez Granero	polpegra@gmail.com
Xavier Giró i Nieto	xavier.giro@upc.edu
Kevin McGuinness	kevin.mcguinness@dcu.ie

Written by:		Reviewed and approved by:	
Date	22/06/2020	Date	28/06/2020
Name	Pol Pérez	Name	Xavier Giró
Position	Project Author	Position	Project Supervisor

Chapter 1

Introduction

1.1 Motivation

All over the world, there are an estimated amount of 466 million people that are deaf or hard-of-hearing, whose primary means of communication are Sign Languages. The communication between deaf and not-deaf people is sometimes challenging because they usually use different modalities for communication: visual (sign language) or acoustic (spoken language). A machine translation tool between the two types of languages would improve the social interaction or access to digital content, among other applications.

Machine translation nowadays is based on deep learning techniques that require large datasets to be trained with. So, a neural machine translation system between speech and sign language may require a large parallel corpus of speech and videos depicting human poses. These poses can be represented with 3D skeletons that would encode some relevant *keypoints* of the human body for sign language. These 3D skeletons may be used to animate synthetic avatars for speech to signs translation, but may also facilitate the translation task in the signs to speech direction.

However, collecting large datasets is challenging in its most basic format of standard 2D video as it requires a recording studio with controlled conditions such as a solid background or fixed camera poses. The difficulty of such recordings grows exponentially when considering a dataset with 3D poses, which actually translates into a multi-camera set up in a very unique recording set up.

This project aims at leveraging the challenge of using 3D poses for sign language translation or animation by transforming 2D datasets into 3D ones. In particular, the proposed models exploit a smaller scale 3D dataset of American Sign Language recordings together with the automatic pseudo-labels of 3D skeletons provided by an existing multi-camera set up. The goal is, using this dataset, to train a deep neural network that will predict the depth coordinates from the video stream coming from one single predefined camera of the whole array. Specifically, it will be explored a Long Short-Term Memory (LSTM) network, a deep neural architecture broadly used for sequence to sequence tasks.

This work is the first one exploiting the 3D skeletons provided by the How2Sign dataset [5], a novel corpus developed at the Barcelona Supercomputing Center (BSC), Universitat Politecnica de Catalunya (UPC) and Carnegie Mellon University (CMU), with the support of Facebook, the European MSCA program and "la Caixa" Foundation. This dataset of American Sign Language (ASL) was recorded by the doctoral candidate Amanda Duarte in the singular Panoptic Studio [6] at CMU. This bachelor thesis is the first academic work that is benefiting from it.

1.2 Aims and objectives

In particular, as Dr Xavier Giró i Nieto pointed out, the corpus collected by A. Duarte is in a very clear need of predicting 3D *keypoints* from 2D video information, since it only has 2D and 3D skeleton pose estimations on a small subset of the How2Sign dataset [5] (section 4.1), recorded on the Panoptic Studio [7]. And this can be accomplished by means of Deep Learning.

The work, then, focuses in building this needed tool for predicting video 3D poses from 2D poses, so that it can be applied to annotate not only the rest of the How2Sign dataset, but any given 2D labelled sign language dataset. Specifically, the goal is: using the recordings from one of the 480 cameras of the Panoptic Studio, with the 2D pose *keypoints* already extracted with OpenPose [3], and its 3D keypoints already estimated, to build and train a Deep Learning model (see section 2.3.1) capable of predicting 3D skeletons (set of body *keypoints*) from 2D skeletons on any new unseen data.

Of course, since the idea is to use this tool for completing the sign language dataset, the skeletons it aims to estimate from 2D to 3D are that of sign language poses (it is a restricted domain).

1.3 Project Scope

This project, then, can be considered a complement for the much wider research on [5]. The solution comprises the appropriate manipulation of the dataset, for it to be suitable for training and evaluating the deep learning model, and the building, training, and evaluating processes for the aforementioned model.

Given that the project requirement was that given a sequence of JSON files (in the specified format) containing a set of 2D *keypoints* corresponding to a frame each, output a sequence of 3D points coordinates corresponding the 3D skeleton pose for each frame of the video; and that the specification demanded an acceptable performance, similar to that of other models in alike tasks; it was decided to implement a solution based on a neural network architecture known to work well in sequence-to-sequence tasks: the Long Short-Term Memory network [8].

Neural networks work by progressively extracting higher level features from raw input, even when they are really complex. But that also means that it must exist an actual relation between input features and targeted ones. Then, what does make think that this task can be accomplished by means of neural networks?

Part of the answer is that, despite the individual differences, it does exist a relation between the width, height (input features) and depth (target) of the human body -all humans have a similar general structure. Moreover, the reason to believe the network can predict the third coordinate from the other two is because it is a video and the LSTM uses the temporal information in it: having multiple frames in which the same *keypoints* are detected (thus, inherently having the distance between them) and how they're positions evolve, allow the model to infer the third coordinate.

Previous work has been done in 2D to 3D pose estimation [9, 10, **?**]. But they do not take into account finger and face keypoints, that really important on the context this project is framed. Also, a work was released this year that includes a very similar task to the one tackled on this thesis [11].

The code for building, training and evaluating the chosen architecture for this project is programmed in Python, and developed on Jupyter Notebooks (see 4.1).

It must be mentioned that the major constraint of this project is the time given to accomplish these objectives. And for that reason, this project has not achieved the desired results yet. However, because this project is conducted collaboratively both Dublin City University and Universitat Politècnica de Catlalunya, the finishing deadline for the project is not that of the submit of this document, but the 29th of June 2020. The development of the described solution in below chapters, continues until that date.

1.4 Document structure

Over the next chapters all the designing, implementation and results for this project will be explained.

First, the necessary technical background and related work already done by other authors will be presented. After that, there will be a brief listing of the hardware and software equipment used for developing the research, and the working environment will be described.

After those, the next chapter will explain the design of the created solutions. It will specify the task and accurately describe the Deep Learning models designs, as well as the data manipulation that must be done. Then, the succeeding chapter explain the actual implementation of those solutions, with precise description of the Jupyter Notebooks structure and the code developed in it.

Following the previous chapters, there is the report of the obtained results and some discussion about them. And finally, the document ends with a brief explanation of the ethics involved on the project, the budget calculation, and a concluding chapter with the author's assessment about the project and indications for future work.

1.5 Work Plan and Gantt Diagram

The development of this project was divided in several phases. First of all, an initial phase to learn the theory behind RNNs and LSTMs, and also to learn how to use Pytorch (python library for deep learning) to implement these kind of neural networks. Once achieved a good understanding of these technologies, it followed the ideation of the specific architecture to be used as a first approach for 3D pose estimation.

After these initial phases, the actual implementation was carried out. Once finished an approach, it was performed the solution testing, and depending on the results, either was tried to improve the implementation (if it was not as good as expected), was designed another approach (if the previous didn't work at all), or was continued to the following phase. Finally, the last one was the generation of comprehensive documentation for the whole project.

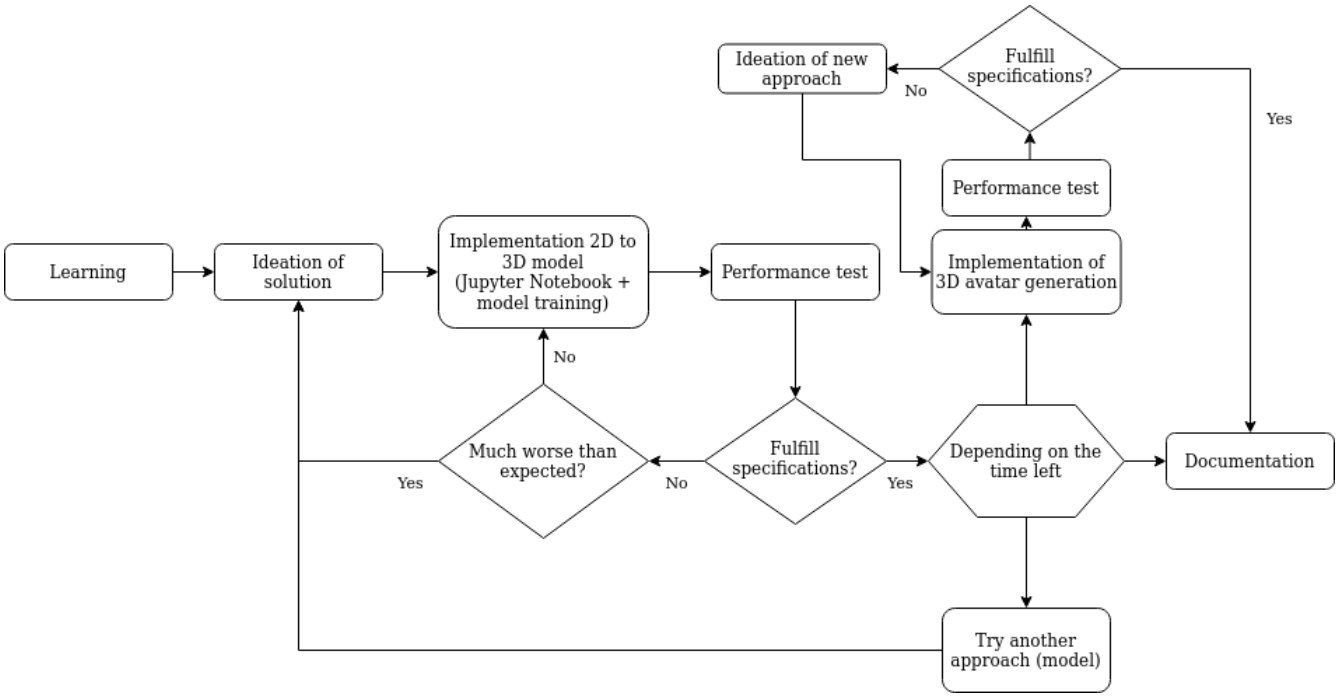


Figure 1.1: Work package breakdown.

There were the following work packages:

- Work Package 1: Previous knowledge acquirement.
- Work Package 2: Ideation of the first approach.
- Work Package 3: Implementation of the first approach.
- Work Package 4: Testing the first approach.
- Work Package 5: Ideation and implementation of the second approach.
- Work Package 6: Testing the second approach.

- Work Package 7: Animated avatar generation with the results achieved.
- Work Package 8: Final documentation.
- Work Package 9: Improve the first approach.

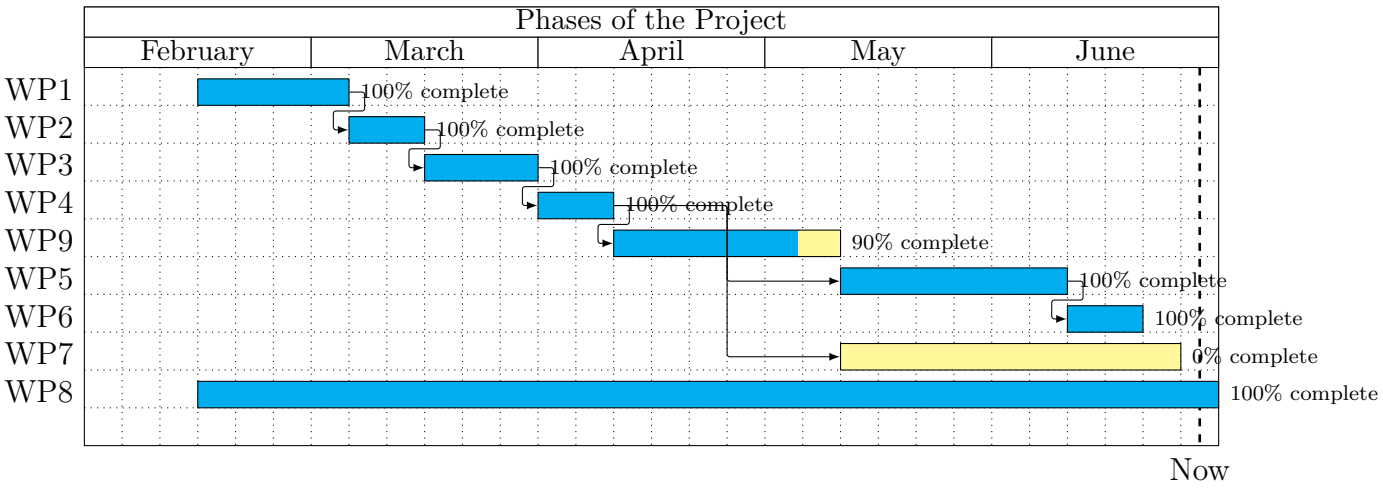


Figure 1.2: Gantt diagram of the project

So, the first milestone was after the Work Package 4 (present the first approach). The second was set after Work Package 6 (second approach results), and the final milestone was the actual finished project (at the end of June).

1.6 Issues and deviations

A project with these specifications, can be easily deviated from the original timeline due to the nature of training Deep Learning models. The training process is can be really long when it is difficult to find the appropriate configuration of parameters to accomplish results, and that can't be known beforehand. That was the case for this project, so every milestone was delayed a little bit.

Furthermore, work load was much more than the expected and Work Package 7 could not be conducted. Some issues that caused this were:

- the normalization process, it was changed several times before arriving to the final one explained in this report;
- several architecture sizes (different number of layers and parameters) were tested because the results weren't good enough at first, but those increased capacity networks didn't improve them;
- the visualization (plot) of results was more complex than expected, because some of the *keypoints* in the dataset were not indexed in the standard format;

- a linear regressor model was created to estimate the scaling factors needed to denormalize the data before plotting the results.

Chapter 2

Related Work

In this chapter the literature survey is presented, dividing each topic on a different subsection.

2.1 Sign Language

2.1.1 ASL linguistics

American Sign Language (ASL) is a natural language, with its specific lexicon, grammar and syntax. In the last decade, Stokoe [12] set the initial linguistic analysis that helped establish ASL as a language, breaking it down into five features: hands shape, location, orientation, movement, and relative position. Since all five components are important and a small change in just one of them might result into a different meaning for the sign, Sign Language translation is a difficult task. Furthermore, the translation between sign language and its corresponding spoken language cannot be done word-by-word, because they have different syntax.

2.1.2 Translation and generation

Translation from and to sign language is hard, but previous work has been done in that direction. Oong et al. [13] proposed to compose sentences by recognizing a single set of signs without taking into account the special linguistic syntax of Sign Language. There are others like Necati et al. [14], which was the first to formalize the sign language translation task in the Neural Machine Translation (NMT) framework, that approach it by using a sequence-to-sequence model. And following the same direction, Sangki et al. [15] used face hand and body keypoints for a sequence-to-sequence model to translate from sign language videos to Korean text.

All the above mentioned approaches focus on translation *from* sign language, but there are some other works that have explored translating from spoken language *to* sign language. The work by Sansegundo et al. [16], aims at translating Spanish speech into Spanish Sign Language. This approach, however, does not appear to being able to achieve reasonable

coverage, as its evaluation is limited to the comprehension of signs from the manual alphabet. More recently, some Stoll et al. [17] explored the translation from text sequence into sequence of skeletons that represent signs.

2.1.3 Datasets

One of the most important factors that has hampered the progress of automatic Sign Language Translation (SLT) is the absence of large annotated dataset. Existing ones appear segmented on either the letter, word or sentence. There is just one, used by [18], aside from How2Sign, that appears to have the speech modality that is needed for automatic SLT, but it only covers 370 phrases matched between English and British Sign Language.

An important factor for the lack of large scale, non-constrained datasets, is that the gathering and annotation of continuous sign language data is expensive and tedious. In this work it is used the How2Sign dataset [5], a novel dataset that not only includes the speech modality, but also is larger than other public datasets that can be used for text-to-sign language translation.It has been developed by the Barcelona Supercomputing Center (BSC), Universitat Politècnica de Catalunya (UPC) and Carnegie Mellon University (CMU), with the support of Facebook, the European MSCA program and "la Caixa" Foundation. This dataset of American Sign Language (ASL) was recorded by the doctoral candidate Amanda Duarte in the singular panoptic studio [6] at CMU. And this bachelor thesis is the first academic work that benefits from it.

2.2 Pose estimation

Different works [19, 20] have addressed 3D pose estimation for hands, but they have focused with still images and do not exploit the temporal correlations contained in video sequences. In particular, both Zimmermann et al. [21] Boukhayma et al. [20] centered most of contributions on estimating the 3D coordinates given *keypoint* detections over 2D images, as in our set up. Zimmermann et al. [21] proposed a three-stage pipeline that first segments the hand over the input image to later estimate its 2D *keypoints*. The final block estimates a 3D view of the hand skeleton depending on a set of camera parameters. Among other the model was tested on 30 static gestures taken from the RWTH German Fingerspelling Database [22] for the task of sign language recognition.

On the other hand, Boukhayma et al. [20] proposed a convolutional neural network that predicts the view, shape and and pose parameters of a 3D model of the hand. While the model can issue its predictions from an RGB image only, results improve significantly when 2D *keypoints* are provided.

Interestingly, they observed that training with weak supervision in the form of 2D joint annotations on datasets of images in the wild, in conjunction with full supervision in the form of 3D joint annotations on limited available datasets, allows a good generalization to 3D shape and pose predictions on images in the wild. That work is partially tested with poses from the New Zealand Sign Language (NZSL) Exercises of the Victoria University of Wellington [23].

The estimation of 3D human pose from video has actually been explored, as in the current state of the art work by Pavllo et al [24]. However, the human skeleton model considered did not contain the joints of the hands, so the available models or datasets are not suitable for sign language translation.

The works by Rayat et al. [9] and Lee et al. [10] are the most similar to this project, as they aim at predicting sequences of 3D *keypoints* from 2D *keypoints* with LSTMs. These works, however, did not consider the *keypoints* in the hands, which are the most important ones in sign language.

2.3 Machine Learning

Machine Learning is the study of a subset of artificial intelligence algorithms. Specifically, it encompasses those algorithms that can improve its performance at a defined task through experience. These are the kind of techniques that are being used for automatic Sign Language Translation.

In particular, a family of those methods that have been successful in a wide variety of areas, including translation and pose estimation, is called **Deep Learning**. And this project is conducted using Deep Learning algorithms.

2.3.1 Deep Learning algorithms

Deep Learning is a set of Machine Learning techniques that use Artificial Neural Networks (ANN or NN) [25, 26] technology to accomplish their tasks. They are based on the universal approximation theorem [27] and have proven to be very effective in signal processing, classification tasks and prediction tasks, among others.

Neural Networks consist in a collection of units called artificial neurons (sometimes perceptrons) that are interconnected forming layers (see figure 2.1). Each layer is capable of extracting higher level features from the previous layer. The likes of the problem this project focuses on, are framed within the supervised learning paradigm. This type of learning consists in learning a function that maps an input to an output based on example input-output pairs [28], i.e. an input object and the desired output value. The way supervised learning works with NNs is the following:

- the last layer of the network gives the output from the inference (whether it is a value or a tensor),
- a cost function -e.g. Mean Squared Error- is used to compute the loss of that inference,
- the cost function is back-propagated [29] through the network,
- every trainable parameter is updated using the corresponding back-propagated cost function in an optimizer algorithm (SGD, Adam, etc.).

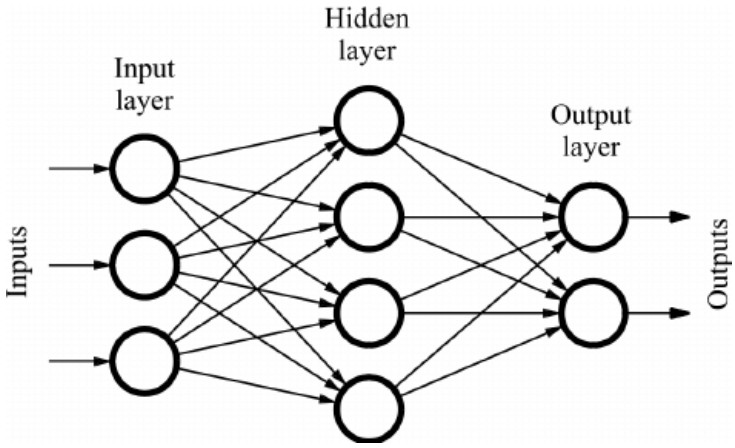


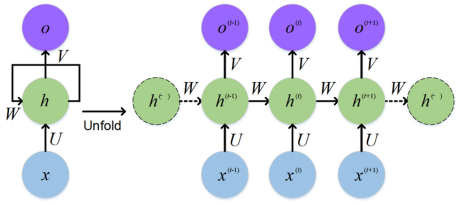
Figure 2.1: Feed Forward Neural Network (Source: Medium, 2019 [1]).

For the task this project is addressing, 2D to 3D pose estimation, we explored the results with the most popular architecture for the sequence processing, which are are Recurrent Neural Networks (RNN).

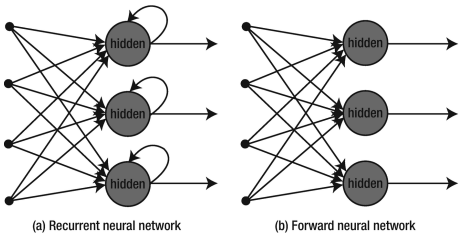
2.3.2 Recurrent Neural Networks for pose estimation

When in a Neural Network the connections between artificial neurons form a directed graph along a time sequence (as illustrated in Figure 2.2a), i.e. they receive feedback either from later layer nodes or from themselves in previous time steps, the network is called a Recurrent Neural Network (RNN) [30]. In these networks the cost function is not only propagated backwards in space, but also in time (Back-Propagation Through Time [31]).

RNNs are really useful when dealing with time series as well as any other kind of sequence. And since this work handles sequences of data (2D and 3D skeletons), RNNs are suitable for the task.



(a) Unfolded RNN (Source: W. Feng, 2017 [32]).

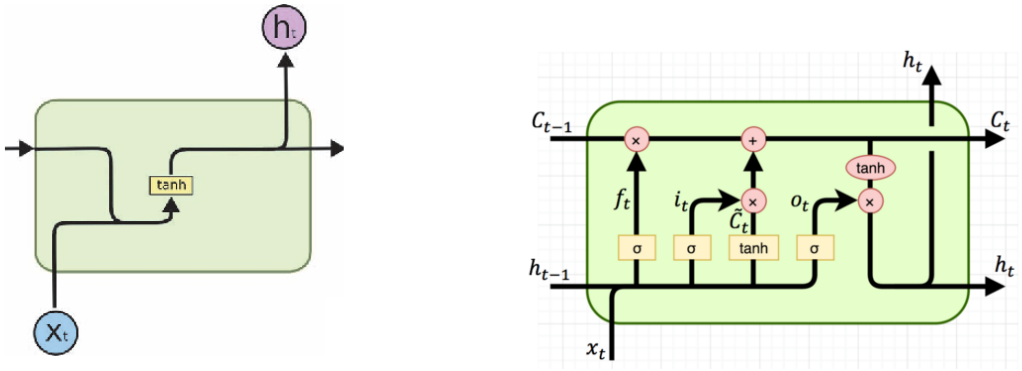


(b) RNN structure (Source: [33])

Figure 2.2: Recurrent Neural Network.

More precisely, there are RNN architectures that can control the stored state of their units, i.e. they allow the network to decide: which information from previous time-steps

is not useful anymore, which information should it take from the current time-step input, and just show the output whenever it is prepared for doing it. One of its implementations is the Long Short-Term Memory network (LSTM) [8].



(a) Simple RNN cell.

(b) LSTM cell.

Figure 2.3: RNN vs LSTM (Source: Medium, 2018 [2]).

Chapter 3

Methodology

3.1 Requirements

The How2Sign dataset [5] contains two parts, depending on the recording studio. 60 hours of sign language videos were recorded on a green studio equipped with 2 RGB cameras with depth sensors from 2 different views. And a much smaller subset (4 hours) of videos were collected in The Panoptic Studio [7], a system equipped with 480 VGA cameras, 31 HD cameras and 10 RGB-D sensors all synchronized; the cameras are placed all over a geodesic dome surface [1].

The RGB videos of the dataset recorded in the green background studio has already been manually cut in utterances, which are the basic training units when working with neural machine translation systems. Other researchers were working concurrently in the translation task which, when solved, it would translate spoken words into sequences of 2D skeletons. However, most existing solutions for avatar generation would require 3D skeletons.

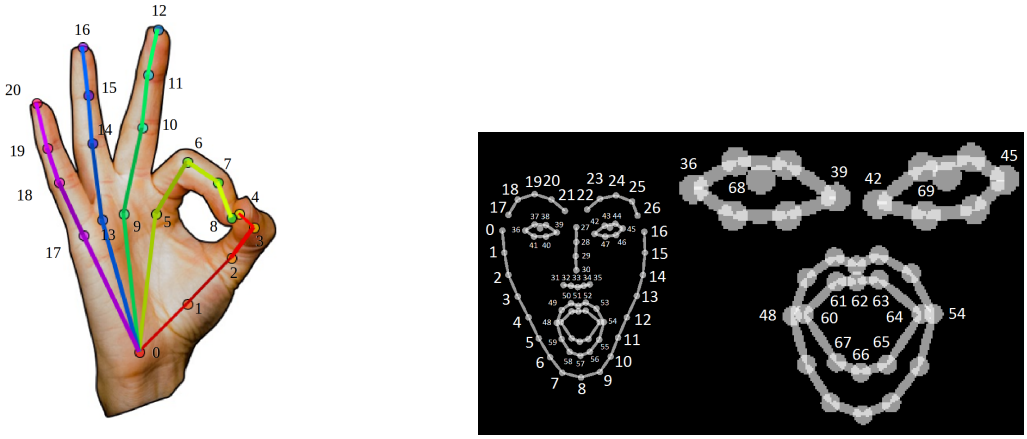
On the other hand, the videos collected at the Panoptic Studio were already analyzed with the human pose estimation tools to provide both 3D skeleton *keypoints* of the sequence, as well as the 2D *keypoints* from each camera view. These poses should be considered as pseudo-labels because they were generated by video analysis algorithms, not annotated manually.

The specific goal of this work is, using the available dataset collected in the Panoptic Studio, to predict the depth coordinates of the 2D skeletons, so that the resulting work could be used in the future to animate the avatars from the 2D skeletons produced by the machine translation engines.

[1] http://www.cs.cmu.edu/ hanbyulj/panoptic-studio/

3.2 Data specifications

The How2Sign Panoptic Studio sub-dataset is structured in folders, each corresponding to a recording. Every recording folder contains all the videos from the different cameras and the corresponding English transcription. But more importantly for this work, they also contain JSON files, one per frame of the recording, that hold the coordinates for every *keypoint* in every frame. Particularly, there is a JSON (frame) sequence per set of *keypoints* - face, hands and body- (see figures 3.1 and 3.2). That is, inside each recording folder, apart from the videos and the transcripts, there are 3 subfolders: one contains JSON files, each corresponding to a frame, that include the triplets of coordinates from the face *keypoints*, for their pertinent frames; another contains JSON files, each corresponding to a frame, that include the triplets of coordinates from the hands *keypoints*, for their pertinent frames; and the last one contains the same structure as the previous but with body *keypoints* coordinates.



(a) Hand keypoints order.

(b) Face keypoints order.

Figure 3.1: Face and hand keypoints (Source: OpenPose, 2018 [3]).

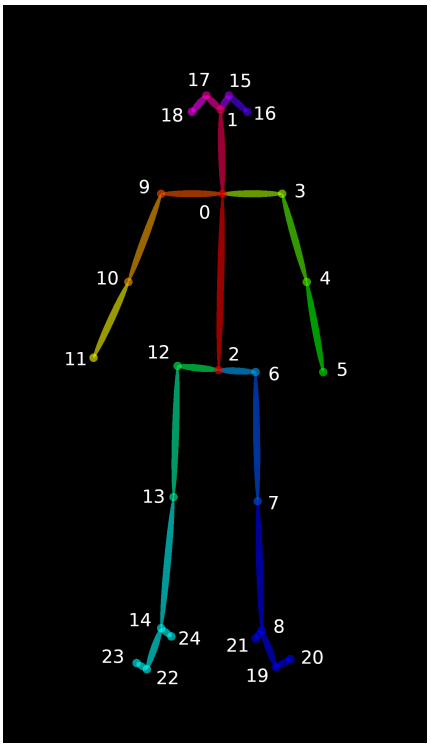


Figure 3.2: Body keypoints order in How2Sign dataset.

3.3 Regression or classification approach

The present task consists on building a system that takes a sequence of vectors as input, in which each vector contains the x and y coordinates of every *keypoint* in a given frame, and outputs another sequence of vectors, in which each vector contains the estimated z coordinate of every *keypoint* in the given frame. Since z coordinate is a continuous value that we are trying to predict from two other continuous values (x, y), this is a **regression** problem.

However, the LSTM architecture was applied in two different set ups which have been considered in the literature to predict sequences of the Z coordinate given its input sequence of XY coordinates. Despite that dealing with continuous values in the depth naturally results in a regression task, the generative nature of the problem has been addressed with success in the past by formulating it as a classification task, by partitioning the output space in bins. This was the case, for example, of the WaveNet model [34] for speech synthesis in which, despite being waveform defined as continuous values, the best results were obtained with a classification set up. This observation motivated a two-fold study considering both approaches.

3.4 Design of the first approach: Regression

For the given task of estimating 3D video poses from 2D coordinates, it is required a sequence-to-sequence model, and, as seen in 2.3.2, LSTM network is a good architecture for building this kind of models. But, in a Machine Learning project like this, another thing as important as the architecture design and the training process itself, is the previous parsing and manipulation of the data. Therefore, the pipeline followed by the code was:

- Collect the data from the JSON files and frame them in the appropriate format for posterior manipulation.
- Restructure the data so that they can be correctly fed into the LSTM model.
- Build the model's specific architecture.
- Train the model and fine-tune it.
- Run inference on test data.
- Interpret the results.
- Go over training again change the required parameters when the results weren't good enough.

3.4.1 Structuring the data

The first thing to do with the Panoptic Studio data after their collection, is to store them in tensors [35] represented as n-dimensional arrays. Tensors are an efficient way of storing *keypoints* data and they are also appropriate structures to feed the network with. Given that the information contained on the dataset is a set of videos, each with a sequence of frames, each frame with a set of *keypoint* coordinates, it can be organized as a 3-dimensional tensor.

The *keypoints* coordinates data stored on the previously mentioned JSON files, come in the format: $[x_1, y_1, z_1, x_2, y_2, z_2, ...x_n, y_n, z_n]$, where x_1, y_1, z_1 are the coordinates of the first *keypoint*, and n is the number of *keypoints*. After loading these data, x and y coordinates and z coordinates should be separated, since the first are used as input for the network and the latter are the desired values to predict. These values are used to compute the cost function (also called criterion or loss function) for the optimization of the network (see 2.3.1). Therefore, the generated tensors are of shape [N videos, max sequence length, $2*$ (m keypoints)] and [N videos, max sequence length, m keypoints], respectively. That is (each row being a video):

$$A = \begin{pmatrix} [x_1, y_1, x_2, y_2, \dots x_n, y_n]_1^1 & [x_1, y_1, x_2, y_2, \dots x_n, y_n]_2^1 & \cdots & [x_1, y_1, x_2, y_2, \dots x_n, y_n]_S^1 \\ [x_1, y_1, x_2, y_2, \dots x_n, y_n]_1^2 & [x_1, y_1, x_2, y_2, \dots x_n, y_n]_2^2 & \cdots & [x_1, y_1, x_2, y_2, \dots x_n, y_n]_S^2 \\ & \vdots & & \\ [x_1, y_1, x_2, y_2, \dots x_n, y_n]_1^N & [x_1, y_1, x_2, y_2, \dots x_n, y_n]_2^N & \cdots & [x_1, y_1, x_2, y_2, \dots x_n, y_n]_S^N \end{pmatrix} \tag{3.1}$$

and

$$B = \begin{pmatrix} [z_1, z_2, \dots z_n]_1^1 & [z_1, z_2, \dots z_n]_2^1 & \cdots & [z_1, z_2, \dots z_n]_S^1 \\ [z_1, z_2, \dots z_n]_1^2 & [z_1, z_2, \dots z_n]_2^2 & \cdots & [z_1, z_2, \dots z_n]_S^2 \\ & \vdots & & \\ [z_1, z_2, \dots z_n]_1^N & [z_1, z_2, \dots z_n]_2^N & \cdots & [z_1, z_2, \dots z_n]_S^N \end{pmatrix} \tag{3.2}$$

where n is the number of *keypoints* in a frame, S is the maximum frame sequence length among all videos, and N is the total number of videos. Of course, not all the videos are of length S, so the remaining frames to S (the last $S - L$ frames, where L is the actual length of a video) are filled with padding. Notice that the tensor generated by the neural network model containing the predictions has the same shape as the latter (3.2).

Another thing to take into account is that on every recording there are two interpreters. They speak in sign language at the same time, but they aren't saying the same, so each interpreter performance should be considered as a separate video. That also means that every JSON file contains two sets of *keypoints*, one per signer. So, they are loaded as different videos and, therefore, are stored in distinct rows of the tensors.

Furthermore, despite having multiple videos of the same recording from different views, there is only one set of coordinates per person per recording. That means that the axes are oriented according to a single reference view (see figure 3.3), and in order to obtain the coordinates from different reference views, the coordinate system should be properly rotated. Since the body of an speaking interpreter doesn't turn much, if the model is trained using a single reference view, it will learn the body structure seen from that view and it will only perform well on future data that uses the same reference. For that reason, if wanted a better generalization independent from the view, the data should be applied different rotations before using them to train the model.

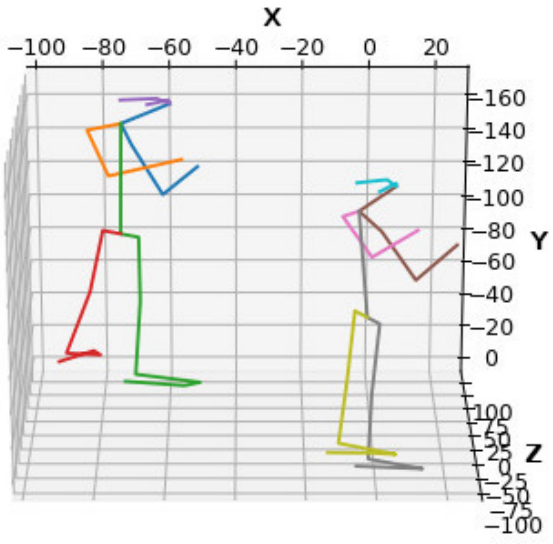


Figure 3.3: Body keypoints from a recording frame with the original axis orientation.

Taking into account the nature of the task, though, it can be noticed that there is no necessity for the network to generalize to multiple views: almost all sign language videos will be recorded from a frontal view; in particular, the ones on the green screen studio dataset, for which this project is intended to work, are recorded from a frontal view. Thus, it is only needed to rotate every skeleton around its centroid so that it ends oriented in a frontal view.

As explained before, each skeleton in 3.3 is treated like a separate video when loaded to a tensor. The easiest way to rotate every skeleton (in every frame of every video) around its centroid is to align them all to zero (coordinate origin) and then apply the rotation matrix to each -see A.

3.4.2 Pose Normalization

Once the data are correctly oriented, they are suitable to feed to the network. However, every interpreter has a different body structure and size and that increases the complexity of the learnable general function that maps the x,y coordinates to z coordinates for any body. In order to reduce that complexity, making it easier to learn, and improve the performance, the following normalization process is applied to all videos.

Let A (3.1) be the input tensor, and B (3.2) the tensor of ground-truth z, A_x^i i-th row (video) of the subtensor of A containing just x coordinates, A_y^i the analogous for y coordinates, and B^i the i-th row of B. Apply:

1. Mapping range of coordinates to [-1, 1].

$$\frac{A_x^i - (\frac{\max A_x^i + \min A_x^i}{2})}{\frac{\max A_x^i - \min A_x^i}{2}} \quad \forall\ i,\ 1 \leqslant i \leqslant N \tag{3.3}$$

$$\frac{A_y^i - (\frac{\max A_y^i + \min A_y^i}{2})}{\frac{\max A_y^i - \min A_y^i}{2}} \quad \forall\ i,\ 1 \leqslant i \leqslant N \tag{3.4}$$

$$\frac{B^i - (\frac{\max B^i + \min B^i}{2})}{\frac{\max B^i - \min B^i}{2}} \quad \forall\ i,\ 1 \leqslant i \leqslant N \tag{3.5}$$

   where N is the number of rows (videos).

2. Apply normalization by mean and standard deviation.

$$\frac{\mathbf{A_x^i} - \mu_{\mathbf{A_x^i}}}{\sigma_{\mathbf{A_x^i}}} \quad \forall\ i,\ 1 \leqslant i \leqslant N \tag{3.6}$$

$$\frac{\mathbf{A_y^i} - \mu_{\mathbf{A_y^i}}}{\sigma_{\mathbf{A_y^i}}} \quad \forall\ i,\ 1 \leqslant i \leqslant N, \text{ where} \tag{3.7}$$

$$\mu_{T_c^i} = \frac{1}{S}\frac{1}{n}\sum_{j=1}^{S}\sum_{k=1}^{n} T_c^{i,j,k} \tag{3.8}$$

$$\sigma_{T_c^i} = \sqrt{\frac{\sum_{j=1}^{S}\sum_{k=1}^{n} T_c^{i,j,k} - \mu_{T_c^i}}{S \times n}} \tag{3.9}$$

$$\frac{\mathbf{B^i} - \mu_{\mathbf{B^i}}}{\sigma_{\mathbf{B^i}}} \quad \forall\ i,\ 1 \leqslant i \leqslant N, \text{ where} \tag{3.10}$$

$$\mu_{T^i} = \frac{1}{S}\frac{1}{n}\sum_{j=1}^{S}\sum_{k=1}^{n} T^{i,j,k} \tag{3.11}$$

$$\sigma_{T^i} = \sqrt{\frac{\sum_{j=1}^{S}\sum_{k=1}^{n} T^{i,j,k} - \mu_{T^i}}{S \times n}} \tag{3.12}$$

N being the number of rows (videos), S the maximum sequence length, and n the number of *keypoints* in a frame.

Note that the shift factors $\frac{\max T_c^i + \min T_c^i}{2}$ and $\mu_{T_c^i}$, and the scale factors $\alpha_{\mathbf{T_c^i}} = \frac{\max \mathbf{T_c^i} - \min \mathbf{T_c^i}}{\mathbf{2}}$ and $\sigma_{\mathbf{T_c^i}}$ are scalars. These latter factors are kept in memory on the implementation to scale the results for better interpretability.

Given that the Panoptic Studio dataset from How2Sign is the only suitable and available for the project, it is used both for the learning process of the proposed approach and for evaluating its performance. Hence, part of the data is used for training the model and another part is exclusively kept for testing (more details on the data split on 4).

The previously described normalization process is applied independently to each split of the data -since it is done per video-, because training data must not be normalized using the inherent characteristics of other data. Furthermore, since test data emulates future data for which there won't be ground-truth labels, and for which the model will want to predict z coordinates, α_{B^i} and σ_{B^i} should not be used to scale the predicted tensor. Because in real inference there won't be a B tensor (desired values tensor). The scale factors for better interpretability of the test predictions (z values), then, should be inferred from the scale factors for test x and y values - that is done with a **linear regression model** (more details in 3.4.3).

3.4.3 Long Short-Term Memory Network

Video pose estimation from 2D to 3D, can be accomplished with a model based on Long Short-Term Memory [10, 9] architectures, because a video is a temporal sequence of

frames. In this particular case, a video is represented by a sequence of *keypoint* coordinates sets, each containing the pose information for a given frame. The approach developed in this project is basically a LSTM network with added feed-forward layers (more commonly known as **fully connected** layers or linear layers).

A LSTM network receives information and generates output in time-steps, creating, thus, a sequence-to-sequence model. In each instant, an LSTM receives 3 pieces of information and produces 2 of them: it receives the previous time-step hidden-state, the previous cell-state, and the current time-step input feature vector; it generates the current time-step hidden-state and the cell-state. Both states are the memory of the network, they contain the stored temporal information.

In this project, the input feature vector on a time-step is the 2D pose skeleton for a given frame (see figure 3.4), and the output on a time-step is the remaining coordinate to produce the 3D pose skeleton.

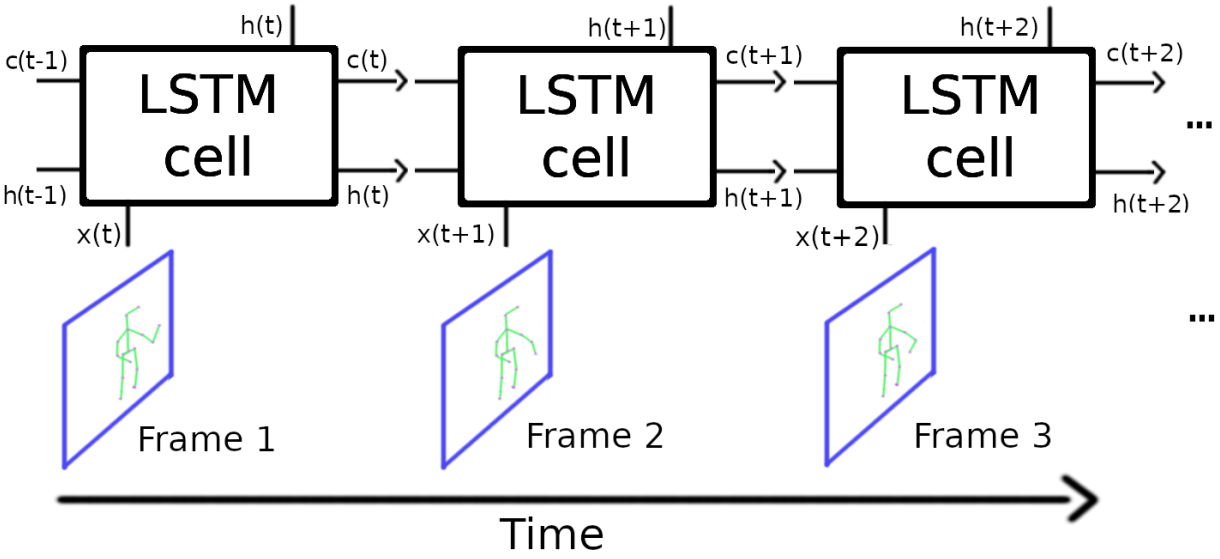
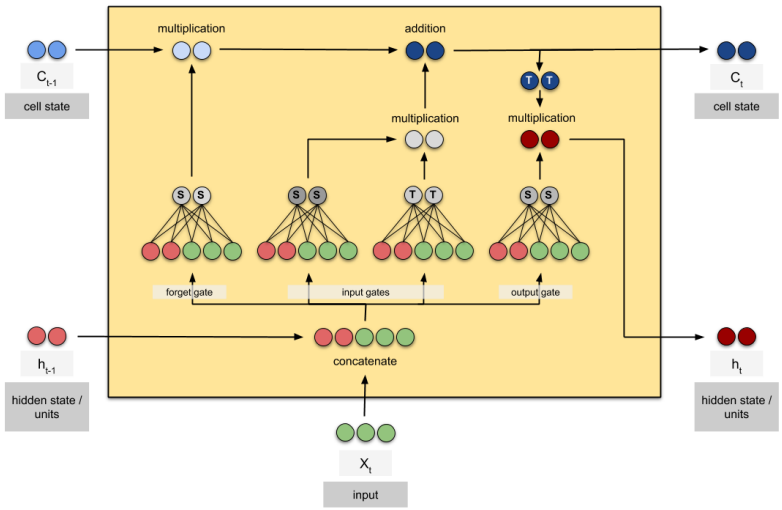
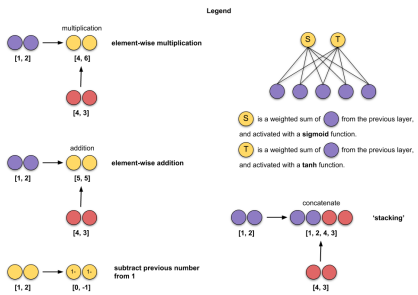


Figure 3.4: Time-steps of the project's LSTM.

In the previous image it may seem that there are more than one LSTM cell, but actually it is the same cell that appears unfolded on time. That means that input feature vectors are fed one at a time, each on a time-step, and that the hidden and cell states from one instant are fed to the same cell on the next time-step. That way, the network can handle any length of sequence, its architecture is not length-dependent. Now, the internal operation of a LSTM cell is illustrated in the next figure 3.5.



(a) Internal operation of LSTM cell.



(b) Legend.

Figure 3.5: LSTM cell (Source: Raimi Karim, 2018 [4]).

On the LSTM used for this project, in particular, the X_t illustrated on the above picture will be of size $2 \times n$, with n the number of *keypoints*, i.e. $X_t = [x_1, y_1, x_2, y_2, \dots x_n, y_n]_t$. In fact, the network is fed in batches. A batch A is a tensor like the mentioned on the previous subsection of this report (3.1) but with $N(\text{videos}) = \text{batch size}$. Then, at a time-step what the network receives as input is the j-th column of A: $A_{:,j}$, that contains a set of *keypoints* from the corresponding frame, for every video in the batch.

The LSTM outputs a tensor of shape [N (batch size), max sequence length, hidden size], but the model needs a tensor of shape [N (batch size), max sequence length, m keypoints] as output. That is why this design has a fully connected layer after the LSTM network that converts the last dimension size to the desired size, that is, gives n z coordinates per time-step per batch element.

It must be emphasised that a LSTM network can have more than one layer. That just means that an N-layer LSTM network has N-1 additional cells that take as input the output of the previous layer cell. In each time-step the output of the network is that of the N-th layer cell. Evidently, a N-layer LSTM will have N times the hidden size (dimension of cell and hidden states) of a single-layer LSTM with the same characteristics.

The final design for the architecture used in this project, namely "LSTM 2D to 3D", is the following: A 2-layer LSTM network with hidden size 512, followed by a fully connected layer with input size 512 and the number of *keypoints* as output size - nearly 3.8 million trainable parameters. Larger architectures were designed and tried, with increased hidden size, more number of layers or making the LSTM bidirectional, but the results didn't improve, and they are not included in chapter 5.

The approach was designed so that four models were produced with the aforementioned "LSTM 2D to 3D": one that uses and predicts all the skeleton *keypoints*, and three that are dedicated exclusively to a set of *keypoints* each (one for the face, one for the hands and one for the body). That was done to see whether it improved the performance when using the set-dedicated models (see 5).

Predicting scaling factors

It has been mentioned that, when using the produced models to run inference on new non-labeled data, there will not be a ground-truth tensor from which the scaling factors for visualization can be computed. The designed method to produce those scaling factors in that case, is to learn them from the input data factors by using another machine learning algorithm: a linear regressor. The linear regressor works similarly to a Neural Network but has a much simpler structure. The idea is to try fitting a training data, that is, to learn the function that maps an input with the shape of a vector $v \in \Re^N$ to, typically, an output value y (though it could be another vector). Specifically in this case, the input vector values would be both scaling factors from x and y in a given video, and the output the z scale factor.

The linear regressor tries to learn the parameters θ_1,θ_2 and b that minimize an error function (typically MSE) between y and $\hat{y}$, where $\hat{y} = \theta_1 v_1 + \theta_2 v_2 + b$. The parameters are updated, like a neural network, with an optimizer algorithm (normally Stochastic Gradient Descent). The linear regressor would be trained using the scaling factors from every video in the train dataset of this project.

3.5 Design of second approach: Classification

The second idea was to convert the problem to a classification task, because it may be easier for a network to classify an input feature vector between pre-defined classes than to predict a continues value. The pipeline followed in this approach, though, is exactly the same as the explained in the previous one.

3.5.1 Structuring data and pose normalization

As far as data structuring goes, collection, data processing and normalization processes are the same as in the first approach. However, there is an extra step here after normalizing the data.

In this case, once the normalized tensors are created, an additional quantization is required. Every z-coordinate needs to be split into several bins, each corresponding to a discrete value (typically, the upper boundary of the bin). The bins are labelled with an integer from 0 to N-1 -where N is the number of bins. Then, each value on B (desired values tensor) is substituted for the label of the bin it falls into.

Also, the only scaling factor that is stored/inferred for predicted values in this approach is the mean z-range in each video. No other are necessary because the de-normalization procedure in this case is: divide each result by N, knowing that each value will be a

number between 0 and N-1 (label of the bin); and then multiply by the computed z-range, returning that way to the original scale of the axis.

3.5.2 Long Short-Term Memory network

In this case, the LSTM network will solve a classification task by generating an output tensor of shape:
$[N$ (batch size), max sequence length, m keypoints, C classes$]$.

The last dimension corresponds to the encoding of the classes (bins), with the one-hot encoding method. The number in each class for a given *keypoint* prediction represents the "probability" of pertaining to that class. Therefore, a post-processing method selects the class with maximum probability and substitutes the last dimension for the class label -returning to a 3-dimensional tensor.

The architecture design of "LSTM 2 to 3D" in this approach, then, is the following: A 3-layer LSTM network with hidden size 512, followed by 2 fully connected layers - the first with input and output sizes 512 and 256, respectively, and the second with 256 as input size and number of *keypoints* as output size - nearly 5.6 million trainable parameters. The architecture is larger than the one from regression approach, because in this case it did improve a bit the results when increasing the capacity.

Chapter 4

Development

4.1 Hardware

It was necessary to set up the appropriate working environment for developing the project. Given the software nature of the project, there has not been much physical material involved but the computers used to develop the code and test the solutions. In particular, most of the project was developed on the own personal computers of the student. However, as it often occurs with Deep Learning projects, when training the models proposed as solution for the problem tackled in this project, considerable computational power was needed. That is why, if required, the code was also run on a Docker[1] container allocated on Dublin City University servers, or on machines with GPU backend provided freely by Google with limited execution time[2].

Due to the limited power the personal hardware of the author has, tests using all the videos from same interpreter and the whole dataset coult not be run for the classification approach. The local GPU would run out of memory. It might have been possible to allocate the scripts and database to another machine capable of performing those tests, however, since the second approach was not working as well as expected it was decided not to waste time and resources on that process.

4.2 Software

All the project was implemented in Python 3 programming language. In order to build a dedicated developing environment, a Docker container was created with no more and no less than the required tools for conceiving, implementing and testing the solutions. All the code was developed in Jupyter Notebooks[3], and these notebooks are hosted on a specially created for the project Github repository [4].

[1]https://www.docker.com/

[2]https://colab.research.google.com

[3]https://jupyter.org/

[4]https://github.com/imatge-upc/asl-2d-to-3d The repository is currently private and can only be accessed by the author and the supervisors of the project, as well as any other specifically granted access

Python[5] is an interpreted, high-level programming language. It is widely used for many purposes because of its incredible versatility -Python is multi-paradigm, it supports functional, imperative, object-oriented and structured programmings. Furthermore, it has a huge community, so a lot of help and documentation can be easily found online.

In addition, most data-science implementations and virtually all machine learning algorithms implementations are currently built around a particular data structure from Python's NumPy[6] library: the NumPy array, an efficient interface to store and operate dense data buffers [36]; it can be generalized to N-dimensions with numpy.ndarray structure. Finally, the most used libraries for Deep Learning are Python's. For all that, it was the chosen programming language to implement this project.

Among the aforementioned Python libraries, Pytorch was the one used to develop the designed network. PyTorch[7] is an open source machine learning library that provides a class called "torch.Tensor" to store and operate multidimensional arrays of numbers. The PyTorch Tensor class is similar to the NumPy Array, but can also operate on CUDA-capable Nvidia GPUs.

Other utility libraries like *matplotlib*[8] for plotting, and SciPy for coordinates rotation were used. And, as mentioned on 4.1, all the code was developed on Jupyter Notebooks.

4.3 Notebooks organization

As explained in 3 Methodology, two designs were implemented for this model: the first one addresses the problem as it is (a regression task); and the second, although basically using the same neural network type (LSTM), tackles it as a classification task, i.e. make the z values discrete and build a classification model with each class representing a discrete value.

For every design, four main Jupyter notebooks were created, each containing the development of a model: one trained and tested using all the *keypoints*, one for face *keypoints*, another for hands *keypoints*, and the last for just body *keypoints*. All four Notebooks replicate the same structure, they only differ on the data collection (each Notebook only loads its corresponding sets of keypoints), on the implementation of an evaluation metric and on the plotting part for the interpretation of results.

The structure of the notebooks follow the pipeline designed on the previous chapter. Thus, first of all, the pertaining data is acquired, each of the two people on every recording is treated as a separate video, and both A (3.1) -x, y tensor- and B (3.2) -z tensor- are generated. This is the corresponding code on the notebook that uses all *keypoints*.

```
1 def get_keypoints(data_path):
2     dataset = []
3     groundtruth = []
```

person.

[5] https://www.python.org/
[6] https://numpy.org/
[7] https://pytorch.org/
[8] https://matplotlib.org/

```
    # Look over just the folders inside the directory
    just_folders = filter(lambda x: isdir(join(data_path, x)), listdir(
data_path))
    for p in list(map(lambda x: join(data_path, x), just_folders)):
        # Gets 2 list of n_frames lists, one for the 2D coordinates and
one for the third coordinate.
        # Each list of the n_frames lists contains, either the (x and y)
 or the z of each keypoint for the face(first line), hands(second),
body(third).
        # e.g. the first line will result in [[x1,y1,x2,y2...x70,y70]
sub1...[x1,y1...x70,y70]subN], [[z1,z2...z70]sub1...[z1..z70]subN]
        # Actually, as there will be two of each list above because
there are two people en each video.
        face_2d, face_3d = get_face(p)
        hands_2d, hands_3d = get_hands(p)
        pose_2d, pose_3d = get_body(p)

        # Concatenates the coordinates for the face, hands and body on
the last dimension, for each person.
        vid_input_p1, vid_input_p2 = ([fa+ha+po for fa, ha, po in zip(
face_2d[i], hands_2d[i], pose_2d[i])] for i in range(2))
        vid_labels_p1, vid_labels_p2 = ([fa+ha+po for fa, ha, po in zip(
face_3d[i], hands_3d[i], pose_3d[i])] for i in range(2))

        dataset.append(vid_input_p1)
        dataset.append(vid_input_p2)
        groundtruth.append(vid_labels_p1)
        groundtruth.append(vid_labels_p2)
        print(f'Completed folder {p}')
    return dataset, groundtruth
```

The `get_face()`, `get_hands()` and `get_body()` functions are in charge of obtaining the corresponding sets of keypoints on video frame sequences. The other notebooks will only have one of those functions each. The output of the above function is both A and B tensors, this time represented as nested lists.

After creating those, since not every list in the first dimension (not every sequence of frames) is of equal length, padding is inserted on the shorter sequences, so that all the lists all have maximum sequence length. Once that is accomplished, the nested lists are converted into multidimensional NumPy arrays; specifically in this case, into 3-d arrays. The value used for padding is NaN (IEEE 754 floating point representation of Not a Number), provided by NumPy library.

Given that reading the data directly from the JSON files is a really slow process, after creating the NumPy arrays on the first run, they were stored into Python pickles for faster loading in posterior execution of the notebooks.

Once data acquisition is completed, each notebook proceeds with the data structuring code:

- all skeletons centroids are aligned with coordinate origin; a rotation is applied so that every skeleton is facing a frontal view;

- the normalization process explained on 3.4.1 is conducted;
- **(regression notebooks)** the scale factors -range of values and standard deviation for x,y,z- for each video of the training data are stored, and the "z-axis" ones for the evaluation data are inferred with a linear regressor model (further explanation below);
- **(classification notebooks)** only the range of values is stored/inferred as a scaling factor, because the results' values will be numbers in range $[0, 20]$ (see next point) and then dividing them by 21 and multiplying by just the range, the proper aspect ratio is obtained;
- **(classification notebooks)** the z-values on the ground-truth array are converted to integers in the range $[0, 20]$ - the range of z is divided into 21 "bins", each represented by an integer value-;
- then, NumPy `ndarrays` are converted into `torch.Tensor` and they are stored in `TensorDataset` PyTorch structures, where they can be grouped into batches;
- and finally, the torch device is set to GPU if available.

After that, Neural Network module classes, called `LSTM_2D3D`, are defined for both regression and classification solutions. The classes allow to implement the designed "LSTM 2D to 3D" architectures, but they are more general. They have selectable input size, output size, hidden dimension and even bidirectional optionality for the LSTM (see 4.4). The class version defined for classification task also have selectable number of bins (classes).

They also have an option for activating dropout, which is a regularization method to prevent overfitting on the training data [37]. Dropout works by randomly dropping nodes (leaving them to 0) with certain probability, forcing that way to break the co-dependency that neurons on a given layer may develop among each other and cause a braking on the generalization power of individual neurons. For the final results, though, dropout was not used.

The output of the regression task models instantiation can be seen in the next figure 4.1.

```
LSTM_2D3D(
  (lstm): LSTM(276, 512, num_layers=2, batch_first=True)
  (fc): Sequential(
    (0): Linear(in_features=512, out_features=138, bias=True)
  )
)
3789962
```

Figure 4.1: First output is the model is the model architecture; the second is the exact number of trainable parameters.

Once created the model, training process is conducted. The Jupyter Notebook format allows to just re-execute the training cells, enabling an efficient way of fine-tuning the network hyperparameters.

Finally, it is run the inference on the testing data and the evaluation metrics are computed. After that, each notebook provides an interpretation section for appropriate visualization of the results, with 3D plots on both training and testing data.

Notice that the data is normalized. For that reason, in the interpretation section, a de-normalization process is conducted to recover the original aspect ratio before plotting the skeletons. It is then that the scaling factors learned during the data structuring are used. The problem is that, the test dataset represents new data that has not been seen by the network and for which there is no ground-truth annotation, and then it would not be possible to have the scaling factors for the z-axis (target values). For that reason, the z scaling factors for test data are inferred from the ones of x and y with a Linear Regression model.

Since 2D to 3D video pose estimation is not an easy task, and because of the fact that every person's body has slightly different proportions (even after normalizing them), the problem was tackled first from a simplified version and then with the original specifications. That is, first it was tried to solve the pose estimation for one video (splitting its frames into train and test), since that task would not have the extra complication of generalizing to any body structure and the sign language domain would be restricted to that of the topic the video is talking about. Later, the problem was tried to handle for more than one video but from the same interpreter - this specification assumes all future data will have the same body structure as this interpreter. And, at the end, it was tried with all videos from all signers in the dataset. All notebooks then, have a "mode" variable that can be set to three different values in order to run the notebook in either one of the aforementioned modes.

Note that in the case of just one video, the data tensors would be of shapes
[1, sequence length, 2*(m keypoints)] (x and y coordinates, input) and
[1, sequence length, m keypoints] (z coordinates, expected output), respectively; those can't be separated into training data and testing data. For that reason, the selected video was cut into several smaller sequence length parts that were taken as different videos, i.e. tensors of shape [P parts, $\frac{\text{original sequence length}}{P}$, 2*(m keypoints)] and
[P parts, $\frac{\text{original sequence length}}{P}$, m keypoints].

4.4 Selected code snippets

Given that this a software project, the coding was the main component of it, and the most important part. That being said, not all the code is necessary to put in this report, because the complete Jupyter Notebooks can be found on the Github repository of the project 4.1.

So, in this section, only the considered most relevant code (with comments) will be included. The first interesting piece of code worth putting here, then, is the implementation of the normalization process:

```
1 def norm_uniform(tensor, coordinates=1, factor=None):
2     scale = []
3     mean_ranges = []
```

```
    for n_vid in range(tensor.shape[0]):
        coord_scale = []
        max_value = [np.nanmax(tensor[n_vid, :,i::coordinates]) for i in range(coordinates)]
        min_value = [np.nanmin(tensor[n_vid, :,i::coordinates]) for i in range(coordinates)]
        center = [(max_value[i]+min_value[i])/2 for i in range(coordinates)]
        ranges = np.ndarray((tensor.shape[1],coordinates))
        for n_frame in range(tensor.shape[1]):
            rang = [np.nanmax(tensor[n_vid, n_frame,i::coordinates])-np.nanmin(tensor[n_vid, n_frame,i::coordinates]) for i in range(coordinates)]
            ranges[n_frame] = np.asarray(rang)
        mean_range = [np.nanmean(ranges[:,i]) for i in range(coordinates)]
        for j in range(coordinates):
            subtensor = tensor[n_vid, :, j::coordinates]
            subtensor[:] = np.subtract(subtensor, center[j])
            if factor is not None:
                subtensor[:] = np.divide(subtensor, factor[n_vid])
            else:
                subtensor[:] = np.divide(subtensor, max_value[j]-center[j])
            coord_scale.append((max_value[j]-center[j] if factor is None else factor[n_vid]))
        scale.append(coord_scale)
        mean_ranges.append(mean_range)
    return mean_ranges, scale
```

The above function implements the first step of the normalization process explained in 3.4.1, that is, mapping the coordinates original range to [-1, 1] range. As it can be seen, it is programmed in a flexible way that allows to apply the process to tensors with different last dimension sizes depending on the number of coordinates they contain -this function can be applied both to tensors like (3.1) or like (3.2). The returned values are the scaling factors that can be learned in this normalization step -the mean range of z in each video and the actual denominator of the mapping procedure. Only one of them has to be used in the de-normalization process for correct visualization and, in the final implementation, the first (average z-range) was chosen, because it gave better predictions on the linear regressor.

As for the second part of the normalization process previously mentioned, the actual standardization, is implemented on the following function:

```
def normalize(tensor, coordinates=1, std=None):
    moments = []
    std_centroids = []
    for n_vid in range(tensor.shape[0]):
        coord_moments = []
        mean_value = [np.nanmean(tensor[n_vid, :,i::coordinates]) for i in range(coordinates)]
        std_value = [np.nanstd(tensor[n_vid, :,i::coordinates]) for i in range(coordinates)]
```

```
        centroids = np.ndarray((tensor.shape[1],coordinates))
        for n_frame in range(tensor.shape[1]):
            centroid = [np.nanmean(tensor[n_vid, n_frame, i::coordinates
    ]) for i in range(coordinates)]
            centroids[n_frame] = np.asarray(centroid)
        std_centroid = [np.nanstd(centroids[:,i]) for i in range(
    coordinates)]
        if std is not None:
            std_value = [std[n_vid]]
        for j in range(coordinates):
            subtensor = tensor[:, :, j::coordinates]
            subtensor[:] = np.subtract(subtensor, mean_value[j])
            subtensor[:] = np.divide(subtensor, std_value[j])
            coord_moments.append((mean_value[j], std_value[j]))
        moments.append(coord_moments)
        std_centroids.append(std_centroid)
    return moments, std_centroids
```

It is programmed in the same general form as the other function. Notice that both functions apply the normalization in-place (they overwrite directly the old tensors); and instead, their output values are the scale factors for each coordinate on each video. In this second function the returned values are $\sigma_{T_c^i}$ $\forall i$ $1 \leqslant i \leqslant N$ videos, and c belonging to [x,y] and the standard deviation of the skeleton centroids in each video, respectively. Again, only one was needed for de-normalization and the first was selected, because it gave better estimations on the Linear Regressor.

The other piece of code worth mentioning here is the definition of `LSTM_2D3D` class. It will be shown here the one used on regression task notebooks, the classification version of the class can be found in B.

```
class LSTM_2D3D(nn.Module):

    def __init__(self, input_size, output_size, hidden_dim, n_layers,
    bidirectional, dropout=0.):
        super().__init__()
        # Save the model parameters
        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim
        self.bi = bidirectional

        # Define the architecture
        self.lstm = nn.LSTM(input_size, hidden_dim, n_layers,
    batch_first=True, bidirectional=bidirectional, dropout=dropout)
        self.fc = nn.Sequential(
        nn.Linear(hidden_dim*(2 if self.bi else 1), output_size)
        )

    def forward(self, x, state, lengths):
        # Describe the forward step
        batch_size, seq_len = x.size(0), x.size(1) # We save the batch
    size and the (maximum) sequence length

```

```
21        # Need to pack a tensor containing padded sequences of variable
    length
22        packed = nn.utils.rnn.pack_padded_sequence(x, lengths=lengths,
    batch_first=True, enforce_sorted=False)
23        ht, hidden_state = self.lstm(packed, state) # ht will be a
    PackedSequence
24
25        # Need to flatten and reshape the output to feed it to the
    Linear layer
26        ht = ht.data.contiguous() # ht will be of shape [sum(lengths),
    hidden_dim]
27        ot = self.fc(ht) # ot will be of shape [sum(lengths), ouput_size
    ]
28
29        l_ot = [ot[:int(length)] for length in lengths] # list of batch
    elements, each shape [lengths[i], output_size]
30        packed_ot = nn.utils.rnn.pack_sequence(l_ot, enforce_sorted=
    False) # PackedSequence
31        # Finally return to shape [batch_size, seq_len, output_size]
32        ot, _ = nn.utils.rnn.pad_packed_sequence(packed_ot, batch_first=
    True, total_length=seq_len)
33
34        return ot, hidden_state
35
36     def init_hidden(self, batch_size):
37        weight = next(self.parameters()).data
38        hidden = (weight.new(self.n_layers*(2 if self.bi else 1),
    batch_size, self.hidden_dim).zero_().to(device),
39        weight.new(self.n_layers*(2 if self.bi else 1), batch_size, self
    .hidden_dim).zero_().to(device))
40        return hidden
```

It is shown that the class inherits from PyTorch nn.Module, and that it is structured in 3 main parts: declaration of the network parameters, definition of the forward step (main description of the network functioning), and the states initialization -as explained in previous chapters, an LSTM has two stored states, hidden and cell, and every time-step takes those states from the previous one, so on the first-time step they need an initialization.

It was previously explained that some padding should be added to the end of shorter sequences in order to form a rectangular tensor. These padding, though, are not real values the network wants to take as input, if they were, they could lead to an incorrect solution for the task. It can be seen, then, that in the forward step, the network not only takes the input tensor but also the real length of each sequence, so it knows whether a value is padding or not -detailed notes on how that process reshapes the tensor through the network can be found on the code comments.

Chapter 5

Experimentation and Discussion

Several experiments were carried out to evaluate the proposed methods. It has been mentioned that there are four models (three specific models for each body part, full skeleton general) for each approach. They are evaluated against its corresponding test split of the dataset; and, as noted in the explanation of the notebook "modes" (see 4.3), three experiments were designed for every model -one using information from just one video, one using information from all the videos from the same interpreter, and one using all the available information. After running the inferences, the results were displayed in 3D plots conveniently oriented. The code for generating the visualization of the results can be found at the end of every Jupyter Notebook[1].

5.1 Training details

The training is conducted on the typical way of a supervised learning problem. That is:

- The data are split into three sets: train, validation and test. In this case 80%, 10% and 10%, respectively.
- The Neural Network is trained over a number of epochs, each going through the complete train set. In an epoch, the network is fed the data in batches; for each batch the loss and the backward step are computed, and the weights (network parameters) are updated according to the selected optimizing algorithm.
- At the end of each epoch, inference is run over validation set. And the loss and the performance metric obtained are compared to those of the training in that epoch.
- Finally, after all the epochs, the loss and metric evolution for both train and validation data are plotted.

This process allows to see whether the the model overfits to training data (learns too well the training data structure and has no generalization power), the model is underfitting (it isn't learning well enough the function it is trying to reproduce) or it is just performing as desired. The hyperparameters of the network can be adjusted, then, to improve

[1] https://github.com/imatge-upc/asl-2d-to-3d

its performance, and the aforementioned training process is repeated as many times as considered necessary.

For the specific training carried out in this project, the criterion selected as cost function to minimize was: Mean Squared Error ($MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$) for the first approach; and for the second approach ()classification), Negative Log Likelihood -assume x_i a vector with the estimated probabilities for each class on the sample i and y the target class (ground-truth), then $NLL = \frac{1}{n}\sum_{i=1}^{n} -\log(x_{i_y})$. And the chosen optimizer for both approaches was Adam[2].

Let g_t be the gradient of the cost function w.r.t. parameters θ at time-step t; m_t and v_t the moving averages of the gradient and the squared gradient, respectively, with decay-rate controlling hyperparameters β_1, β_2; and α the stepsize. Adam update step is $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}$, where $\alpha_t = \alpha \cdot \sqrt{1-\beta_2^t}/(1-\beta_1^t)$ [38].

In addition to that, a learning rate scheduler is applied to the learning process. The learning rate is an hyperparameter that represents the length of the step to take in the direction specified by the gradient when performing the optimizer step (update of the weights). This scheduler implements the "1-cycle policy" [39]. The 1cycle policy changes the learning rate after a batch has been used for training. It anneals the learning rate from an initial learning rate to some maximum learning rate and then from that maximum learning rate to some minimum learning rate much lower than the initial learning rate.

Once the the model is considered to have gone through the proper training, inference is run over the test data, and both the loss and the selected metrics are computed to evaluate the model's performance.

5.2 Evaluation metrics

5.2.1 Regression metrics: PCK and MPJPE

The regression task was assessed with two commonly used metrics for pose estimation: Percentage of Correct Keypoints (PCK) [40] and Mean Per Joint Position Error (MPJPE).

As mentioned, accuracy would be a relatively bad metric, e.g. if the desired value was 3.5, both estimated values 3.1 and 9.2 would count as "wrong", but 3.1 is clearly a better estimation. The problem with accuracy, then, is that it only computes as correct the exact same value as the ground-truth in each *keypoint*. And in this problem that is not as important as predicting a body with approximately the same pose as the label. PCK overcomes the explained accuracy restriction by allowing a small specified error on the estimated value to be considered correct. Specifically, PCK considers correct a *keypoint* when the distance between prediction and ground-truth is less than $\alpha \cdot \max h, w, d$, h, w, d being the dimensions of the "space" or "bounding box" the skeleton is placed on.

In the problem addressed here, the three dimensions (x, y and z coordinates) are normalized -first mapped on [-1,1] range and then standardized- as explained in above chapters.

[2]look at PyTorch documentation for the implementation

When computing PCK those dimensions are de-standardized so that in every video the range of values on each axis is [-1,1]. Taking that into account and the fact that only the z coordinate is actually predicted (the other two are taken from input), the actual implementation only computes the difference between predicted and ground-truth z. And the selected α factor is 0.1, meaning that the metric tolerates an error of 20% z range.

Returning to the example illustrated before, accuracy does not evaluate how close the result was to the desired value, which is actually important in this case. And, although allowing certain error on precision of the prediction, neither does it PCK. That is why a distance-oriented metric was also included. MPJPE computes the mean Euclidean distance between the predicted and the ground-truth skeletons (that is to say, mean *keypoint* Euclidean distance) after aligning them with their root keypoints. The formula is the following:

$$\text{MPJPE} = \frac{1}{T}\frac{1}{N}\sum_{t=1}^{T}\sum_{i=1}^{N}\|(J_i^{(t)} - J_{root}^{(t)}) - (\hat{J}_i^{(t)} - \hat{J}_{root}^{(t)})\| \tag{5.1}$$

where N is the number of joints, and T the number of samples. In this particular implementation N equals to the sum of all the sequence lengths across the dataset this metric is computed on.

As seen in the formula above, the root joints of the labels and the joints from the network output are aligned. That is done to give importance to the relative placement of the keypoints on the pose rather than the absolute distance between prediction and ground-truth. Furthermore, PCK is already a metric that indicates absolute correctness. In order perform the alignment, it was defined a function `substract_root_PJPE` (see B) that subtracts the root joint of each *keypoint* set (face, hands, body) from corresponding set.

It is applied to both the predicted and the label tensors, and it returns the root-subtracted versions of them. A defined function implements MPJPE for a batch when the root-subtracted tensors are passed as arguments. Then, when all batch MPJPEs are averaged, the epoch MPJPE is obtained. Note that since the model is predicting z coordinates only, the Euclidean distance is the same as the absolute difference.

MSE loss reflects the squared distance per keypoint between prediction and ground-truth. MPJPE, similarly, reflects the distance between predicted and ground-truth skeletons but once its roots are aligned. Also, considering that the normalized data has a standard deviation of 1 and a mean of 0, MPJPE gives some notion about the mean relative error of the approximation. Or rather, it tells that the error percentage over the full range of values ($\frac{dist.predicted-groundtruth}{rangeofz}$) is less than half the MPJPE score. PCK is basically an accuracy score with a tolerance of 20% error.

Note that both MSE loss and MPJPE in each case are computed using the normalized tensor of the corresponding sets. That means that the metrics cannot be used to compare directly which model will have visually better results, considering that each model normalizes its own data. That way, the same number for MPJPE, for example, in face-only model and the body-only model, would mean that the qualitatively observed error on the body is higher, because the original domain (range of values) of the data is wider.

5.2.2 Classification metric: Accuracy

Here the problem is approached as a classification task. We adopted the classic accuracy -(True positives+True negatives)/(Total number of keypoints)-, with the below justification.

In this particular scenario, the z-axis range of values is split in 21 ”bins” each corresponding to a class. That implies that a correct *keypoint* classification inherently admits an error of 5% of z range. That would be the equivalent of using PCK with $\alpha = 0.025$ in the first approach.

5.3 Quantitative results

The quantitative results are presented in the following tables.

Table 5.1: Averaged (from 3 runs) MSE loss, MPJPE and PCK scores for every model in regression task.

		MSE loss	**MPJPE**	**PCK**
		Test	Test	Test
All kp	1 video	0.9558±0.0897	0.4019±0.0294	30.52±4.25%
	1 interpreter	**0.8049**±0.2463	0.3446±0.0596	31.97±4.78%
	All videos	0.9221±0.2025	**0.2975**±0.0678	**36.32**±5.15%
Body kp	1 video	0.5489±0.1847	0.4307±0.0947	39.23±10.43%
	1 interpreter	**0.3917**±0.0737	**0.3903**±0.0592	**42.46**±1.36%
	All videos	0.9379±0.4556	0.5125±0.0638	28.51±13.08%
Face kp	1 video	**0.9352**±0.4467	0.6928±0.1237	**46.67**±11.57%
	1 interpreter	1.3249±0.1816	**0.4044**±0.0393	29.52±2.78%
	All videos	2.1018±0.4789	0.4807±0.0949	25.86±3.79%
Hands kp	1 video	**1.3475**±0.3017	0.9168±0.0893	23.15±1.57%
	1 interpreter	1.4379±0.1999	**0.5073**±0.0714	11.79±0.64%
	All videos	1.9615±0.0935	0.6014±0.0417	13.47±0.96%

Table 5.2: Averaged (from 3 runs) NLL loss and accuracy score for every model in classification task.

	NLL loss	**Accuracy**
	Test	Test
All kp, 1 video	3.0096±0.0014	15.37±0.32%
Body kp, 1 video	3.0350±0.0014	14.49±0.95%
Face kp, 1 video	3.0223±0.0024	15.40±0.16%
Hands kp, 1 video	3.0298±0.0023	10.25±0.17%

Every experiment was executed three times. The above results are the averaged of the three runs. Complete tables with results from train and validation can be found on C.

Classification task results are far more consistent than regression task's. However, no experiments were performed for one interpreter videos or all the videos, and its stability of results cannot be known.

5.4 Qualitative results

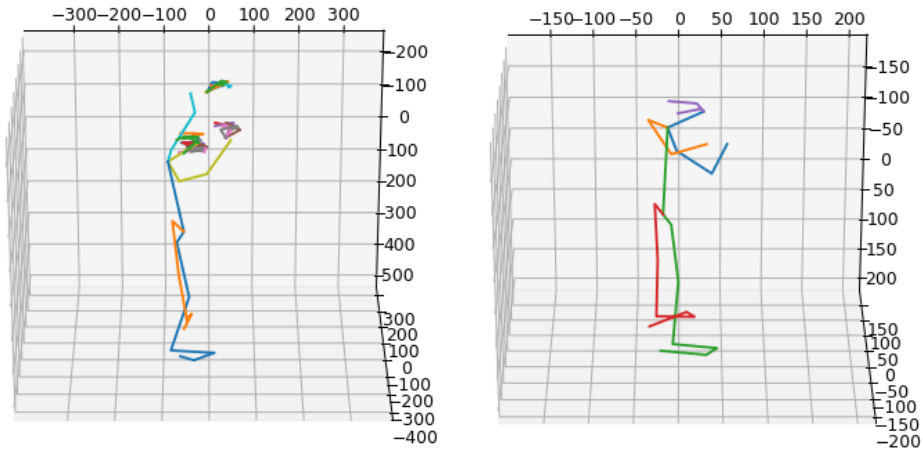
The qualitative results have been plotted for each model in all the experiments (three per regression approach model, and one per classification approach model). For example, for the designed model that uses all the *keypoints* from the JSON files of the first solution, the presented results are: the ones obtained by training and testing with information from just one recording from the dataset; training and testing with information from all the recordings of the same signer; and training and testing with all the recordings in the dataset. The analogous results are displayed for face, hands and only body models of the regression approach. Each experiment results contain plots of predicted skeletons from test data and its corresponding ground-truth.

For every experiment there will be a sample frame plotted per model. But note that all these plots are from different frames, they are **not** the prediction of the same frame by each model. Every one of them was selected either because they represented the majority of good results of its model (but not the very best) or to show a specific behaviour of the network.

Notice that when plotting a skeleton, its coordinate system is rotated so that z values end on the horizontal direction, letting that way to compare results better. And, as the plots shown here are from testing data, they were de-normalized using the inferred scaling factors. That means that some of them needed to be applied a manual "zoom" on z-axis (i.e., an extra scaling factor manually inputted) to correct some bad estimations of the scaling factors and obtain the desired aspect ratio.

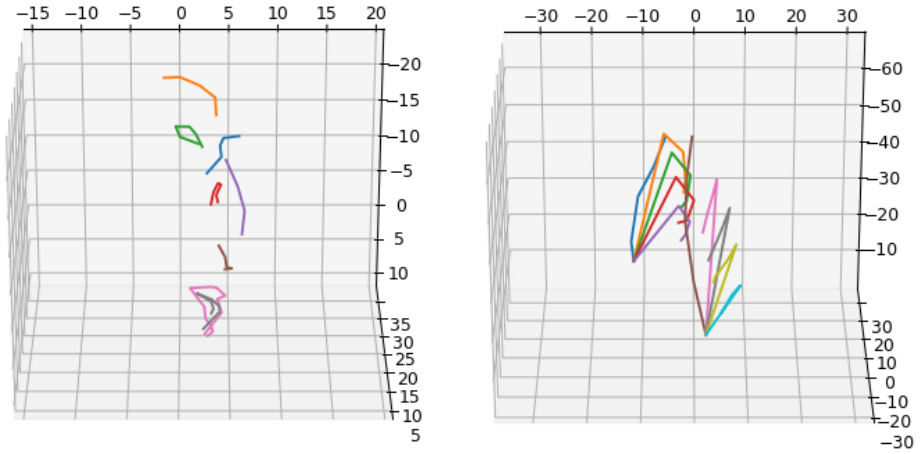
5.4.1 Regression approach

Results of using just one video



(a) Sample frame from all-*keypoints* model.

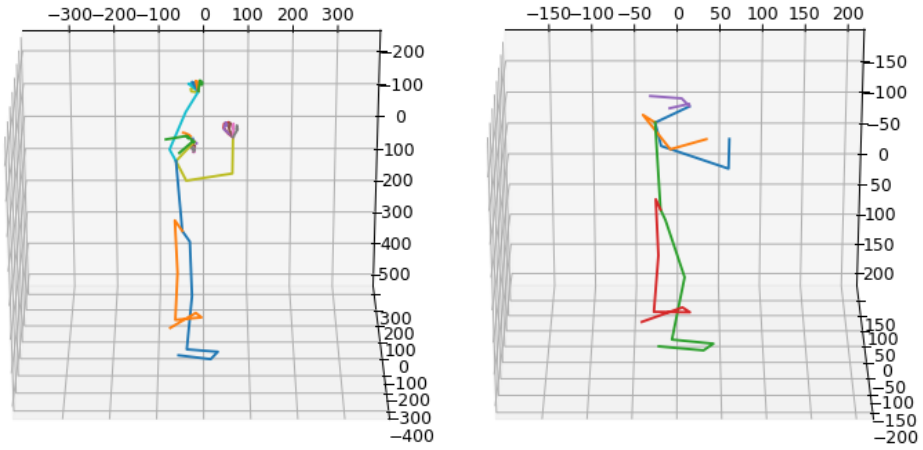
(b) Sample frame from only-body model.



(c) Sample frame from only-face model.

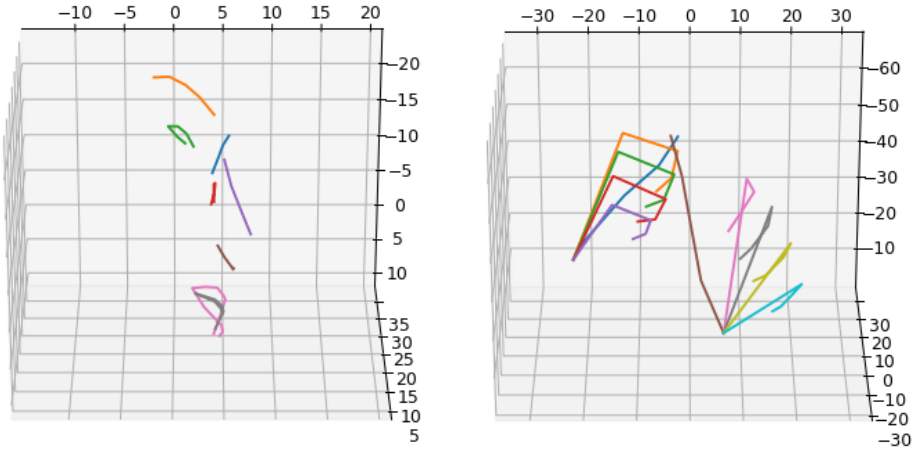
(d) Sample frame from only-hands model.

Figure 5.1: Predictions using one video.



(a) Sample frame from all-*keypoints* model.

(b) Sample frame from only-body model.



(c) Sample frame from only-face model.

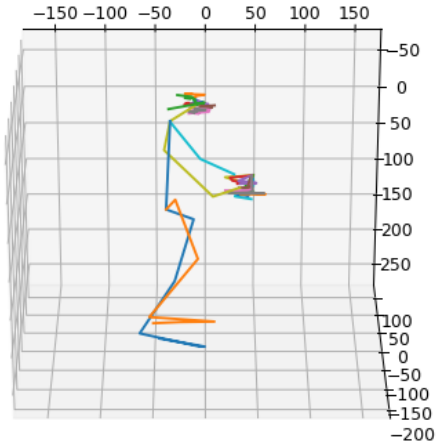
(d) Sample frame from only-hands model.

Figure 5.2: Corresponding ground-truth of

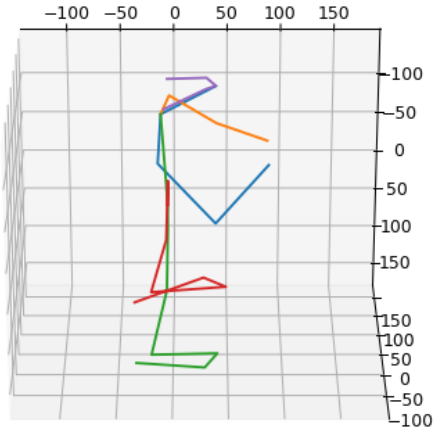
As seen in the above figures, the network performs better when trained for a specific subset of *keypoints*. The model that uses all the *keypoints* is able to predict more or less the general human body structure but lacks precision on *keypoint*-dense parts, i.e. body *keypoints* are acceptable but hands and face appear more distorted. That is caused because the network is not precise enough, and when predicting more separated *keypoints* (body) that imprecision is less noticeable. However, the part-specific models, deal always with the same "dimension" of bone length (i.e. separation between *keypoints* in an skeleton), and that shows much better results.

Also, the above hands and all-*keypoints* plots were applied a large manual scaling because they appeared squeezed in the z dimension (bad prediction of the scaling factors).

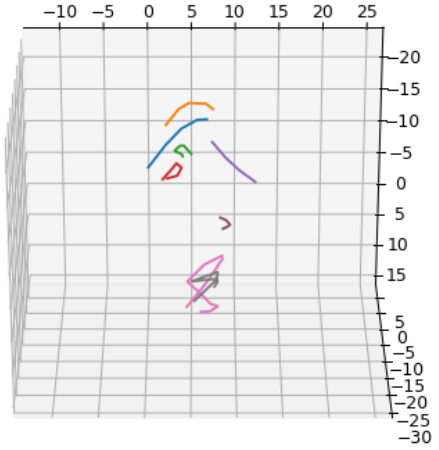
Results of using videos from the same interpreter



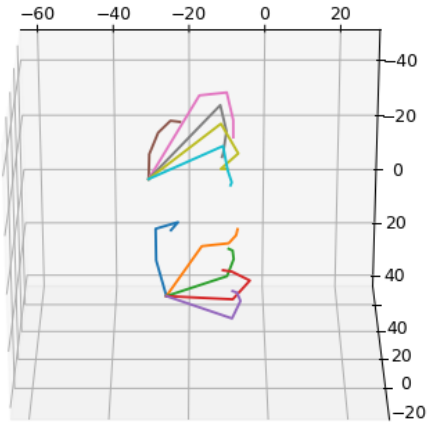
(a) Sample frame from all-*keypoints* model.



(b) Sample frame from only-body model.

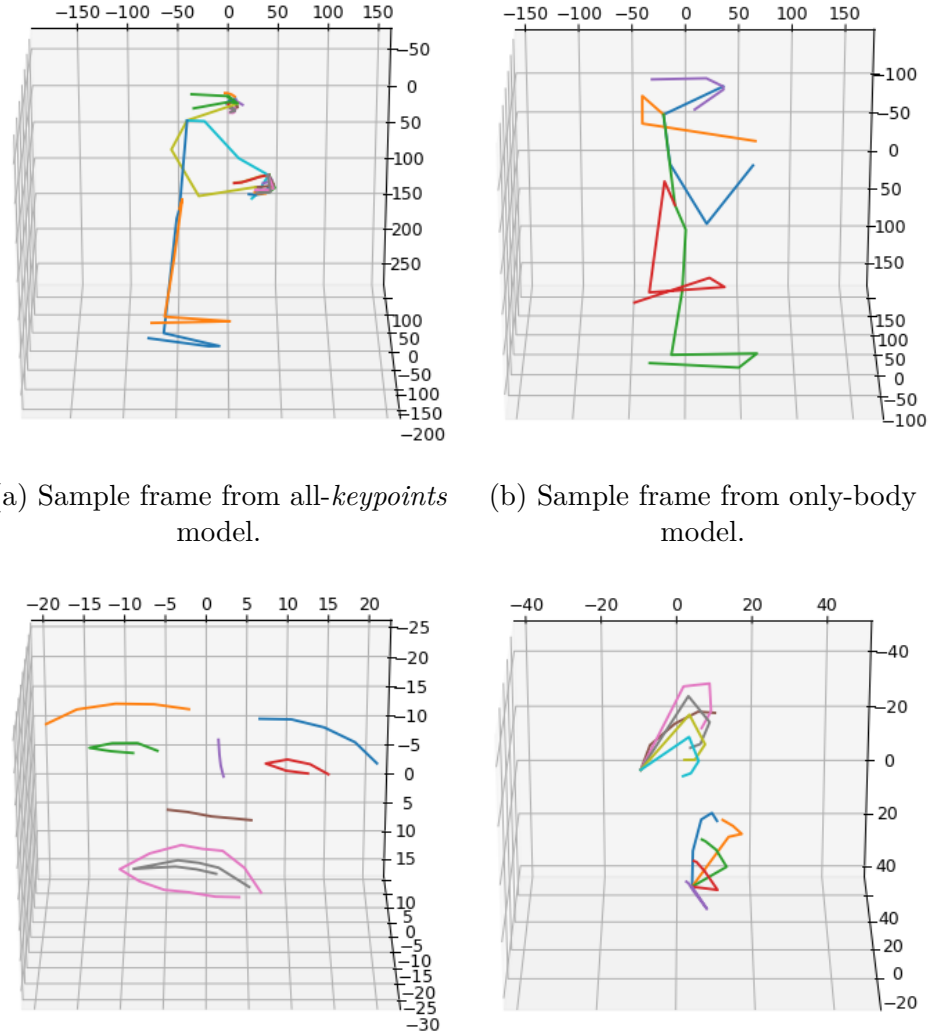


(c) Sample frame from only-face model.



(d) Sample frame from only-hands model.

Figure 5.3: Predictions using videos from the same interpreter.



(a) Sample frame from all-*keypoints* model.

(b) Sample frame from only-body model.

(c) Sample frame from only-face model.

(d) Sample frame from only-hands model.

Figure 5.4: Corresponding ground-truth of

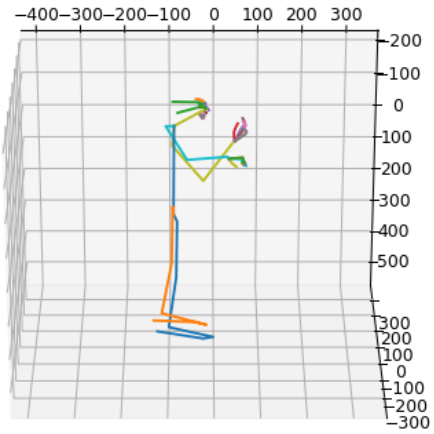
Here again, the same behaviour is observed: the individual models perform better than the general one, as expected. But when using this many more videos, it can be noticed another effect. When trying to predict poses that are rotated with respect to the frontal view, that is to say, they are facing diagonally or sideways if observed from the frontal camera, it does not perform well (see 5.3c and 5.4c).

Since in sign languages the body and the face are mainly facing forward, it seems that the models are learning the "most common pose", or more precisely, the most common z for each *keypoint*. That is why when a face or body should be facing either left or right, it appears facing forward.

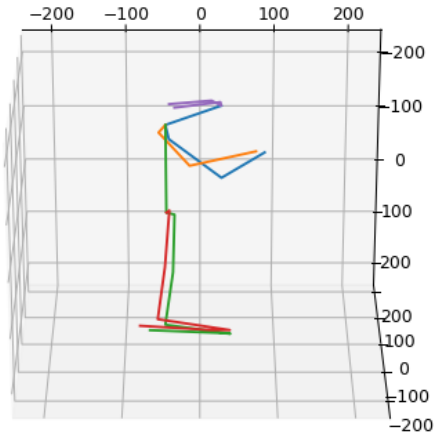
On the contrary, hands are a highly variant structure, and seeing that many different poses cause that network cannot generalize well. This may improve with more training data.

The above face and all-*keypoints* results were actually a little bit squeezed on z, so they were applied a small manual scaling factor. The hands, depending on the frame, still needed a larger correction of the scaling factor.

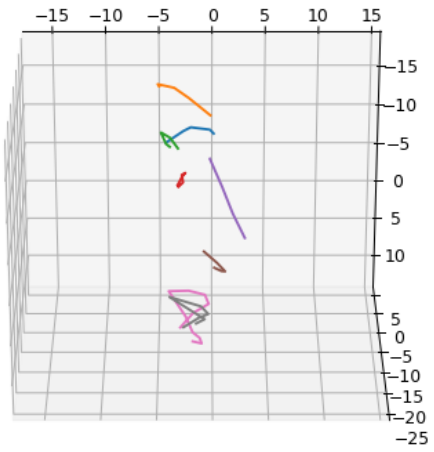
Results of using all the videos



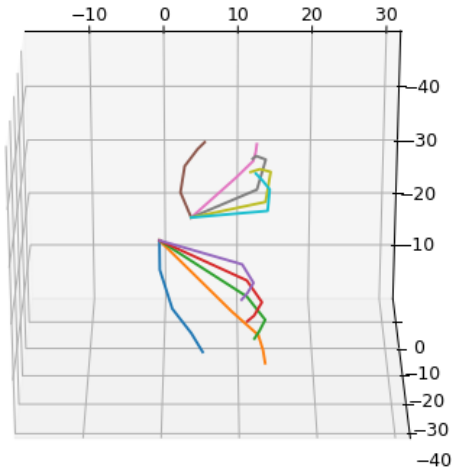
(a) Sample frame from all-*keypoints* model.



(b) Sample frame from only-body model.

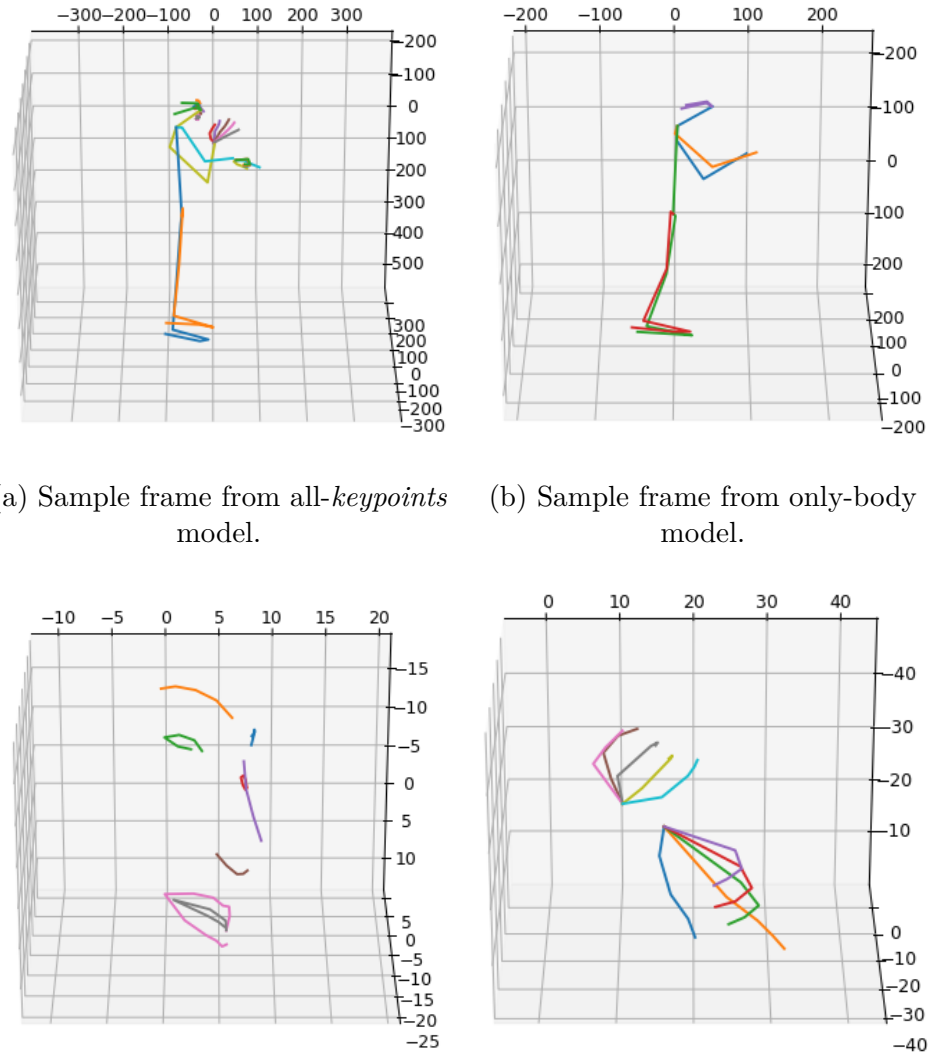


(c) Sample frame from only-face model.



(d) Sample frame from only-hands model.

Figure 5.5: Predictions when using



(a) Sample frame from all-*keypoints* model.

(b) Sample frame from only-body model.

(c) Sample frame from only-face model.

(d) Sample frame from only-hands model.

Figure 5.6: Corresponding ground-truth of

This results using all the videos are better than expected, seeing the previous experiments. Here, even the all-*keypoints* model predicts plausible face and hands. Despite not being equally reflected on the quantitative results for specific-part models, it is clear that having more data is really valuable for improving results. On only-body and only-face models, it makes sense that the differences between interpreters and the improvement achieved by having more data compensate each other (that way having similar qualitative and quantitative results, or even better when using 1 interpreter); because those parts are the most dependent of the interpreter body structure.

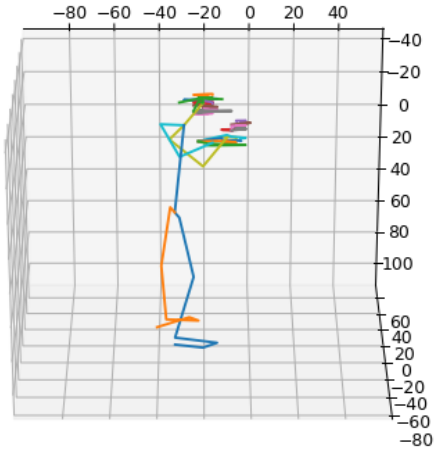
The fact of having different interpreters (hence, different body structures) on the dataset has not had much negative impact. Rather, having that many more data have improved

some results. So, it would seem that the difference between body structures is less important than the diversity of poses.

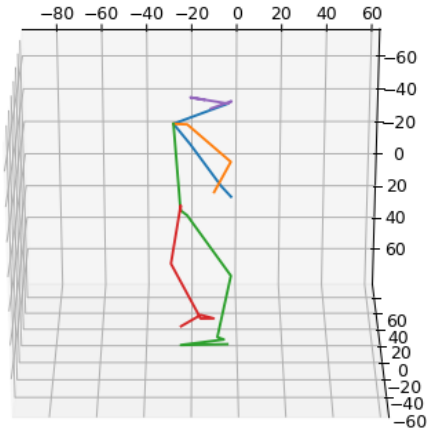
Incidentally, the scaling factors for the body here were the worst predicted among all body results -and they applied the manual ”zoom”, as explained before.

5.4.2 Classification approach

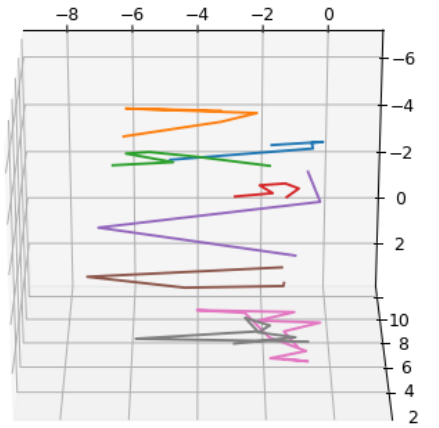
Results of using just one video



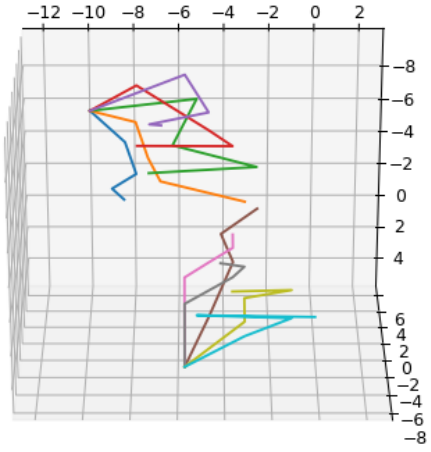
(a) Sample frame from all-*keypoints* model.



(b) Sample frame from only-body model.

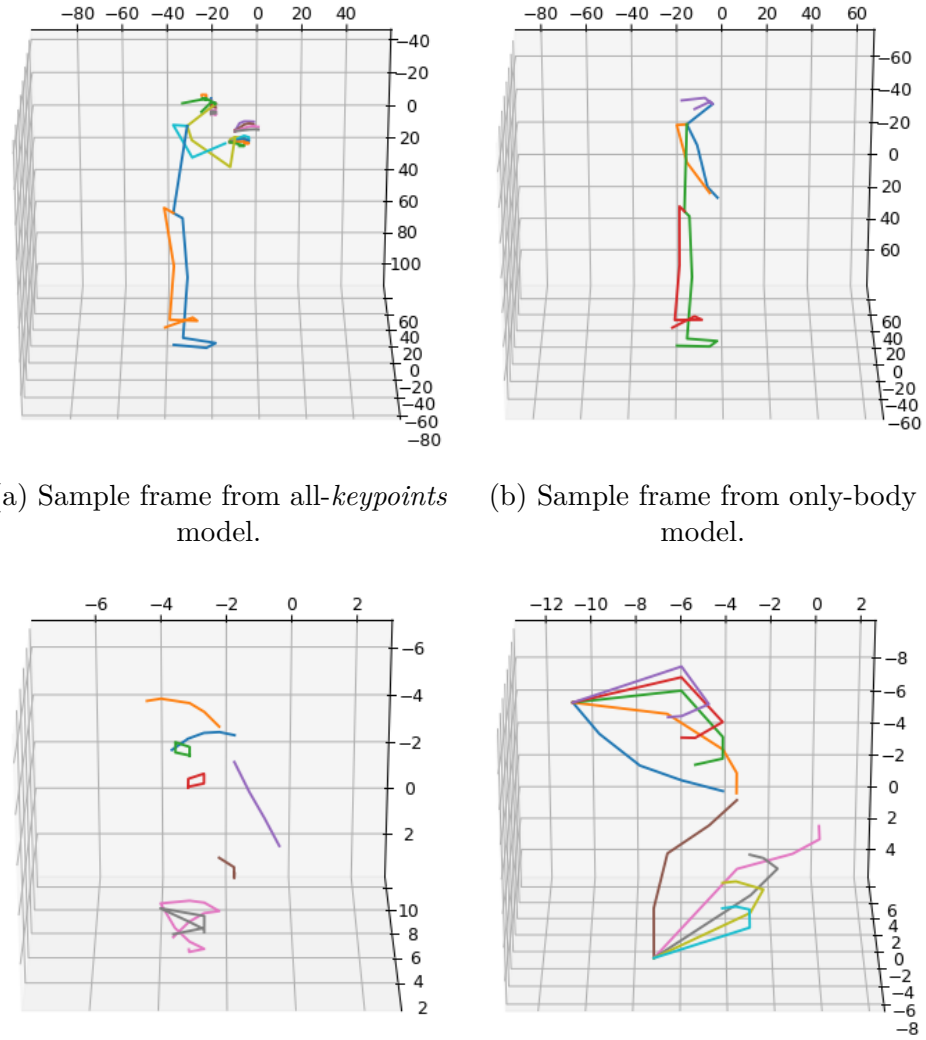


(c) Sample frame from only-face model.



(d) Sample frame from only-hands model.

Figure 5.7: Predictions when using one video.



(a) Sample frame from all-*keypoints* model.

(b) Sample frame from only-body model.

(c) Sample frame from only-face model.

(d) Sample frame from only-hands model.

Figure 5.8: Corresponding ground-truth of

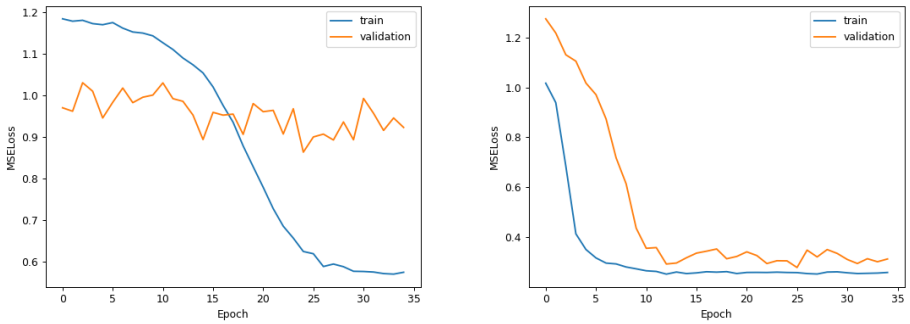
It is shown that the results for the classification task are far worse than the corresponding from regression task. It is clear from the NLL loss plots (on the following section) that models are perfectly capable of memorizing the training data. Nevertheless, it seems that the generalization is more difficult in this case.

In the case of the all-*keypoints* model, the same phenomenon as in regression task is observed: hands and face badly estimated and the body a little better. However, in this case the predictions for part-dedicated models were far worse than the corresponding in regression task. The scaling factors inferred for the classification task de-normalization

were only the range of z (as explained in above chapters), and contrarily to the results, they were perfectly estimated. None of the results obtained in classification task notebooks needed the manual extra scaling.

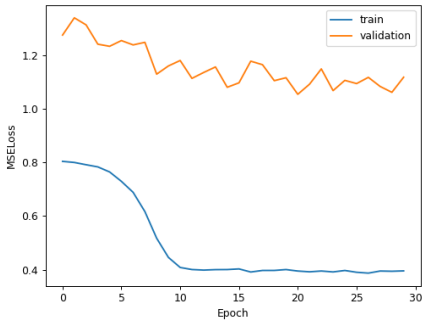
5.5 Training and validation loss curves

Experiments using one video

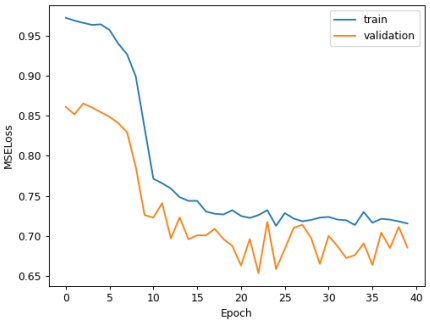


(a) All-*keypoints* model.

(b) Only-body model.



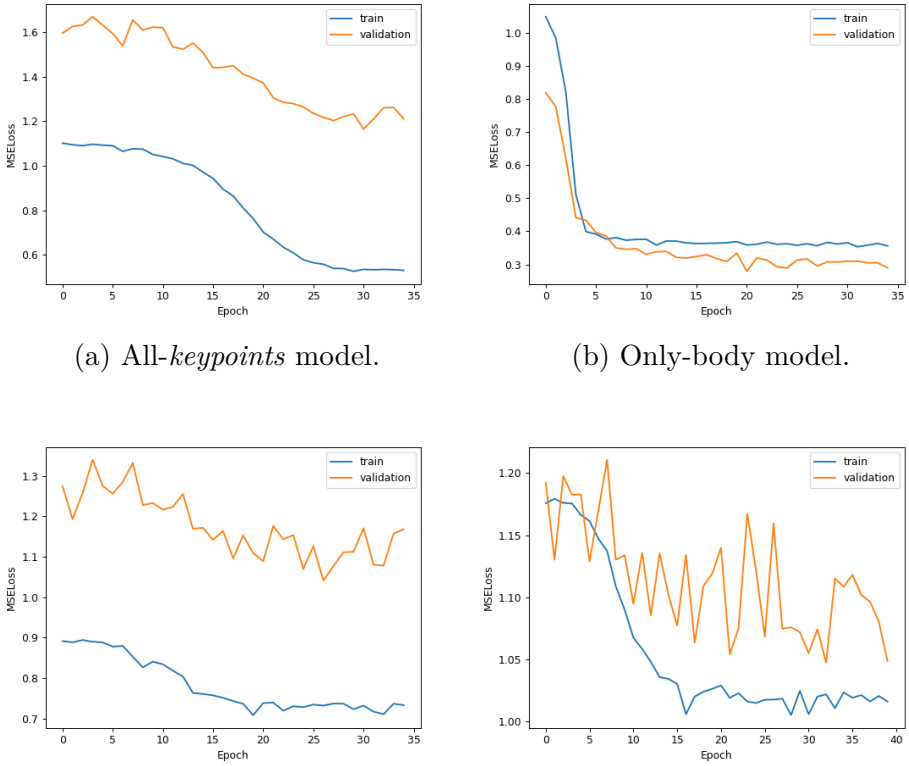
(c) Only-face model.



(d) Only-hands model.

Figure 5.9: MSE Loss vs epochs when using one video.

Experiments using all the videos from one interpreter



(a) All-*keypoints* model.

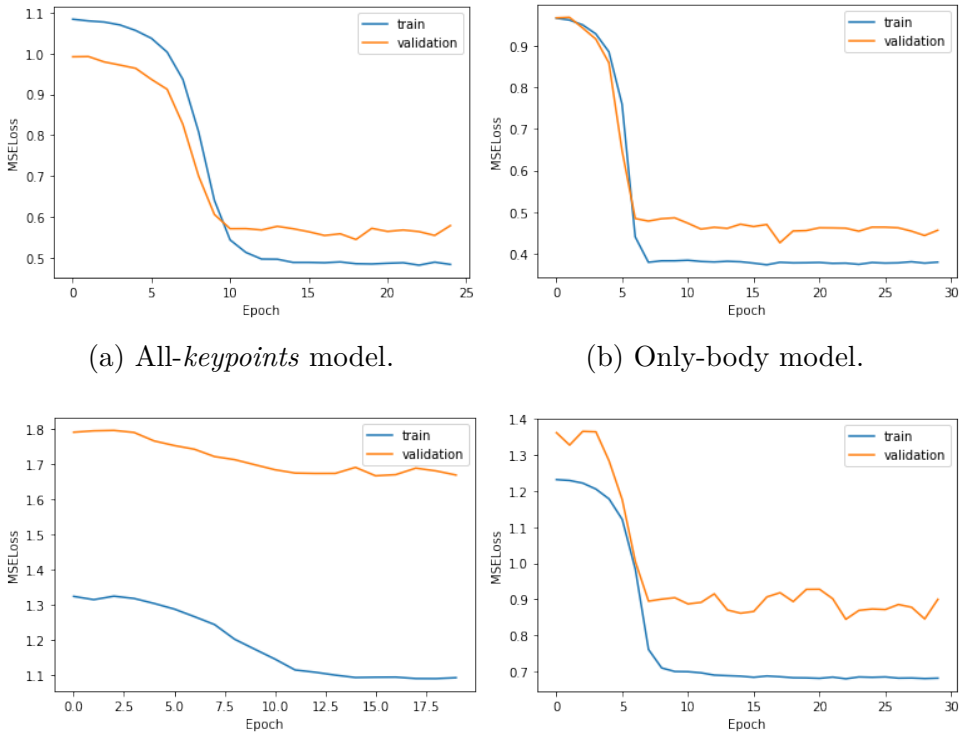
(b) Only-body model.

(c) Only-face model.

(d) Only-hands model.

Figure 5.10: MSE Loss vs epochs when using all the videos from a single interpreter.

Experiments using all the videos



(a) All-*keypoints* model.

(b) Only-body model.

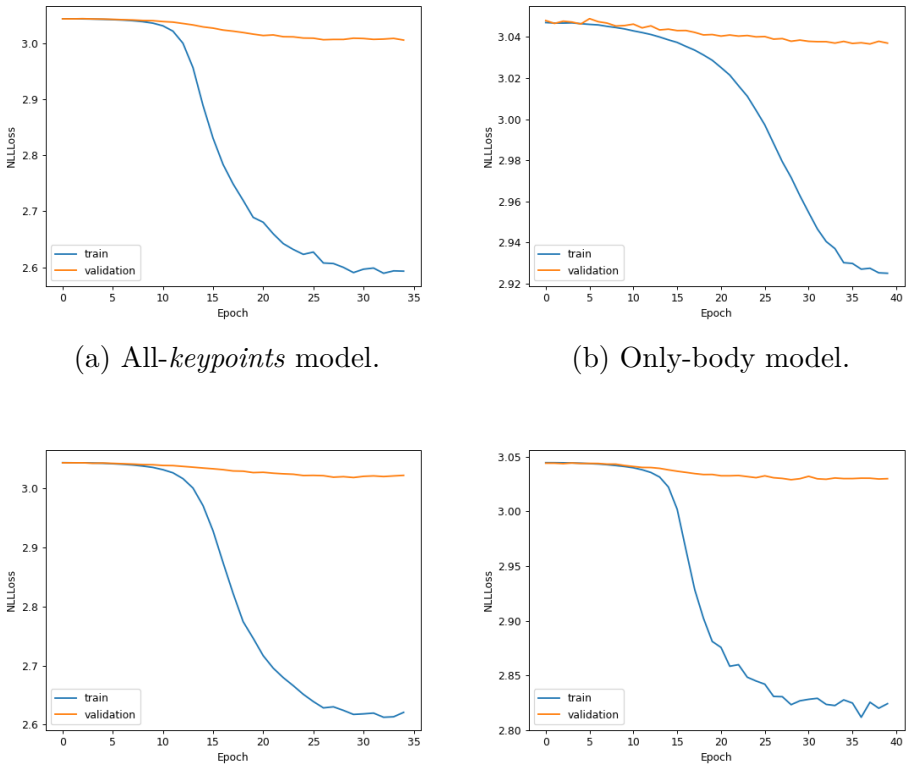
(c) Only-face model.

(d) Only-hands model.

Figure 5.11: MSE Loss vs epochs when using all the videos in the database.

5.5.1 Classification approach

Results of using just one video



(a) All-*keypoints* model.

(b) Only-body model.

(c) Only-face model.

(d) Only-hands model.

Figure 5.12: NLL Loss vs epochs when using one video.

In both approaches, it is clear that the network is capable of memorizing the training data with the proper learning rate and enough epochs. But when trying that not only the training loss but the validation loss decreases, classification models are not as successful as regression ones.

In classification, the validation loss noticeably descends in every case, but much more slowly than the training loss. It couldn't be found the appropriate tuning of parameters that led to a better generalization. That is why, given the time restriction, there weren't more tests performed on classification models.

5.6 Discussion

The results displayed on this chapter show that the proposed first solution works well enough for general body estimation. Compared to results from related work, however, it falls a bit behind. But none of that works neither have as high demanding restrictions and requirements as this project nor have been tested against the same dataset.

Results shown for models using all the videos are quite good and they are able to replicate a plausible human pose. Despite that, sign language demands high accuracy on the gestures, mostly hands', and that is why this work needs to be improved before being used for sign language annotation.

It is needed to say that larger architectures than the explained on the designing chapter were tested. That is, architectures with higher number of neurons for hidden states, more layers and even with bidirectional LSTM layers. However, they didn't accomplish better results than the ones shown here. So that isn't the right direction to go for further improvements -at least for now. Several suggestions on how to continue this work will be explained on the conclusions chapter 8.

Chapter 6

Budget

This is a research project for a final bachelor's thesis that is mainly of software development. Hence, since all the software used was publicly available and the hardware was mainly provided by the author and the university, there were almost no real costs.

However, for this section the project is considered as if it was conducted from zero resources and with wages for the contributors -in order to calculate the cost of the project if done independently. Thus, author's and DCU computers used will be included at cost computation, and the author will be presented as an intern researcher (with typical wage of 8€/hour). It also will be included the publishing fee as if the work was to be published on a journal. Online provided hardware such as Google Colabs won't be included because they can be used freely by anyone.

Table 6.1: Machinery cost

Concept	Cost/unit	Amortisation/unit	Units	Total amortisation	Lifetime	4 month amort.
Personal Computers	1000€	900€	3	2700€	7 years	128,57€

Table 6.2: Wages

Type	Wage	Number	Month (without SS)	Month (with SS)	Total (4 month)
Supervisor	1000€	2	2000€	2680€	10400€
Intern researcher	597,01€	1	597,01	800	3200€
External consultant	-	-	-	-	0€
Total		5	2597,01€	3480€	**13920€**

Table 6.3: General expenses

Concept	Cost (4 months)
Energy supply	256€
Internet	160€
Computers	2871.43€
Total	**3287.43€**

Table 6.4: Total cost

Type	**Cost (4 months)**
General expenses	3287,43€
Wages	13920€
Publishing fee	100€
Total	**17307,43€**

Chapter 7

Ethics

In a research project like this, a series of ethical considerations must be and are addressed:

1. The participation of the author, the supervisors and any other contributors was voluntary. The first party was voluntarily enrolled to University module this project belongs to; the second agreed in participating on this research project; and the third only contributed at their will.
2. The participation of the respondent was done under informed consent, since it was accepted the project proposal.
3. The use of offensive, discriminatory, or other unacceptable language needs to be and **is** avoided along all the project work.
4. Privacy and anonymity or respondents is of a paramount importance.
5. Acknowledgement of related and previous works of other authors used in any part of the dissertation can be is correctly provided (see 2 and references).
6. The highest level of objectivity in discussions and analyses is maintained throughout the research, and the objectives and aims of the project are explained without any deception or exaggeration.

Furthermore, this project strives to create and promote social good and help to the Deaf[1] community, without harming any other social environment. It could be objected the commonly ethical issue found in Machine Learning, that is, the replacement of a human profession by the designed ML model. However, in this case, professional interpreters would be still needed after once, not only this work, but the bidirectional translator from speech to sign language was completed. Because the motivation for that translator is to use it on content or situations where the professionals are not available. And, in any case, signers would still be needed to properly build and improve the ML solutions.

[1]I follow the recognized convention of using the upper-cased word Deaf to refers tothe culture and describe members of the community of sign language users and, in contrast, the lower-cased word deaf to describe the audiological state of a hearing loss

This work helps to the process of constructing a bidirectional translator between speech and sign language that not only will not replace the work of professional translators, but rather it will contribute on a wider integration of the Deaf into many social environments. And that is a huge positive ethical impact.

Chapter 8

Conclusions and future work

The conclusions that can be extracted from this work, as well as suggestions on how to continue it, are explained in this chapter. Overall, the main conclusion is that the proposed design, specifically the first approach (regression task), works well enough to predict general body poses, and in the case of part-specific models, even some plausible faces and hands. However, sign languages need to be precise on its poses and so this project cannot be used to label databases like How2Sign yet. The obtained results could be used in a lot less demanding context, but it cannot be said that the project is able to successfully complete the task it was designed for.

Nevertheless, the results show that the network is actually learning, in fact, it easily learns the most common pose of every body structure. This indicates potential for better predictions. The solutions proposed here yet have room for improvement, and they can be continued with some of the suggestions explained on the next section or they can serve as starting line for another solution. Then, detailing the conclusions, it can be said that:

- This project has an important impact on addressing the problem of pose estimation from 2D to 3D for sign language, because it contributes in setting a baseline for domain-restricted and highly precise skeleton estimation tasks. Specifically, it contributes in deciding whether LSTM based approaches have enough potential to perform the task.
- Pose estimation for highly variant structures such as hands, is the most complicated part in this kind of problems.
- The fact of having different interpreters (hence, different body structures) on the dataset has not had much negative impact -and normalization helped in being so. Rather, having that many more data have improved some of the results, especially all-*keypoints* ones. So, the difference between body structures is less important than the diversity of poses.
- The classification task solution shows a much less noisy training process and the variance between runs for ”only one video” experiment was lower. That leads to the conclusion that, possibly with more regularization methods and a better fine-tuning

of the parameters, it could achieve similar performance as the regression approach -not much better, though. However, it appears more difficult to train.

- Since increasing network capacity does not improve much the results, it can be concluded that there's not much more to extract from the current feature vectors (from plain x and y coordinates). Adding some other features that include visual information should be useful for further improvements (see next section).

It is clear that Deep Learning techniques are really useful in complex tasks where one knows it exists a correlation between features and target, but does not know how to model it exactly -like this one. The key factors in this kind of works are: selecting the appropriate feature space to run the inference from, and having enough data.

Therefore, despite some of the results being good enough to be used for actual sign language annotation, the majority of them lack the precision to do so, and they are too variant with respect to the dataset split. All being said, this work aids on the advancement towards a more socially-aware society, since it contributes on the progress made to help Deaf community have more accessibility.

8.1 Future Work

Some guidelines and suggestions for continuing this work and make further improvements:

- Despite the conclusion on the last point above, it would be interesting to perform better experiments with larger architectures (and maybe try Bidirectional LSTM) -for that, much more powerful hardware is needed.
- The results from the part-dedicated models (body, face and hands) are better than the all-*keypoints* model ones. However, to actually annotate a dataset those results need to be fused. Then, a post-processing code needs to be develop to perform the fusion with the proper scale for each part. Once done, the combined results can be plotted and the metrics can be computed, allowing a better comparison with the all-*keypoints* model.
- As seen that part dedicated models are more precise, a good try to improve hands prediction would be use a dedicated model for each hand. That would make the centroids of the frame to be always in the actual hand, and could lead to a better prediction.
- For classification task, use regularization methods and use much more powerful hardware to try experiments with all the videos.
- Since it is concluded that there is not much more information to extract from plain x and y coordinates features, a good continuation for the project would be to include some new features to the input. For example, the How2Sign dataset has also labels for the visibility of *keypoints*, i.e. whether each *keypoint* is visible from a given camera. Then, including the visibility annotations from the frontal camera on the input tensors could add quite relevant information.

- As suggested in some related work [9], use a slightly different LSTM-based architecture. Instead of producing the third coordinates prediction for a frame in each time-step, where the information from the previous time-steps are stored on the hidden states, use an encoder-decoder approach: one LSTM to serve as encoder, which does not output anything; and then, another LSTM (decoder) that takes the last hidden states from the encoder (containing all the frames information) as the initial hidden states, and outputs one frame prediction at a time using the previous time-step output as input for each time-step.

Bibliography

[1] C. Warke. (2019) Simple feed forward neural network code for digital handwritten digit recognition. [Online]. Available: https://medium.com/random-techpark/simple-feed-forward-neural-network-code-for-digital-handwritten-digit-recognition-a234955103d4

[2] S. Rathor. (2018) Simple rnn vs gru vs lstm: Difference lies in more flexible control. [Online]. Available: https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57

[3] Z. Cao, G. Hidalgo, T. Simon, S. Wei, and Y. Sheikh. (2018) Openpose. [Online]. Available: https://github.com/CMU-Perceptual-Computing-Lab/openpose

[4] R. Karim. (2018) Animated rnn, lstm and gru. [Online]. Available: https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45

[5] A. Duarte, "Cross-modal neural sign language translation," in *Proceedings of the 27th ACM International Conference on Multimedia (MM'19)*, ACM, New York, NY, USA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3343031.3352587

[6] H. Joo, T. Simon, X. Li, H. Liu, L. Tan, L. Gui, S. Banerjee, T. Godisart, B. Nabbe, I. Matthews *et al.*, "Panoptic studio: A massively multiview system for social interaction capture," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 1, pp. 190–204, 2017.

[7] H. Joo, H. Liu, L. Tan, L. Gui, B. Nabbe, I. Matthews, T. Kanade, S. Nobuhara, and Y. Sheikh, "Panoptic studio: A massively multiview system for social motion capture," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov 1997.

[9] M. Rayat Imtiaz Hossain and J. J. Little, "Exploiting temporal information for 3d pose estimation," *arXiv*, pp. arXiv–1711, 2017.

[10] K. Lee, I. Lee, and S. Lee, "Propagating lstm: 3d pose estimation based on joint interdependency," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[11] J. Zelinka and J. Kanis, "Neural sign language synthesis: Words are our glosses," in *The IEEE Winter Conference on Applications of Computer Vision (WACV)*, March 2020.

[12] W. C. S. Jr, "Sign language structure: An outline of the visual communication systems of the american deaf," *Journal of deaf studies and deaf education*, vol. 10, no. 1, pp. 3–37, 2005.

[13] E.-J. Ong, H. Cooper, N. Pugeault, and R. Bowden, "Sign language recognition using sequential pattern trees," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, Providence, Rhode Island, USA, 2012.

[14] N. C. Camgoz, S. Hadfield, O. Koller, H. Ney, and R. Bowden, "Neural sign language translation," in *Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7784–7793.

[15] S.-K. Ko, C. J. Kim, H. Jung, and C. Cho, "Neural sign language translation based on human keypoint estimation," *Applied Sciences*, vol. 9, no. 13, 2019.

[16] R. San-Segundo, J. M. Montero, J. Macías-Guarasa, R. Córdoba, J. Ferreiros, and J. M. Pardo, "Proposing a speech to gesture translation architecture for spanish deaf people," *Journal of Visual Languages & Computing*, vol. 19, no. 5, pp. 523–538, 2008.

[17] S. Stoll, N. C. Camgöz, S. Hadfield, and R. Bowden, "Sign language production using neural machine translation and generative adversarial networks," in *BMVC*, 2018, p. 304.

[18] S. Cox, M. Lincoln, J. Tryggvason, M. Nakisa, M. Wells, M. Tutt, and S. Abbott, "Tessa, a system to aid communication withdeaf people," in *5th International ACM conference on Assistive technologies. ACM*, 2002, pp. 205–212.

[19] Y. Zhou, M. Habermann, W. Xu, I. Habibie, C. Theobalt, and F. Xu, "Monocular real-time hand shape and motion capture using multi-modal data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 5346–5355.

[20] A. Boukhayma, R. d. Bem, and P. H. Torr, "3d hand shape and pose from images in the wild," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 843–10 852.

[21] C. Zimmermann and T. Brox, "Learning to estimate 3d hand pose from single rgb images," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 4903–4911.

[22] P. Dreuw, T. Deselaers, D. Keysers, and H. Ney, "Modeling image variability in appearance-based gesture recognition," in *ECCV workshop on statistical methods in multi-image and video processing*, 2006, pp. 7–18.

[23] D. McKee, D. McKee, and E. Pailla, "Nz sign language exercises," *Deaf Studies Department of Victoria University of Wellington*, vol. 5, 2018.

[24] D. Pavllo, C. Feichtenhofer, D. Grangier, and M. Auli, "3d human pose estimation in video with temporal convolutions and semi-supervised training," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7753–7762.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[26] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan 2015.

[27] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural Networks*, vol. 6, pp. 861–867, Jan 1993.

[28] P. N. Stuart J. Russell, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, oct 1986.

[30] S. Dupond, "A thorough review on the current advance of neural network structures," *Annual Reviews in Control*, vol. 14, pp. 200–230, 2019.

[31] M. Mozer, "A focused backpropagation algorithm for temporal pattern recognition," *Complex Systems*, vol. 3, 01 1995.

[32] W. Feng, N. Guan, Y. Li, X. Zhang, and Z. Luo, "Audio visual speech recognition with multimodal recurrent neural networks," 05 2017, pp. 681–688.

[33] P. Goyal, S. Pandey, and K. Jain, *Unfolding Recurrent Neural Networks*. Berkeley, CA: Apress, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4842-3685-7_3

[34] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[35] R. Sharpe, *Differential Geometry: Cartan's Generalization of Klein's Erlangen Program*. Springer Science & Business Media, November 2002.

[36] J. VanderPlas, *Python Data Science Handbook*. O'Reilly Media, November 2016.

[37] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[39] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay," *CoRR*, vol. abs/1803.09820, 2018. [Online]. Available: http://arxiv.org/abs/1803.09820

[40] S. Ginosar, A. Bar, G. Kohavi, C. Chan, A. Owens, and J. Malik, "Learning individual styles of conversational gesture," in *Computer Vision and Pattern Recognition (CVPR)*.

Appendices

Appendix A

Appendix A

The rotation matrix for a rotation R by angle θ around an axis $\mathbf{u} = (u_x, u_y, u_z)$:

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

in the case of this project, it the rotation is applied around the y ($\mathbf{u} = (0,1,0)$) axis, that simplifies a lot the above matrix.

$$R = \begin{bmatrix} \cos\theta & (1-\cos\theta) & \sin\theta \\ (1-\cos\theta) & \cos\theta + (1-\cos\theta) & (1-\cos\theta) \\ -\sin\theta & (1-\cos\theta) & \cos\theta \end{bmatrix}$$

Every triplet of coordinates (in the form of a vector) is multiplied by this matrix to rotate them θ degrees.

SciPy library provides an implementation of that matrix and the code for applying it to the tensor of this project is:

```
r = R.from_euler('y', 110, degrees=True)
shapes = dataset.shape
dataset = dataset.reshape(-1, 2)
groundtruth = groundtruth.reshape(-1,1)
xyz = np.concatenate((dataset, groundtruth), axis=1)
xyz = r.apply(xyz)
dataset, groundtruth = xyz[:, :2].reshape(shapes), xyz[:,2].reshape(
    shapes[0], shapes[1],138)

print(dataset.shape, groundtruth.shape)
```

Appendix B

Apendix B

Implementation of MPJPE metric. The first function is in charge of aligning predicted and ground-truth skeletons (it subtracts their corresponding root *keypoints*), and the second computes the MPJPE for a batch once predictions and ground-truth have been aligned.

```
def substract_root_PJPE(output):
    jf = torch.chunk(output[:, :, :70], max_seq, dim=1)
    jhl, jhr = torch.chunk(output[:, :, 70:91], max_seq, dim=1), torch.
   chunk(output[:, :, 91:112], max_seq, dim=1)
    jb = torch.chunk(output[:, :, 112:], max_seq, dim=1)
    joints_merged = []
    roots = [33, 0, 0, 8]
    for i, joints in enumerate((jf, jhl, jhr, jb)):
        n_joints = []
        for chunk in joints:
            n_joints.append(chunk.sub(chunk[:,:,roots[i]].unsqueeze(2)))
        joints_merged.append(torch.cat(tuple(n_joints), dim=1))
    joints_merged = torch.cat(tuple(joints_merged), dim=2)
    return joints_merged
```

```
def mpjpe(rooted_o, rooted_l, seq_lens):
    MPJPE = []
    for i in range(len(seq_lens)):
        MPJPE.append(rooted_o[i,:int(seq_lens[i])].sub(rooted_l[i,:int(
   seq_lens[i])]).abs().mean().item())

    return np.mean(MPJPE)
```

Appendix C

Apendix C

Table C.1: MSE loss, MPJPE and PCK scores for every model in regression task.

		MSE loss			**MPJPE**			**PCK**		
		Train	Validation	Test	Train	Validation	Test	Train	Validation	Test
All kp	1 video	0.5143	1.0229	0.9558	0.2869	0.2869	0.4019	45.87%	17.44%	30.52%
	1 interpreter	0.6005	1.2401	**0.8049**	0.3125	0.4007	0.3446	42.85%	27.59%	31.97%
	All videos	0.5901	0.9081	0.9221	0.2573	0.2641	**0.2975**	43.83%	33.18%	**36.32**%
Body kp	1 video	0.2752	0.3072	0.5489	0.3384	0.3368	0.4307	55.36%	31.21%	39.23%
	1 interpreter	0.4507	0.4072	**0.3917**	0.3878	0.4021	**0.3903**	52.69%	38.13%	**42.46**%
	All videos	0.4495	0.6385	0.9379	0.4473	0.5099	0.5125	44.16%	33.94%	28.51%
Face kp	1 video	0.4255	1.1641	**0.9352**	0.3703	0.7517	0.6928	62.16%	28.14%	**46.67**%
	1 interpreter	0.9561	1.1698	1.3249	0.3451	0.4446	**0.4044**	31.30%	25.49%	29.52%
	All videos	1.0318	1.5528	2.1018	0.3456	0.4552	0.4807	30.20%	29.84%	25.86%
Hands kp	1 video	0.6909	0.8005	**1.3475**	0.7736	0.8851	0.9168	27.64%	25.28%	23.15%
	1 interpreter	1.0652	1.2074	1.4379	0.4654	0.4665	**0.5073**	13.79%	8.02%	11.79%
	All videos	0.8660	1.1167	1.9615	0.4317	0.4439	0.6014	13.81%	11.62%	13.47%

Table C.2: NLL loss and accuracy score for every model in classification task.

	NLL loss			**Accuracy**		
	Train	Validation	Test	Train	Validation	Test
All kp, 1 video	2.5792	3.0068	3.0096	15.82%	15.59%	15.37%
Body kp, 1 video	2.9077	3.0368	3.0350	15.39%	14.87%	14.49%
Face kp, 1 video	2.6295	3.0233	3.0223	15.85%	15.69%	15.40%
Hands kp, 1 video	2.8272	3.0303	3.0298	10.60%	10.41%	10.25%

Table C.3: Standard deviation of C.1 metrics.

		MSE loss stdev			**MPJPE stdev**			**PCK stdev**		
		Train	Validation	Test	Train	Validation	Test	Train	Validation	Test
All kp	1 video	**0.0534**	**0.0893**	**0.0897**	0.0561	**0.0264**	**0.0294**	3.32%	**3.60%**	**4.25%**
	1 interpreter	0.1231	0.2481	0.2463	**0.0113**	0.0572	0.0596	2.87%	5.49%	4.78%
	All videos	0.1069	0.5891	0.2025	0.0230	0.0690	0.0678	**0.32%**	7.89%	5.15%
Body kp	1 video	**0.0223**	**0.0776**	0.1847	**0.0154**	**0.0311**	0.0947	0.69%	1.95%	10.43%
	1 interpreter	0.1654	0.1344	**0.0737**	0.0658	0.0773	**0.0592**	0.50%	6.77%	**1.36**%
	All videos	0.0622	0.1648	0.4556	0.0269	0.1093	0.0638	**0.12**%	13.20%	13.08%
Face kp	1 video	**0.0621**	0.3775	0.4467	0.0294	0.1594	0.1237	0.83%	6.97%	11.57%
	1 interpreter	0.1968	**0.0387**	**0.1816**	0.0637	0.0696	**0.0393**	0.88%	2.21%	2.78%
	All videos	0.1207	0.1232	0.4789	**0.0163**	**0.0255**	0.0949	**0.13**%	**0.80**%	3.79%
Hands kp	1 video	**0.0359**	**0.1025**	0.3017	0.0277	**0.0270**	0.0893	0.17%	**0.97**%	1.57%
	1 interpreter	0.0995	0.3362	0.1999	**0.0199**	0.0296	0.0714	**0.02**%	1.09%	**0.64**%
	All videos	0.3641	0.6697	**0.0935**	0.1408	0.0856	0.0417	0.29%	1.22%	0.96%

As expected, the less variant results are the ones from one video experiments, since however the dataset is split all frames come from the same video. Contrarily, the face only results show less standard deviation when using more videos, because the face is the less variant structure -thus the more videos the better generalization and the lower results variance. Finally, taking a look into only-hands model results, it can be noted that they have an experiment that clearly has less standard deviation. The reason behind that may be because hands are the most variant structure and that makes that where they come from (same or different videos) less important on the stability of results.

Table C.4: Standard deviation of C.2 metrics.

	NLL loss			**Accuracy**		
	Train	Validation	Test	Train	Validation	Test
All kp, 1 video	0.0123	0.0022	0.0014	0.36%	0.36%	0.32%
Body kp, 1 video	0.0161	0.0003	0.0014	1.00%	0.94%	0.95%
Face kp, 1 video	0.0077	0.0014	0.0024	0.16%	0.16%	0.16%
Hands kp, 1 video	0.0084	0.0022	0.0023	0.26%	0.22%	0.17%