Proceedings of the 7th International Workshop on Sign Language Translation and Avatar Technology (SLTAT 7), pages 73–78
Language Resources and Evaluation Conference (LREC 2022), Marseille, 20-25 June 2022

A Software Toolkit for Pre-processing Sign Language Video Streams

Fabrizio Nunnari
German Research Center for Artificial Intelligence (DFKI)
Saarland Informatics Campus D3.2
fabrizio.nunnari@dfki.de

Abstract

We present the requirements, design guidelines, and the software architecture of an open-source toolkit dedicated to the pre-processing of sign language video material. The toolkit is a collection of functions and command-line tools designed to be integrated with build automation systems. Every pre-processing tool is dedicated to standard pre-processing operations (e.g., trimming, cropping, resizing) or feature extraction (e.g., identification of areas of interest, landmark detection) and can be used also as a standalone Python module. The UML diagrams of its architecture are presented together with a few working examples of its usage. The software is freely available with an open-source license on a public repository.

Keywords: sign language, video pre-processing, open source toolkit, software engineering

1. Introduction and Related Work

In these years, there is a consistent amount of public-funded research on sign language recognition and translation. In particular, two EU-funded projects, SignOn[1] (Shterionov et al., 2021) and EASIER[2], attempt to provide bi-directional translation from and to spoken and sign languages of different European languages.
In addition to its contribution to the EASIER project, the German Research Center for Artificial Intelligence (DFKI) works on the nationally funded AVASAG[3] (Nunnari et al., 2021a) and SocialWear[4] (Nunnari et al., 2021b) projects. All of those projects share the use of the latest generation of artificial intelligence techniques, based on neural networks, for video analysis. In all cases, video material needs to be analysed and pre-processed before being fed to convolutional neural networks (CNN) architectures.

In machine learning, data pre-processing is a common task that ensures some form of data normalization and possibly some pre-computation of features that facilitates the training of the neural architectures.
In the realm of sign language, such video pre-processing might include identifying body parts (hands, face, lips, eyes) and cropping the portion of video frames containing a higher resolution of such items. Other pre-processing steps might include the identification of landmarks (i.e., transiting from pixel-based features to 2D/3D vector information).
However, despite being recognized as a necessary step, pre-processing is performed again and again among different projects using highly customized scripts that are hardly reusable across projects or datasets. This is due to many factors. One of them is the storage format of the video material, which is for example available as images sequence in the PHOENIX corpus (Forster et al., 2012) and as compressed videos in the Hamburg DGS corpus (Hanke et al., 2020). Other typical differences relate to different ways of organizing and naming the video sources.

For those reasons, the DFKI started a software project with the goal of collecting in a single open-source repository all of the algorithms broadly needed to perform pre-processing of sign language videos, to maximize reusability across projects, but leaving out the specific details that are hindering its portability.
The project is called Sign Language Video processing Tools[5] and it is available as a public open-source repository on the popular GitHub platform. The software package is essentially a collection of command-line tools, usable also as Python modules, developed by aggregating several popular open-source libraries and tools such as ffmpeg[6], MediaPipe (Lugaresi et al., 2019), OpenCV (Bradski and Kaehler, 2000), MTCNN (Xiang and Zhu, 2017). Figure 1 shows some examples of the toolkit in action.
The added values of this toolkit, compared to directly using directly its underlying libraries, are described in detail in section 2. Section 3 lists the tools implemented so far. Finally, section 4 summarizes the paper and describes future work.

2. Framework Goals and Design

The framework has been designed to fulfill the three following requirements:

1. Usable both as command line tools as well as Python functions;

[1] https://signon-project.eu
[2] https://www.project-easier.eu
[3] https://avasag.de
[4] https://affective.dfki.de/socialwear-bmbf-2020-2024/

[5] Sign Language Video Processing Tools code repository: https://github.com/DFKI-SignLanguage/VideoProcessingTools
[6] https://ffmpeg.org

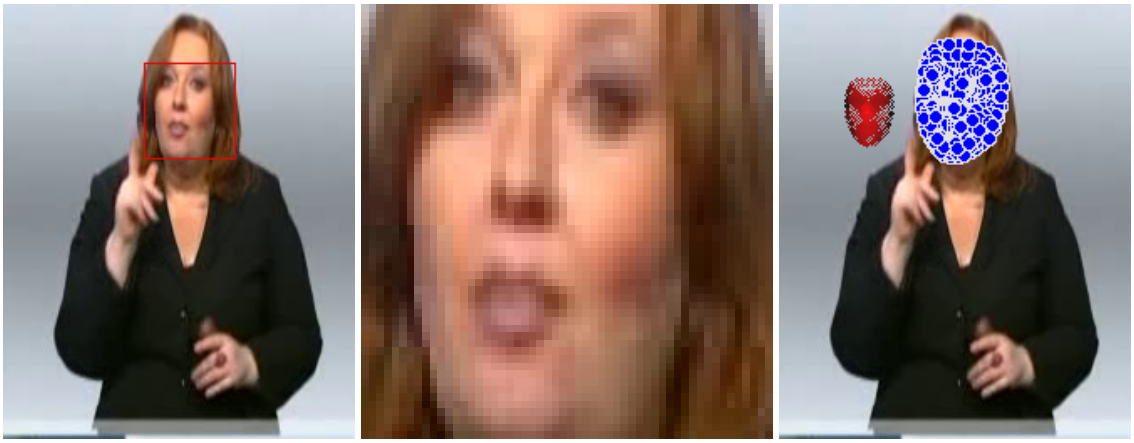

Figure 1: Examples of the toolkit applied to a test video of the PHOENIX corpus. From left to right: face bounds detection, cropping, detection, and normalization of facial landmarks. For the latter, the blue dots are the landmarks detected by MediaPipe, while the red dots are the landmarks after normalization of the head orientation.

2. Support video streams both as encoded videos and as image sequences;

3. The parameters of the command-line tools are designed to be concatenated with build automation tools.

In the following, we describe how those requirements have been addressed.

As for **Requirement 1**, all of the video processing tools are organized as stand-alone Python modules. Figure 2 shows a UML diagram of the top-level `slvideotools` package, which contains a sub-package for each of the available tools. Packages and sub-packages are implemented as Python sub-modules. Every sub-module acts as *wrapper* for a specific functionality. Each wrapping sub-module contains a top-level code acting as the `main` execution point, parsing the command line arguments, and invoking the corresponding video processing function; the latter has the same name as the containing sub-module.
For example, the `crop_video` tool is implemented in the `slvideotools.crop_video` sub-package and can be invoked as CLI command:

```
python -m slvideotools.crop_video\
    --inframes myface.mp4\
    --inbounds face_bounds.json\
    --outframes cropped_frames/
```

At the same time, the function `crop_video(...)` is available within pure Python code and can be imported and reused:

```
from slvideotools.crop_video import crop_video
from slvideotools.datagen import\
    create_frame_producer, create_frame_consumer

with create_frame_producer(
        dir_or_video="myface.mp4") as prod,\
     create_frame_consumer(
        dir_or_video="cropped_frames") as cons:

    with open("face_bounds.json", "r") as bounds_fp:
        bounds = json.load(bounds_fp)

    crop_video(frames_producer=prod,
        bounds_tuple=bounds,
        frames_consumer=cons)
```

The next paragraph describes what are frame producers and consumers.

Requirement 2 Sign language video material is often stored as a video stream. However, in some cases, to more easily feed single frames to a convolutional classifier, or to avoid video compression artifacts, videos are stored as a sequence of single images, usually collected inside a folder. To seamlessly support frame sequencing from both videos and image folders, the class structure depicted in Figure 3 was adopted. The production and the consumption of frames are managed through two abstract classes: `FrameProducer` and `FrameConsumer`. Their subclasses are responsible for implementing a method to read frames from a video or a directory, and to store frames in a video or directory. To further facilitate code flexibility, two factory methods (Gamma et al., 1994) create the correct Producer/Consumer subclass by checking if the source or the destination is a video file or a directory. Finally, the Producer/Consumer top classes support the *context management* interface[7], allowing for for automatic resource disposal through the `with ... as ...` statement.
As a result, the typical recipe to process frames from/to video containers or directories is illustrated in the following code snippet.

```
from slvideotools.datagen import\
    create_frame_producer, create_frame_consumer

with create_frame_producer(
        dir_or_video="my/frames/") as prod,\
     create_frame_consumer(
        dir_or_video="my_final_video.mp4") as cons:
```

[7] https://docs.python.org/3/reference/datamodel.html#context-managers

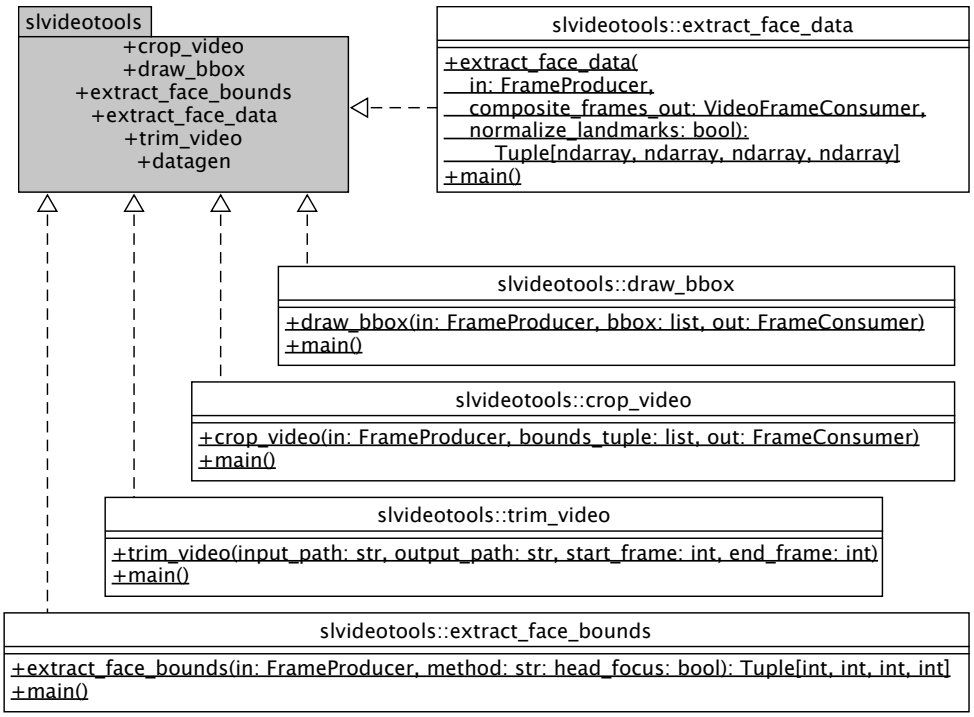

Figure 2: The UML diagram of the data video processing tools package.

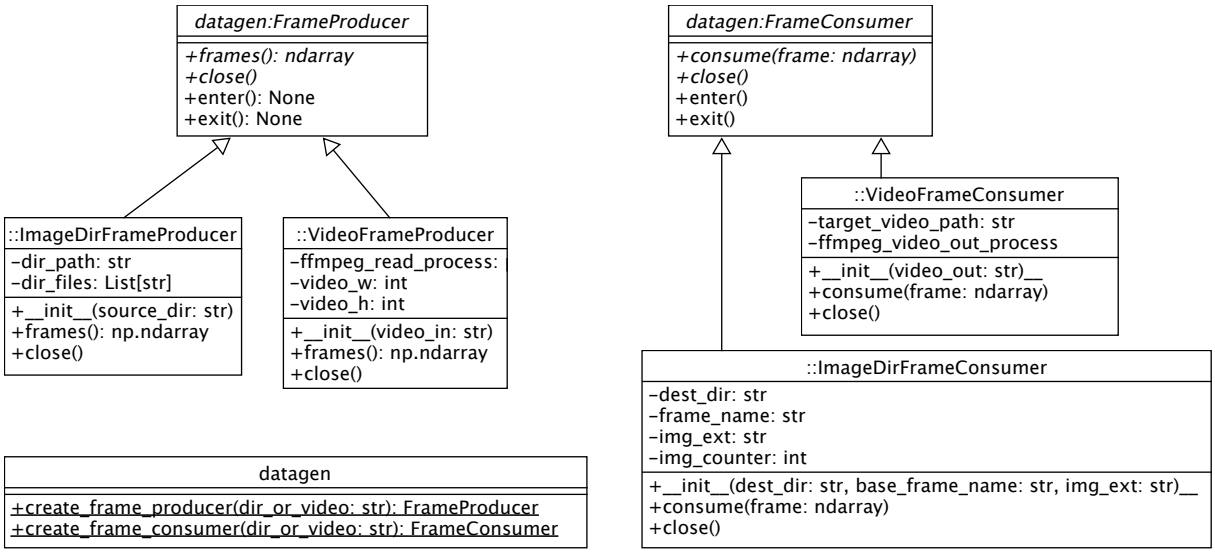

Figure 3: The UML diagram of the data generation subpackage.

```
for in_frame in prod.frames():
    # Process your frame
    # out_frame = ...

    # Feed the frame to output video
    cons.consume(frame=out_frame)
```

Finally, for **Requirement 3**, the command line tools must be usable from wrapping build automation systems like the popular GNU Make[8] or equivalent more advanced systems like Luigi[9].

Video datasets can grow consistently, and running video preprocessing over many video samples can be time and resource-consuming. Hence, when preparing a new dataset, it is important to avoid the repetition of a video processing step (e.g., feature extraction) when not required, e.g., when only a few new samples are added or a few samples are updated (e.g., re-takes).

To fulfill this requirement two simple guidelines were

[8] https://www.gnu.org/software/make/

[9] https://github.com/spotify/luigi

followed. First, the command line interfaces have been designed to take explicit filenames as input and output, avoiding any automatic generation of filenames or any custom convention about naming. For example, automatic filename composition, like appending a timestamp to a base file naming, is avoided. Automatic filename generation can be handy to run several tests while avoiding overriding previous results, but hinders reproducibility and increases the complexity in the maintenance of data folders (also, potentially leading to uncontrolled space occupation). Hence, file names must be unambiguously provided and naming conventions are left to project-specific needs. Second, every script is designed to manage single files (or single folders containing video frames). Iteration over files or directories, which normally requires dealing with peculiar naming conventions, is left to external automation tools.

For example, the following `Makefile` scans a directory for videos with extension `.mp4` and for each video generates a corresponding `.bounds` JSON file with information on the bounding box containing the face of the speaker in the video.

```
# Directory containing the .mp4 files
DIR=videos
# Lists all of the MP4 videos
invideofiles := $(wildcard $(DIR)/*.mp4)
# Compose the names of output .bound files.
boundfiles := $(subst .mp4,.bounds,$(invideofiles))

all: $(boundfiles)
        @echo "Extracted face bounds."

$(boundfiles): $(DIR)/%.bounds: $(DIR)/%.mp4
        @echo "Finding bounds for video $< ..."
        python -m slvideotools.extract_face_bounds \
            --invideo $< \
            --outbounds $@
        @echo "Saved to $@."
```

Every time the `make` command is invoked, each `.bounds` file will be created or updated if the corresponding source video is renewed.

The use of automated dependency checking systems is of extreme advantage when dealing with evolving datasets where single animation clips might be added or updated as the dataset is populated. Using dependency systems ensures that only the minimal set of video processing operations is performed to keep the dataset in a consistent state.

3. Implemented Tools

At the moment of writing, the following command-line tools and functions are fully implemented.

extract_face_bounds This tool analyzes a video clip and identifies frame-by-frame a bounding rectangle containing the face of a speaker. The bounding box (upper-left x and y corner, width, height) of the full video is then computed so that the face is always visible during the whole video. For sign language analysis this approach helps in dealing with frames where the hands cover the face. In those situations, face detection tools fail. By gracefully skipping frames without a visible face, the global bounds containing the face for the whole video can still be inferred from the other video frames, where the face is detected. Two detection methods are currently supported: using the MediaPipe library (Lugaresi et al., 2019), which is faster, or the MTCNN (Zhang et al., 2016), which is more robust for faces at variable distances from the camera. The bounding information is saved into a simple JSON array file.

draw_bbox This tool takes as input a video and bounding box information and produces a new video with the bounding information as an overlay. This is useful for debugging the face detection procedure.

crop_video takes as input a video and bounding box files and outputs a cropped video. This is useful for cropping the face, hands, lips, or any other information which requires zooming on a body part for normalizing image size, increasing resolution, removing noisy information, and thus improving further analysis.

extract_face_data is a complex tool able to extract four kind of information. First, it uses Mediapipe to extract the set of 468 landmarks describing the movement of the face of a subject. Second, it outputs the position of the tip of the nose, which can be used as reference to identify the position of the face in a video frame. Third, it infers the rotation of the head; this is done with vector operations involving the landmarks at the border of the forehead, which are not involved by facial muscle activation (Figure 4 shows the process). Fourth, it calculates a scaling factor, estimating the distance of the face from the camera, useful to normalize the face size before using it in further facial expression recognition algorithms, such as the classification of facial expressions (Savchenko, 2021).

trim_video is useful to trim out initial and ending frames of a video, for example to insulate stroke and hold phases of a motion while removing the preparation and release phases. This script is the only one not using the Producer/Consumer mechanism, but takes as parameters the input and output video file paths, and relies of the `ffmpeg` core functionality to trim a video while avoiding uncompressing and recompressing the stream (which might hinder video quality).

Command	Elapsed	frames/sec
extract_face_bounds (MTCNN)	44s	5,5
extract_face_bounds (MediaPipe)	<1s	>242
draw_bbox	6s	40,33
crop_video	2s	121
extract_face_data	9s	28,89
trim_video	< 1s	> 242

Table 1: Results of the speed test measuring the execution time on a 242-frame sample video.

To measure performances, we monitored the time needed for the execution of all the implemented com-

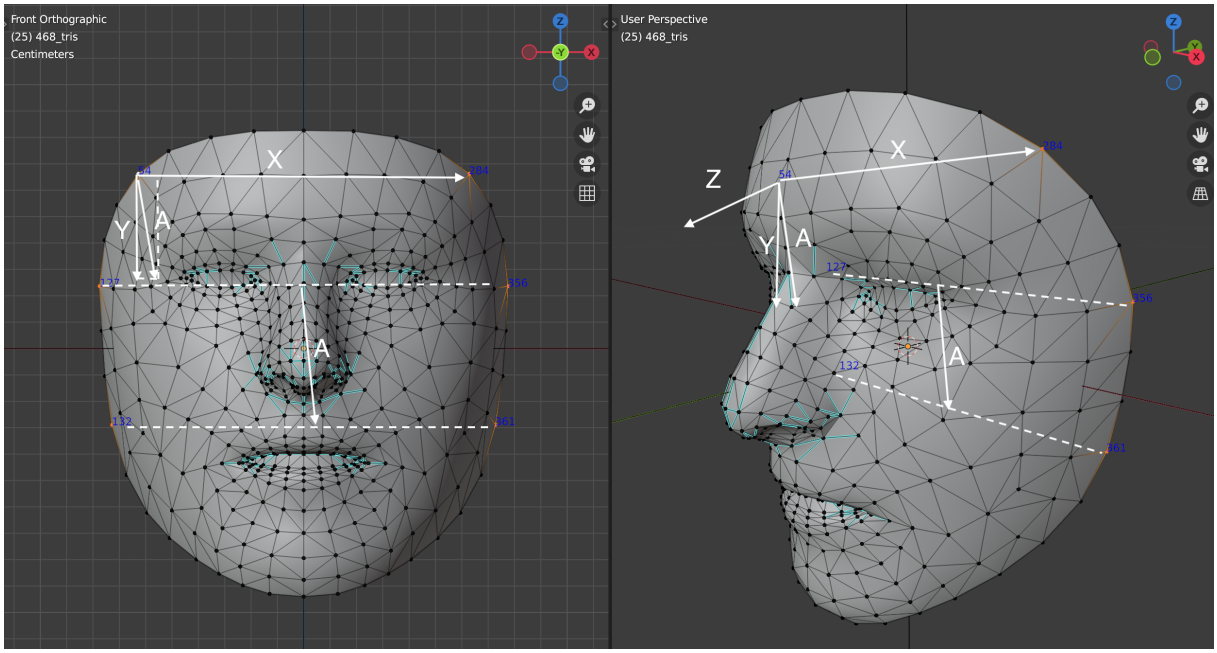

Figure 4: Computing the head rotation from the Mediapipe 3D facial landmarks. To calculate the rotation of the head, we need to define a new orthogonal system with axes X, Y, and Z, which must be already approximately aligned with the absolute reference axes x,y,z when the subject is looking straight forward. The new system must be computed using landmarks that do not move with the facial muscles. The new X axis is computed by considering two landmarks on the forehead. A new A axis is computed considering the midpoints of two horizontal segments joining the sides of the face. Because of noisy information and approximations, A is rarely perfectly orthogonal to X; hence, the new Y is computed from A by subtracting its projection on X. Finally, the new Z is the cross-product between X and Y. The new XYZ reference system is then compared with the global xyz axes to produce a 3x3 rigid rotation matrix.

mands on a sample sign language video. The sample video is the longest found in the PHOENIX 2014-T corpus: 242 frames, resolution 210 X 260 pixels. The reference hardware is a MacBookPro (model 2019) with Intel i9 CPU. Table 1 reports the results. It can be noticed that the slowest process is the extraction of the face bounds with MTCNN, while the same process executed with MediaPipe lasts less than one second. It is worth specifying that the test machine doesn't support GPU acceleration, meaning that MTCNN might perform significantly better on other hardware.

4. Conclusions

We presented the requirements and architecture design of an open-source software toolkit dedicated to the pre-processing of sign language videos. The goal of such a toolkit is to centralize, into a single repository, pieces of code that are often copied and scattered around many projects requiring pre-processing for developing sign language recognition systems. The software architecture of the toolkit has been designed with extensibility in mind.

The toolkit offers already the scripts needed to process face information and will be extended to integrate ad-hoc analysis of other body parts (head, upper body, hands) and features (eye blinks, eye gaze, etc.). Other tools will be likely dedicated to color normalization.

We are updating this toolkit with the code that we developing for three different projects dedicated to sign language analysis and translation. Our goal is to help the research community in speeding up video material pre-processing, without re-implementing it from scratch, and involve other researchers in sharing other pre-processing techniques in a common open repository.

Acknowledgements

The author would like to thank Yasser Hamidullah for his contribution to some parts of the code, and Cristina España-Bonet and Eleftherios Avramidis for their collaboration on the several sign language projects and for the review of this manuscript.

This work has been partially funded by BMBF (German Federal Ministry of Education and Research) within project AVASAG (Avatar-basierter Sprachassistent zur automatisierten Gebärdenübersetzung, grant number: 16SV8491) and project SOCIALWEAR (Socially Interactive Smart Fashion, DFKI Kst 22132), and by the EU Horizon 2020 programme within the EASIER project (Grant agreement ID: 101016982).

5. Bibliographical References

Bradski, G. and Kaehler, A. (2000). Opencv. *Dr. Dobb's journal of software tools*, 3:2.

Forster, J., Schmidt, C., Hoyoux, T., Koller, O., Zelle, U., Piater, J., and Ney, H. (2012). RWTH-PHOENIX-Weather: A Large Vocabulary Sign Language Recognition and Translation Corpus. In *Language Resources and Evaluation*, pages 3785–3789, Istanbul, Turkey, May.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Hanke, T., Schulder, M., Konrad, R., and Jahn, E. (2020). Extending the public DGS corpus in size and depth. In *Proceedings of the LREC2020 9th Workshop on the Representation and Processing of Sign Languages: Sign Language Resources in the Service of the Language Community, Technological Challenges and Application Perspectives*, pages 75–82.

Lugaresi, C., Tang, J., Nash, H., McClanahan, C., Uboweja, E., Hays, M., Zhang, F., Chang, C.-L., Yong, M. G., Lee, J., Chang, W.-T., Hua, W., Georg, M., and Grundmann, M. (2019). MediaPipe: A Framework for Building Perception Pipelines. *arXiv:1906.08172 [cs]*, June. arXiv: 1906.08172.

Nunnari, F., Bauerdiek, J., Bernhard, L., España-Bonet, C., Jäger, C., Unger, A., Waldow, K., Wecker, S., André, E., Busemann, S., Dold, C., Fuhrmann, A., Gebhard, P., Hamidullah, Y., Hauck, M., Kossel, Y., Misiak, M., Wallach, D., and Stricker, A. (2021a). AVASAG: A German Sign Language Translation System for Public Services. In *1st International Workshop on Automatic Translation for Signed and Spoken Languages (AT4SSL)*. Association for Machine Translation in the Americas, August.

Nunnari, F., España-Bonet, C., and Avramidis, E. (2021b). A Data Augmentation Approach for Sign-Language-To-Text Translation In-The-Wild. In Dagmar Gromann, et al., editors, *3rd Conference on Language, Data and Knowledge (LDK 2021)*, volume 93 of *Open Access Series in Informatics (OASIcs)*, pages 36:1–36:8, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 2190-6807.

Savchenko, A. V. (2021). Facial expression and attributes recognition based on multi-task learning of lightweight neural networks. In *2021 IEEE 19th International Symposium on Intelligent Systems and Informatics (SISY)*, pages 119–124. IEEE.

Shterionov, D., Vandeghinste, V., Saggion, H., Blat, J., Coster, M. D., Dambre, J., Heuvel, H. V. d., Murtagh, I., Leeson, L., and Schuurman, I. (2021). The SignON project: a Sign Language Translation Framework. In *Proceedings of the 31st Meeting of Computational Linguistics in The Netherlands (CLIN 31)*, July. event-place: Ghent.

Xiang, J. and Zhu, G. (2017). Joint face detection and facial expression recognition with mtcnn. In *2017 4th international conference on information science and control engineering (ICISCE)*, pages 424–427. IEEE.

Zhang, K., Zhang, Z., Li, Z., and Qiao, Y. (2016). Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks. *IEEE Signal Processing Letters*, 23(10):1499–1503, October.