Polytechnic of Turin
Computer engineering
Department of Computer Engineering,
Cinema and Mechatronics

Real-Time Sign Language Detection Bound By User Attention Recognition

Course of Computer Vision and Image Processing

s281657 Angela D'Antonio — Professors
s281330 Federica Moro — Bartolomeo Montrucchio
s286312 Marzio Vallero — Luigi De Russis

a.y. 2020/2021

Contents

1 Abstract

This paper describes all the theoretical and implementation aspects regarding the generation of a neural network to be used as means for recognizing the alphabet of the Italian sign language. The first part describes the software libraries and techniques used, alongside others which were ultimately discarded. We later describe in depth the dataset generation and labeling process and its fundamental role in increasing the final network's detection stability, without incurring in the overfitting and underfitting problems. The training process is performed through Tensorflow's Object Detection API by taking advantage of transfer learning from a pre-trained model of Tensorflow's Model Zoo. After that, we discuss RetinaFace, the Single-Shot Multi-Level neural network used for attention recognition, which acts as a barrier for gesture recognition, such that any gesture input is ignored when not detecting a user looking towards the camera. Ultimately, the networks are tested in order to measure stability and performance.

2 Introduction

Nowadays more and more people rely on automated information systems to improve everyday life. Computer vision and image processing, alongside machine learning, aim to provide advanced tools which can also be used for programming functionalities and user interfaces which simplify interaction with machines. Taking advantage of these resources, our project tries to imbue a machine with the ability to understand the inputs of a user which communicates through the use of sign language, in our specific case, the Italian Sign language. This alternative input method can prove to be useful in environments where a keyboard cannot be employed or would detract from the user experience, enforced in the case in which a user cannot use his or her voice to speak, for example in the case of smart home assistants for deaf people with camera interfaces, or whenever the environment presents too much background noise to allow for clear voice recognition. Moreover, the user's attention span must be measured, in order to let the machine discern intentional inputs from spurious ones coming from the user's behaviour when he or she is not trying to interact with the machine itself. The paper focuses on explaining the steps we took to tackle this challenge. The first section, *Libraries and Techniques Used*, describes in depth the various software libraries and techniques currently available in the field and the ones we chose to use, highlighting advantages and disadvantages of each one. Moreover, we talk about the related image processing projects we used as a reference. The subsections tackle *Dataset creation and labelling*, *Neural network training*, *Gaze recognition* and finally *Real time sign language detection*, and include specifications with brief comments on their respective parts. The subsequent section, *Resuts and analysis*, provides insight on the program's final behaviour, alongside comments on the accuracy of the two neural networks used and overall performance. In last section we express our opinion about the work, the learning goals we got from this project and discuss the evolution of this branch of the Computer Vision field in the near future.

3 Libraries and Techniques Used

Real-time Sign Language Detection is a Python and OpenCV based project that provides all the steps required to produce and label a dataset in order to train a neural network for object recognition through transfer learning and use it to perform real time gestures recognition. The input

frame is processed in search of gestures only if a face is detected within the image frame and the predicted gaze direction of the user is toward the computer's screen.

3.1 Dataset creation and labelling

The neural network used for gesture recognition must first be trained on a dataset containing the gestures we wish to detect. In order to do so, we followed the steps which are also described in the Python Notebooks GenerateDataset.ipynb and LabelDatasetAndTrain.ipynb. The first step is responsible for the definition of the set of gestures and for automated sample capture. After defining the number of images we want to collect for each object (in our case a hand gesture), we proceed to actually capture the frames, which are automatically saved in separate directories according to their label of reference. It is recommended to produce for each label the same amount of frames and to insert a single gesture per frame. This step can be skipped if an already existing set is to be used instead. Then we define a label map containing the same set of labels on which we want to train the model onto. The *nImages* variable specifies how many images of a single gesture shall be taken: in our case, we chose to use ten different frames per gesture, of which seven have been used for training and three for testing, per label. The main variance between sets of images for the same label were different spatial positions of the hand with respect to the frame. It's important to note that a wider dataset for each gesure will increase variance, possibly increasing accuracy and resilience of the final trained model. However, be warned that all the images of the dataset must be manually labeled, so choose wisely.

The labeling process is in fact done with *LabelImg*, a python based graphic tool that lets us to manually select a region of interest within each frame and label it according to a specific class. This process produces, for each frame, an output *.xml* file that embeds the coordinates of the region of interest and label of a specific frame. At the end of the process, we will have an *.xml* for each frame of the dataset.

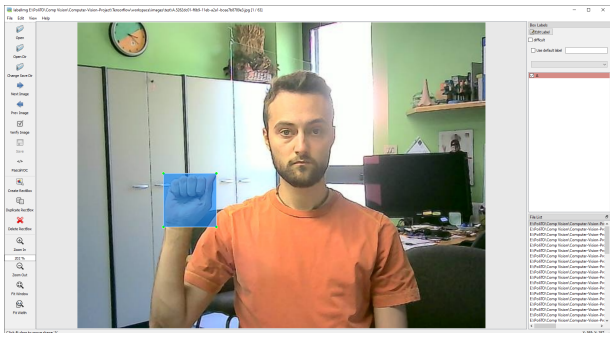

Figure 1: The LabelImg interface after drawing a bounding box around the *A* gesture.

3.2 Neural network training

Once the dataset has been labeled we can train the neural network. To do so we have to split the dataset into two parts, one to be used for training and the other for testing. It's important to note that each frame must be moved alongside its corresponding *.xml* file. The following steps are also described in the Python Notebook LabelDatasetAndTrain.ipynb. We create a Label Map for the different labels in the dataset, then embed all the the *.xml* files associated to each frame to two *.record* files, *train.record* and *test.record*, a specific file format necessary to interface a datset with Tesorflow. Trainig and testing is automated through *Tensorflow's Object Detection API*, a free open source software library for machine learning focused on object recognition. The training technique involves transfer learning from a general purpose pre-trained model, the *SSD MobileNet V2 FPNLite*

320x320, a Single Shot MultiBox Detector Feature Pyramid Network, which is available from *Tensorflow's Detection Model Zoo*, trained on the *COCO dataset*. We chose this network as we needed a very fast Box based detection network with sufficient accuracy, and it is, with a detection speed of $22ms$ on average and a COCO mAP score of 22.2. Other pre-trained models lacked in one or more of the desired characteristics and were thus ultimately discarded as candidates for our project. The SSD's approach is different from other object detection system that usually follow these steps: assume bounding boxes, resample pixels or features for each box, and apply a high quality classifier, repeating the step for a fixed number of iterations. With this approach there is a fundamental improvement in terms of speed because bounding box proposal is deleted, as well as the subsequent phase of pixel or feature resampling, and all computation is encapsulated in a single network, as seen in the *Single Shot MultiBox Detector paper*. The training process is thus sped up through transfer learning from the SSD FPN to the final network trained on our custom image set. With this technique, the final model boasts high accuracy despite using a reduced dataset, cutting down on time requirements for labeling and training. The main principle which allows the trasfer of knowledge across different but related source domains is the idea of generalization of experience, as illustrated in the *Comprehensive Survey on Transfer Learning*.

3.3 Gaze recognition

Whilst developing the project, we noticed that the program had no way to recognize willingness of a gesture: this ultimately lead to gestures being detected also when the user wasn't intentionally interacting with the application. For this reason we introduced a gaze detection algorithm based on the trained deep neural network *Faster RetinaFace* provided as an improvement on the provided alongside the *RetinaFace: Single-ShotMulti-Level Face Localisation in the Wild* paper. The Gaze recognition portion of the software is based on a cascade approach at different levels of granularity to effectively increase detection speed. The first layer of the algorithm employs the *Faster RetinaFace*, used to detect the regions of interest in the frame containing a face. This step, as opposite to the Haar cascade used for masking, is extremely precise and allows for detection of faces at variable positions and rotations. Each region of interest is then processed by applying the *Deep Face Alignment* process, using the stacked Hourglass heatmap based approach for robust 2D and 3D face alignment. The topology of the deep face alignment layer is built with alignment network multi-scale (HPM) residual block and channel aggregation blocks (CAB). The topology of the deep face is sent to the *Head Pose Estimation* which applies cascaded regression trees to predict the face's shape (i.e. feature locations) change in every frame. Splitting nodes of trees are trained in random, greedy, maximizing variance reduction fashion. This step produces a point cloud representing the feature locations of the detected face. The feature locations are used to pinpoint the orientation of the segmented iris with respect to the eye's position according to the *Realtime and Accurate 3D Eye Gaze Capture with DCNN-Based Iris and Pupil Segmentation* paper. The iris position and face alignment are then used to compute the gaze direction of the user, producing two 2D vectors with origin at the iris's center and pointing towards the approximate gaze direction of the detected face. The norm of the two gaze vectors equals zero when the user stares directly into the camera and grows as he or she looks further away from it: it has thus been used as the decision metric to separate frames in which

the user is looking towards the screen from in which it's not doing so.

3.4 Real time sign language detection

The previously trained model is thus used to detect hand signs in image frames, however we must try to reduce input error as much as possible. During the initial testing phase, we noticed that the trained model had some difficulties in correctly identifying the hand due to the detection noise from the background other objects and varaible ambient illumination during the day or from different angles. We thus applied a first step consisting in a double masking approach: the frame is HSV thresolded for skin-toned pixels, then an Haar cascade detection algorithm is used to detect a bounding box for the faces in the frame and removing them. However, variable illumination conditions and skin tones required a broader HSV filter, which still let trough too much noise. In order to address this issue, we included an additional *interactive script CreateHSVProfile.py*, available also in Notebook form, that lets the user create a custom HSV thresolding profile, which are automatically saved in a configuration file *config.dat* and loaded in by the *SignDetection.py*. If no config.dat file is found, a temporary dictionary with statistically relevant default values is used instead.

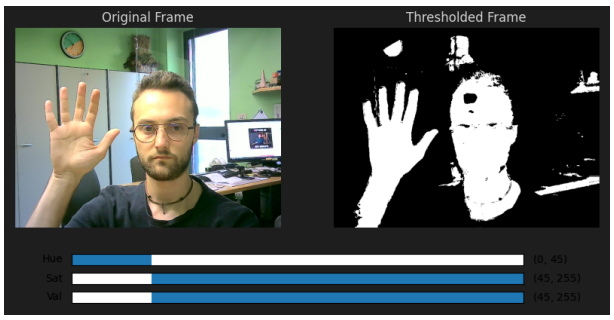

Figure 2: The interactive custom HSV profile script, with range sliders for the Hue, Saturation and Value parameters.

The gaze detection network is then fed with the original not-masked frame input, producing two 2D eye gaze direction vectors, on for each eye. If the norm of the gaze vectors is collectively smaller than a certain threshold, it means that the user is looking towards the camera. If the user looks away for more than five consecutive frames, or no user is detected, the sign detection is suspended until a user looking towards the camera is detected again. If the gaze detection step is passed, the sign detection neural network is invoked, passing as argument the previously masked frame. The network outputs a structure, containing an array of tuples (label, accuracy) and other arrays, alongside which one accounts for the bounding box coordinates of each detected sign. If a gesture is kept stable for five frames, it's added to the output string buffer. In order to improve the software's input capabilities, specific temporal combinations of basic signs have been set up to be recognized as special actions to be used as additional means of interaction. This reduces the stress on the neural network's side, reducing the chance of sign prediction error, other than avoiding the addition of extra non-standard gestures for the user to learn. The sign combinations have been chosen to be easy to perform and logically coherent with the action they represent. At last, the resulting output and string buffer are displayed on a window through the *matplotlib* python library.

4 Results Analysis

The result of our work is a software capable of detecting in real time the Italian Sign Alphabet [figure 3] if it actively finds a user trying to interact with it. It accomplishes its goal of being a reliable mean of interaction, as the user can input any character plus and additional set of special actions. The currently provided network is a right hand trained model, able to recognize static

gestures. Using left-handed gestures may incur in aliasing problems.

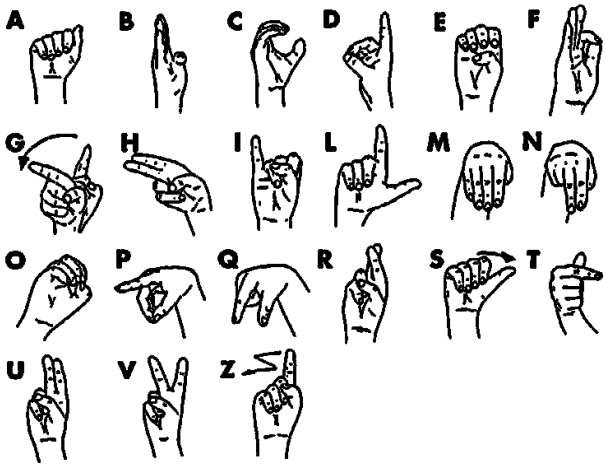

Figure 3: The characters in the standard Italian alphabet.

On a performance standpoint, it must be noted that the neural network portion of the software is the most computationally intensive and as such, should be run on dedicated CUDA enabled GPUs. In our case, since we did not have a CUDA enabled GPU available for accelerating the detection, it has been instead run on the CPU: as such, this impacted the final performance in output frames per second. On the other hand though, the current configuration of the project be run on any machine with a powerful enough consumer CPU. In our case, a *Intel i7-4770* was able to output up to 10-12 frames per second when both the Gaze Detection network and the Sign Detection networks were running together, and up to 25-30 frames per second when only the Sign Detection network was running.

5 Usage

The following guide takes as granted that a user is detected in the frame and is looking towards the camera. When a character is detected, a bounding box surrounds the hand gesture in the frame, displaying the detected sign's label as feedback and the detection accuracy score as a percentage [figure 4]. All detected signs with detection accuracy lower than 70% are ignored and, in any case, no more than one bounding box is displayed per frame, although more than one sign in usually detected: this is done as a measure to reduce as much as possible user-perceived aliasing.

Figure 4: Letter H has been detected.

When a sign is held stable for five frames, the corresponding letter is stored in a string buffer and shown on screen, in order to provide additional feedback to the user [figure 5].

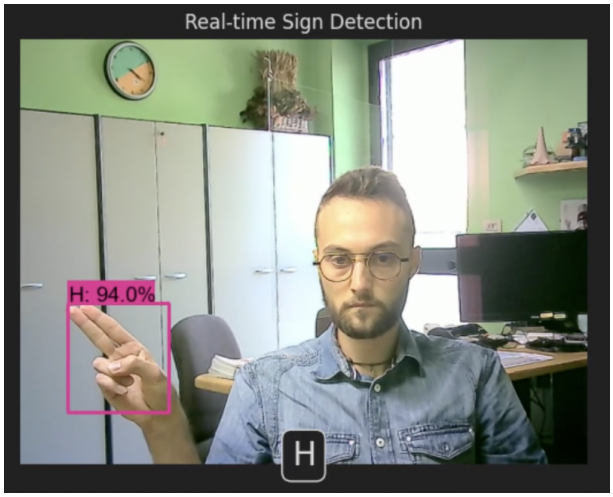

Figure 5: Letter H displayed on screen.

As stated in the previous section, special temporal gestures have been introduced in order to let the user perform high level interaction with the software. These special gestures are registered as specific couples of characters detected in quick succession:

- An "A" gesture followed by an "S" gesture, appends a space to the active string buffer.
- A "V" gesture followed by a "U" gesture, deletes the last character in the active string buffer.
- An "E" gesture followed by an "A" gesture, deletes all the characters in the active string buffer.
- A "D" gesture followed by an "O" gesture, prints the active string buffer to stdout, then deletes all the characters in the active string buffer.

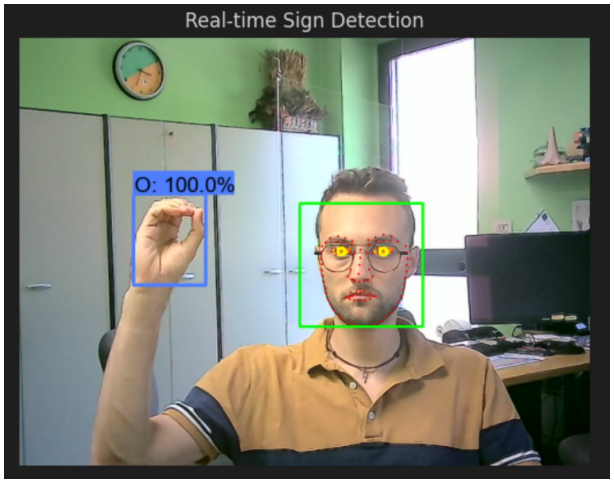

Figure 6: The frame with all features printed on screen. The gaze vectors have norm almost zero, thus are drawn as orange dots.

Running the script with the *–Debug* option, gives the user the possibility to enter a debug mode which allows the person to see every landmark used by the program to process the face and the eyes movement, as well as the gaze direction vectors. The green bounding box represents the face detection performed by the Haar classifier and is used for masking purposes: since after HSV skin-tone thresholding the face is still visible, sometimes certain parts of the face could be recognized as signs. The facial landmarks are drawn as red dots, the pupils are highlighted with yellow circles and the gaze direction vectors are printed as arrowed lines in orange.

Figure 7: The user is looking away, thus the *A* sign is ignored.

6 Conclusions

During development, we stumbled upon multiple issues. Various object classifiers didn't provide the expected results, such as Haar's Classifiers for eyes recognition, or were too unstable. Tensorflow's APIs, being extremely vast required a lot of know-how to be setup properly. In order to produce a working model with sufficient accuracy, multiple datasets and training sessions had to be done. For gaze detection, multiple segmentation algorithms have been tried before finding an approach that suited our needs. However, we refused to stop and kept on trying. At the end of development, we were glad of what we had accomplished. We understood that Computer Vision can pave the way for better means of interaction between humans and machines. We hope that machines will have the chance to improve humanity's tomorrow.

7 Acknowledgments

The project has been developed for the Computer Engineering Masters Degree exam Image Processing and Computer Vision, taught by professors B. Montrucchio and L. De Russis, during the a.y. 2020/2021 at the Polythecnic of Turin.

8 References

- Liu W., Anguelov D., Erhan D., Szegedy C., Reed S., Fu C., Berg A. C.: Single Shot MultiBox Detector. 2016.
- Lin T., Maire M., Belongie S., Bourdev L., Girshick R., Hays J., Perona P., Ramanan D., C. Zitnick L., Dollár P.: Microsoft COCO: Common Objects in Context. 2015.
- Zhuang F., Qi Z., Duan K., Xi D., Zhu Y., Zhu H., Xiong H., He Q.: A Comprehensive Survey on Transfer Learning. 2020.
- Deng J., Guo J., Ververas E., Kotsia I., Zafeiriou S.: RetinaFace: Single-Shot Multi-Level Face Localisation in the Wild. 2020.
- Ansar A., Daniilidis K.: Linear pose estimation from points or lines. 2003.
- Ke T., Roumeliotis S.I.: An Efficient Algebraic Solution to the Perspective-Three-Point Problem. 2017.
- Wang Z., Chai J., Xia S.: Realtime and Accurate 3D Eye Gaze Capture with DCNN-Based Iris and Pupil Segmentation. 2021.
- Tensorflow Object Detection API installation
- Tensorflow Object Detection API GitHub
- TensorFlow 2 Detection Model Zoo
- LabelImg GitHub
- Faster RetinaFace
- Deep Face Alignment
- Head Pose Estimation