Hand gesture recognition: overview and approach to hand shape classification

Author: Andrea Trevisin

Abstract
This report's goal is to present my final project for the Cognitive Services course, consisting in a convolutional neural network implemented with Tensorflow and Keras and trained to classify 16 different hand shapes. This report will introduce the subject of hand gesture recognition, a problem that I came to find very interesting, and some of the most recent and successful approaches. It will then cover the definition of the hand shape classification problem, the explanation of the chosen approach, the selection and preprocessing of the dataset, the experiments and tuning of the model, and a practical application of the trained model (implemented in Python) for real-time hand shape recognition. The trained CNN obtains approximately 88% accuracy on the validation dataset, and is shown to be fairly accurate in real-time evaluation.

Contents

Introduction

Preamble In the last decade, with the advent of augmented and virtual reality systems, a lot of research has been dedicated to finding more natural ways to interact with computers, as we would interact with humans. One way to do this, which is already rather common, is voice: smart assistants like Amazon Alexa and Google Home are widely used, and voice recognition technology is now part of our everyday lives. Another mean of communication, less popular but certainly with great potential, are hand gestures and shapes. which can be associated to specific commands and instructions. Unfortunately this approach is less practical as a camera is required, and hands are not so simple to detect due to a number of factors, such as skin colour and different possible finger positions. As a final project for the Cognitive Services course at the University of Padua,I decided explore the subject of hand gesture recognition and then build a simple application for hand shape classification. An objective of this project is to put in practice the knowledge gained during the course to design a sufficiently accurate model capable of performing well in a real-time application.

Organization of the paper In Section 1 the problem of hand gesture recognition is discussed, along with state of the art solutions; in Section 2 the subproblem of hand shape recognition is expanded upon, as well as the approach of choice and how it was influenced by existing work; in Section 3 the dataset used to train the network is presented with an explanation the data augmentation pipeline in place; Section 4 details the CNN model architecture and hyperparameters, and in Section 5 the training and validation results are shown and analysed. Finally, Section 6 briefly describes my small application using the model to recognize hand shapes in real time via webcam input.

1. Inspiration

1.1 Hand gesture recognition

Of the many data sciences challenges out there, hand gesture recognition is a particularly interesting one where many approaches are possible. It is not a widely diffused way of interacting with devices, mainly because of the aforementioned need of at least a camera (which, by itself, isn't sufficient to achieve good accuracy in most scenarios); it has mostly been explored in areas regarding simulations, interactions with smart cars and gaming - and most times a tracking device of sort is used to facilitate the task. While it is not a widely used

form of managing devices (a close ”relative” face recognition, has only recently seen success as many smartphones started to implement face-based security features), it could have many beneficial applications if it became more widespread - namely, touchless interfaces could be a substitute for voice activation for speech-impaired and hearing-impaired people [1][2]. As a challenge, it presents several layers of difficult, the first of which is recognizing the hand. While it may seem trivial at first, recognizing and tracking a human hand as a separate object from the body is not as intuitive as it sounds given, for example,the variety of perspectives, skin colours and lighting conditions (three of the most notable variations used when recording datasets for the problem - which are, most of the times, scarce or too small feature-wise). It may be difficult to recognize and track hand movement from a simple RGB camera feed, as exposed skin and background can heavily influence colour-based edge detection. Then comes the problem of classifying the hand shape, an easier task given the distinct features provided by finger positioning relative to the palm. The main difficulties are recognizing different perspectives (as point of view can influence the significance of finger positioning), and the back of the hand from the palm (differences between an open hand facing either way are very slight). The last challenge is establishing relationships between the aforementioned features and the spatial and temporal dimensions of the gesture: the hand moves freely in a three-dimensional space and the same gesture can be performed at different speed and with larger or stricter movements. Some gestures share the same spatial movement but with different hand shapes, and so on. As it can be seen, there are many challenges to be tackled and many ways to approach them - which will be discussed shortly.

1.2 State of the art approaches

Most practical gesture recognition applications can be observed in the gaming and automotive fields, with new technologies on the rise in healthcare, and are usually able to recognize simple hand motions or whole body movements. One of the most famous motion tracking devices is the Microsoft Kinect, a multi-modal sensor equipped with a classic RGB camera and an infrared projector, which projects a 3-dimensional grid enabling the sensor to ascertain the depth of an object in addition to its X and Y coordinates in space, effectively granting an easy way to track an object in a 3-dimensional space. As of today Kinect is widely used for research on gesture recognition (as well as similar sensors like Leap Motion or Intel RealSense), since it has been established that depth-based datasets yield the best accuracy when used to train neural networks on the task at hand. As it is shown in this paper [3], depth-based recognition is always more precise than the more ”standard” RGB approach, with differences in accuracy as high as 6.2% on the nvGesture dataset (NVIDIA Dynamic Hand Gesture Dataset). The paper concludes that the best results are obtained by combining RGB, depth and infrared features, reaching a recognition accuracy as high as 83.8% when using combination of 5 different sensors (RGB, optical flow, depth and infrared image/disparity). It’s worth noting, though, that it represents only a slight 3.5% improvement from a purely depth-based. The neural network architecture of choice is 3D-CNNs, or three-dimensional neural networks, which use 3D convolutional kernels (as opposed to traditional 2D ones) in order to leverage context from adjacent inputs - in a video format, that means that by using a 3D CNN it is possible to make use of temporal features together with spatial features of the frames. The downside of 3D CNNs is that the number of parameters to be tuned during training is really high and are rather resource-intensive. A more recent research [4] elaborates on the optimal pipeline for hand gesture recognition that is able to circumvent the aforementioned issue. The proposed pipeline consists of a lightweight CNN to be used as a detector (ResNet-10), which is used to establish when a gesture begins (acting as a binary classifier), and a deep CNN (ResNeXt-101, pretrained on the Jester dataset) for the actual classification task. The results show how the pipeline, when trained on depth-based input, reaches an accuracy of 83.82% on the same dataset mentioned earlier, achieving a 0.52% increase in precision while only utilizing one sensor.

2. Approaching hand shape classification

While the aforementioned problem is indeed interesting and challenging, it is unfortunately really demanding in terms of computation, hardware and dataset size (the nvGesture dataset alone takes up over 60 GB of space). For the sake of this project, one of its many sub-problems has been explored - namely the problem of hand shape recognition. While it does not present challenges as interesting as temporal and spatial correlation between frames, it still requires a methodical approach in order to develop a model that can be applied in a dynamic, real-time situation. This project aims to train a CNN model able to recognize different hand shapes from an image obtained by a standard laptop webcam with a sufficient degree of accuracy, to eventually develop an actual application that makes use of said model. In the project, a dataset acquired using an infrared sensor with a frontal perspective is chosen to train the models; data augmentation is then applied, to highlight important features and reduce overfitting; a CNN is then trained on the training data and evaluated against a test set. For the real-time application, background subtraction is used to obtain an image with similar features to the training dataset, which is then pre-processed and evaluated by model.

2.1 Existing work

While the inspiration for this project stemmed from the papers cited above, the technical aspects in approaching the subproblem of hand shape classification were influenced by a conference paper [5] by Núñez Fernández and Bogdan who propose a CNN architecture on which I designed the one used for this project. The aforementioned paper [4] by Gunduz

et al. provided data augmentation modalities and parameters that were adapted to the dataset in use.

3. Dataset

The dataset of choice is the Multi-Modal Dataset for Hand Gesture Recognition [6] that can be found on Kaggle.

Description The dataset is composed by a set of near infrared images of various hand gestures and poses, acquired with a Leap Motion sensor. The images depict 16 different hand gestures (15 if we consider that the palm gesture is counted twice, one per side of the hand) performed by 15 different subjects, respectively 5 women and 10 men. This dataset was chose due to infrared images facilitating image thresholding as the (white) hand is very distinct from the (dark) background, as well as the fact frontal position of the hand resembles the perspective that would be used later. There is a total of 44189 train images and 10812 test images, each labeled with a value from 0 to 15 and the name of the gesture.

3.1 Pre-processing

Images in both the training and test sets are first pre-processed in order to highlight relevant features as well as obtain a suitable input size. The pre-processing pipeline is as follows:

1. Resize to 192x128 resolution
2. Apply Gaussian filter with 5x5 kernel
3. Apply binary thresholding using Otsu's method
4. Normalize

Resizing helps reducing the input size, and by keeping the original/resized ratio close to 0.5 there is no significant feature loss. After resizing a Gaussian filter smoothens the image, which removes possible downsizing noise due to pixelation and makes thresholding more precise. Binary thresholding is done with Otsu's method, which searches for the threshold that minimizes the intra-class variance: the algorithm iterates through all the threshold values and calculates a measure of spread for the pixel levels each side of the threshold. Finally the image is converted to a numpy array and normalized, obtaining a matrix of zeros and ones.

$$\sigma_W^2 = W_b\,\sigma_b^2 + W_f\,\sigma_f^2$$

Figure 1. Intra-class variance is defined as a weighted sum of variances of the two classes.

Data augmentation Given that the dataset contains similar images, and that the number of features has been reduced, a data augmentation pipeline is in place to avoid overfitting:

1. Apply random zooming with magnification a range of 0.8 to 1.2
2. Apply random horizontal shift up to 15% of the input width
3. Apply random vertical shift up to 15% of the input height
4. Apply random rotation up to 17 degrees in either sense

Data augmentation is provided as a part of the Keras module and is applied during training; having different kinds of spatial augmentations mixed together lowers chances of overfitting the model and will enhance the real-time accuracy of the model. To maintain consistency, the test data is only zoomed. Data augmentation is executed before training by creating an `ImageDataGenerator` with the desired parameters and batch size. The generator is then fit to the data, which means that the parameters for augmentation are adjusted to the data, and passed to the model instead of the actual data. Labels are also augmented with one-hot encoding, a process that transforms categorical features in a binary array with the same length as the cardinality of the labels, and that contains zeroes in all positions except for the one corresponding to the label.

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→ One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

Figure 2. Example of one-hot encoding

4. Architecture

Convolutional neural networks, or CNNs are deep neural networks particularly suited for learning from image data as they allow 2 and 3-dimensional arrays as input layer. The great advantage of CNNs is that, thanks to convolutional layers, they are able to take advantage of the hierarchical pattern in image data, and derive complex features from simpler ones; this while also preserving spatial features. Each convolutional layer in a CNN can be seen as a filter that detects progressively higher level features.

4.1 LeNet-5

The architecture of choice is loosely based on the concept of the LeNet-5 network [7], a very entry-level CNN for handwritten character recognition. This is due to the fact that thanks to the pre-processing pipeline, important features in our dataset are prominent and require a less complex CNN. The LeNet-5 architecture consists of two convolutional layer/average pooling layer pairs, a flattening convolutional layer, followed by two fully-connected layers and a softmax classifier.

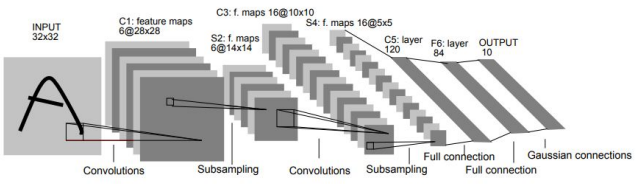

Figure 3. LeNet-5 architecture

4.2 CNN structure

The CNN designed to solve this problem can be considered an upscaled version of the LesNet-5. It maintains a similar structure with a first convolutional layer, followed by two convolutional-pooling layer pairs for feature extraction and subsampling, and two fully-connected layers before a softmax classifier. Differently from the LeNet model, this model uses ReLU activation functions for the convolutional layers (which were first proposed in the Alexnet CNN) and includes a dropout regularization layer that deactivates random neurons to prevent overfitting. The CNN structure can be described as follows:

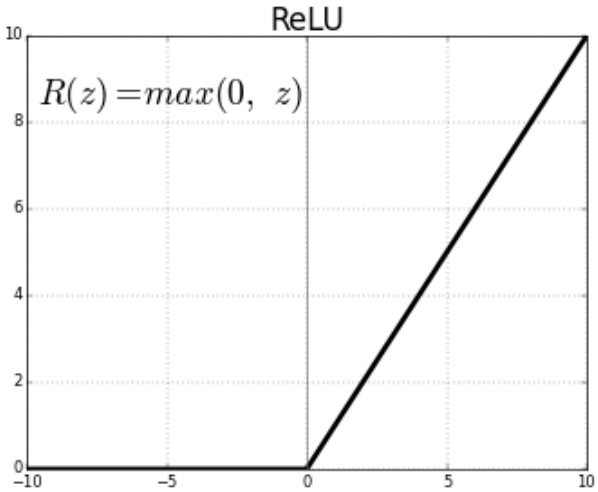

Figure 4. Rectified Linear Unit (ReLU) activation function

1. **Convolutional layer** with filter size of 32, input size of 128x192x1 (a single thresholded image), 3x3 kernel applied with stride 2 and ReLU activation function
2. **Max pooling layer** with same input size as the previous layer's output and 2x2 pool size
3. **Convolutional layer** with filter size of 64, same input size as the previous layer's output, 3x3 kernel applied with stride 1 and ReLU activation function
4. **Max pooling layer** with same input size as the previous layer's output and 2x2 pool size
5. **Convolutional layer** with filter size of 96, same input size as the previous layer's output,3x3 kernel applied with stride 1 and ReLU activation function
6. **Max pooling layer** with same input size as the previous layer's output and 2x2 pool size
7. **Convolutional flattening layer** that flattens the output in preparation for the densely connected classifier
8. **Densely-connected layer** with 512 hidden units and ReLU activation function
9. **Dropout regularization layer** with 0.3 ratio
10. **Densely-connected layer** with 16 hidden units (one for each class) and softmax activation function

4.3 Implementation

The aforementioned model is implemented using the Keras module. Keras is an open-source neural network Python library designed to enable fast neural network development; its `layers` component offers different kinds of layers for building neural networks that are highly capable of customization but require little configuration (e.g.: by adding layers to a sequential model it is only necessary to specify input size for the first one; Keras will automatically derive the right input size for inner layers). For the implementation of the CNN, 4 kinds of layers were used:

- `Conv2D` layers, corresponding to convolutional layers, perform 2D convolution with kernel and stride of the desired size; it is also possible to specify padding. The filter size determines how many filters the layer will learn - ideally it should start as a small number in shallow layers and progressively increase in deeper layers.
- `MaxPooling2D` layers are used to downscale the input of the following layer; the pool size is the tuple of downscaling factors (vertical, horizontal); the pooling stride defaults to the same as the pool size.
- `Dropout` layers apply, obviously, dropout to the input: they set some of the input units to 0 according to the specified rate (which should usually range from 0.2 to 0.5).
- `Flatten` layers flatten the input into a 1D array, without modifying the batch size; this is usually done in preparation for classification layers.
- `Dense` layers are standard densely-connected layers with a specified number of hidden units and an activation function; the dot product between the kernel weights (automatically generated by the layer) and the input is computed, then the bias (also automatically generated) is added and the activation function is applied to the result. The softmax activation function is used in the last layer to obtain class likelihoods.

As it can be seen in Figure 5 the CNN model has 3,032,560 trainable parameters.

4.4 Learning parameters

4.4.1 Optimizers

Optimizers are optimization algorithms used to determine how to adjust the weights of the network's nodes after a training epoch, based on the difference between the actual and the expected results, in order to achieve better accuracy. Two different adaptive optimizers were tested with default hyperparameters.

RMSprop An adaptive learning method proposed by G. Hinton [8], it is somewhat similar to a gradient descent algorithm

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 64, 96, 32)	832
max_pooling2d_1 (MaxPooling2	(None, 32, 48, 32)	0
conv2d_2 (Conv2D)	(None, 30, 46, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 15, 23, 64)	0
conv2d_3 (Conv2D)	(None, 13, 21, 96)	55392
max_pooling2d_3 (MaxPooling2	(None, 6, 10, 96)	0
flatten_1 (Flatten)	(None, 5760)	0
dense_1 (Dense)	(None, 512)	2949632
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 16)	8208

Total params: 3,032,560
Trainable params: 3,032,560
Non-trainable params: 0

Figure 5. Summary of the Keras model

with momentum. The main difference lies in the way gradients are calculated: nodes with an history of large gradients are given large gradients, and nodes with historically small gradients are assigned small gradients. The gradient history is defined by an exponentially decaying average of all previous **squared** gradients, which is used to compute an adaptive learning rate by dividing the current learning rate by the square root of the decaying average. This is shown in the formula:

$$\begin{aligned} E[g^2]_t &= \alpha E[g^2]_{t-1} + (1-\alpha) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t \end{aligned} \tag{1}$$

where α is the decay coefficient, $E[g^2]_t$ is the decaying average and $\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}}$ is the adaptive learning rate.

Adam The *Adaptive Moment Estimation* (Adam) algorithm expands on the idea of RMSprop by multiplying the adaptive learning rate by an exponentially decaying average of past gradients rather than the past gradient:

$$\begin{aligned} E[g^2]_t &= \alpha E[g^2]_{t-1} + (1-\alpha) g_t^2 \\ A[g]_t &= \beta A[g]_{t-1} + (1-\beta) g_t \end{aligned} \tag{2}$$

where α and β are the decay coefficients, $E[g^2]_t$ is the decaying average of squared gradients, $A[g]_t$ is the decaying average of the gradients. The problem here is that these values are biased towards 0, given that the gradient vector is initialized as zeros on the first run. Therefore the following adaptation must be made:

$$\begin{aligned} \hat{E}[g^2]_t &= \frac{E[g^2]_t}{1-\alpha} \\ \hat{A}[g]_t &= \frac{A[g]_t}{1-\beta} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{E}[g^2]_t + \varepsilon}} \hat{A}[g]_t \end{aligned} \tag{3}$$

This bias-corrected formula, when compared to RMSprop, introduces both additional momentum and additional "friction" (in the form of bias correction), and is generally regarded as one of the best adaptive learning algorithms. While RMSprop is utilized with the default hyperparameters, the learning rate for Adam will be set to 0.008 for a more interesting comparison.

4.4.2 Loss function

Given that the challenge at hand is a single label categorization problem (in which each input can be only attributed one category), the obvious choice is the categorical cross-entropy function. Categorical cross-entropy loss, also known as softmax loss (as it is usually employed when the last layer is a softmax classifier) is defined as $-\sum_{n=1}^{C} t_i log(f(s)_i)$, where C is the number of classes, t_i is the ground truth and $f(s)_i$ is the result of the softmax function. Assuming that labels are one-hot encoded, the only element in the ground truth vector that is non-zero is $t_i = t_p$; by discarding null elements in the summation, the formula now becomes:

$$-log\left(\frac{e^{s_p}}{sum_{i=1}^{C} e^{s_i}}\right) \tag{4}$$

with s_p being the score for the positive class. What this formula achieves is that when the prediction is very far from the ground truth the loss value is very high; as the prediction becomes more certain, the loss lowers sharply at first and then more slowly the closer the prediction gets.

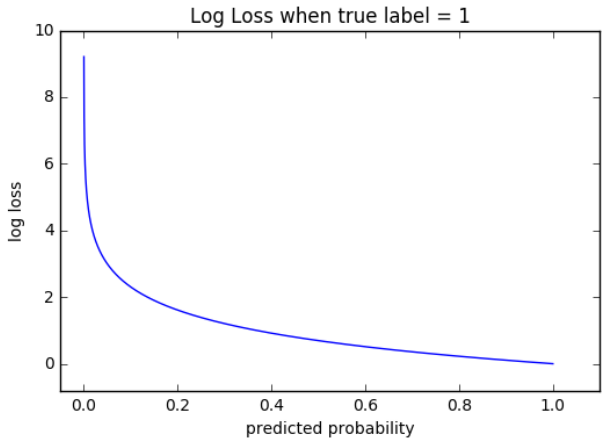

Figure 6. Graph for the categorical cross-entropy function

5. Training and results

As mentioned in Section 3.2, data augmentation takes place during training. Training is run for 15 epochs with a batch size of 64, categorical cross-entropy loss function and two different optimizers (Adam with 0.005 starting learning rate and RMSprop with default hyperparameters). The test data is only augmented with random zooming for the sake of consistency between training and testing. The test to train data percentage is approximately 24.7%. When training, the following callbacks are specified: early stopping when the validation loss varies by less than 0.01, with a patience of 10 epochs, and learning rate reduction when the validation loss varies by less than a 0.1 factor within 3 epochs. After training, the following results are obtained for the two models using different optimizers:

Table 1. Training results

Optimizer	Accuracy	Loss
RMSprop	0.9762	0.1236
Adam	0.9631	0.1120

Table 2. Validation results

Optimizer	Accuracy	Loss
RMSprop	0.8650	0.6839
Adam	0.8805	0.6174

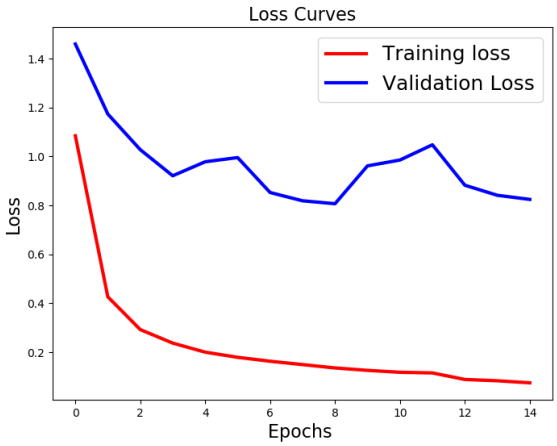

Figure 7. Loss curves when using the RMSprop optimizer

RMSprop does not appear to be a good choice for this network architecture: after the 9th iteration the validation loss 7 starts rising significantly, triggering the learning rate reduction callback in an attempt to avoid overfitting (attempt which succeeds, suggesting that setting a learning rate lower than normal could be better). RMSprop's worse results than Adam's might stem from the former's tendency to "snowball" the gradient more than the latter. The result of this are more

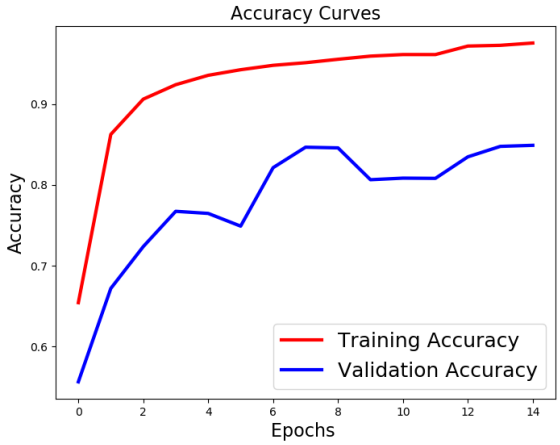

Figure 8. Accuracy curves when using the RMSprop optimizer

Figure 9. Loss curves when using the Adam optimizer

accurate training results, but worse validation accuracy and loss. When observing the results attained with Adam, the validation loss 9 and accuracy 10 curves are also noticeably bumpier than the respective training curves, although they do maintain a descending trend: while training curves are usually smooth, this kind of validation curves suggest that the model tends to overfit data and then correct itself. It can be observed how both accuracy and loss stop improving around the 12th epoch, suggesting that it might be the ideal time to stop training as successive epochs lead to overfitting. Other possible precautions would be increasing data augmentation or increasing dropout regularization.

6. Real-time application and possible improvements

Applying this model to a real-time situation required some adaptation: first of all, the input would be acquired with a simple laptop webcam - a very different type of sensor

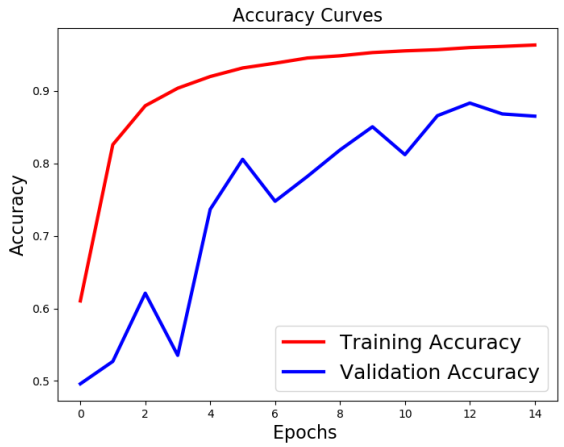

Figure 10. Accuracy curves when using the Adam optimizer

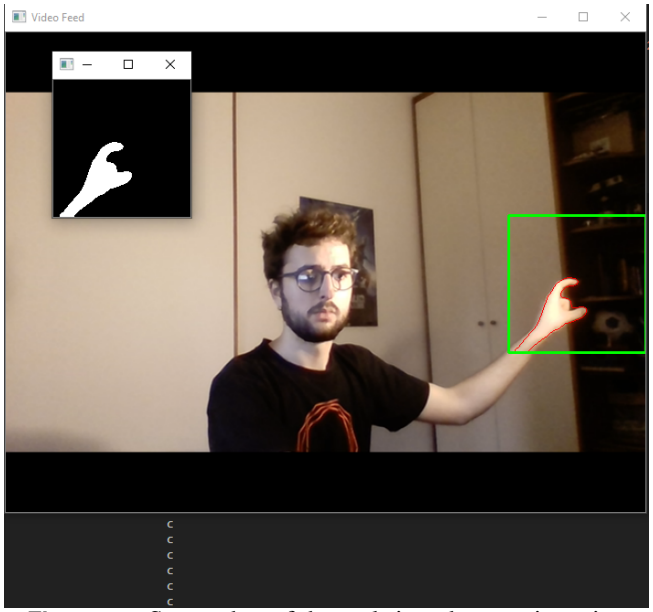

Figure 11. Screenshot of the real-time detector in action

when compared with the infrared camera the dataset was acquired with. The first challenge was obtaining an image of the hand in a format suitable to be evaluated by the trained model. The approach of choice is background elimination: an unobstructed mage of the background is acquired at the very start of the program, which is then subtracted to the camera feed; the weighted average of the resulting image is computed, allowing - if this process is restricted to a fixed area - to obtain a clear distinction between the hand and the background (N.B.: his whole ordeal could easily be ignored if we were to use a depth sensor - which as mentioned, has the best performance when it comes to hand recognition tasks as it makes it easier to extract features). In the program, the first 30 frames are captured to establish the background; the area where the hand should be positioned is then highlighted. The resulting image is then thresholded and pre-processed in order to be fed to the classifier. The model shows to be fairly accurate with small windows of uncertainty, especially when trying to distinguish an open hand facing forward from one facing backward. Possible improvements would be to either train the model to recognize hand shape purely based on RGB feed (task for which a large dataset would be needed, as well as a more complex network architecture), or to use a depth sensor with a depth-based dataset, which would prove more accurate and would prove interesting as the features being detected would be entirely different.

References

[1] Balakrishnan P. and Pradhan G. Hand-gesture computing for the hearing and speech impaired. *IEEE Computer Society*, 2008.

[2] Anagha J. Jadhav and Mandar P. Joshi. Hand gesture recognition system for speech impaired people: A review. *IRJET*, vol. 03 iss. 07, 2016.

[3] Molchanov P. Yang X. Gupta S. Kim K. Tyree S. Kautz J. Online detection and classification of dynamic hand gestures with recurrent 3d convolutional neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[4] Köpükü O. Gunduz A. Kose N. and Rigoll G. Real-time hand gesture detection and classification usingconvolutional neural networks. *IEEE International Conference on Automatic Face and Gesture Recognition*, 2019.

[5] Dennis Núñez Fernández and Bogdan Kwolek. Hand posture recognition using convolutional neural network. 12 2018.

[6] Multi-modal dataset for hand gesture recognition. `https://www.kaggle.com/gti-upm/multimodhandgestrec`. Accessed: 31-08-2019.

[7] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20:5, 2015.

[8] Overview of mini-batch gradient descent. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: 31-08-2019.

”” ””