Hand gesture detection

Erik Finnesand, Kristian Horve Tjessem and Brage Solheim

1 Summary

The project presented in this paper goes through how a hand gesture detection model is built. The project is split into two subproblems. These are hand detection in images and hand gesture detection. For the hand detection problem the python library Mediapipe is used to find hand landmarks and to crop the hand out of the image. For the hand gesture detection problem a convolutional neural network is trained and validated by images from the HaGRID dataset[1] which consists of 18 different types of hand gestures. Another set of images from the HaGRID dataset is then used to test the model. The two subproblems are then combined into a single python function which takes a full size RGB image as argument, crops and resizes it and returns the gesture the model detects.

2 Introduction

We have been tasked with detecting hand gestures from an image. A task that is simple for a human, but very tricky for a computer. Following the principle of breaking stuff into smaller steps, we take the problem of detecting a hand gesture to first being able to detect the hand itself, irregardless of gestures. If we can figure out where the hand is in an image, we can crop that part of the image, process it faster than the entire image and gain more accurate results while training our model.

The next problem after we are able to detect the hand and determine its location, is to be able to detect the gesture it's performing. There are some different techniques that can achieve this, some are mentioned in our theory section, and our eventual solution is explained later on.

3 Theory

This chapter explains the theory of the methods that is used for the hand gesture detection.

3.1 Representation of images in computers

Computers are limited to binary in how it structures data. Behind the scenes an image is just a bunch of ones and zeroes. To structure a color image in binary, the standard is to represent it as a 3d matrix. You break the image down to colored pixels whose intensity value is mapped to the matrix. An image matrix with m rows and n columns, means that an image has m pixels in height and n width, giving $m * n$ pixel resolution.

The intensity value can be called color depth, where the most common value range is from 0 to 255. In a gray scale image, 0 would be completely black, and 255 completely white. For colored images the value would be a intensity of a color, since you can create all colors some combination of red, green and blue. We therefore say there are three channels: red, green and blue channels that each have their own value between 0 to 255. The combination of those intensities will create the final color.

The mentioned 3d matrix is therefore 3 dimensions for colored images. The dimensions are the intensity values for red, green and blue. Put together and we can represent an image with the coordinates x,y,c. x and y are coordinates of the 2d image, c has the information about the intensity of the pixels color at position (x,y). So when constructing a color image to represent on screen, check all 3 dimensions intensity value at (x,y) and combine them with some algorithm to create the final color of a pixel. Do this for all pixels and you have the full image.

3.2 Hand detecting techniques

To be able to detect a gesture from a hand, we first need to be able to detect the hand itself. In the paper [14], the authors examine and review a bunch of techniques that utilize cameras with some type of depth sensing integrated into them. Although we do not have depth information in our images, the paper does also review some techniques that segment the hand using skin-color maps and or Haar-like features.

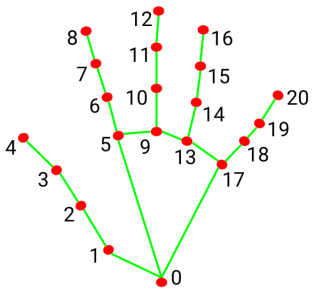
Using skin-color maps brings up some problems mentioned in both [14] & [13], mainly that lighting has a very strong effect on the results. If you train a classifier on some data, it may not be able to detect the hand due to the difference in lighting.

Another challenge in detecting hands is that even the size can be different, not just because people have different size hands, but the distance to the camera lens can affect the size on the output image. Even a small coin can cover the sun if close enough to the eye.

Comparing the features of the human hand to that of the face, we can see that there are patterns in the face that have high contrast, like the mouth and eye region [5]. The hand does not posses such contrast patterns that would have made it easier to detect in an image.

The creators of `mediapipe`, a library we use to detect hands, use machine learning to first figure out where the palm(s) are in an image. They have achieved this by creating a single-shot detector model. It uses a single deep neural network to detect objects in images [12]. The object in this case is then the palm of a hand, "since estimating bounding boxes of rigid objects like palms and fists is significantly simpler than detecting hands with articulated fingers."[5].

After the palm has been detected from an image, the area of the image that contains the palm then gets sent to another model. The next models is specifically trained to figure out structures of the hand. They end up representing the hand as 21 points that is specific to the wrist and knuckles of the hand. The model is even able to handle hands that are partially visible.



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

Figure 1: Landmarks of the hand. Sourced from [5]

3.3 Convolutional neural network

Convolutional neural networks is a class of neural networks commonly used for detecting and analyzing images. It is a combination of deep neural networks and kernel convolution. Convolutional neural networks takes the 3D image matrix as a input and outputs which class it thinks the image belongs to. Between the input and output layer some hidden layers exist.

3.3.1 Layers

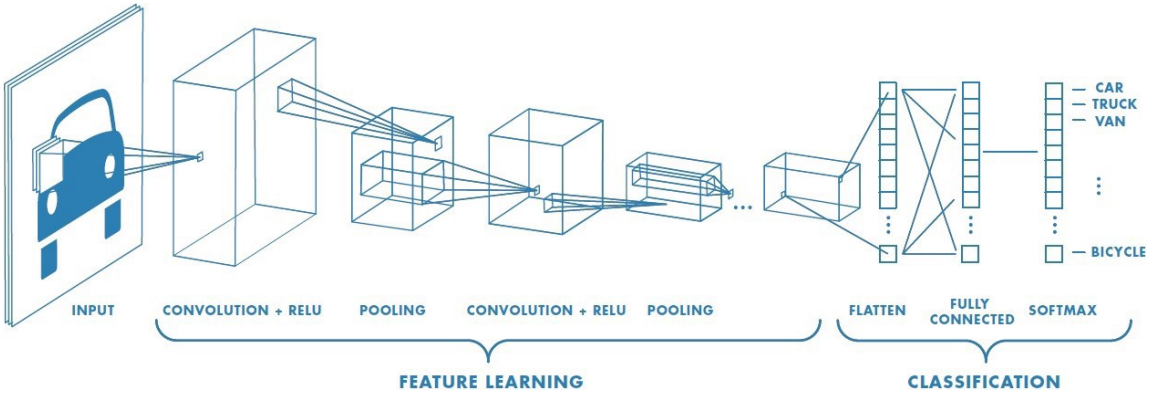


Figure 2: Example of a convolutional neural network 2

Convolutional neural networks consists of different hidden layer. The main layers are convolutional layers and pooling layers. These are the layers used for feature learning.

Convolution is a operation between two matrices that takes the product of the same index element in both matrices for all indexes before adding them together.

Convolutional layers are layers that perform convolution on an image matrix. The convolution are done with filters which are $hxwxd$ matrices used as pattern detectors, where h is the height, w is the width and d is the depth of the filter. The depth of the filter has to be the same as the depth of the image (3 if the image is RGB and 1 if the image is greyscale). The filters start at completely random values that are updated during the training. Compared to a normal neural network the filters can be looked at as weights between neurons. The convolution starts at the upper left corner of the image matrix and shifts one column to the right for every turn. For each turn the convolution result of the filter and the part of the image it overlays are computed and stored in a new matrix. When the filter reaches the far right of the image matrix it shifts down one row before moving back to the left side of the image matrix to repeat the process. After the filter has convolved the whole image matrix the layer stores the matrix containing the convolution results. This is the new representation of the image. The size of the final output from the convolutional layer is the size of this matrix times the number of filters used in the layer.

A convolutional neural network consists of multiple of these convolutional layers. The first layers normally starts with only a few basic filters used to detect large noteable edges in the image. As we go deeper into the network more complex filters are used to gradually detect more sophisticated patterns.

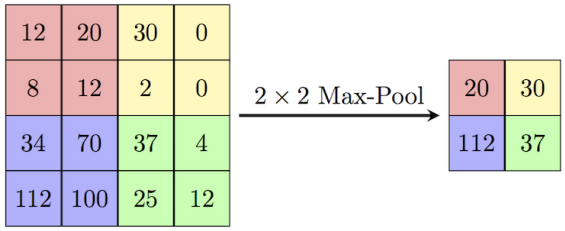


Figure 3: Example of max pooling [4].

Each convolutional layers is normally followed by a pooling layer. The pooling layers goal is to reduce the dimensions of the input image as seen in figure 3 where a $4 * 4$ image gets reduced to a $2 * 2$ image. This is done by downsampling the image. The pooling layer splits the image into multiple $2 * 2$ matrices. The next step depends on the pooling technique used. The two technique are max pooling and average pooling. Max pooling takes the maximum value in the $2 * 2$ matrix and places it in a new matrix, while average pooling does the same with the average value of the $2 * 2$ matrix. This layer will mainly reduce computational load as we go deeper into the network, but it will also reduce overfitting.

After the feature learning phase the classification phase can start. The first step of the classification phase is to flatten the multiple image matrices returned by the last pooling layer to a one dimensional array. After this a last hidden layer of neurons are used before the image is classified.

3.3.2 Training a convolutional neural network

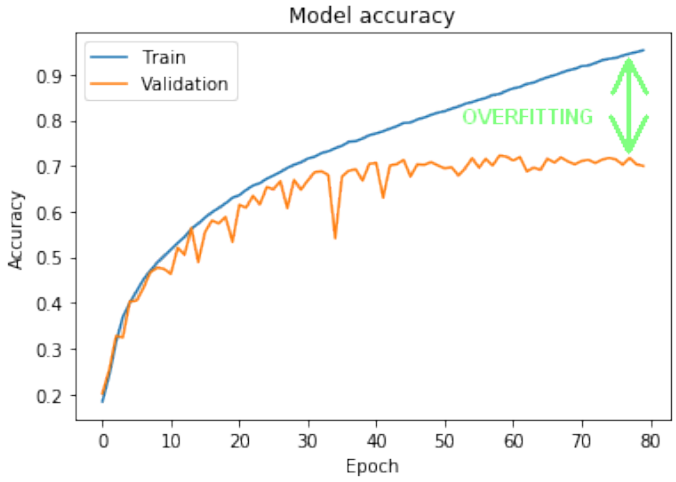


Figure 4: Example of overfitting [8].

A convolutional neural network is trained using a train dataset and validated using a validation dataset. The validation dataset is used to test the model after each epoch. The reason a validation

dataset is used is to keep track of the model training and to see if overfitting occurs. Overfitting occurs when the classification accuracy of the training set is larger than the validation set as seen in figure 4. This happens when the model starts to learn from noise and other details in the training set. Multiple techniques can be used to avoid overfitting such as pooling, removing unecessary details from the input image, data augmentation and performing dropout.

Data augmentation is used to create artificial data from the images in the dataset such that it appears larger than it is. This can be done by flipping, rotating, zooming, cropping etc. images in the training dataset.

Dropout is performed in by ignoring a percentage of nodes in the layer. For each training stage a new set of nodes are ignored.

4 Implementation

The project was created using Python in Jupyter Notebook. This chapter explain how the hand gesture detection was implemented.

4.1 Libraries

The libraries used for this project are listed in the table below.

Libraries	**Description**	**Uses**
OpenCV[7]	OpenCV is a library mainly aimed at real-time computer vision, but it can also be used for processing images.	OpenCV is used for reading and writing images from/to the file directory. It is also used to crop images, create text on images and to draw points on images.
Matplotlib[3]	Matplotlib is a library used for creating static, animated, and interactive visualizations in Python.	Matplotlib is used to plot images and graphs in the notebook.
NumPy[6]	NumPy is a library that adds support for large, multi-dimentional arrays and matrices, as well as mathematical functions to operate on these arrays.	NumPy is used to return the maximum value from lists.
Seaborn[10]	Seaborn is a visualization library based on Matplotlib. Seaborn provides more customization for plots plotted in Matplotlib.	Seaborn is used to create grids in some of the Matplotlib plots. It is also used to create a heatmap in the confusion matrix when testing our model.
Scikit-learn[9]	Scikit-learn is a machine learning library built on top of SciPy. It has support for various classification, regression and clustering algorithms. It also supports various ways to evaluate models.	Scikit-learn is used to create a confusion matrix and classification report when testing our model.
TensorFlow[11]	TensorFlow is a machine learning and artificial inteligence library. It support various ways to easily prepare data, as well as creating, compiling, training and testing custom and preset models.	TensorFlow is used to import datasets from the file directory, creating a convolutional neural network, as well as compiling, training and testing the neural network.
Mediapipe[5]	Mediapipe is a cross-platform library that offers customizable machine learning solutions. Supporting both live and steams of data. Built for speed even on common hardware. Free and open source	We use a built in model that detects hands in images. It finds landmarks on the hand that are used to crop out the hand from the rest of the image.

4.2 Data and images

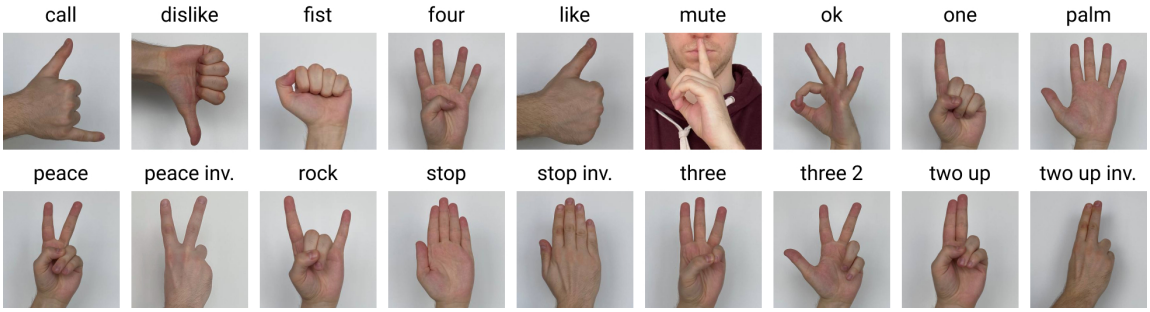


Figure 5: The 18 different classes in the HaGRID dataset[2].

The images used for this project came from the HaGRID (HAnd Gesture Recognition Image Dataset) dataset. The size of the full dataset is 716GB and contain 552,992 1920×1080 pixels RGB images divided into 18 classes of gestures as seen in figure 5.

Each image in the dataset has annotations stored in a JSON file for the class it belongs to. The annotations for each image are:

- `bboxes`: Normalized coordinates for boxes around detected hands. `[top left X pos, top left Y pos, width, height]`.
- `labels`: List of class labels of the detected gestures. e.g. `like`, `peace`, `no_gesture`.
- `landmarks`: List of normalized hand landmarks. `[x, y]`.
- `leading_hand`: `right` or `left` for which hand is showing the gesture.
- `leading_conf`: Leading confidence for `leading\_hand`.
- `user_id`: ID of user submitting the image.

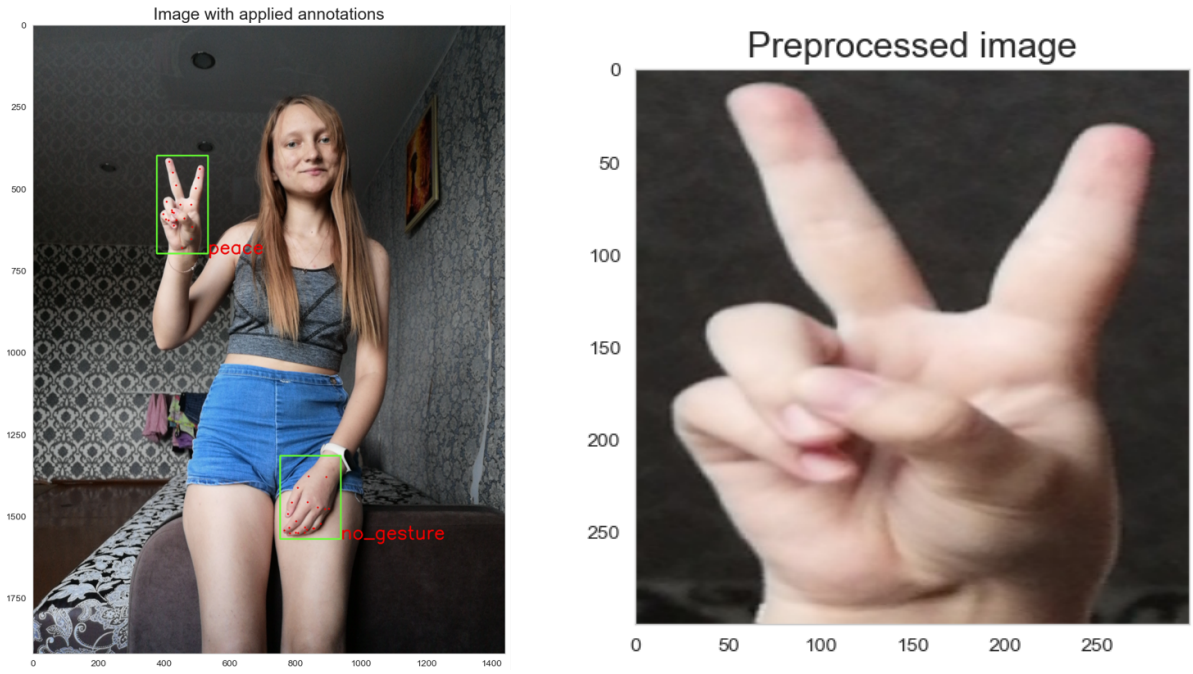
For training and validating the model 500 images of each class from the HaGRID test dataset is used. For testing the model the HaGRID subset is used. The subset contains 100 images of each class. The images is stored in a test and train folder with subfolder for each class containing the images belonging to the class. The same is the JSON annotation files.

When importing the data to Jupyter notebook these functions are used:

- `files_to_list()`: Takes `directory_path` and `file_extension` as argument. Iterates through every file in the directory using `os.listdir(directory_path)` and returns a list of every file containing the `file_extension`.
- `img_to_dict()`: Takes `directory_path` as argument. Iterates through every file in every subfolder in `directory_path` and adds them to a list in a dictionary where the class is the key and the list of images belonging to the class is the value.
- `json_to_dict()`: Takes `files` and `img_dict` as argument. `files` is returned by `files_to_list()` and `img_dict` is returned by `img_to_dict()`. Iterates through every JSON file using `json.load()` and returns a dictionary. The dictionary contains the classes as keys and sub-dictionaries as

values. The sub-dictionaries contains image names belonging to that class as keys and their annotations as values.

4.3 Pre-processing



(a) Image with annotations

(b) Pre-processed image

Figure 6: Image before and after being pre-processed

Since the training dataset is small the model can learn from noise and background details in the images. To avoid overfitting the images are pre-processed. Since the goal of the project is to detect hand gestures, we can crop out the hands to remove unnecessary details from the image.

This is done by a function called `preprocess()`. This function takes the dictionary returned by `img_to_dict()`, the source path of the images that should be pre-processed and the destination path of where the pre-processed images should be stored as arguments. The function iterates through every image and reads the image in RGB format with the use of `cv2.imread()`. The hands in each image is then cropped with the help of the `bboxes` and the cropped image are resized to 300x300 pixels. This is because the size of each image should be the same when training the model.

4.4 Train and validation datasets

As mentioned in chapter 4.2 500 images of each class from the HaGRID test dataset is used for training and validating the model. This is done using the 80/20 split on the pre-processed images.

400 images of each class is used for training the model and 100 images of each class is used to validate the model.

The TensorFlow function `keras.utils.image_dataset_from_directory()` is used when creating the datasets. This function takes the path of the folder containing the class subfolders of the pre-processed images, the size of the validation split, the subset that are being created(training or validation), a seed, the image size and the batch size as argument. Since this function is called twice, once for the training set and once for the validation set, the same seed has to be used to avoid some of the same images being in both sets. The batch size is the number of samples processed before the model is updated

4.5 Convolutional Neural Network

Layer (type)	Output Shape	Param #
sequential (Sequential)	(None, 300, 300, 3)	0
rescaling (Rescaling)	(None, 300, 300, 3)	0
conv2d (Conv2D)	(None, 300, 300, 16)	448
max_pooling2d (MaxPooling2D)	(None, 150, 150, 16)	0
conv2d_1 (Conv2D)	(None, 150, 150, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 75, 75, 32)	0
conv2d_2 (Conv2D)	(None, 75, 75, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 37, 37, 64)	0
dropout (Dropout)	(None, 37, 37, 64)	0
flatten (Flatten)	(None, 87616)	0
dense (Dense)	(None, 128)	11214976
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 18)	2322

Total params: 11,240,882
Trainable params: 11,240,882
Non-trainable params: 0

Figure 7: Summary of the convolutional neural network

Building a convolutional neural network requires a lot of trial and error. This is because we don't want to either overfit or underfit the model. The model used for this project is seen in figure 7.

4.5.1 Create and compile model

The model is made using the TensorFlow function `keras.Sequential()`. This function groups a linear stack of layers into a `keras.Model`. The layers used to build the model is imported from `keras.layers`.

At the top of the model some data augmentation is done with the layers `RandomFlip("horizontal")` which randomly flips the input image horizontally and `RandomRotation(0.05)` which randomly rotates the input image in the range of $[-0.05 \cdot 2\pi, 0.05 \cdot 2\pi]$. These layers augment the images so that the train dataset appears larger than it is and reduces the chance of overfitting.

The data in the input image array is then normalized to a value between 0 and 1. This is done with the `Rescaling()` layer.

After the image normalization 3 sets of alternating `Conv2D()` layers and `MaxPooling2D()` layers are used. These are both explained in 3.3. The first convolutional layer uses 16 filters, the second layer uses 32 filters and the third layer uses 64 filters. The size of all these filters are $3 * 3$.

After the third max pooling layer a dropout is performed using the function `Dropout(0.2)`. This layer in only active during training and drops 20% of the nodes in the layer. This is to prevent overfitting. The output image matrix is then flattened to a one dimensional array using the layer `Flatten()`. The output of the flatten layer is then sent to a `Dense()` layer of 128 neurons. Another dropout layer is then applied after these neurons before reaching the final `Dense()` layer of 18 neurons. These 18 neurons represent the classes of images.

The model is compiled using the TensorFlow `Adam` optimizer, the TensorFlow `SparseCategoricalCrossentropy` loss function and accuracy as metric to be evaluated.

4.5.2 Train and test model

The model was trained for 30 epochs. One epoch is a complete pass through the training set. An example of a test with the gesture `ok` is seen in figure 8. More information about the training and testing of the model can be seen in chapter 5.0.1 and chapter 5.0.2.

4.6 Hand detection

We use the `mediapipe` implementation explained in 3.2 to do the heavy lifting in our detection of hands. We provide the library with an image, it will then return a list of hands, where a hand is a list of the 21 landmarks of the hand. We can then loop through these lists to figure out which points are at the maximum and minimum positions, essentially finding the corners of a box that contains the entire hand.

When we have the entire box that contains the hand, a cropping of the original image is performed, we resize it such that it becomes the standardized size that was used to train the model. We take both the cropped and the resized cropped image and put them into a new instance of a `humanHand` class. It's a simple data class that stores the two mentioned images, and the list of landmarks for the particular hand. Its this cropping and resizing that constitutes the pre-processing of the image.

The function named `detectHand` does the process described above. It returns a list of the created `humanHand` objects if any hands where found, else if no hands were detected, return a Python `None`.

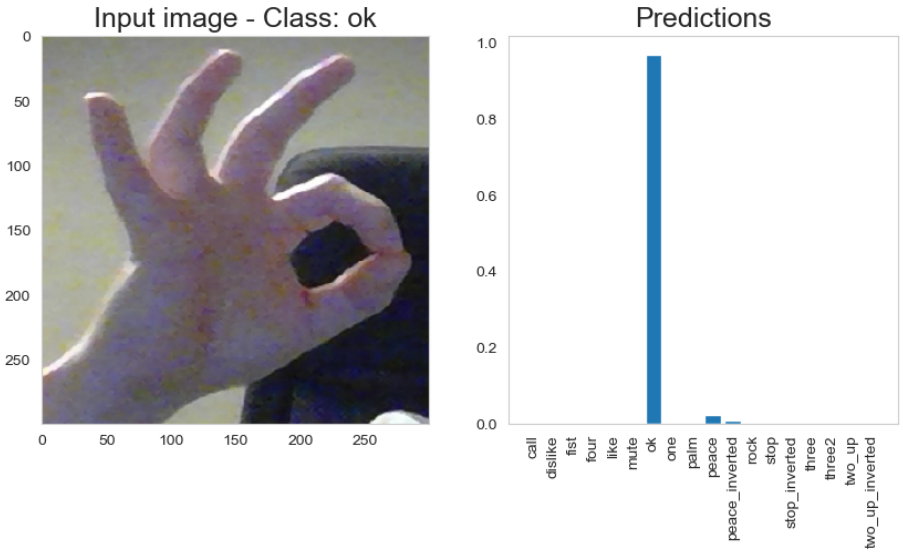


Figure 8: Test example of trained model

We can then simply loop through the list of returned hands and send their resized image through too the trained model.

4.7 Applying model with hand detection

The last part of the project combines the hand detection mentioned in 4.6 with the convolutional neural network mentioned in chapter 4.5. A function named `hand_gesture_detection` combines both these parts. It takes the path of the image as argument. The function calls the `detectHand` function, packs the cropped and resized image into a `tensor` object and then calls the predict function of the model. After this it plots the input image together with a graph of the predictions.

5 Experiments and results

This chapter goes through and discusses the experiments and results of the project.

5.0.1 Training results

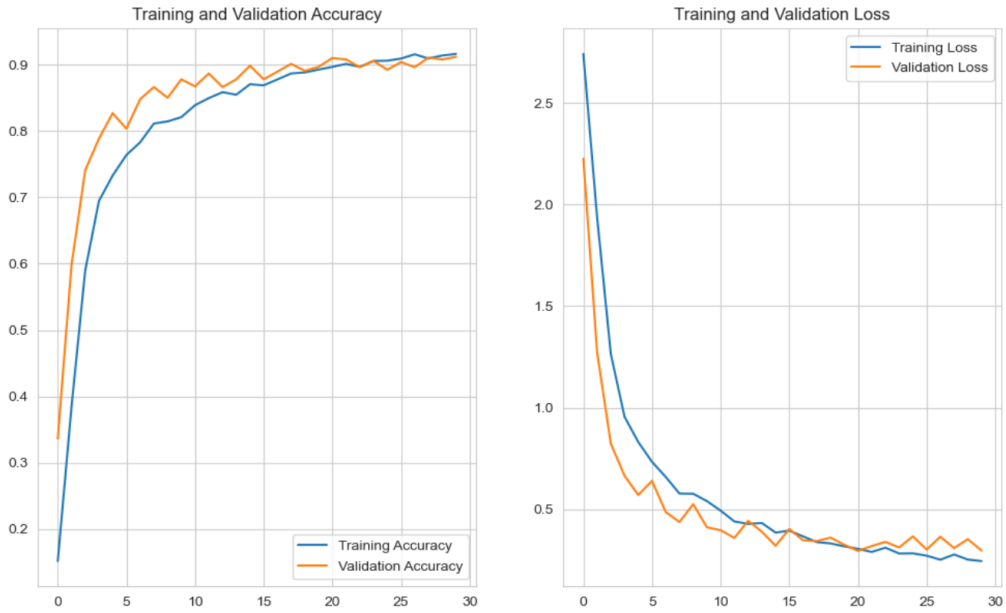


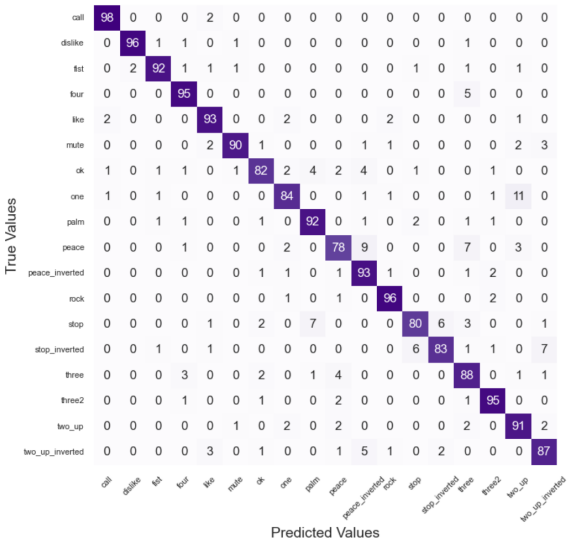
Figure 9: Results of training the model for each epoch

As seen in figure 9 the training of the model went very well. At the beginning of the training process the validation accuracy is much higher than the training accuracy. This is because of the data augmentation used on the training data set. Ideally the training should have stopped at epoch 22 when the accuracy of the training dataset intersected with the accuracy of the validation dataset. However at epoch 30 there is a minimal difference between both accuracies which makes us conclude that almost no overfitting occurred. The training and validation accuracy was both at around 91% at the end of the last epoch.

5.0.2 Testing results

	precision	recall	f1-score	support
call	0.96	0.98	0.97	100
dislike	0.98	0.96	0.97	100
fist	0.95	0.92	0.93	100
four	0.91	0.95	0.93	100
like	0.90	0.93	0.92	100
mute	0.96	0.90	0.93	100
ok	0.90	0.82	0.86	100
one	0.89	0.84	0.87	100
palm	0.88	0.92	0.90	100
peace	0.86	0.78	0.82	100
peace_inverted	0.82	0.93	0.87	100
rock	0.94	0.96	0.95	100
stop	0.89	0.80	0.84	100
stop_inverted	0.91	0.83	0.87	100
three	0.79	0.88	0.83	100
three2	0.92	0.95	0.94	100
two_up	0.83	0.91	0.87	100
two_up_inverted	0.86	0.87	0.87	100
accuracy			0.90	1800
macro avg	0.90	0.90	0.90	1800
weighted avg	0.90	0.90	0.90	1800

(a) Summary of testing results



(b) Confusion matrix of testing results

Figure 10: Testing results

As mentioned in chapter 4.2 100 images of each class were pre-processed and used for testing the model. Figure 10 shows the main metrics for each class and the confusion matrix for the test. As when training our model the accuracy of the test is 90%. This is a sign that our model is not overfitted.

As seen in the figure there is some variations between the results of the classes. An example is the class `call` which has a recall of 0.98 and the class `peace` which has a recall of 0.78, a difference of 0.2. This is because the call gesture is unique, while the peace gesture has lookalikes such as `peace_inverted` and `three`. This is the case for alot of the gestures which is why we see variations between the classes.

5.0.3 Resulting model with hand detection

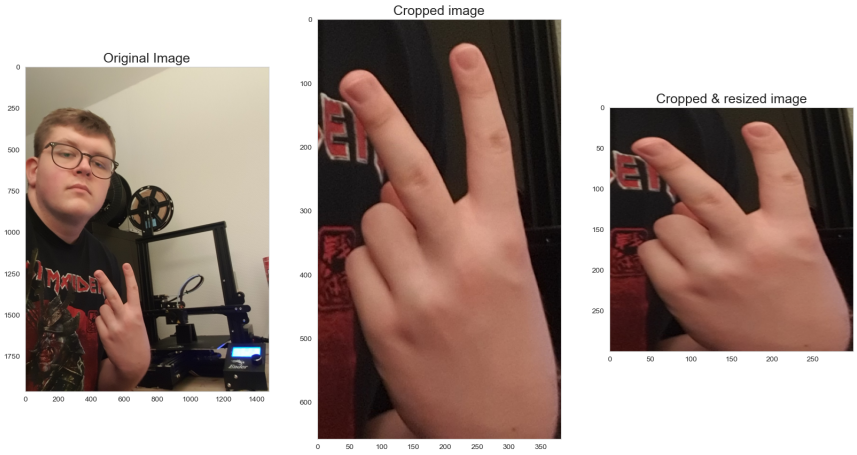


Figure 11: Result from detectHand function

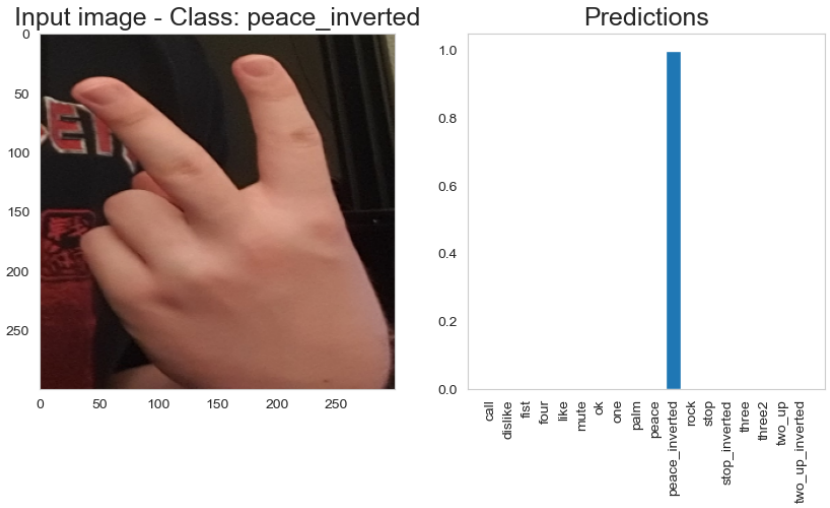


Figure 12: Result from hand_gesture_detection function

The solution is able to correctly figure out where the hand is located. It correctly cropped the image and resized it, processed it in the trained model, that then gave a correct prediction.

6 Conclusion

Our model is surprisingly good at predicting hand gestures relative to the size of the dataset used to train the model. With an average f1-score of 0.8967 we are still far away from the pre-trained

models the creators of the HaGRID dataset has provided on their Github page. All of their models have f1-score in the upper 0.9 range, where the best pre-trained model has an f1-score of 0.9928. Their model is however trained on the full training dataset consisting of 28 times more images than our training dataset.

References

[1] Hagrid. `https://github.com/hukenovs/hagrid`. Accessed: 2022-11-05.

[2] Hagrid classes. `https://github.com/hukenovs/hagrid/raw/master/images/gestures.jpg`. Accessed: 2022-11-10.

[3] Matplotlib. `https://matplotlib.org/`. Accessed: 2022-11-07.

[4] Max pooling. `https://production-media.paperswithcode.com/methods/MaxpoolSample2.png`. Accessed: 2022-11-12.

[5] Mediapipe hands. `https://google.github.io/mediapipe/solutions/hands.html`. Accessed: 2022-11-09.

[6] Numpy. `https://numpy.org/`. Accessed: 2022-11-07.

[7] Opencv. `https://opencv.org/`. Accessed: 2022-11-07.

[8] Overfitting. `https://aigeekprogrammer.com/wp-content/uploads/2019/12/model-accuracy-cnn-2-1.png`. Accessed: 2022-11-12.

[9] Scitkit-learn. `https://scikit-learn.org/stable/`. Accessed: 2022-11-08.

[10] Seaborn. `https://seaborn.pydata.org/`. Accessed: 2022-11-09.

[11] Tensorflow. `https://www.tensorflow.org/`. Accessed: 2022-11-07.

[12] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.

[13] Munir Oudah, Ali Al-Naji, and Javaan Chahl. Hand gesture recognition based on computer vision: a review of techniques. *journal of Imaging*, 6(8):73, 2020.

[14] Jesus Suarez and Robin R. Murphy. Hand gesture recognition with depth images: A review. In *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*, pages 411–417, 2012.

Appendix

Github

Our source code is located on github.
`https://github.com/erikx50/Hand-gesture-detection`
The file that contains the all the code is `HandGestureDetection.ipynb`