HANDS – FACE RECOGNITION USING MACHINE LEARNING

Ma422 – Introduction Machine Learning

Antoine Castel / Diego De Sousa / Luca Lizon / Adrien Monteiro
SET

Table des matières

Introduction

Artificial intelligence (AI) and machine learning (ML) are being incorporated into a wider range of applications in today's technologically advanced society. The development of detective systems that improve security and user engagement is one important area of progress. This project aims to create a hands and face recognition system that can be easily connected to a platform.

This project's main objective is to create and put into use a system that can precisely identify and recognize human faces and hands in real-time. With the use of sophisticated AI algorithms and computer vision techniques, the system will have great precision in identifying and tracking these traits. Then, this capacity offers up a wide range of possible applications, from better safety and monitoring in different situations to improved user authentication and engagement.

This project provides a practical platform for applying different machine learning algorithms such as face detection or hand gesture recognition, fostering a deeper understanding of AI and ML principles. Integrating this detection system with a connection platform will allow for real-time communication and interaction between users and the system. This connection platform will serve as a bridge, facilitating seamless data transfer and connectivity, thereby enabling a wide range of functionalities.

This hands-and-face detection system linked to a connection platform will show via extensive testing and development that it can completely transform security protocols.
The project is structured into several distinct phases, each designed to build upon the previous one to create a robust hands-face recognition model:
First, we created our dataset to develop our face detection model. Then, we compared it to two other face detection models. Afterward, we used the Mediapipe model for the hand detection part. In the final phase, we aimed to create an application for registration and login using hand gesture recognition. For this, we built upon a project that utilized the Mediapipe model for recognizing several hand gestures. However, we needed to train the model further to recognize hand gesture numbers.

For access to the complete project, please visit the following GitHub link:
https://github.com/antoinecstl/ML-project-Hands-Face-Recognition-connection-platform-

1. Face Detection Model

Construction of the dataset

The goal of this project is to develop a hands-face recognition model linked to a connection platform using machine learning techniques. For that we need to create our dataset of images to train our model. In this part you will understand the different steps taken to collect data, process images, and apply transformations to prepare the dataset for model training.

The initial phase involves collecting images using a webcam. We capture 30 images per run and ran the code 4 times to have approximately 100 images saved in a specified directory. The webcam is accessed using OpenCV's cv2.VideoCapture(0), and a loop is set up to capture images iteratively. Each iteration involves capturing a frame and checking if the capture was successful. If successful, the image is saved with a unique filename in the IMAGES_PATH directory. The captured frame is displayed on the screen to monitor the process, and a short delay of 0.5 seconds is introduced between captures to allow for slight movements and variations in the images. Additionally, the loop can be exited early by pressing 'q'. This step ensures that we have a diverse set of images, which is crucial for training, validating, and testing the recognition model.

After collecting the images, the next step is to load and visualize them using TensorFlow. We create a dataset of image file paths using tf.data.Dataset.list_files(). A function is defined to read and decode JPEG images, which is then applied to the dataset. Using Matplotlib, a subplot is created to display these images. This visualization step helps verify that the images have been correctly captured and can be read into the model pipeline without issues.

To ensure proper model evaluation, the dataset is split into training, validation, and test sets. Separate directories are created for each of these sets. The images are randomly shuffled to ensure a random distribution. The dataset is split into 70% training, 15% validation, and 15% test sets. A custom function is used to copy the images and their corresponding labels to the respective directories. This organization and splitting of data into distinct sets are essential for training the model, validating its performance, and testing its generalization capabilities.

Before growing our dataset, we used labelme to train our model to detect faces on images. Labelme is a software which we use on all images of our dataset to draw a rectangle where the face is on the image. The rectangle coordinates is then registered into a json file.

At the beginning, we had only 100 images and it wasn’t enough at all to create our model so to enhance the dataset and improve the model's robustness, various augmentations are applied using the albumentations library. We define several transformations such as random cropping, flipping, brightness/contrast adjustments, gamma corrections, RGB shifts, and vertical flipping. An image and its corresponding label are loaded from the training set. The bounding box coordinates are normalized relative to the image dimensions, and the defined augmentations are applied to both the image and the bounding box. This augmentation step helps create a more diverse dataset, which can lead to a more robust and generalizable model by simulating different scenarios and variations that the model might encounter in real-world applications.

The next phase involves visualizing bounding boxes on augmented images and performing additional data augmentation.

We start by applying transformations to the images and displaying the resulting bounding boxes. The coordinates of the bounding boxes are scaled appropriately, and rectangles are drawn on the augmented images using OpenCV's cv2.rectangle function. The image is then displayed using Matplotlib to visually confirm the augmentation process.

Following the visualization, we process all images in the training, validation, and test sets. Each image is read, and its corresponding label is loaded. If an annotation exists, the bounding box coordinates are extracted and normalized. For images without annotations, a default small bounding box is assigned. This ensures that the model can handle both positive and negative samples.

For each base image, we generate 60 augmented versions using the previously defined augmentation pipeline. Each augmented image and its corresponding label are saved in the aug_data directory. The labels include the bounding box coordinates and a class indicator (1 for faces, 0 for non-faces).

After augmenting the data, we prepare the datasets for training, validation, and testing. This involves loading the images and labels into TensorFlow datasets, resizing the images, normalizing pixel values, and adjusting contrast.

We use TensorFlow's tf.data.Dataset.list_files to load the file paths of the images and labels. The images are then read and resized to 180x180 pixels using tf.image.resize. The pixel values are normalized to the range [0, 1], and the contrast is adjusted.

The labels are loaded using a custom function load_labels, which reads the JSON files and extracts the class and bounding box information. These labels are then mapped to TensorFlow datasets.

Next, we combine the image and label datasets into a single dataset for training, validation, and testing. This is done using tf.data.Dataset.zip, which pairs each image with its corresponding label.

The combined datasets are shuffled to ensure a random distribution of samples and batched into groups of 8. Batching helps in efficient processing during training. The datasets are also prefetched to ensure that data loading does not become a bottleneck during training.

To verify the data preparation, we visualize some sample images along with their bounding boxes. We retrieve a batch of images and labels from the training dataset and draw rectangles on the images using the bounding box coordinates.

Using Matplotlib, we display a set of four sample images. Each image is scaled back to the original pixel value range, and the bounding boxes are drawn to verify that the coordinates are correctly aligned with the faces in the images.

Model training

In this section, we construct a deep learning model designed for face detection, leveraging TensorFlow's Functional API. Our model is built upon the pre-trained VGG16 architecture, which is commonly used for image recognition tasks. The VGG16 model is included without its top classification layer to serve as the backbone of our custom model.

The build_model function is defined to create our neural network architecture. This function starts by specifying the input layer, which expects images of size 180x180 with three color channels (RGB). The pre-trained VGG16 model is then applied to this input layer.

Our model has two distinct output heads to address two separate tasks. The first head is a classification model that determines whether an image contains a face. It uses a GlobalMaxPooling2D layer followed by a dense layer with ReLU activation and a final dense layer with sigmoid activation to output a binary classification.

The second head is a regression model responsible for predicting the bounding box coordinates of the detected face. Similar to the classification head, it also uses a GlobalMaxPooling2D layer, followed by dense layers with ReLU and sigmoid activations, respectively. The sigmoid activation ensures that the output values are scaled between 0 and 1, which is suitable for bounding box coordinates.

After defining the model architecture, the facetracker model is instantiated and summarized to inspect its structure and parameters.

With the model architecture in place, we proceed to prepare for the training process. First, we obtain a batch of training data and use the model to make initial predictions. This step helps verify that the data pipeline and model outputs are correctly configured.

To optimize the model, we define a learning rate decay strategy tailored to the number of training batches per epoch. The Adam optimizer is chosen for its adaptive learning rate capabilities, which are beneficial for training deep neural networks.

A custom localization loss function is implemented to handle the bounding box regression task. This function calculates the difference between the true and predicted coordinates, focusing on both the position and size of the bounding boxes. Additionally, a binary cross-entropy loss function is used for the classification task to handle the binary nature of face detection.

To streamline the training process, we create a custom training loop by defining a subclass of TensorFlow's Model class, named FaceTracker. This class includes methods for compiling the model and performing training and evaluation steps. The train_step method computes the loss for each batch, applies gradients, and updates the model's weights. Similarly, the test_step method evaluates the model on validation data without updating the weights.

Once the model and training loop are set up, we compile the FaceTracker model with the optimizer and loss functions.

The model is then trained for 10 epochs, with the training and validation datasets provided. Throughout the training process, losses for both classification and regression tasks are logged and visualized using TensorBoard.

After training, we evaluate the model's performance on a batch of test data. Predictions are made using the trained model, and the results are visualized to assess the accuracy of face detection and bounding box regression.

Model testing in realtime

After validating our model on our dataset, we wanted to test our model on livestream by using our camera. We know that videos are just images separated with a small delay, we knew that our model could work for livestream videos.

We apply the model on our webcam, and it works as we wanted.

You can check the *Homemade_Model_Test.mp4* video to see how our model works on livestream.

For the continuation of the project, we are going to use better trained models. We tried both MTCNN and Blazerface from mediapipe library. We conclude that we will use mediapipe one because it is faster and has better result.

2. Hand Detection

After creating our model of face detection and after comparing it with others models, we decided to add to our project an hand detection also. For the hand detection, we used the mediapipe hand detection model.

For that, first, we import the necessary libraries, cv2 for video capture and display, and mediapipe for face and hand detection models. Then, we initialize the MediaPipe models for face detection, hand detection, and drawing utilities, which help visualize the detected landmarks on the frames.

Next, we create instances of the face detection and hand detection models with specified parameters. For face detection we set a min_detection_confidence of 0.5, which is the minimum confidence threshold for detecting faces. Similarly, for hand detection, we set min_detection_confidence to 0.5, and min_tracking_confidence to 0.5 also for detecting and tracking hands.

We set up video capture from the webcam using OpenCV and configure the frame width and height for the captured video stream.

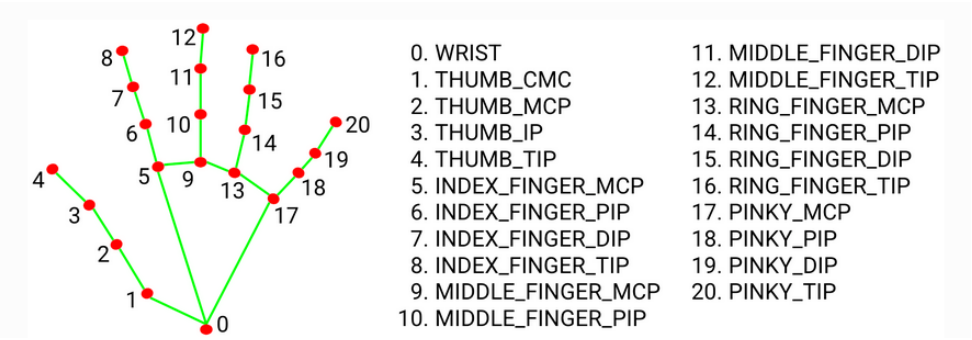
We enter a loop to process each frame from the webcam. If a frame is not successfully read, we skip to the next iteration. Inside the loop, we convert the frame from BGR to RGB because the mediapipe model uses RGB images and set the image to non-writeable to optimize performance during processing.

We then run the face and hand detection models on the current frame. After processing, we revert the image to writeable and convert it back to BGR format for display.

If faces are detected, we draw the detection results on the frame using the drawing utilities provided by MediaPipe. Similarly, if hands are detected, we draw the hand landmarks with specified drawing specifications, such as color, thickness, and circle radius.

3. Hand Gesture Recognition

The objective of this project is to have a connection page using hand gesture to type the password. For that, despite having a hand detection model we will need one that interpret what gesture or number the user is doing.
The good point is that the MediaPipe hand detection model gives us the coordinates of different point on the hand :



Having that information, we can construct another model that will treat these coordinates and recognize what number the user is showing to the camera.

To build this idea, we found a project on GitHub of someone that made something close to what we wanted.

You can find it here: https://github.com/Kazuhito00/hand-gesture-recognition-using-mediapipe

This project creates a model based on those coordinates and give us a snippet of code to train it on hand gesture.

The base model given for hand gesture recognition only has "Ok, Open, Close, Pointer" recognition pattern but we want to modify it to have number recognition from 0 to 5 on each hand so the user can enter their password which will consist of numbers.

To do so we have modified the keypoint_classifier_label.csv file which register the coordinates of those hand points linked to a value which must be between 0 and 9 in our case 0 to 5 which are the numbers that we can represent with one hand. We registered more than 3000 hand coordinates linked to the hand representation of number with the right and left hand.

After finishing the hand position dataset, we ran the keypoint_classification_EN.ipynb to train the model on those coordinates. The model that is used is the Sequential one given by keras. This TensorFlow Keras Sequential model is designed for a classification task. It begins with an input layer that expects data with 42 features. This specific dimensionality is determined by the input shape of 21×221×2.

The model incorporates several layers to process the input data. First, a Dropout layer is introduced with a dropout rate of 20%. This layer randomly sets 20% of the input units to zero at each training step. The purpose of this dropout layer is to prevent the model from overfitting by ensuring that it does not rely too heavily on any individual input feature.

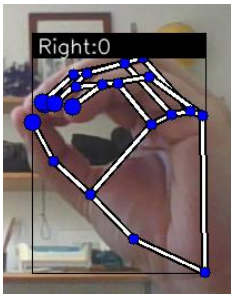
Following the initial dropout layer, the model includes a fully connected Dense layer with 20 neurons. This layer uses the ReLU (Rectified Linear Unit) activation function, which helps introduce non-linearity into the model, allowing it to learn more complex patterns in the data.

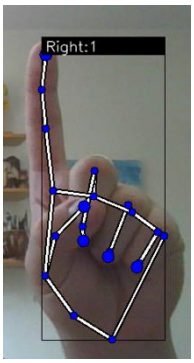
To further enhance regularization, the model includes another Dropout layer, this time with a higher dropout rate of 40%. This layer further reduces the risk of overfitting by randomly setting 40% of the input units to zero during training.

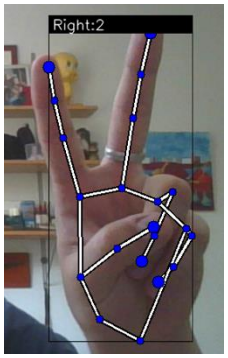
The model then adds a second Dense layer, this time with 10 neurons, again using the ReLU activation function. This layer continues to process the data, extracting relevant features and patterns necessary for classification.

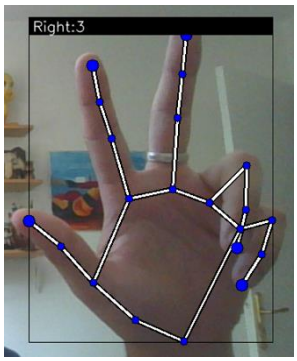
Finally, the model concludes with an output Dense layer. The number of neurons in this layer is equal to NUM_CLASSES, representing the different classes the model aims to predict. It uses the softmax activation function, which converts the raw output scores into probabilities, effectively categorizing the input data into one of the predefined classes.

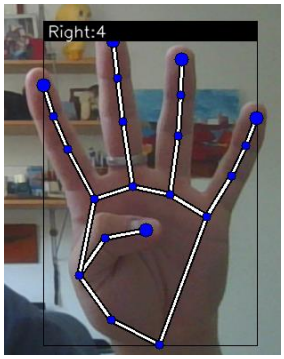
Now trained, our model detects different hand positions related to number. You can see some of them here and in the video named : *Hand_number_recognition.mp4*

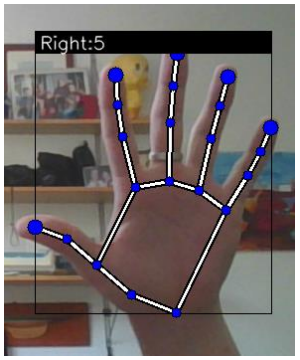










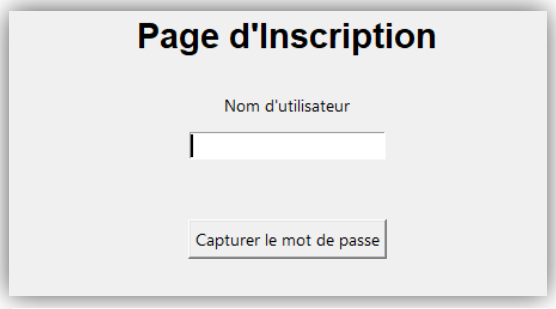


4. Tkinter connection platform

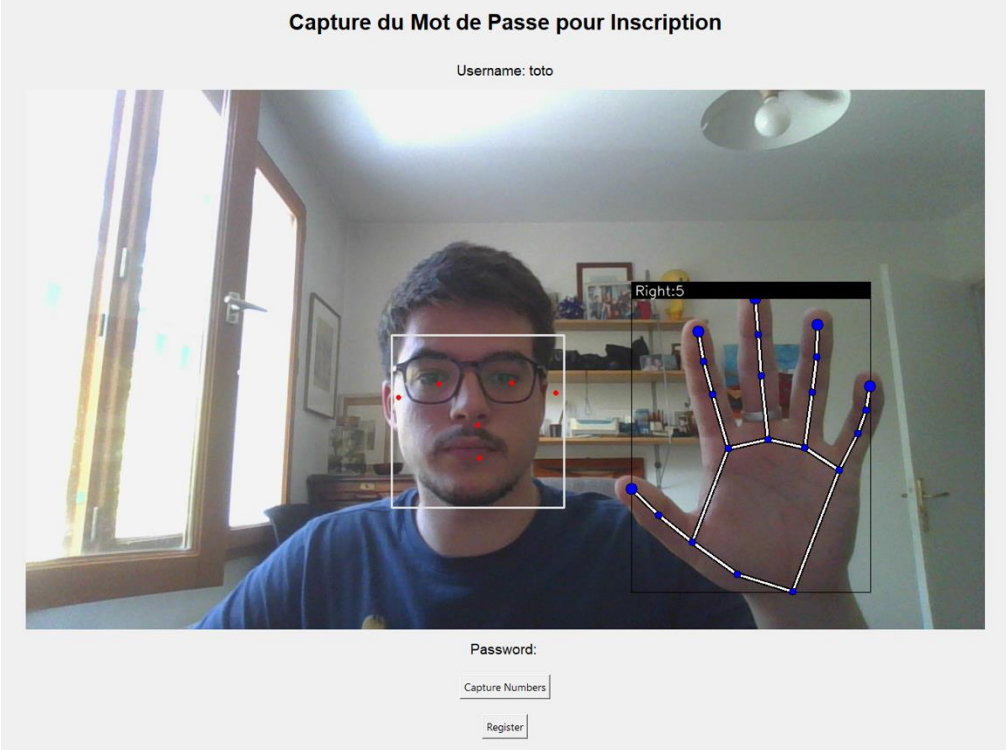
For this project, we wanted to create something where we could use our models that we created before. We thought of a connection platform where people can register and login in by using a username and a password. The password is made of numbers captured by our webcam using the hand gesture recognition.

The application's graphical user interface (GUI) is built using Tkinter, offering various pages for the registration, login, password capture, and dashboard. Hand gesture recognition is achieved using MediaPipe model as we explained before, which captures hand gesture numbers and displays them as a password during registration and login.

Home page



Register page



Password capture page to register

Here are some pictures of the registering part of the platform. You can see a more in depth example of usage on this video : *Platform_usage.mp4*

User data (username and password) is stored in an SQLite database (passwords.db) for verification during login and to store new user registrations. Our code also deals with error handling for duplicate usernames, empty fields, and incorrect passwords.

The application initializes video capture and updates the video frames using OpenCV.

Overall, our code provides a straightforward implementation of hand gesture recognition for user authentication within a Tkinter-based GUI application, combining computer vision techniques with user interface design to create an interactive and secure authentication system.

5. Improvements

While our project has successfully developed a robust hands and face recognition system integrated with a connection platform, there are several advanced features that we have not yet implemented. These include the ability to differentiate between individual faces and the capability to quantify hand movements for conversion into specific information. Below, we explain these potential improvements and the challenges associated with their implementation.

Facial Recognition for Individual Identification:

Our current system can detect faces but cannot distinguish between different individuals. Implementing facial recognition involves creating a model that can not only detect a face but also identify it as belonging to a specific person. This requires several additional steps:

1. Dataset Requirements: We would need a comprehensive dataset containing multiple images of everyone to train the model to recognize different faces accurately. Gathering such a dataset can be time-consuming and resource-intensive, especially ensuring that it includes sufficient variations in lighting, angles, and expressions.
2. Model Complexity: Adding facial recognition increases the complexity of the model. It would require advanced deep learning architectures such as convolutional neural networks (CNNs) combined with additional layers for feature extraction and classification. Popular models like FaceNet or the usage of the Dlib library can be explored, but these would significantly increase computational demands.
3. Performance Optimization: Real-time facial recognition necessitates efficient processing to ensure that the system remains responsive. Optimizing the model for speed without compromising accuracy would be a challenging task, involving techniques like model quantization and leveraging GPU acceleration.

Hand Movement Quantification:

Currently, our system can detect hand gestures, but it does not quantify hand movements to transform them into actionable information. Implementing this feature involves several complexities:

1. Data Annotation: We need annotated data that captures various hand movements and the corresponding information they represent. This involves extensive manual labeling and ensuring that the dataset covers a wide range of gestures.
2. Gesture Recognition Model: Developing a model capable of recognizing and quantifying hand movements involves understanding not just static gestures but dynamic sequences of movements. Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM) networks could be used to handle the temporal aspect of gestures.
3. Integration with Application Logic: The recognized gestures need to be integrated seamlessly into the application logic to perform specific actions. This requires designing a robust interface between the gesture recognition model and the application backend.
4. User Variability: Different users might perform gestures differently, and the system must be trained to handle such variability. Ensuring the model generalizes well across different users is crucial, which might require diverse and extensive training data.
5. Latency and Real-time Processing: Facial recognition and gesture recognition need to be processed in real time. This necessitates efficient algorithms and potentially leveraging hardware accelerations, such as GPUs or dedicated AI processors.

To summarize:

Our hands-and-face recognition system would be much improved by the addition of these cutting-edge capabilities, increasing its adaptability and usefulness in a range of scenarios. These advancements do, however, provide a unique set of difficulties, chiefly those about data requirements, model complexity, processing demands, and privacy issues. To achieve successful implementation, addressing these issues would need meticulous planning, in-depth study, and maybe large resources.