9 OCR

9.1 What is OCR?

OCR is an abbreviation for *Optical Character Recognition*. It is a field of technology that allows computers to recognize printed or handwritten text in images, scanned documents, or even a live camera feed and translate it into machine-readable text. OCR is a crucial component of many text-based applications, including document management systems, the digitalization of antiquarian books, and automated data entry.

OCR technology uses a variety of methods, such as *machine learning*, *pattern recognition*, and *image processing*. OCR employs machine learning algorithms to discover and identify letter, numeric, and symbol patterns in image data. These algorithms allow the OCR software to continuously learn from the data it processes, allowing it to increase its accuracy over time.

OCR is considered a part of machine learning because it uses supervised learning algorithms to recognize and classify text in images. The training data used for OCR algorithms are typically labeled images of characters and words, which the algorithms use to learn and recognize patterns in the data.

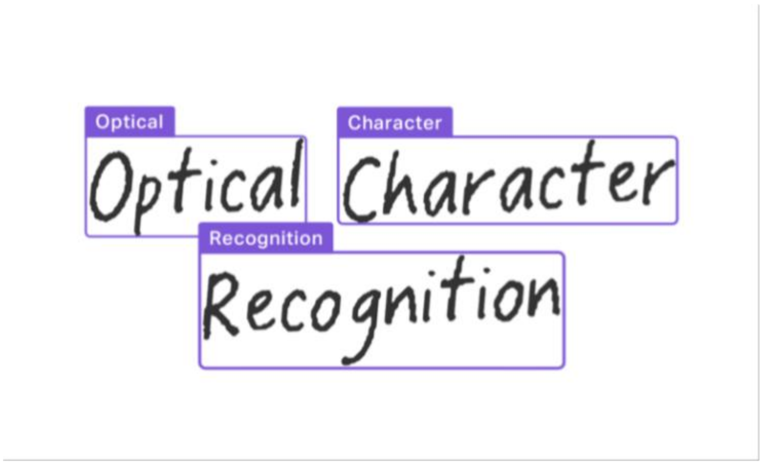


Figure 9.1: Example of a OCR showing recognition of characters

9.2 OCR on Microcontrollers

9.2.1 Requirements

General Requirements

To perform OCR, we need the following components:

Optical scanner or camera: A device that can capture the image of the printed or handwritten text that you want to recognize. This can be an optical scanner or a camera.

Image pre-processing software: OCR requires high-quality images of the text to be recognized. Hence, the captured image needs to be pre-processed to correct for distortions, remove noise, enhance contrast, and normalize the size and orientation of the text.

Optical character processing: OCR uses machine learning algorithms to recognize and convert the text in the pre-processed image into machine-readable text. The procedure uses a combination of image processing, pattern recognition, and machine learning techniques to recognize characters, words, and sentences in the image.

Language model: OCR needs a language model that defines the set of characters, symbols, and language rules to be recognized in the image. The language model helps the OCR software to recognize and classify text in different languages and scripts.

Output format: OCR produces the recognized text as output. The output can be in various formats.

These components work together to perform OCR and convert printed or handwritten text in images into machine-readable text that can be used for various applications such as document management, data entry automation, and digitization of printed materials.

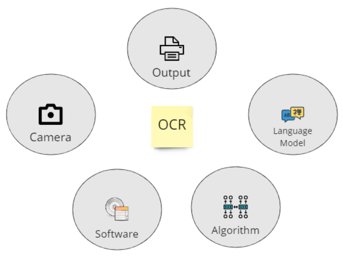


Figure 9.2: Components of OCR

Hardware Requirements

The hardware requirements for OCR (Optical Character Recognition) can vary depending on the complexity and volume of the OCR tasks that need to be performed. Here are some of the key hardware requirements for OCR:

CPU: OCR requires a processor with sufficient speed and processing power to handle the image pre-processing, character recognition, and language modeling tasks. A multi-core processor is recommended for faster performance.

RAM: OCR software uses a large amount of memory to store image data, language models, and other processing information. Therefore, a minimum of 4GB of RAM is recommended for OCR tasks.

Storage: OCR tasks require a significant amount of storage to store the input images, pre-processed images, and output text files. The amount of storage required depends on the size and number of documents to be processed.

Graphics card: A dedicated graphics card can help speed up the image pre-processing tasks, especially for large images or batches of images.

Scanner or camera: OCR requires a device that can capture high-quality images of the text to be recognized. This can be an optical scanner or a camera.

Display: A high-resolution display can help visualize the OCR results and identify any errors or issues in the recognition process.

Connectivity: OCR may require internet connectivity for language model updates or cloud-based OCR services.

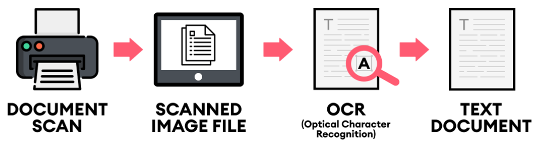


Figure 9.3: OCR Process Flow : Simple
Source: https://www.verypdf.com/wordpress/202302/pdf-to-text-ocr-converter-sdk-for-net-46522.html

9.2.2 Feasibility

OCR on Microcontrollers is feasible, but it can be challenging due to the limited processing power and memory resources of most Microcontrollers. However, there are several OCR libraries and techniques that have been developed specifically for Microcontrollers that can help overcome these limitations.

Approach 1 : To use *pre-trained machine learning models* that have been optimized for microcontrollers. These models are typically smaller in size and have lower computational requirements than their full-sized counterparts, making them more suitable for deployment on microcontrollers. One such example is the *TinyOCR* library(command-line), which is a lightweight OCR library specifically designed for microcontrollers.

Tesseract OCR

Figure 9.4: Example of pre-trained model
Source: https://www.wikipedia.com

Approach 2 : To use image processing techniques such as *binarization*, *edge detection*, and *contour detection* to segment text regions within an image. Once the text regions have been identified, OCR algorithms can then be applied to extract the text content. This approach is computationally less demanding than using machine learning models but may have lower accuracy.

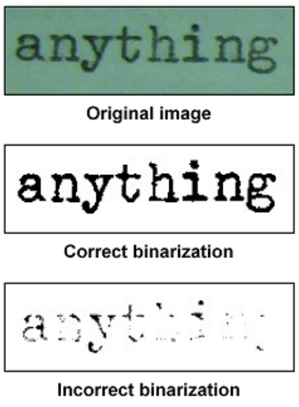


Figure 9.5: Binarization Example
Source: https://www.wikipedia.com

Approach 3 : To use *cloud-based OCR services*, which offload the heavy processing requirements to a remote server. The Microcontroller can then send image data to the cloud OCR service for text recognition and receive the recognized text back. This approach requires an internet connection and may have latency and privacy implications. In summary, implementing OCR on Microcontrollers requires careful consideration of

Figure 9.6: Cloud based OCR Example
Source: https://pyimagesearch.com

the available resources and the desired level of accuracy and performance. Depending on the use case, pre-trained machine learning models, image processing techniques, or cloud-based OCR services may be a viable solution.

9.3 Obstructions of OCR on ESP32

9.3.1 General Restrictions

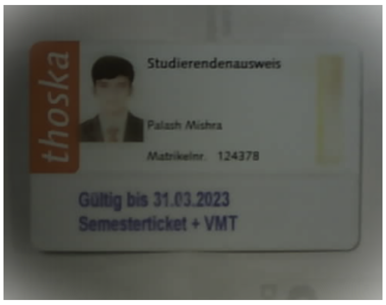
While the ESP32 Microcontroller can provide some capabilities for performing OCR along with external processing support, there are also some restrictions to keep in mind:

Processing power: While the ESP32 has a dual-core processor, it may not have enough processing power to perform OCR on images with different characters or with complex algorithms. This can lead to slow performance or errors in the OCR process.

Memory limitations: The ESP32's memory is limited and may not be sufficient to store large image datasets or complex OCR models. This may require the use of external memory or cloud-based storage.

Image quality: The quality of the images being processed can significantly impact OCR accuracy. Low-quality images, such as those captured by low-resolution cameras or with poor lighting, may result in incorrect OCR results.

Limited camera support: While the ESP32 has interfaces for connecting to cameras, it may not support all types of cameras or offer the resolution or image quality necessary for OCR.



thoska

Studierendenausweis

Palash Mishra

Matrikelnr. 124378

Gültig bis 31.03.2023

Semesterticket + VMT

Figure 9.7: Sample Image Quality of ESP32 in close proximity

Limited OCR libraries: There may be limited OCR libraries or software packages available that are compatible with the ESP32 microcontroller.

Overall, while the ESP32 microcontroller can provide some capabilities for performing OCR, it is important to keep these restrictions in mind when considering OCR applications on this platform. Careful consideration of the specific application requirements and limitations of the ESP32 can help ensure successful OCR performance.

9.3.2 Implementation Restrictions

Prohibition of Model Storage : OCR models typically require a minimum file size of more than 4MB, which is beyond the storage capacity of the ESP32. This limitation was taken into account during implementation to prevent the need for external storage. Moreover, support for storing large models in SD-Cards has also been disregarded.

Prohibition of Cloud Services : In order to leverage cloud-based OCR services, which effectively delegate the computationally intensive processing requirements to a remote server, a reliable internet connection is a prerequisite. However, due to the constraints imposed by the requirements, any reliance on internet access has been strictly avoided in the implementation.

Prohibition of External Processing : OCR demands a high processing power, which is considerably greater than the processing capacity of ESP32. This is due to the fact that OCR entails not only recognition but also segmentation, thereby necessitating the assistance of external processing for ESP32 to perform OCR with precision and efficiency. However, the implementation has refrained from employing any form of external processing support.

Prohibition of higher-resolution camera module : The implementation must be executed with consideration for the limitation of OV2640 camera module usage. Given that OCR requires high resolution images to effectively perform character recognition, utilizing the OV2640 module may result in suboptimal accuracy. As such, the implementation is configured to use the minimum framesize of FRAMESIZE(QVGA) to avoid the occurrence of multi-threading errors or memory allocation issues that can arise from processing greater framesizes on the ESP32. In addition, this module doesn't support auto-focus.

Prohibition of higher-datasets with multi-features : When considering the matriculation number to be extracted from the thoska, it is important to note that it consists of 6 digits utilizing the digits 0-9, resulting in approximately 5,040 possible combinations. However, as the number of possible features increases, so does the processing power required for feature extraction. Given the current restrictions, if we were to use 6 classes for a number such as 124378 (1,2,4,3,7,8) on an image dataset of 100 images, with 6-9 defined classes to train the model, we would obtain approximately 30,000-307000 features. This would typically require 2-3 MB of RAM, and any further increase in features would necessitate additional RAM which is beyond the capacity of the ESP32.

Prohibition of pre-built libraries : The primary objective is to integrate machine learning on the ESP32 platform. Therefore, the utilization of pre-existing libraries on the ESP32 is not permissible and, as such, will not be entertained. It is noteworthy that there are minimal libraries tailored for the Arduino IDE that are pertinent to OCR, which can be adopted for implementation on the ESP32 platform.

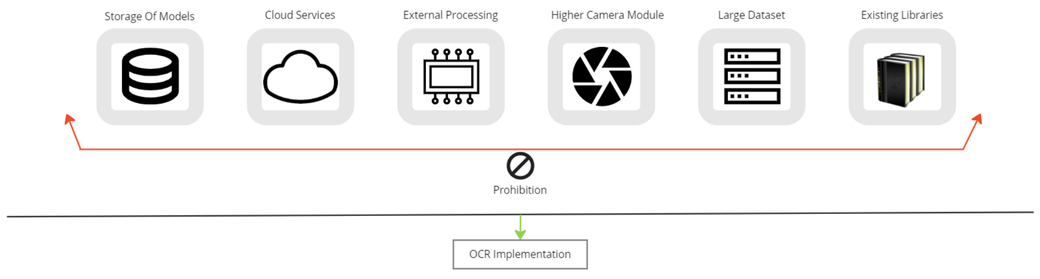


Figure 9.8: Implementation Prohibits the usage of the above components

9.4 Implementation Research So Far

9.4.1 Capability Check Approach : PyTesseract + OpenCV + Pillow

Overview

PyTesseract is a Python wrapper for the Tesseract OCR engine, which provides a convenient interface to perform OCR on various types of images.

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library that can be used to process and analyze images and video streams.

Pillow is a Python Imaging Library (PIL) fork that adds support for opening, manipulating, and saving various image file formats.

Approach Overview

This approach was attempted to explore the OCR capabilities of the ESP32 camera by using a Python script to fetch OCR from video footage captured by the ESP32 camera webserver. However, the accuracy of OCR was not satisfactory. To improve accuracy, potential factors such as image quality, OCR algorithm, and computational resources should be considered.

Technical Overview

GitHub Reference

This approach captures a video stream from an IP camera and uses the Tesseract OCR library to recognize text in the frames. It then overlays the recognized text and bounding boxes around the characters on the frames, and displays them using the OpenCV library. Here, it was required to have Tesseract built-library in the system.

This approach is using several Python libraries to capture video from a network camera (using RTSP protocol), apply optical character recognition (OCR) on each frame, and then display the frame with text overlayed on top of the recognized characters.

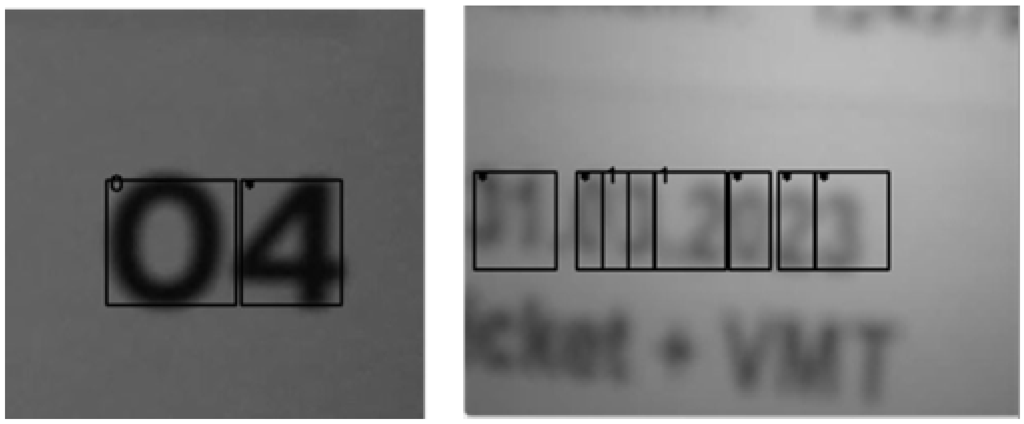


Figure 9.9: Inference Results

Shortcomings

This approach could not work as expected and the accuracy was very poor. As a result, the true capability of ESP32 for OCR could not be judged.

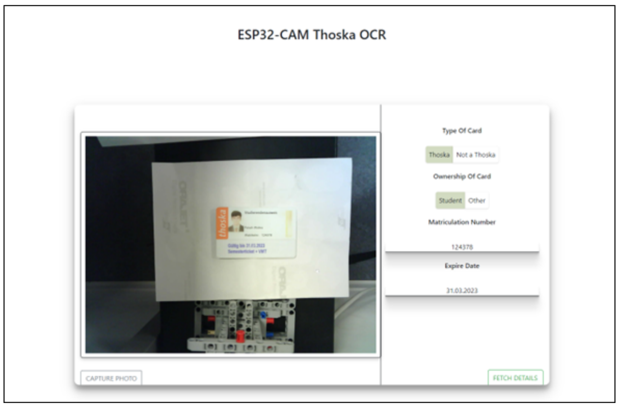
9.4.2 Cloud Based OCR Approach : Tesseract JS API

Overview

Tesseract JS API is a JavaScript library that provides OCR (Optical Character Recognition) capabilities and allows the extraction of text from images.

Approach Overview

The approach involved testing text recognition using Tesseract API on an Arduino and found that it worked well with large and clear words. The approach stored the image in flash memory using SPIFFS and built a frame using LEGO blocks to capture stable images for better detection accuracy. The wide aperture made text detection difficult, but it was found that capturing images from 35cm away produced good quality images of Thoska. By fixing the position of the ESP32 cam, it was able to capture better and stable images. The pre-processing of the captured images involved cropping them to eliminate unnecessary details and used Gaussian blur, sharpening, opacity, and RGB correction to improve image quality.



ESP32-CAM Thoska OCR

Type Of Card

[✓ Thoska] Not a Thoska

Ownership Of Card

[✓ Student] Other

Matriculation Number

124378

Expire Date

31.03.2023

Figure 9.10: Application Overview - Desktop

Technical Overview

GitHub Reference

Breaking down on main functions :

Wifi-Server Module

This module sets up a WiFi connection with a static IP address (192.168.0.122), sets the gateway IP address and subnet mask, and defines the network credentials (SSID and password). It creates an AsyncWebServer object on port 80 and connects to the WiFi.

Image-Capture Module

This module captures a photo using an ESP32 camera and saves it to the SPIFFS flash memory. When the "CAPTURE" button is pressed on the web server, a request is sent to the ESP32 to take a new photo. The photo is then captured and stored in a file named FILE_PHOTO using the SPIFFS.open() function.

Tesseract Module

This module defines a JavaScript function called scan() which is executed when a button is clicked. The function fetches an image from a web server (the URL is specified as img parameter) and uses the Tesseract OCR library to recognize text in the image. The OCR is configured to recognize German language (deu) and logs its progress using a custom function named showStates(). Once the OCR is completed, the function passes the recognized text to another custom function called checkIfImageLethal() for further processing. The resulting processed text is then passed to a function called extractMat() for formatting a matriculation number.

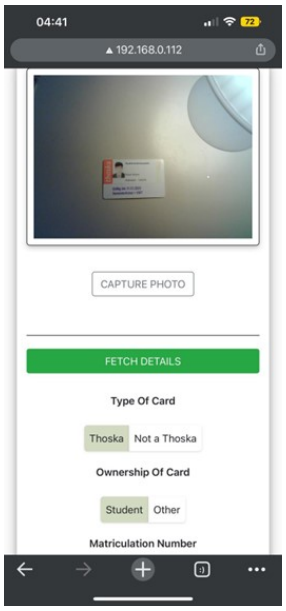


Figure 9.11: Application Overview - Mobile

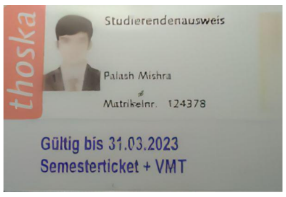


Figure 9.12: Pre-processing involved in the application

Shortcomings

This approach worked accurately and could perform OCR smoothly. However, since the aim of the project was to completely remove any internet usage, this approach could not be carried forward.

9.4.3 Capability Check Approach : Keras Model Training

Overview

Keras is an open-source neural network library written in Python that facilitates the creation of complex machine learning models.

Approach Overview

Prior to adopting the current approach, the built-in pipeline in Keras was utilized to assess the accuracy of the captured images. Unfortunately, the results were unsatisfactory. As a result, it was determined that training a custom Keras model may be a more viable option. Consequently, efforts were directed towards developing a custom model using the Keras library.

Results from the custom model and pipeline:

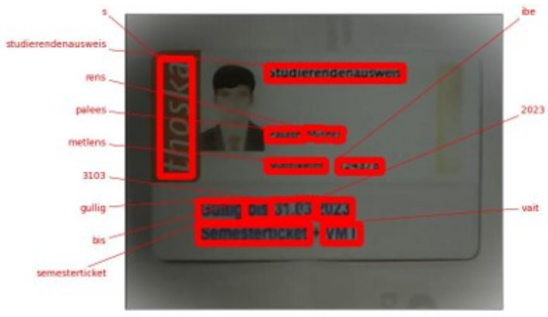


Figure 9.13: Results from pipeline

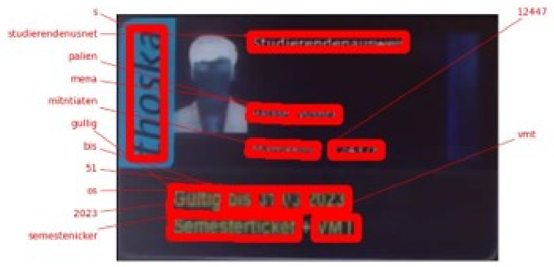


Figure 9.14: Results from custom model

Technical Overview

Pipeline :

Github Reference

To perform optical character recognition on images captured by an ESP32 device, the process uses the Keras OCR library. The first step is to set up the OCR pipeline by calling the keras_ocr.pipeline.Pipeline() function. Then, images from a specific folder path are read using the keras_ocr.tools.read() function and saved in a list called "images". After that, the pipeline.recognize() function is used with the "images" list as an argument to generate text predictions from the images. The predicted text is stored in "prediction_groups". Finally, the keras_ocr.tools.drawAnnotations() function is used to plot the text predictions on top of the original images, and the resulting images are displayed using Matplotlib's plt.subplots() function. The plot displays one row for each input image, with the predicted text overlaid on the image.

Custom Model :

Github Reference

This code implements optical character recognition using the keras_ocr library. It sets up a detector and a recognizer object for detecting and recognizing text in images, respectively. The detector is initialized using the keras_ocr.detection.Detector() function with the clovaai_general model, and the recognizer is initialized using the keras_ocr.recognition.Recognizer() function with the "kurapan" model. The detector is then trained to recognize and localize objects in images using a batch generator, and the existing generator is converted to a single-line generator for the text recognizer. The code also includes various configuration parameters, such as batch size and callbacks to monitor the training process.

Shortcomings

The primary objective of this approach was to develop a custom-trained keras model for text detection, which would later be converted into C++ code using the keras2cpp library. However, the results obtained from the model were unsatisfactory as the text within the target image was not accurately recognized. Despite being an improvement over the direct pipeline approach, which produced the matriculation number as an alphabetical value, the accuracy was still deemed insufficient for the intended application. Given the sensitivity of the operation, it was decided that this approach would be abandoned, and further research would be pursued.

9.4.4 Compiled Library Approach : Tesseract + Leptonica

Overview

Tesseract is an optical character recognition (OCR) engine developed by Google that is widely used for converting scanned images of text into digital text.

Leptonica is an open-source image processing and analysis library that provides extensive support for image processing operations, such as image scaling, cropping, and color space conversions.

Approach Overview

This approach was aimed to generate a build library so that it could be used directly in ESP to perform OCR

Technical Overview

Github Code Reference

Github Procedure Reference

This approach involved integrating the Tesseract optical character recognition (OCR) library into a Microsoft Visual Studio environment.

To get started with Tesseract in Visual Studio, there are several steps that need to be followed. The first step was to build the latest library using the Software Network client, which is a tool that allows for the easy management and installation of software packages. Once the latest library has been built, the next step was to install Git and Vcpkg, which are required to download the libraries needed for the approach.

After installing Git and Vcpkg, the Tesseract language data needs to be obtained, and the Tesseract libraries need to be set up for use in Visual Studio. This involves integrating Vcpkg with Visual Studio and adding the Tesseract libraries to the project. Finally, the Tesseract OCR engine can be used to recognize text in images by creating an instance of the Tesseract class and passing the image to be recognized.

Shortcomings

This particular approach was observed to be more time-consuming as compared to other methods due to the requirement of careful generation of libraries. Any small error or omission of a specific ".dll" file during the generation process could result in the entire process being interrupted. However, even after successfully generating the libraries, integrating them with the Arduino IDE proved to be infeasible due to the need for an additional command-line toolchain. Furthermore, it was also observed that the generated libraries were not compatible with the ESP32 used in the Arduino, and after several attempts, this approach was eventually abandoned.

9.4.5 TFLite Model Approach : TFLite Model + V8 Engine

Overview

TFLite Model: TFLite Model is a machine learning model designed to run on mobile and embedded devices, with optimized performance and reduced memory footprint.

V8 Engine: V8 Engine is an open-source, high-performance JavaScript engine used in Google Chrome and Node.js. It compiles and executes JavaScript code at lightning speeds, enabling efficient and smooth web browsing and server-side application development.

Approach Overview

The approach utilizes the MobileNetV2 architecture to perform digit detection in real-time. The system receives live feed from a camera, captures the image, and then analyzes the image to recognize the digits present. This is achieved by defining Regions of Interest (ROIs) on the captured image.

To enable real-time digit detection, an efficient image processing pipeline leverages the usage of the MobileNetV2 architecture.

To define ROIs, an algorithm is used that first identifies the location of each digit in the image and then creates a rectangular box around it.

Technical Overview

The proposed application employs the MobileNetV2 architecture for the TensorFlow Lite (TFlite) model to perform Optical Character Recognition (OCR) while simultaneously displaying live feed and associated values through a browser. The TFlite model is pre-trained, but its processing time takes approximately 5 minutes, as it must be fed at set intervals to perform OCR again. Unfortunately, this approach did not yield the desired outcome, as the Thoska detection is a random event.

To address these concerns, a user-friendly TFlite wrapper was incorporated , along with inline image processing, including feature detection, alignment, and Region of Interest (ROI) extraction. Despite these enhancements, the processing time remains slow, making it unfeasible to use for Thoska detection. The camera capture quality compatible for the model too hinders the process of detection, since the digits in thoska are too small.

Shortcomings

The proposed approach for Thoska detection necessitated the card to remain steady, and the frame to remain in place, as any movement would change the Region of Interest (ROI) for digit detection. However, due to the varying qualities and placements of different Thoska cards, it became challenging to define the ROIs accurately for every card. As a result, the overall operation was slow and impractical for Thoska detection.

The model employed a threshold-based classification approach for digit detection, with a few classes often clashing, leading to incorrect outputs or missing digits. This required significant processing time and capacity, and integrating it with face recognition would have been challenging.

Considering these limitations, the approach was eventually abandoned, and alternative solutions were explored for Thoska detection.

9.4.6 Local Model Approach : Tesseract JS Model

Overview

Tesseract JS : It is a JavaScript library that provides OCR (Optical Character Recognition) capabilities and allows the extraction of text from images.

V8 Engine: V8 Engine is an open-source, high-performance JavaScript engine used in Google Chrome and Node.js. It compiles and executes JavaScript code at lightning speeds, enabling efficient and smooth web browsing and server-side application development.

Approach Overview

A JSON file is retrieved from a server by an ESP32 device that is positioned 100mm away from a designated platform for the Thoska. A second ESP captures an image of the Thoska placed on the mount and performs OCR to extract the matriculation number and other relevant details. The OCR result is compared with the value received over an HTTP server. Access is granted to the user if the matriculation number matches. Language models for OCR are stored on an SD card, and Tesseract JS library, which uses these models, is executed using the browser v8 engine. The most accurate and robust approach is used to retrieve the matriculation number.

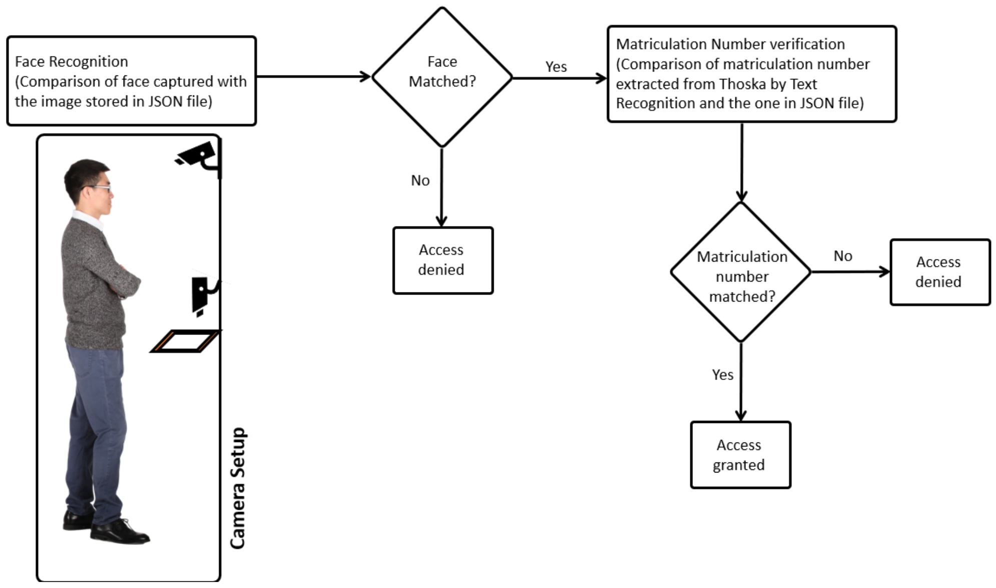

Face Recognition
(Comparison of face captured with the image stored in JSON file)
Face Matched?
Yes
No
Access denied
Matriculation Number verification
(Comparison of matriculation number extracted from Thoska by Text Recognition and the one in JSON file)
Matriculation number matched?
No
Access denied
Yes
Access granted
Camera Setup

Figure 9.15: Process Flow

Technical Overview

Github Reference

Primary Functions :

Loading the Model **:**

This module uses the Tesseract.js library to perform optical character recognition (OCR). The module initializes a Tesseract worker, which is responsible for processing the OCR tasks. The worker is configured with the workerPath, langPath, and corePath parameters, which specify the paths to the necessary files for running Tesseract.

The async function is then called to load the worker, language data, and initialize it. In this case, the language 'deu' is loaded, which represents the German language. The logger parameter is also set, which logs the progress of the OCR process as a progress bar.

Overall, this module initializes the Tesseract worker and loads the necessary language data for OCR processing in German language.

Utilising the Model :

An asynchronous function that uses the OCR technology to extract a matriculation number from an image. The module takes an image file and passes it to the worker for recognition.

Once the recognition process is complete, the text extracted from the image is stored in a variable called "text". A regular expression is then used to find a six-digit number that corresponds to the matriculation number on the Thoska card. The extracted number is stored in a variable called "matNoFromOCR".

In the event that an error occurs during the OCR detection process and no matriculation number is found, a function called "handleOCRdetectionError()" is called to handle the error.

Sending request to Face-recognition ESP to fetch recognition response :

The module makes a GET request to a specified URL.

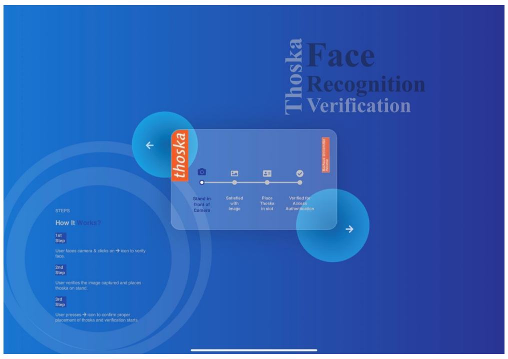
Once the request is successful, the response is returned as a text format. Then, the retrieved data is passed as an argument to the setResponseConstraints() function, which sets the constraints for the response.

In case of an error during the request process, the catch block logs the error message to the console for debugging purposes.

Overall, the module serves as a means to retrieve verification data from a specified URL and set response constraints based on the data returned.

Finally, the request object's send() method is called to send the response to the client making the request. The response includes an HTTP status code of 200, indicating a successful response, and the content type of text/plain. The response body contains the verification payload obtained from the external resource and returned by fetchVerification().

Overall, this code sets up a simple HTTP endpoint that allows the javascript to fetch the face-recognition response stored in it's own server.



User faces camera & clicks on → icon to verify face.

User verifies the image captured and places thoska on stand.

User presses → icon to confirm proper placement of thoska and verification starts

Figure 9.16: Application Interface

Shortcomings

The approach that utilized browser-based optical character recognition (OCR) was found to be the most robust and accurate method for detecting digits printed on thoska cards. However, since the project's objective was to perform all processing on the ESP32 microcontroller, the approach that relied on the browser to compile the OCR operation was ultimately discarded.

9.5 Final Implementation

9.5.1 Abstract

The objective of this project was to train a machine learning model using an ESP32 microcontroller to recognize digits printed on a Thoska card, specifically the matriculation number. A dataset consisting of over 650 images was collected and used to train the model with labels of different digits present on the Thoska card.

To accomplish this, a deep learning framework was utilized to train the object detection model, which involved several stages including data preprocessing, model training, and model evaluation. The trained model was then deployed on an ESP32 microcontroller to enable real-time object detection.

Overall, the project was successful in achieving the desired objective, demonstrating the potential of ESP32 microcontrollers and deep learning frameworks in the field of computer vision.

This project has numerous potential applications, including but not limited to access control, attendance tracking, and identity verification. With further development and optimization, the project could potentially be scaled up for use in larger organizations and institutions.

9.5.2 Tools Used

Software Used :

- Arduino IDE
- EdgeImpulse SDK

Hardware Used :

- ESP32-OV2640
- SD-Card

9.5.3 Approach Architecture

Concept Map

An object detection model based on MobileNetV2 (alpha 0.1) designed to coarsely segment an image into a grid of background vs objects of interest. These models are designed to be <100KB in size and support a grayscale input at any resolution.

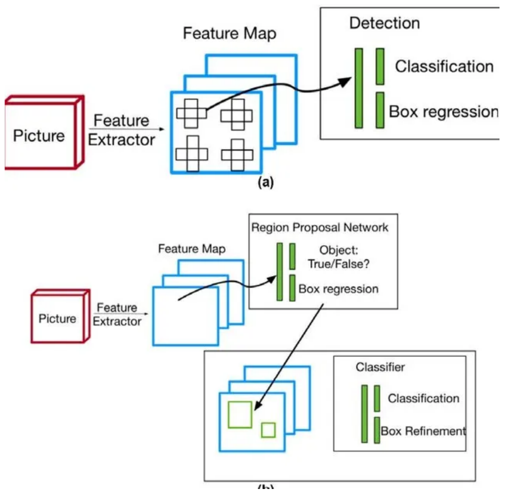


Figure 9.17: Working of mobilenet
Source: https://www.medium.com

Definition

FOMO (Faster Objects, More Objects) is a cutting-edge machine learning algorithm designed to facilitate object detection in highly constrained devices. With FOMO, it becomes possible to count objects, determine their location in an image, and track multiple objects in real-time using a significantly lower amount of processing power and memory compared to existing solutions like MobileNet SSD or YOLOv5. Although the output of the image classifier generated by FOMO is limited to a binary distinction between the presence or absence of a face, it is important to note that the underlying neural network architecture is composed of several convolutional layers. Each of these layers creates a diffused lower-resolution image of the previous layer, with some degree of locality preserved in the outcome. FOMO leverages this architecture by cutting off the last layers of a standard image classification model and replacing them with a per-region class probability map, such as a 4x4 map. A custom loss function is then

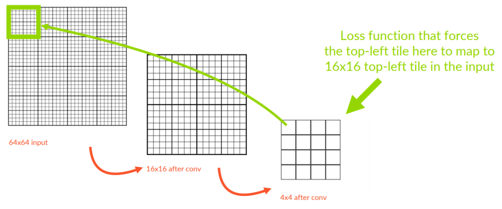

Loss function that forces
the top-left tile here to map to
16x16 top-left tile in the input
64x64 input
16x16 after conv
4x4 after conv

Figure 9.18: FOMO Explanation
Source: https://www.medium.com

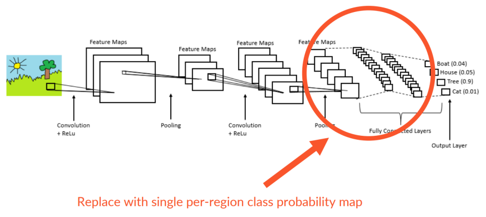
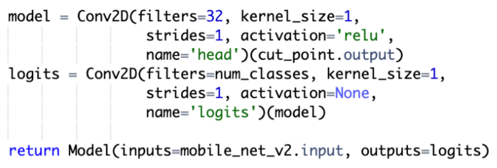


Figure 9.19: Layering of FOMO
Source: https://www.medium.com

used to compel the network to preserve locality in the final layer, generating a heatmap that indicates the presence and location of objects. The resolution of the heatmap is



```
model = Conv2D(filters=32, kernel_size=1,
               strides=1, activation='relu',
               name='head')(cut_point.output)
logits = Conv2D(filters=num_classes, kernel_size=1,
                strides=1, activation=None,
                name='logits')(model)

return Model(inputs=mobile_net_v2.input, outputs=logits)
```

Figure 9.20: FOMO Convolutional Classifier
Source: https://www.medium.com

determined by the layers that are cut off from the network. For instance, the FOMO model trained on beer bottles cuts off the layers when the size of the heatmap is eight times smaller than the input image, resulting in a 20x20 heatmap from a 160x160 input image. However, this can be adjusted to provide pixel-level segmentation for counting small objects. While FOMO does not output bounding boxes, it is easy to derive these from the heatmap by drawing a box around a highlighted area. It is worth noting that FOMO is sensitive to the ratio of objects to background cells in the labelled data. To address this, the configuration includes a weight for object output cells, which is set at 100 by default to balance the majority of background. In situations where objects are relatively rare, this weight can be increased to 1000 to enhance the focus on object detection, albeit at the cost of a potentially higher number of false detections. To implement this technique, FOMO partitions the input image into tiles of a predetermined size, such as 8x8 pixels. Each grid is then independently processed by a classifier, which generates a heat map indicating the approximate locations of objects within the image. The precision of the heat map is increased with a smaller tile size, resulting in a more accurate representation of the object locations. FOMO uses MobileNetV2 as a base model for its trunk and applies a spatial reduction of 1/8th from input to output, cutting MobileNet off at the intermediate layer block_6_expand_relu. In effect, FOMO can be viewed as the first section of MobileNetV2 followed by a standard classifier applied in a fully convolutional manner. The default FOMO classifier is equivalent to a single dense layer with 32 nodes, followed by a classifier with num_classes outputs.

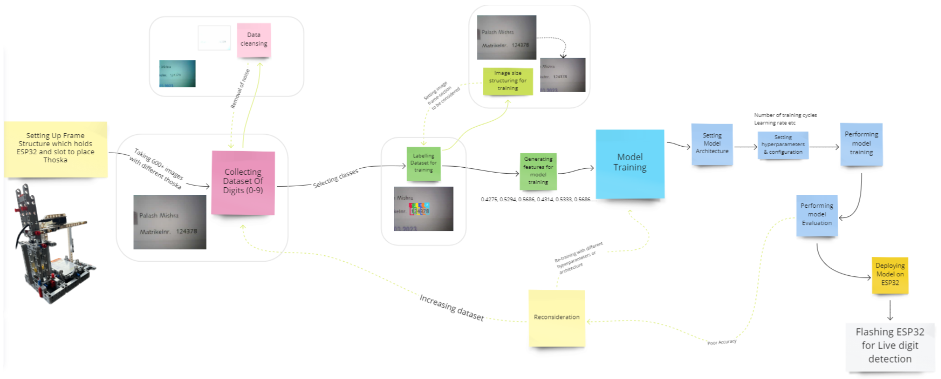

Data cleansing
Removal of noise
Setting Up Frame Structure which holds ESP32 and slot to place Thoska
Taking 600+ images with different thoska
Collecting Dataset Of Digits (0-9)
Palash Mishra
Matrikelnr. 124378
Selecting classes
Setting image frame-section to be considered
Image size structuring for training
Labelling Dataset for training
Generating features for model training
0.4275, 0.5294, 0.5686, 0.4314, 0.5333, 0.5686.....
Model Training
Setting Model Architecture
Number of training cycles
Learning rate etc
Setting hyperparameters & configuration
Performing model training
Performing model Evaluation
Deploying Model on ESP32
Flashing ESP32 for Live digit detection
Re-training with different hyperparameters or architecture
Reconsideration
Increasing dataset
Poor Accuracy

Figure 9.21: Complete procedure of Model

9.5.4 Steps Involved

The implementation can be visualised in the following steps :

- Creating frame-structure to hold ESP32 and a slot to place Thoska for detection
- Collecting data-set
- Labeling data-set
- Feature Extraction
- Model Training
- Model Evaluation
- Deployment on ESP32
- Live rendering

9.5.5 Detailed procedure

Creating Frame to hold ESP32 and Thoska

The aim of this step was to develop a lego frame that could hold an ESP32 microcontroller and be used for digit detection on a Thoska card. The step involved the design and construction of a custom frame that could accommodate both the ESP32 and the Thoska card in a secure and stable manner. The frame features a slot that is specifically

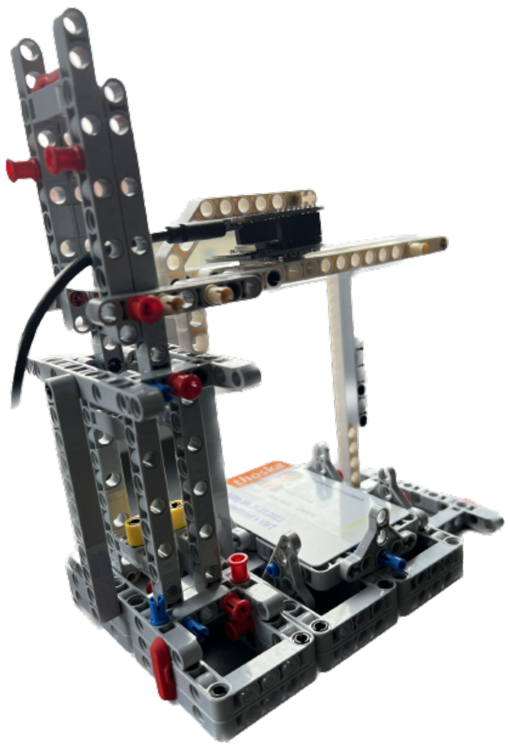


Figure 9.22: Frame structure holding ESP32 and Thoska

designed to hold the Thoska card, which is then captured by an onboard camera connected to the ESP32. Once an image of the Thoska card is captured, the ESP32 utilizes a pre-trained object detection model to detect the digits on the card.

Collecting dataset

In this step, a code was developed to capture images of Thoska cards at regular intervals and save them onto an SD card. The objective of this activity was to collect a dataset of images for digit recognition training purposes. In total, around 600 images were collected, featuring various Thoska cards with different digits ranging from 0 to 9.

The program was designed to automate the process of capturing images, saving valuable time in collecting data manually. The images were captured every 3 seconds to ensure that the dataset was adequately diverse, with a range of lighting conditions, angles, and backgrounds captured.

As part of the data cleansing process, images that were deemed unusable, such as those that were blurry or poorly lit, were removed from the dataset to ensure the highest quality of training data.

9.5.6 Labeling dataset

In this step, a dataset of 600 Thoska card images was manually labelled with six different classes, each corresponding to a digit from 0 to 9. The labelling process involved identifying and marking the position of each digit present in the image to ensure that the model was trained on accurate and reliable data. The labelling process was done

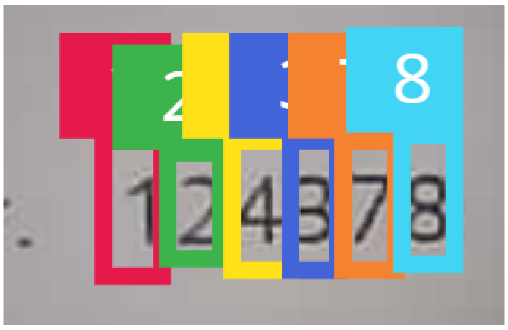


Figure 9.23: Labelling of classes

manually and involved a significant amount of effort and attention to detail. To label each image with the six different classes, a total of 3,600 labels had to be assigned. This means that for each image, it had to be carefully examined and identify the position of each digit present in the image, and assign the correct label.

The labelling process was a critical step in developing the digit recognition model. By providing labeled data, the model was able to learn and recognize the digits present in Thoska card images accurately.

9.5.7 Feature Extraction

After collecting and labeling the dataset of 600 Thoska card images with six different classes, the next step was to extract features from the images. In this step, color(RGB) was used as the primary feature for digit recognition. The result is a set of numerical

```
0.3961, 0.5333, 0.5569, 0.3961, 0.5333, 0.5569, 0.4039, 0.5373, 0.5647, 0.4353, 0.5451, 0.5686,
```

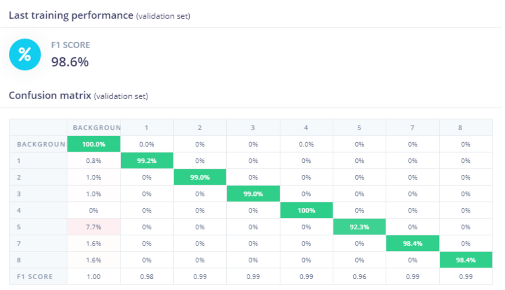
Figure 9.24: Sample Features

values that represent the key characteristics of each image. These numerical values are used as input to the machine learning model.

9.5.8 Model Training

After feature extraction, the dataset was split into training and testing sets, with 80 of the data used for training and the remaining 20 used for testing. The training data was further split into smaller batches for improved training performance.

A neural network model was selected for training, and the hyperparameters were tuned to optimize model performance. The model was trained using the training dataset, and the progress of the training process was monitored. After training, the



Last training performance (validation set)

F1 SCORE
98.6%

Confusion matrix (validation set)

	BACKGROUN	1	2	3	4	5	7	8
BACKGROUN	100.0%	0.0%	0%	0%	0.0%	0%	0%	0%
1	0.8%	99.2%	0%	0%	0%	0%	0%	0%
2	1.0%	0%	99.0%	0%	0%	0%	0%	0%
3	1.0%	0%	0%	99.0%	0%	0%	0%	0%
4	0%	0%	0%	0%	100%	0%	0%	0%
5	7.7%	0%	0%	0%	0%	92.3%	0%	0%
7	1.6%	0%	0%	0%	0%	0%	98.4%	0%
8	1.6%	0%	0%	0%	0%	0%	0%	98.4%
F1 SCORE	1.00	0.98	0.99	0.99	0.99	0.96	0.99	0.99

Figure 9.25: Sample initial training with digits

model was evaluated using the testing dataset. The model's performance was measured in terms of accuracy and loss. The confusion matrix was also analyzed to determine which digits were correctly identified and which were misclassified.

Based on the evaluation results, the model was further refined, and the training process was repeated to improve the model's accuracy. The final model achieved an F1 rate of over 89, demonstrating its ability to recognize the digits present in Thoska card images.

9.5.9 Model Evaluation

After the model was trained, it was important to evaluate its performance to ensure it could accurately recognize digits in new, unseen Thoska card images.

To do this, the model was first tested using the reserved testing dataset which was not used during training. The testing dataset contained images of Thoska cards with digits that the model had not seen before. These images were fed into the model, and the model's prediction was compared to the actual digit present in the image. The

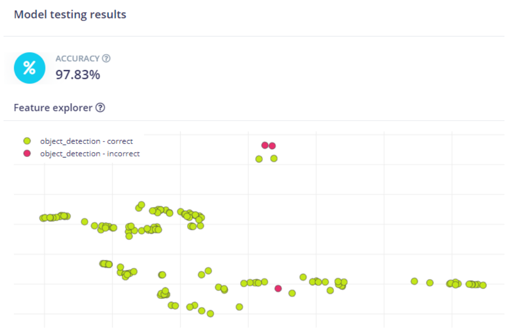


Figure 9.26: Sample initial training with digits

evaluation of the model's performance was based on metrics such as accuracy, precision, recall, and F1 score. The accuracy measures the proportion of correct predictions made by the model, while precision measures the proportion of true positive predictions among all positive predictions made by the model. Recall measures the proportion of true positive predictions among all actual positives in the dataset, and F1 score is a weighted average of precision and recall.

In addition to these metrics, a confusion matrix was also analyzed to determine the model's performance in classifying each digit. The confusion matrix displays the number of true positives, true negatives, false positives, and false negatives for each digit, providing insights into the model's strengths and weaknesses.

Based on the evaluation results, adjustments were made to the model to improve its performance. The evaluation process was repeated until the model achieved a satisfactory level of accuracy and precision in recognizing digits in Thoska card images.

Once the model was evaluated and refined, it was ready to be deployed on the ESP32 microcontroller for real-time digit recognition in the Thoska card images captured by the camera.

9.5.10 Deployment on ESP32

The deployment comprised of the following steps :

- Opening the Arduino IDE and clicking on "Sketch" in the menu bar, then selecting "Include Library", and finally click on "Add .ZIP Library".
- Navigating to the location where the Edge-Impulse library zip file was saved on the computer, and then selecting the file, and clicking "Open".
- The IDE will then import the library and display a confirmation message in the status bar at the bottom of the window.
- Making changes accordingly for the required output and creating a sketch.
- Selecting the appropriate board and port from the "Tools" menu.
- Clicking the "Upload" button to compile and upload the sketch to the ESP32 board.

The IDE will compile the sketch, upload it to the ESP32 board, and display the upload progress in the status bar at the bottom of the window.

After the upload is complete, the sketch will begin running on the ESP32 board.

In summary, to import a zip library in the Arduino IDE, we need to add the library by selecting "Add .ZIP Library" in the "Sketch" menu. Then, to use the library, we make changes as per our requirements and create a sketch. Finally, we can upload the sketch to the ESP32 board by selecting the board and port, and clicking the "Upload" button.

9.6 Integration with Face Recognition

9.6.1 Overview

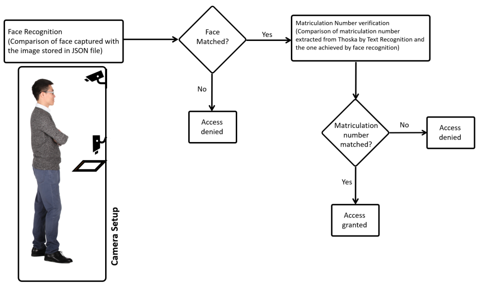

Face Recognition
(Comparison of face captured with the image stored in JSON file)
Face Matched?
Yes
No
Access denied
Matriculation Number verification
(Comparison of matriculation number extracted from Thoska by Text Recognition and the one achieved by face recognition)
Matriculation number matched?
No
Access denied
Yes
Access granted
Camera Setup

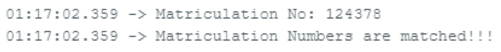
Figure 9.27: Process flow in integration

A scenario can be described in which an individual approaches an ESP32 device designed for facial recognition. Upon the individual's presence, the facial recognition algorithm begins processing. If the individual's facial features are correctly identified, the ESP32 sends a unique matriculation number linked to that individual's face to another ESP32 via a HTTP POST request. The second ESP32 device is intended for scanning thoskas . If the face is not identified, the process ends there itself.

Once the second ESP32 receives the matriculation number, it initiates the thoska detection process. The device scans the thoska and recognizes the matriculation number printed on it. Subsequently, the system compares the number obtained from the facial recognition process with the number derived from the thoska detection process. If the two numbers match, authentication is granted.

9.6.2 Technical Overview

GitHub Reference



```
01:17:02.359 -> Matriculation No: 124378
01:17:02.359 -> Matriculation Numbers are matched!!!
```

Figure 9.28: Results inference from OCR

The program initializes a camera and defines some variables and functions, including capturing an image, handling Matriculation, creating an access point, and performing Thoska detection using the captured image. The access point is created for FaceThoska to connect and send a POST request to the server with a "*matriculation*" parameter. The program then extracts the value of the parameter and sets it to a global variable. After that, the program performs Thoska detection using the captured image and turns on the LED if the detected value matches the global variable. Finally, the program captures an image using the camera and runs the classifier. If the classifier is for object detection, it checks if a bounding box is found.

9.7 System Comparison

9.7.1 Overview

The accuracy of OCR (Optical Character Recognition) depends on several factors such as image quality, font type, font size, and the complexity of the text. The OCR algorithm and implementation also play a critical role in determining the accuracy of the OCR process.

In general, OCR performed on a standalone ESP32 device may have a lower accuracy compared to OCR performed on a more powerful system that utilizes Python, internet connectivity, and machine learning libraries. This is because a standalone ESP32 device typically has limited processing power and memory, which may constrain the OCR algorithm's ability to perform complex operations, such as advanced image processing, machine learning, or deep learning.

On the other hand, when OCR is performed on a system that utilizes Python, internet connectivity, and machine learning libraries, it may be possible to achieve higher OCR accuracy because the system can leverage advanced image processing, machine learning, and deep learning techniques. These techniques can help improve the OCR algorithm's ability to recognize complex characters and symbols, and handle noise and other image artifacts.

9.7.2 System Solution

TesseractJS is an open-source OCR (Optical Character Recognition) engine that is built on Tesseract, which is an OCR engine developed by Google. Tesseract is a machine learning-based OCR engine that uses deep learning techniques to recognize text from images.

TesseractJS was trained using a combination of supervised and unsupervised learning techniques. The training process involved feeding the OCR engine with large amounts of training data consisting of images and their corresponding text. This data is used to train the OCR engine to recognize characters and words in new images.

The training process for *TesseractJS* involved the following steps:

Data acquisition: Large amounts of training data consisting of images and their corresponding text are collected. The training data is sourced from a variety of sources such as books, newspapers, magazines, and other printed materials.

Data preparation: The collected data is preprocessed to remove noise and other image artifacts that may affect the OCR engine's ability to recognize text accurately. This includes techniques such as deskewing, normalization, and binarization.

Training the OCR engine: The preprocessed data is used to train the OCR engine using supervised and unsupervised learning techniques. Supervised learning involves providing the OCR engine with labeled training data, where the images are paired with their corresponding text. The OCR engine learns to recognize characters and words based on the labeled training data. Unsupervised learning involves providing the OCR engine with unlabeled data, where the OCR engine learns to recognize characters and words based on patterns and relationships in the data.

Testing and validation: The trained OCR engine is tested and validated using a separate set of data that was not used for training. The testing and validation data are used to evaluate the OCR engine's accuracy and to identify areas where the OCR engine may need improvement.

Fine-tuning: The OCR engine is fine-tuned based on the results of the testing and validation process. This involves adjusting the OCR engine's parameters and configurations to improve its accuracy.

In summary, TesseractJS was trained using a combination of supervised and unsupervised learning techniques, where large amounts of training data were used to train the OCR engine to recognize characters and words in new images.

9.7.3 Current Solution

The current solution is built with combination of Object-Detection and Classification. Since, specific support for OCR is not possible directly on ESP32 which may involve segmentation, detection and recognition, this solution was considered.

The solution was trained with around 600 images of different thoska which contain digits from 0-9. This way, around 300 features were available for every digit from the dataset with different alignment, brightness and qualities.

The training process for the solution involved the following steps:

Data acquisition: As stated earlier, around 600 images were captured to create a dataset.

Data preparation: The collected data was preprocessed to remove noise and other image artifacts that may affect the solution's ability to recognize text accurately. This was mostly done manually like removing images which are blurred or noised and later image-size structuring was done for 320*320 size which was focussed on the matriculation number printed on thoska to have good range of detection.

Training the OCR engine: The preprocessed data is used to train the splution using supervised learning techniques. Supervised learning involves providing the model with labeled training data, where the images are paired with their corresponding text. The solution learns to recognize characters and words based on the labeled training data. Also, the detection worked similarly.

Testing and validation: The trained solution was tested and validated using a separate set of data that was not used for training. Around 75 images were tested with the F1 rate of 89

Fine-tuning: The OCR engine is fine-tuned based on the results of the testing and validation process. This involves adjusting the solution's parameters and configurations to improve its accuracy.

In summary, the training was way different in terms of size than the system solution.

9.7.4 Differences

Againsts :

- The current solution is *limited* to detecting only the font printed in Thoska, while the system solution can identify a wider range of fonts. Unlike the current solution, the system solution can handle unseen images.
- While the current solution prioritizes detecting the matriculation number, which serves as a unique identifier, *other text* on Thoska is not given equal priority. In contrast, the system solution is capable of detecting all text present on the card.
- The system solution exhibits a 50 accuracy rate when dealing with blurred images, owing to a diverse dataset with images of varying qualities. However, the current solution was trained on good quality images only, making it *unsuitable* for blurred or dark images.

- Attempts to enhance the current solution's ability to read text may prove futile as the ESP32's RAM capacity is *insufficient* for handling large feature sets. On the other hand, the system solution is amenable to different language models.
- The system solution includes a user-friendly interface that simplifies the process for the user. However, with the current solution, the user is notified of successful authentication via three blinks on the ESP32, which may not be an *intuitive* result for the user.

Fors :

- The utilization of models in the system solution is dependent on a compiler engine, whereas the current solution operates *independently* on ESP32.
- The detection process in the system solution employs worker objects for asynchronous operation, taking approximately 2 seconds, whereas the current solution achieves an inference rate of less than *200ms* for detection.
- The system solution requires SD-card storage to load detection models, whereas the current solution is pre-loaded with the necessary modules and does *not require* external storage.
- The utilization of the ESP32 in the system solution is limited to HTTP request handling with face recognition, whereas the current solution effectively leverages the ESP32's *full capacity.*
- In the system solution, an LED light is used to capture images for detection, while the current solution employs a buffer-based image capture method *without* the need for a flash.

9.8 End Discussion

The project successfully utilized machine learning object detection techniques to perform digit detection on printed matriculation numbers on Thoska cards. Despite initial challenges and failed attempts, it was possible to overcome these obstacles and achieve the seemingly impossible task of accurately fetching the digits.

The project was implemented on an ESP32 platform and did not require any external storage, internet connectivity, or high-end camera module. This standalone feature highlights the true potential of ESP32.

The project provided a valuable learning experience, enabling to gain knowledge and practical skills in machine learning concepts while overcoming challenges and failures. Overall, the project was a learning platform and served as a testament to the capabilities of machine learning and the ESP32 platform.