Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

System for inventory monitoring

Petr Ungar

Supervisor: Ing. Michal Janošek, Ph.D.
Field of study: Cybernetics and Robotics
May 2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Ungar** Jméno: **Petr** Osobní číslo: **466329**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Systém pro kontrolu stavu zásob

Název bakalářské práce anglicky:

System for Inventory Monitoring

Pokyny pro vypracování:

Sestavte z bezdrátového modulu kamery LILYGO systém pro vzdálené (cloud-based) monitorování a kontrolu stavu zásob: kamera bude snímat množství produktů v regálových policích, ukládat obraz do vzdáleného systému, kde bude provedena analýza obrazu za pomoci neuronových sítí a výsledný stav ukládán/zobrazován. Uvažte možnost bateriového napájení kamerového modulu.

Seznam doporučené literatury:

[1] Katircioglu, Kaan K., and Ying Li., IBM: Machine vision technology for shelf inventory management,. U.S. Patent Application 14/215,006, filed September 17, 2015.
[2] Cato, Robert Thomas, and Thomas Guthrie Zimmerman, IBM: Using cameras to monitor actual inventory. U.S. Patent No. 8,091,782. 10 Jan. 2012.
[3] Šonka M., Hlaváč V.: Boyle R.: Image processing, analysis, and machine vision. Cengage Learning, Toronto, 2015.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michal Janošek, Ph.D., katedra měření

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **16.09.2019** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce:
do konce letního semestru 2020/2021

Ing. Michal Janošek, Ph.D.	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce		podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Acknowledgements

I would like to express my gratitude to Ing. Michal Janošek, Ph.D. for his guidance during my work on this thesis.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 22, 2020

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 22. května 2020

Abstract

This thesis aims to develop an inventory monitoring system using LilyGO T-Journal camera board, Firebase, and PyTorch framework. A modified T-Journal board is used to capture images and send them to Firebase. These images are processed in a Python program using convolutional neural networks (CNN) and the results are stored back to Firebase. These results are then visualized in a mobile app which is also used to label images for CNN training.

Keywords: Python, PyTorch, convolutional neural networks, embedded systems, Arduino, Firebase, monitoring

Supervisor: Ing. Michal Janošek, Ph.D.

Abstrakt

Cílem této práce je, pomocí vývojové desky LilyGO T-Journal, Firebase a frameworku PyTorch, vytvořit systém pro kontrolu stavu zásob. Upravená deska T-Journal zaznamenává a nahrává fotky do Firebase. Tyto fotky jsou poté zpracovány pomocí Python skriptu a konvolučních neuronových sítí (CNN). Výsledky zpracování jsou nahrávány zpět do Firebase a následně vizualizovány v mobilní aplikaci, která zároveň slouží pro "labelování" obrázků pro trénink CNN.

Klíčová slova: Python, PyTorch, konvoluční neuronové sítě, vestavné systémy, Arduino, Firebase, monitorování

Překlad názvu: Systém pro kontrolu stavu zásob

Contents

Figures

Tables

Chapter 1
Introduction

An inventory monitoring system is any system tracking the current state or level of stock. It is widely used for any planning or reporting activities. In the past, all the inventory monitoring was usually done in person. A responsible employee would manually count all the products and insert the numbers into some reporting software. This process can be very time consuming and prone to errors.

Naturally, many tools have tried to fully or at least partially automate this process for multiple reasons. Some of them are:

- to save costs on human resources,
- to decrease the amount of flawed data,
- to prevent unnecessary manufacturing shutdowns due to missing components,
- to prevent theft.

1.1 Outline

This thesis aims to develop affordable, compact, and easy to deploy solution for remote stock monitoring. It is divided into two parts. The main part is the software. This part is further divided into the main program, the program for hardware (firmware), the machine learning algorithm, and the mobile app used for labeling data for training and reviewing the outputs. The second part is the hardware part where the hardware used in this project is described together with all the alterations made.

1.1.1 Comparison to the state-of-the-art

This section provides a summary of currently used technologies, it points out different use cases and advantages or disadvantages to the given solution.

- Conventional solutions
  This subsection specifies the technologies that do not use machine learning algorithms.

- Smart solutions
  This subsection summarizes the technologies that use machine learning.

1.1.2 Hardware

First, the LilyGO T-Journal is introduced and the technical specifications are provided. Second, the changes to the hardware as part of consumption optimization are described.

1.1.3 Software

This section contains an explanation of different applications written for this project, it also tries to visualize the connections between these applications and the high-level workflow of the whole functionality.

- Architecture
  This chapter contains all the high-level work diagrams and graphs. It should improve the understanding of the relations between the different programs.
- Camera board program
  This section contains information about the firmware for the hardware device used to capture images. It also explains how to correctly set up such a device.
- Main program
  This section summarizes the functionality of the main program which is the backbone of the whole project.
- Mobile application
  The android mobile application developed for this project is described.
- Image processing
  This section is about the program that interacts with the machine learning modules and datasets, however, the machine learning itself is described in section Machine Learning.
- Firebase [6]
  Even though Firebase is not software developed during this project, it is essential to the functionality and therefore a quick overview is provided for better understanding.

1.1.4 Machine Learning

The technology and the framework used in this thesis are described. Further the process of finetuning and the final architecture are discussed.

Chapter 2

Objectives

This thesis aims to develop a simple, quickly deployable, and cheap solution for remote stock monitoring. The aim is to develop both, the software and the hardware, specifically the camera device, which should be able to, once installed, capture and transmit an image to remote storage. This device should ideally be powered from battery and should be able to function for several months (strongly dependent on the frequency of capturing and battery capacity). The software part should implement three functionalities, namely semi-attended stock level evaluation, data visualization, and a program connecting these two functions (back-end). Semi-attended meaning that the results should still be reviewed by a human. Generally, the machine learning component should be used as an automated out-of-stock notification rather than a bulletproof solution, as this thesis does not strive to design a state-of-the-art machine learning algorithm, but rather a handy combination of several useful technologies.

Chapter 3

Comparison to the state-of-the-art

This section describes some of the related projects. Since inventory monitoring is a very broad topic, only a handful of projects were selected, even though some of them may not be considered state-of-the-art, they were included to demonstrate different approaches/technologies and different applications.

Generally, we can divide the demonstrated projects into two groups. The first group uses sensors to measure some physical quantity such as weight, length, pressure, etc., this group will further be referred to as the conventional solutions. The second group makes use of computer vision and machine learning, therefore we will refer to it as smart solutions.

3.1 Conventional Solutions

In general, these solutions work well for smaller-scale projects. Typically, they require installation of some specialized storing container and each product has a dedicated sensor, which is monitoring just the particular product.

Figure 3.1: Example of a conventional solution - MasterShelf [13]

- MasterShelf [13]
  A specialized shelf is installed in a cooler. The system estimates the amount of drinks in a cooler. The evaluation is based on the linear displacement of sensors placed on springs pushing against the bottles/cans.

- StockVue [23]
  Each product is stored in a box that is placed on a scale. The stock amount is derived from the weight of the stored product. The manufacturer claims that you'll get 1 piece per 1000 piece accuracy. The downside is that you have to install special containers and scale for each product you want to monitor.

3.2 Smart Solutions

Unlike conventional solutions, smart solutions usually do not require installation of specialized hardware other than cameras, which are often already installed and used for other purposes such as security.

Figure 3.2: Example of a smart solution - Panasonic out-of-stock detection [17]

- Panasonic out-of-stock detection [17]
  A series of IP cameras sends data to the remote evaluation system. Just like in this thesis only binary monitoring is supported (in stock/out of stock).
- Multiple camera system for inventory tracking [27]
  The core of this system is a self-driven processing unit/robot equipped with multiple cameras. The robot maneuvres between the storage shelves and creates a "realogram" - a virtual map of the current stock. The unit also processes all the inputs and transmit the data to remote storage.
- Machine vision technology for shelf inventory management [9]
  Systematically placed cameras monitor different products and using edge detection the stock amount is determined. This implementation targets consumer stores such as supermarkets or drug stores.

Chapter 4

Proposed solution

The solution introduced in this thesis needs to be designed for simple installation and use.

To satisfy the need for easy installation, we avoid using complex hardware devices like most of the conventional solutions. However, a little trade-off is made, which is the installation of one camera per product. It is to simplify the evaluation process, by eliminating the need to distinguish different products.

The ease of use is achieved by focusing all the user interaction into one location, the mobile app. A user uses this app to manually evaluate the state of a product and to review the current states of different products. User evaluated data is eventually used to train a convolutional neural network (CNN) classifier, which should partly reduce or potentially fully replace the need for user evaluation. As stated above, the solution is designed to run one camera per product, which also means one CNN classifier per product.

The actual image processing and CNN training should run on a designated server. To connect the server to all the other components Firebase is used as it is relatively easy to access its APIs from servers/PCs, mobiles, and embedded devices.

The solution is visualized in figure 4.1

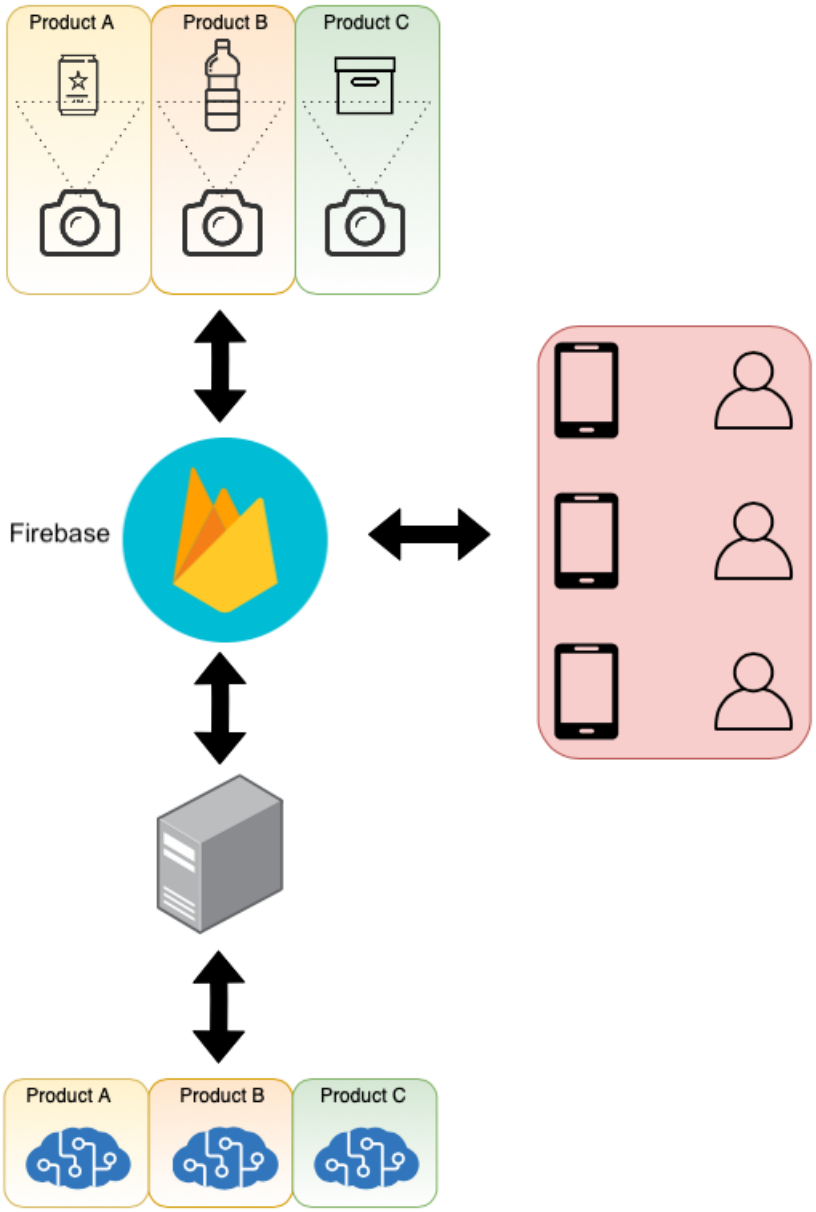


Figure 4.1: Proposed solution

Chapter 5
Hardware

This section provides details of the hardware used for this project, it breaks down the main components and also describes the changes that were made to the original board.

5.1 Overview

The hardware consists of a LilyGO T-Journal board. It uses an ESP32 microcontroller with an integrated Wi-Fi chip and a camera sensor. After initial testing, a couple of changes had to be made to optimize the power consumption.

5.2 Requirements

The main concern for the board is its power consumption, ideally, the board should be able to run off a standard 3.7 V Li-ion battery for months (depending on the capacity and frequency of image capturing). Another requirement is Wi-Fi with a medium-long range (10 m – 100 m), this is somewhat conflicting with the previous requirement because Wi-Fi can have considerably high power consumption. However, since the device will spend the majority of time in a deep sleep mode, we care more about idle consumption rather than the working consumption. Lastly, the board needs to be able to capture images and convert them to any standard image format.

5.3 LilyGo T-Journal

Upon some investigation, the LilyGo T-Journal from a Chinese manufacturer Lily-GO appeared to be the best option. It is equipped with ESP32, OV2640 with `.jpg` compression, integrated Wi-Fi with extended range, Bluetooth Low Energy, and a small OLED display. It supports micro-USB charging and programming and comes with battery outlets straight out of the factory.

Technical specifications	
Chipset	ESPRESSIF-ESP32-PCIO-D4 240MHz Xtensa® single-/dual-core 32-bit LX6 microprocessor
FLASH	QSPI flash/SRAM, up to 4 x 16 MB
SRAM	520 KB SRAM
Display	0.91 SSD1306
USB to TTL	CP2104
Camera	OV2640 - 2Megapixel
On-board clock	40MHz crystal oscillator
Working voltage	2.3V-3.6V
Working current	about 160mA
Size	64.57mm*23.98mm
Power supply	USB 5V/1A
Wi-Fi Description	
Standard	FCC/CE/TELEC/KCC/SRRC/NCC
Frequency range	2.4GHz 2.5GHz(2400M 2483.5M)
Transmit power	22dBm
Communication distance	up to 300m

Table 5.1: LilyGo T-Journal technical specifications [11]

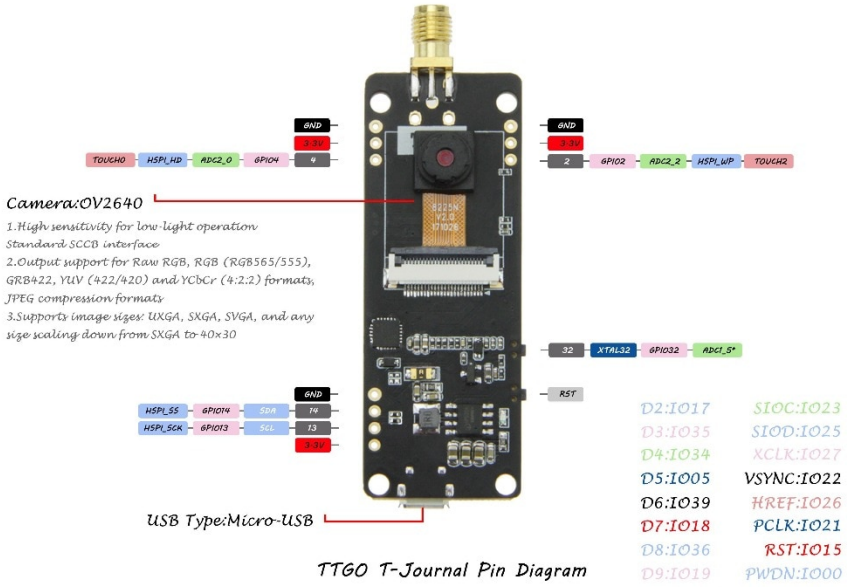


Figure 5.1: LilyGo T-Journal schema [11]

5.4 ESP32

"ESP32 is a series of low-cost, low-power system on a chip microcontroller with integrated Wi-Fi and dual-mode Bluetooth."[26] It is programmable using standard Arduino development frameworks which makes developing the code much simpler. Also according to the manufacturer, the chip in deep sleep only drains less than 100 µA.

5.5 OV2640

It is the world's smallest 2MP camera chip. It has high sensitivity for-low light operations which is suitable for installation in darker places like warehouses. The standby consumption is approx. 600 µA and it supports image resolution of up to UXGA (1600x1200). We opted for Fish-eye lenses, however, it also comes with standard lenses.

5.6 Consumption optimization

After a couple of test runs and initial measurements, it is obvious that the power consumption is too high to meet the criteria and run off battery for months. The board, as it comes from the factory, is not optimized for low power consumption as most of the peripheries are always on and can not be powered down programmatically.

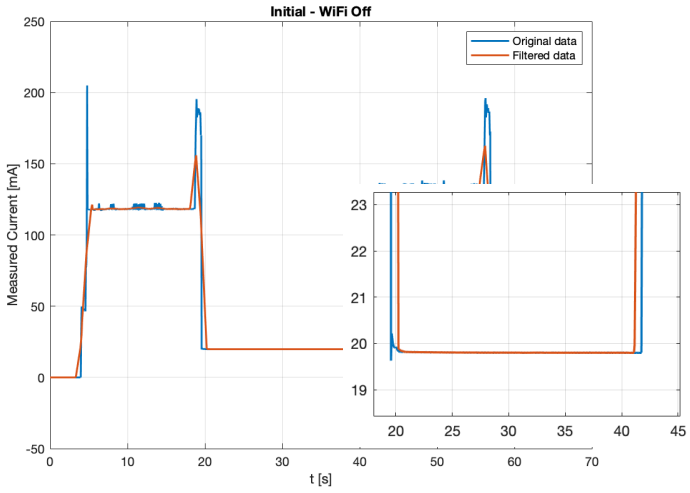


Figure 5.2: Initial consumption without Wi-Fi

Figure 5.2 and Figure 5.3 depict the initial power consumption. The graphs show that the power consumption is approx. 20 mA in the idle state and the mean consumption in the active state is in the range 120 mA − 180 mA.

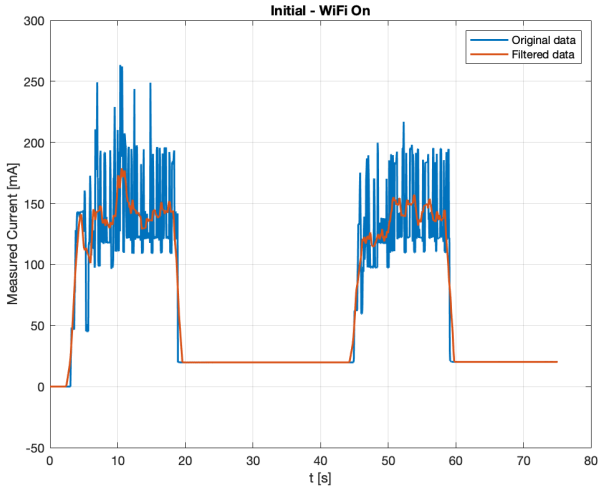


Figure 5.3: Initial consumption with Wi-Fi

There is also a difference between a board that can connect to Wi-Fi and a board that fails to do so. When a board is connected to Wi-Fi the power draw is generally higher and pulsates.

The first step of the optimization was to identify the components that can either be completely removed or at least powered down when not in use.

The components that can be completely removed are:

- The red LED turns out to be connected directly to the source and therefore is on, whenever the board is powered.
- The OLED display is not required for this project.

The second step was to identify the main power consumers in the idle state. These are:

- One of the candidates was the UART circuit. We were unsure whether the circuit enters suspend mode in which the supply current is $100\,\mu A - 200\,\mu A$ or instead the UART stays active and draws $17\,mA - 20\,mA$. It turned out that the circuit actually enters the suspend mode and therefore is not an issue.
- The biggest consumer turned out to be the camera chip. The reason being the `PWDN` pin that has a pull-down resistor (Figure 5.4), this means that the camera is always on and cannot be turned off. We need it the other way around, the camera should always be off, except when in use. The solution is to replace the pull-down with a pull-up and make a connection to a `GPIO` pin, then whenever the camera is initialized, the `GPIO` is set to low, which turns the camera on and when the board enters deep sleep, the camera is powered down again.

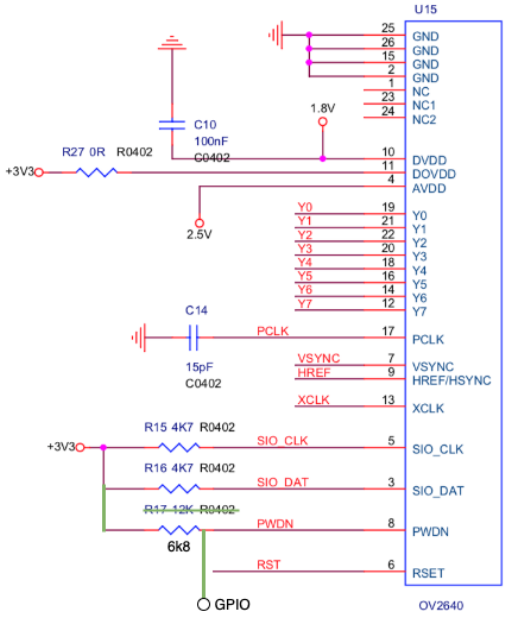


Figure 5.4: Changes to PWDN pin [11]

- We also managed to save some µA by disconnecting the battery connection built into the board. Initially, we wanted to use the built-in battery connector, but this source of powering turned out to be very ineffective and therefore will be replaced with a custom battery control device.

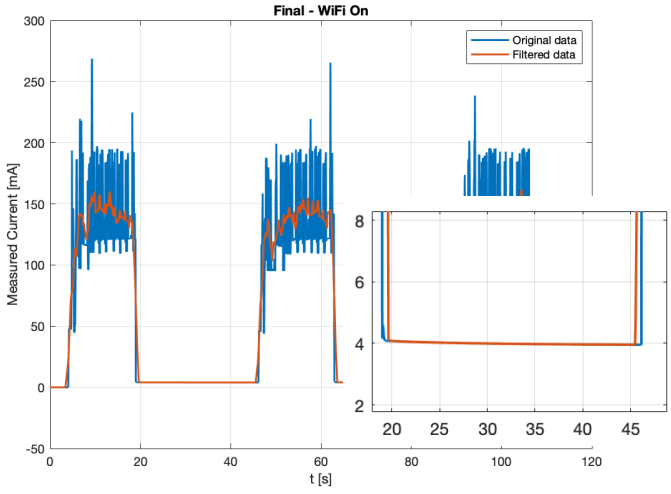


Figure 5.5: Final consumption with Wi-Fi

These changes caused the idle current to drop to around 4 mA extending the expected battery life to up to 5 times. The resulting power consumption can be seen in Figure 5.5. The expected battery lifespan rises from less than a month to around 3 months for 10 Ah battery. The expected lifespan is derived from 5.1.

$$\begin{aligned} days &= \frac{capacity_{battery}}{consumption_{day}} \\ &= \frac{capacity_{battery}}{I_{working} \cdot t_{working} + I_{idle} \cdot t_{idle}} \end{aligned} \tag{5.1}$$

Power consumption comparison		
	Initial	Final
Period	3600s	
Duty cycle	0.42% (15s)	
Working current (approx.)	150mA	150mA
Idle current (approx.)	20mA	4mA
Average current	20.546mA	4.6132mA
Expected battery span - 10Ah	20 days	90 days

Table 5.2: Initial and Final consumption 1 image/hr

5.7 Battery

To demonstrate the ability to run off a battery, the battery circuit which is shown in 5.6 was built. It allows us to power the board using `D-sub 9` connector and to charge the battery from `micro-USB`.

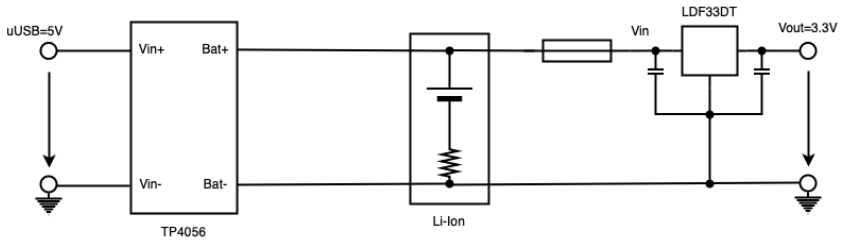


Figure 5.6: Battery connection schema

Chapter 6

Software

This part describes the structure and basic functioning of the different software parts developed for this project. The approach and architecture chosen for the project would not be sustainable for large scale deployments, however, since this project is generally intended to run on-premise and not as SaaS (Software as a Service) many of the challenges of large scale distributed systems could be omitted.

6.1 Overview

The software part can be divided into 2 sections - Architecture and Code, where the Code section can further be divided into 5 subsections, one being the code for the camera board, which handles capturing, compression, and uploading of an image to Firebase. Second, is the image processing module that takes care of training CNN models and later evaluates the images using these models. The third component is the main program that implements general logic, normalizes, and generally handles data in Firebase, it also interacts with the CNN models and updates the information about products monitored. The next part is the mobile app which is used for two purposes - labeling training images and reviewing and managing the evaluated images. Lastly, an overview of Firebase is provided as it is the core of data handling and storing for this application.

6.2 Architecture

As briefly mentioned in this section's summary, the architecture was designed for on-premise, small-medium size deployments. More concretely, the maximum expected number of devices is medium-high tens of monitored products (recording devices) and low tens of responsible users. A couple of entities are mentioned throughout this project, these entities are:

- User
  A user in our case is an instance of Firebase Auth User and is also the abstraction of an end-user/consumer of this service. A user is the one responsible for providing all the manual inputs that might be required

during the runtime - such as labeling training data, confirming/declining generated outputs. A user is also the one using the outputs of this service such as images captured and the status predicted.

- Camera (Device)
  Camera or Device is the representation of a physical device capturing images. It is tightly connected with the actual product it is monitoring as this service runs as one camera per product. Due to this fact, there is no separate "product" class defined within this project and all the information regarding a product should be stored under related camera documents. The relation between a camera and a user is many-to-many, a user can be responsible for multiple devices, but also a device can have multiple responsible users.

- Record
  A record is a simple class used to hold data related to a captured image and to upload the data to Firebase. One instance corresponds to one image. The relation between a record and a camera is many-to-one, a record can only have one source device however any device produces many records.

The following diagram describes the classes defined in this project and the relations between them.

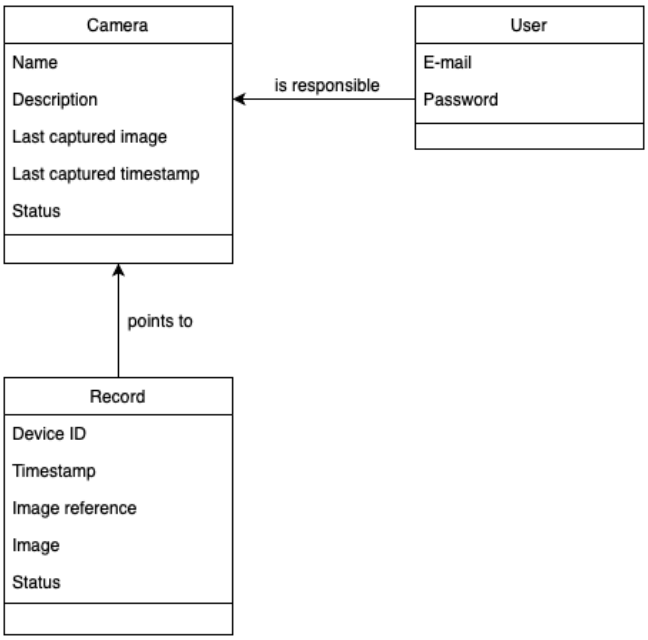


Figure 6.1: Classes and their relations

This is a distributed system and all the data transport is done via Firebase, therefore none of the applications are directly communicating with each other. Figure 6.2 visualizes the virtual communication channels and the data that is being passed through them.

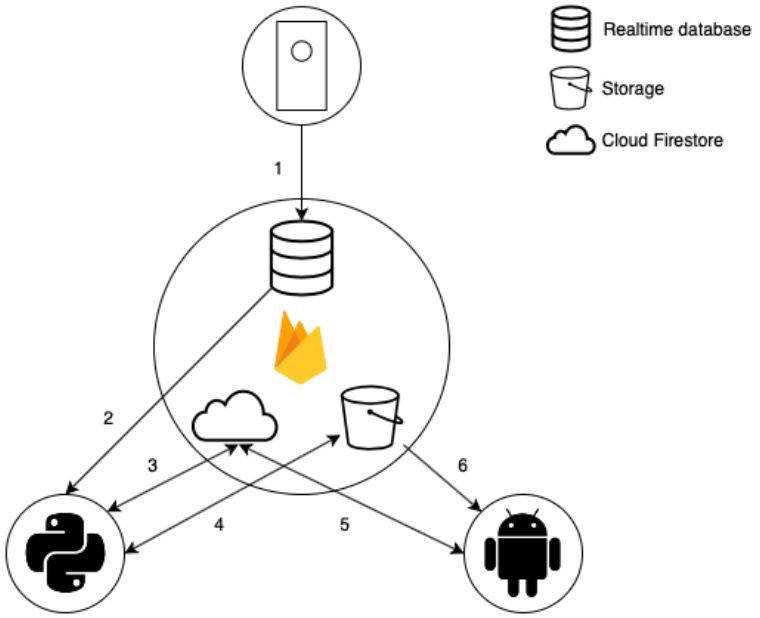


Figure 6.2: Communication between Firebase and different applications

1. In - Camera device uploads images encoded in `Base64`, together with a timestamp and device id. This information is stored as plain text in the Realtime Database.

2. Out - Backend retrieves the items from Realtime Database and represents them as Records (Figure 6.1)

3. In - Records and updated Cameras are uploaded to Cloud Firestore.
   Out - Latest labeled Records are retrieved to update Cameras.

4. In - Images converted from `Base64` to `.jpg` files and uploaded to Storage.
   Out - These images are retrieved for later model training.

5. In - Updated Records and updated Cameras are uploaded to Cloud Firestore.
   Out - Records are downloaded for labeling, Cameras are downloaded for review.

6. Out - Images are downloaded for labeling.

6.3 Firebase

Firebase [6] is a state-of-the-art cloud-based platform that enables us to build powerful and easily scalable infrastructure for app, web, and other software development projects. It offers various services, however, this project only uses those related to data storage.

One of the most powerful features of Firebase is the APIs accessible virtually from any platform.

This project uses the following three services:

- Realtime Database
  `NoSQL` cloud-based database which stores data as `JSON` objects. It is used to store images encoded in `Base64` captured by the camera boards.
- Cloud Firestore
  Like Realtime Database this is a `NoSQL` database, unlike the Realtime Database, the data is stored in objects called Documents that can be grouped into collections. Generally, Firestore allows more advanced operations and is used to store Records.
- Storage
  Storage allows us to store file objects and therefore it is where all the actual `.jpg` images are stored. All the Records refer to the respective files in the Storage.

6.4 Camera board program

The purpose of the camera board is to, in given intervals, capture an image of the monitored product and upload it to the Firebase together with a timestamp and device id. The camera board (described in detail in Hardware section) is equipped with ESP32 microcontroller which can be easily programmed with Arduino-IDE or any similar framework for embedded development. The program itself is based on [12] and [7] and uses multiple 3rd party libraries, mostly to interact with the camera chip, Firebase Realtime Database, and to connect to Wi-Fi.

6.4.1 Setup

Several variables defined at the top of the source code need to be set up properly to be able to run the code successfully. These values are:

- DEVICE_ID - unique id of the camera used to identify the product monitored
- WIFI_SSID - SSID of the Wi-Fi the device is expected to connect to
- WIFI_PASSWORD - password of the Wi-Fi the device is expected to use
- PACKET_SIZE_LIMIT - the maximum size of one packet sent (in bytes, default 4000)[1]
- FIREBASE_HOST - URL of the Firebase project
- FIREBASE_AUTH - Firebase API token
- BASE_PATH - Firebase root document name
- TARGET_RESOLUTION - intended image resolution FRAMESIZE_, where can be replaced with any of the following VGA, CIF, QVGA, HQVGA, QQVGA, UXGA, SXGA, XGA, SVGA
- MINS_TO_SLEEP - how many minutes to wait between two captures[2]

6.4.2 Workflow

As any standard Arduino project, the code is composed of two functions, `void setup()` (called when the device is woken up) and `void loop()` (runs the main body of the code).

[1] there seems to be some limitation to the maximum size of a JSON object in one of the libraries used to upload the image to Firebase and it causes issues when uploading higher quality images

[2] $\Delta = t_{startup_2} - t_{startup_1}$

Normally, we would use the setup to initialize all the peripheries - initialize camera, connect to Wi-Fi, open serial port, and then run the code in the `void loop()` function. However, since one of the intentions is to be able to run the device on batteries, the consumption needs to be minimized and the board is put into deep sleep mode whenever not in use. When the device is brought back to life it reruns the setup and since we only need to capture one image at a time we moved all the logic into the `void setup()` and `void loop()` is never used. The program first initializes all the peripheries, namely connects to the serial port, connects to Wi-fi, and initializes the camera. Second, an image is captured, the image is first compressed into `.jpg` and consequently encoded into `Base64` and URL escaped. Encoded data are then sent to the Firebase Realtime Database where a new node is appended to the BASE_PATH document. Each node has the structure from 6.1

Listing 6.1: Realtime Database node

```
node_id
   |--data
   |   |--image_data_0
   |   |--image_data_1
   |   |      ...
   |   |--image_data_N
   |--device_id
   |--timestamp
   |--image_meta
```

Once the image is transferred, the board enters deep sleep mode and is brought back to life after MINS_TO_SLEEP minutes elapsed since the last awakening.

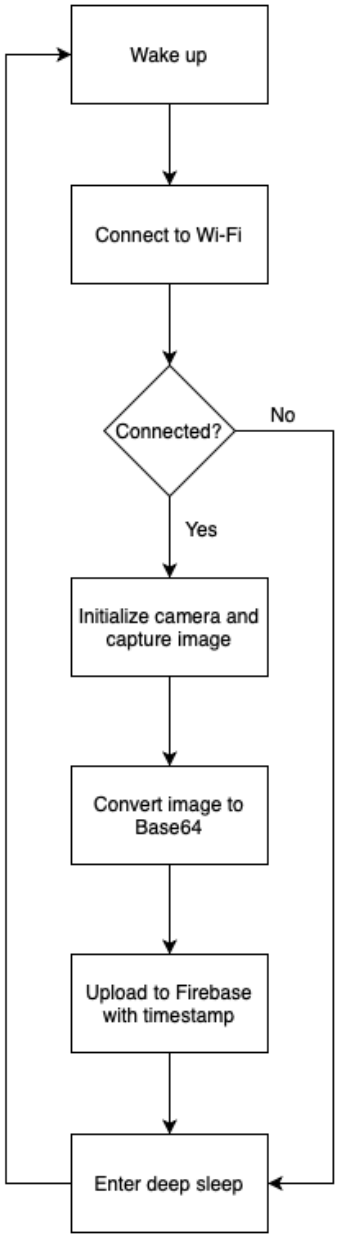


Figure 6.3: Camera board workflow diagram

6.5 Main Program

The main program is a python application that implements all the logic of the back-end and should run on a dedicated hardware (server station, PC). It uses both common ways of parallelism, multithreading, and multiprocessing. Multithreading was introduced to effectively communicate with Firebase, one thread manages all the incoming traffic whereas the second thread uploads the updated data back to Firebase. Multiprocessing was added to improve the performance of image processing and training tasks as working with CNN can be quite computationally expensive.

The main program repetitively checks for new images uploaded into the Realtime Database from any camera boards. It then downloads those `Base64` encoded images and converts them back into original `.jpg` image files. Once the images are converted, the main program uploads them to Firebase Storage and creates new records referring to these images. A record is an instance of a Record class.

As declared, these records are used to store additional information such as time captured, source device, and the status of the monitored product, meaning whether the product is in stock or out of stock. There are two ways to obtain status:

1. If a model of CNN is already trained for the product. The main program creates an evaluation task in the incoming queue and the status is evaluated by a pre-trained CNN model.

2. If a model of CNN is not trained yet, the record is updated with the "Needs labeling" flag and the record is loaded straight into the outcoming queue. Later the image is shown in the mobile app to be labeled by a responsible user. Once a sufficient amount of training images have been labeled the main program creates a training task in the incoming queue and the image processing client trains a new model. Next time a record is created for the same product the main program will perform 1).

6.5.1 Setup

All settings are located in settings.py file distributed with the main module. The following values need to be properly configured:

- CONFIG {
  apiKey - Firebase API key
  authDomain - Firebase project URL
  databaseURL - Realtime Database URL
  storageBucket - Firebase Storage bucket
  }
- FIREBASE_PASSWORD - Firebase user password
- FIREBASE_EMAIL - Firebase user email

- GOOGLE_APPLICATION_CREDENTIALS - location of google cloud credentials
- DEFAULT_IMAGE_PATH - Firebase root document name
- DEFAULT_IMAGE_EXTENSION - default image extension
- DEFAULT_COLLECTION - default collection in Firebase Storage
- SAFETY_TIMECAP_MINS - do not download records from Realtime Database more recent than this time
- LOG_DESTINATION - root folder for logging
- MAIN_LOG_FILE - location of log file
- SLEEP_SECONDS - sleep for n seconds if queue empty (threads and workers)
- ACTIVE_DATA_DIRECTORY - root directory for datasets and models
- ACTIVE_DATASETS_DIRECTORY - directory for datasets
- ACTIVE_MODELS_DIRECTORY - directory for models
- MODEL_NAME - default model naming
- DATASET_NAME - default dataset naming
- ITERS - number of training epochs
- BATCHSIZE - training batchsize
- LEARNING_RATE - CNN model learning rate (SGD)
- MOMENTUM - CNN model momentum (SGD)

6.5.2 Workflow

The main program uses a class called FirebaseClient to interact with Firebase. It is located in `firebase_client.py`

On startup, each thread creates an instance of this class. While being created the FirebaseClient sets up a connection with all three services this project uses - Realtime Database, Firestore, and Firebase Storage, it also authenticates against the server (authentication then needs to be periodically refreshed).

As mentioned, the main program functions as a back-end and implements all the logic of the project. It processes and keeps track of incoming images from cameras, it evaluates these images and then generates outputs based on the results. The task of the main program is both I/O bound (communication with Firebase) and CPU bound (image processing), therefore the program uses both, multithreading and multiprocessing. On startup, two threads and

a pool of workers (additional processes) are created. While each thread has a unique purpose, all the workers are equal and therefore a user can pick an arbitrary number of workers. The only limitation is the number of logical cores - one worker per core.

The program uses two queues, namely `in_Q` and `out_Q`. Their purpose is to share data between processes/workers and the threads. These queues are populated with Tasks.

6.5.3 Task

Tasks are items fed to the queues. A Task is a simple class used to distinguish between an evaluation job and a training job. It contains these members: type of task - evaluate/train, timestamp_in, and timestamp_out and list of Records used for the task or a device id. This class was introduced for two reasons:

- To avoid queue members polymorphism. This way all the members are of the same type, therefore they have the same methods implemented.
- Audit reasons. Thanks to the timestamp in/out properties we can trace back the tasks in logs and easily establish the processing time.

6.5.4 IN Thread

The main job of this thread is to handle incoming Records and feed them as Tasks into `in_Q`. It also adds the Tasks straight to `out_Q` in case no CNN model exists for the device.

Loop

1. All new images from all the boards are downloaded and converted from `Base64` into a temporary `.jpg` file.
2. These images alongside device ids and timestamps are passed into Record constructor and a new record is created for each retrieved image.
3. Images are uploaded to Firebase Storage.
4. The main program now loops thru all the images/records and performs the following steps:
   a. Checks whether the device has an existing model.
   b. Based on the result from a)
      - True —> Task is created and is passed to `in_Q`
      - False —> Record is updated with manual labeling flag and the updated Task is added to `out_Q`. Subsequently, the program checks whether a new model can be trained based on a rule of thumb (amount of labeled images) and accordingly makes a new training Task and adds it to `in_Q`.

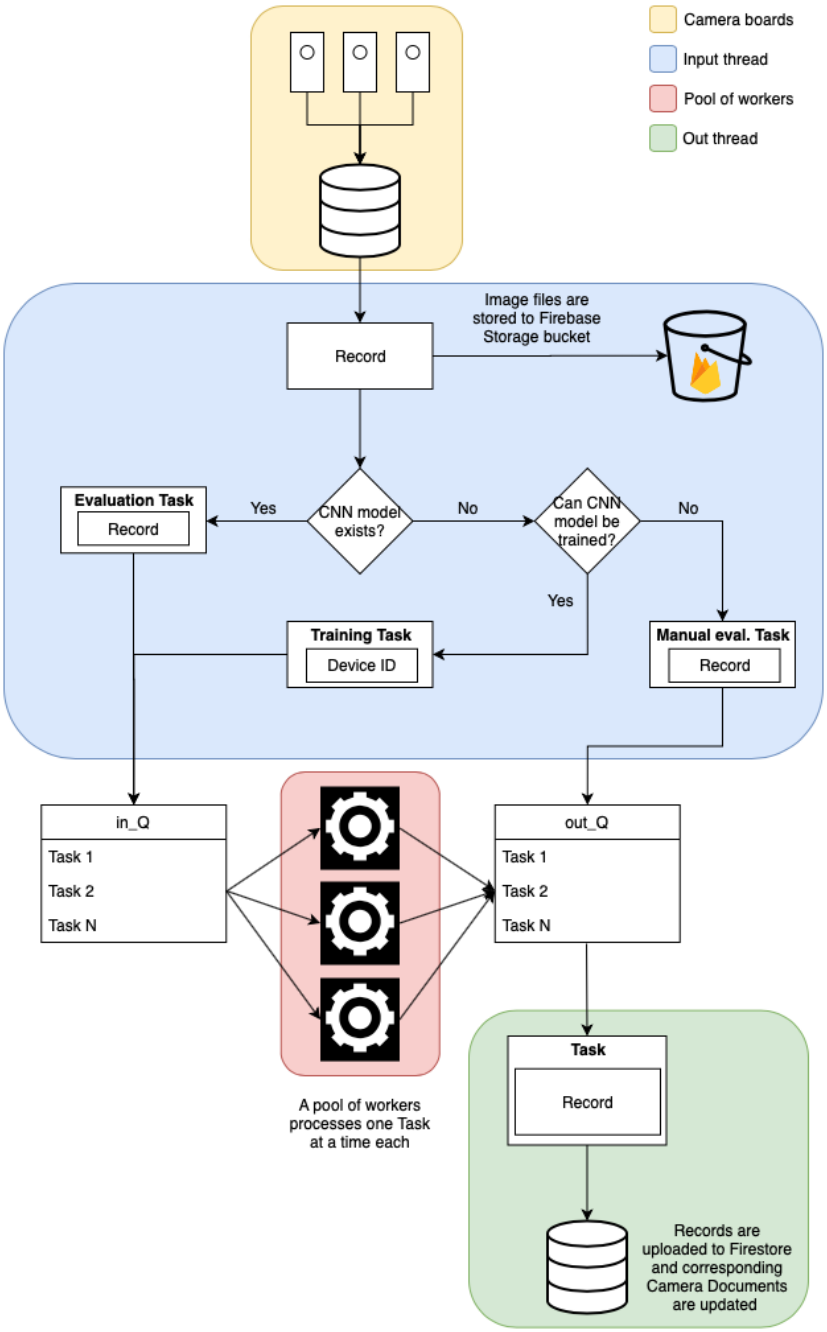


Figure 6.4: Main application workflow

6.5.5 Worker

A worker is a separate process that runs an infinite loop that performs the following steps:

1. Get task from `in_Q`, if no item is found in the queue, the worker sleeps for a while and tries again.
2. Check the nature of the task
   - Evaluation - label the image (set state)
   - Training - train a new CNN model
3. If the task is to evaluate an image the resulting task with updated status is loaded into `out_Q`. The training task has no output as the result is redundant, simply because the result will be retrieved next time the existence of the given model is checked.

6.5.6 OUT Thread

This thread handles all the outgoing communication. Whenever a new task is added to the `out_Q`, it uploads the related Record to the Firebase and updates related Camera Document.

Loop

1. Tasks are retrieved from `out_Q` and the related Records are uploaded to the Firestore.
2. The camera documents are updated. Each camera has a unique document in the Firestore, this document is designed to keep the latest information about the device. Since the amount of Records generated could potentially be very high, the idea is to refrain from accessing raw Records data. Therefore we aggregate all the useful information into these Camera documents and update them as the records come and go.

6.6 Image Processing

This section covers three modules essential for image processing. These modules are:

- `dataset.py`
- `networks.py`
- `image_processing.py`

The `dataset.py` module contains a class which represents images dataset. It also implements all the functionality regarding collecting the dataset from Firebase and image preprocessing. The only preprocessing done in this project is cropping, as the CNN expects input of a given size (480x480), and transformation from `RGB` to normalized greyscale (0.0-1.0).

The `networks.py` contains the definition of CNN classifier and also the method `check_model_exists()` which is essential for the main program - based on the output from this method a new image is either passed on for manual labeling or evaluated using CNN model.

The `image_processing.py` uses both of the previous modules and implements all the functions needed for the image evaluation and training of CNN models. The framework and the techniques used to implement CNN are described in detail in section Machine Learning.

The module contains 2 public functions:

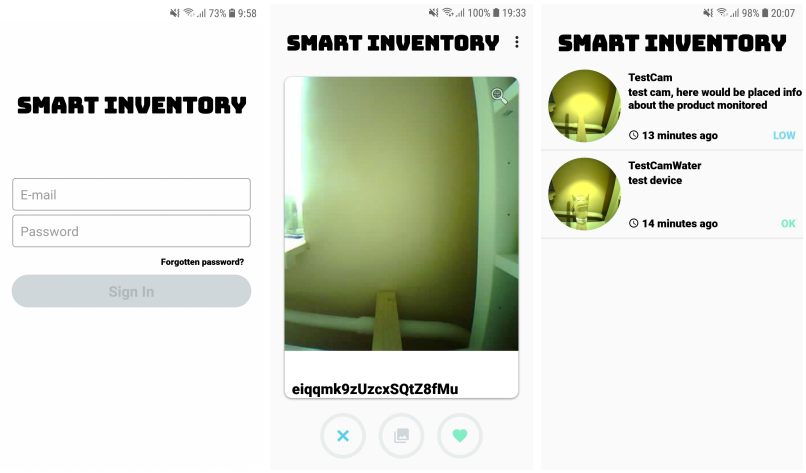
- `process_image()`
  This function expects a Record as an input. First, it transforms the captured image into the appropriate format - crop and greyscale conversion. It then loads the related CNN model and runs the image through the model. Once the result is retrieved it is used to update the Record (Record.status) which is then returned.
  Currently, 2 results can occur:
  - 0 = out of stock
  - 100 = in stock
- `train_CNN()`
  Upon being called, the function first downloads all the previously labeled images by calling the method `gather_test_train_datasets()`. These images are then transformed into the desirable format (crop and greyscale transformation), divided into two locally saved datasets
  - `test_dataset` - generally smaller
  - `train_dataset` - generally larger

  The method then loads these two newly generated datasets and initializes a new instance of CNN model. It then performs training using these datasets. Once the training is finished the trained model is saved to a predefined location where it can be retrieved during future evaluation requests.

6.7 Mobile Application

The mobile application is not required of this thesis, however, it proved to be to the optimal solution for labeling and reviewing the images. However, since it is partly closed-source, it is discussed only briefly to demonstrate its use in this project.

It is a "Tinder" like app running on Android devices. It consists of 3 main activities (screens) - login, labeling activity, results activity.



(a) : Login activity **(b) :** Labeling activity **(c) :** Results activity

Figure 6.5: Screenshots of application activities

6.7.1 Login Activity

This activity is launched whenever the user has no valid session. All the authentication is checked against the Firebase Authentication module.

A user is signed in upon successfully entering a valid e-mail x password combination. A new user is always added by a central authority (system admin) via Firebase Console.

Apart from e-mail and password, Firebase also stores information about devices the user is responsible for, so that users can only see products they are directly responsible for, it also stores the privilege levels - user can label and review or review only.

6.7.2 Labeling Activity

This is the app's main activity and also the launching activity. Whenever the app is launched a valid user session is checked first. If the user is not authenticated he/she is redirected to the Login Activity.

This is the activity where authorized users label images for training but the results are also used to update the products' status until a CNN model is

trained. The images are visualized as floating cards with the product/device IDs at the bottom. Users can either swipe right (In Stock) or left (Low). Once the user votes, the Record is updated at Firestore and the main program handles the rest (training, updating status). Once a user labels all unlabeled images, the app shows just a blank screen. Ideally, the app should be checked regularly and the screen should be kept blank.

6.7.3 Results Activity

This activity shows a list of all the monitored products that the signed-in user is responsible for. One item of the list always contains the following:

- Last captured image - this way even if the automated evaluation triggers incorrect actions, the responsible user can override it.
- Last captured image timestamp
- Product/Device ID
- Product status - this way a user is always able to see the generated status and in case it is faulty, replace it

Chapter 7

Machine Learning

This section presents different approaches considered for image evaluation, it also provides a brief description of convolutional neural networks (CNNs), it's training, finetuning of its architecture and the PyTorch library used for final implementation.

7.1 Discussion

First, it should be reviewed what the actual purpose of machine learning will be. this project does not strive to create a bulletproof solution, that will always return an accurate estimate of the amount of stock. The main power of this project is the almost-realtime remote access to information that would otherwise require physical presence. The machine learning algorithm is rather a supporting technology that further simplifies the inventory monitoring process. It is expected to run in supervised mode meaning that the user should still validate the outputs and only once the functioning has been thoroughly tested can we consider switching the model to fully autonomous mode. To simplify the need process of supervision, the mobile app was designed to minimize the time required to validate and review the outputs, so that the human validation can be done easily and quickly.

Second, the deployment environment should be specified. One of the main intentions was to create a technology that is easily implemented by an unskilled person, can be applied to any situation and any product/material. The goal of simple deployment and training outweighs the accuracy a more complicated model could bring. The accuracy is not crucial for proper functioning, because even if the machine learning performs poorly for any reason, the application can still be successfully used to monitor stock level remotely.

Generally speaking, for applications where an exact amount of stock is expected, this tool is not the right choice as it only makes a binary decision, on the other hand, if the monitoring task is only full vs. empty choice, this tool performs quite well on virtually any material/product.

The technology eventually used is a convolutional neural network classifier, with two categories full (ok) / empty (not ok).

7.2 Considered Approaches

Convolutional neural networks are likely one of the most booming machine learning algorithms of the last decade and were intuitively the go-to technology, however, other approaches were considered before opting for CNN classifier.

- Edge detection/Object detection[18]
  Naturally, edge detection was one of the technologies considered for this project. This technology would be the go-to if the intention was not binary but exact level detection, for example, to count boxes on a shelf or to estimate the amount of sand in a pile by rotating its 2D silhouette. However, edge detection would require a lot of customization for each application and therefore contradicts one of the requirements stated above - easy deployment by an unskilled person.
- SVM classifier[14]
  This algorithm was considered for its simplicity and could replace CNN. The main reason it was not eventually used is the fact that even though only a binary classification is currently required, more classes could be added in the future (e.g. 10 classes of different percentage range 0-10, 10-20...) and SVM does not natively support more than 2 classes.
- CNN classifier[18]
  As mentioned, CNNs are the latest trend in computer vision, it allows an arbitrary number of recognized classes and there is a vast selection of various frameworks with huge communities and many resources - papers, projects, implementations. The only drawback could be the training costs, however, we do not expect to train huge models (small-medium sized datasets of high tens, low hundreds), and therefore it is not an issue. Also, this technology was tested at the beginning of the work and the positive results supported the choice.

7.3 Neural Networks

[5] A neural network is an iterative machine learning algorithm that in some abstraction tries to imitate the functionality of a human brain. One of the huge advantages of this technology is that it requires minimal input data preprocessing or other human interaction.

A convolutional neural network is a subclass of neural networks, it consists of multiple layers which are interconnected, these layers can be - convolutional layer, pooling layer, linear layer, etc. The number of layers is referred to as depth. It is most commonly used for image processing tasks, such as classification, object detection but also images generation or image quality upscaling. A convolutional neural network process the image as volumes, receiving the inputs as combinations of adjacent pixels. These combinations are calculated using a convolution mask (matrix). Determining the optimal values of these matrices' members (weights) is called training of CNN.

7.4 PyTorch

PyTorch[18] is one of the main Deep Learning frameworks together with TensorFlow, Keras, or Caffe. One of the biggest advantages of this framework is the support of CUDA, graphic card computation, which can speed up the learning process by a factor as high as 50 times.

The building block of this library is a Tensor, which is essentially a multidimensional array (matrix) and just like matrix it supports matrix operations such as multiplication or adding.

Another core functionality of the framework is the Autograd package. "The autograd package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different."[18]

7.5 Training the Convolutional Neural Network

The model expects two input datasets, one for training, one for testing. The training dataset is used for the iterative updating of the weights and the test dataset is used to evaluate the model's performance on general data. Using two datasets prevents overfitting - model that performs very well on the training data but very poorly on any general data it has not seen before.

A specific module `dataset.py` implements the Dataset class that is used to represent these datasets. It also implements methods to gather images from Firebase and transform these images into the expected format.

The process of training consists of repetitively iterating through the collected images and retrospectively updating the weights of the model, based on the received and the intended outputs. The relation between the received and the intended outputs brings us to another essential concept of CNN, which is the loss function. It is used to determine how "far" away is the received output from the intended one. This knowledge is then used to update the weights so that the next iteration yields better results.

Thanks to the PyTorch framework the backpropagation process is very simple and done almost automatically. The only thing a user has to define is a loss function and the chosen strategy of updating weights. Cross-Entropy loss and Stochastic Gradient Descend were chosen for this project since they come predefined with the framework and initial testing proved the selection to be working.

Listing 7.1: CNN training implementation

```
def trainCNN(self, DS, TST_DS,
             ITERS, BATCHSIZE,
             lr=LEARNING_RATE,
             mom=MOMENTUM):

    criterion = nn.CrossEntropyLoss()
```

```
    optimizer = optim.SGD(self.parameters(),
                          lr=lr, momentum=mom)

    for i in range(ITERS):
        # get batch from dataset
        inputs, labels, categories = DS.get_batch(BATCHSIZE)

        # map human readable labels to int series
        labels = [self.mapper[label] for label in labels]

        optimizer.zero_grad()

        inputs = torch.from_numpy(inputs)
        labels = torch.from_numpy(np.array(labels))

        # forward + backward + optimize

        outputs = self(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        if i % 20 == 0:
            loss_tst = self.testCNN(TST_DS, BATCHSIZE//2)
            if loss_tst < 0.2:
                return True
    return False
```

7.6 Finetuning and Final Architecture

As there is not such a thing as the correct way to design a CNN classifier, the designing and finetuning of a classifier is strongly test-driven. It requires multiple train and test runs until a well-behaving CNN is created. However, the results are also very data sensitive. The more generalized the dataset is, the better the CNN tends to perform. Apart from the structure of the network a couple of parameters also affects the training. These are the momentum and the learning rate used to optimize SGD.

The CNN used for this project is one of the simpler ones, it uses several convolutional layers with max-pool layers inserted in between and then 3 fully connected layers. Relu is used as the activation function for all the layers.

Chapter 8

Results

The final solution looks as depicted in figure 8.1:

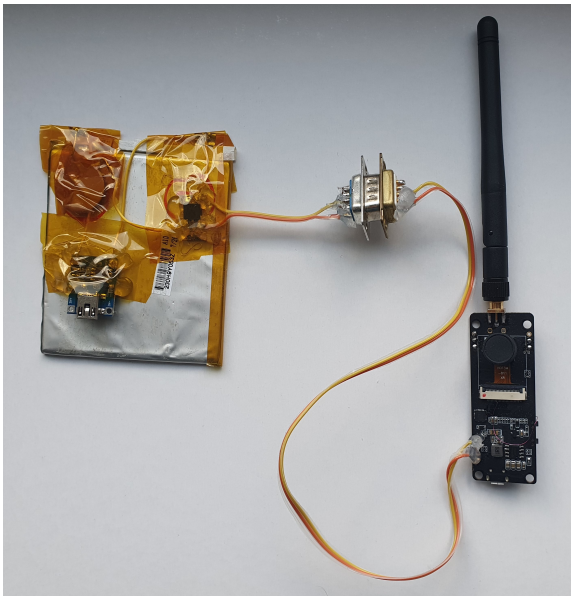


Figure 8.1: Device with battery

To test the solution, a simple "warehouse" like environment was setup. Just like in warehouse, the camera is supposed to be fixed to a certain location, therefore neither the surrounding surface nor the location of the camera changes too much. To imitate this, the camera was fixed to a stand and a white cardboard was used as a background (viz. figure 8.2).

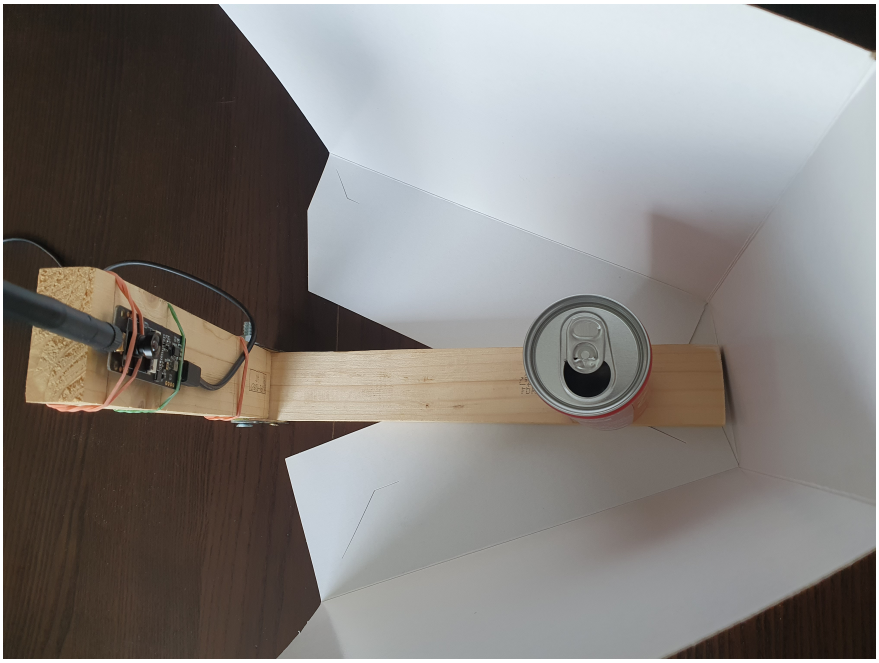


Figure 8.2: Camera setup for testing

The task was to test if a Coca-Cola can is present or missing. For training the CNN we gathered 21 unique images and trained a model. To improve the model performance we shifted around the lighting and the position of the can during image capturing. This should create a less environment-dependent solution. The resulting model was able to distinguish the two states OK/LOW for 10 consecutive times, each time with different lighting and state (present/missing). These results are visualized in figure 8.3 and figure 8.4.

Unfortunately, due to time constraints, more testing could not be done. However, throughout the development, the functionality has been tested multiple times. Usually the task was to determine a volume on different liquids in a glass/bottle. If the amount of liquid dropped below a certain level the CNN was able to set the status to out-of-stock (low).

Figure 8.3: Demo for status OK

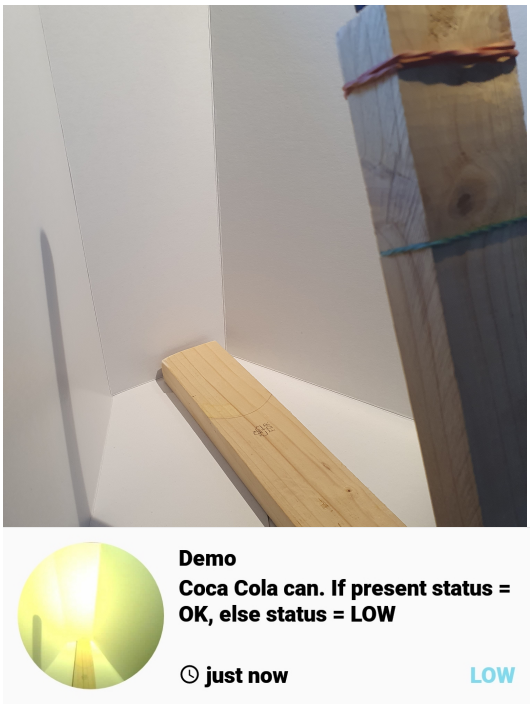


Figure 8.4: Demo for status LOW

Chapter 9

Conclusion

This thesis' objective was to get a working camera device, that captures and transmits an image to remote storage. This image is then either manually evaluated by an end-user in the mobile app or by a pretrained CNN classifier, given enough images have been previously manually labeled.

The result of this thesis is the source code implementing all the back-end functionalities, the android mobile app, and the physical device with firmware.

The core of all back-end functionality is the `main.py` module which makes use of all the other modules developed for this project. It is a Python script and uses both multiprocessing and multithreading to improve its performance.

The android mobile app is a simple mobile app that serves the purpose of visualizing the data to an end-user and is also used for manual labeling of the images.

Last is the physical device. It is a modified LilyGo T-Journal development board. The modifications were carried out to lower power consumption and improve battery life.

In conclusion, the objective of this thesis was met. A working solution of an inventory monitoring system was created. A modified T-Journal development board repetitively uploads images to Firebase, from where these images are downloaded and processed in the main program. The main program either evaluates the image using one of the pre-trained CNN classifiers or requests manual labeling in the mobile app.

A couple of things can be improved, the device is still consuming considerable amount of power, which could be lowered with further modifications to the board. Image processing needs more testing to determine its optimal working environment, potentially other image processing techniques can be applied. The mobile application's user-access settings, overall security and the business logic of working with the outputs need to be developed before deployment to production.

Appendix A

Bibliography

[1] A. Al-Masri. How does back-propagation in artificial neural networks work? https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7, Jan 2019. Accessed on 2020-04.

[2] J. Brownlee. Loss and loss functions for training deep learning neural networks. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks, Jan 2019. Accessed on 2020-04.

[3] R. T. Cato and T. G. Zimmerman. Using cameras to monitor actual inventory. https://patents.google.com/patent/US8091782B2/ens. Accessed on 2020-05.

[4] N. T. P. A. Corp. Tp4056 datasheet. https://dlnmh9ip6v2uc.cloudfront.net/datasheets/Prototyping/TP4056.pdf. Accessed on 2020-05.

[5] DeepAI. Convolutional neural networks. https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network. Accessed on 2020-04.

[6] Firebase. Firebase landing page. https://firebase.google.com/docs. Accessed on 2020-04.

[7] fustyles. Esp32-cam_base64 - program. https://github.com/fustyles/Arduino/blob/master/ESP32-CAM_Base64/ESP32-CAM_Base64.ino. Accessed on 2020-04.

[8] I. P. Hejda. Latex template. https://github.com/tohecz/ctuthesis. Accessed on 2020-05.

[9] K. K. Katircioglu and Y. Li. Machine vision technology for shelf inventory management. https://patents.google.com/patent/US20150262116A1/en. Accessed on 2020-04.

[10] LilyGO. http://www.lilygo.cn. Camera board manufacturer.

[11] LilyGO. Esp32-camera - program. https://github.com/LilyGO/ESP32-Camera. Accessed on 2020-04.

[12] mobitz. Firebase-esp32 - program. https://github.com/mobizt/Firebase-ESP32. Accessed on 2020-04.

[13] MOHAnet. Mastershelf. https://yourstockmonitoring.com. Accessed on 2020-04.

[14] A. Navlani. Support vector machines with scikit-learn. https://www.datacamp.com/community/tutorials/svm-classification-scikit-learn-python. Accessed on 2020-05.

[15] S. U. S. of Engineering. Online courses for convolutional neural networks. https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv. Accessed on 2020-05.

[16] OmniVision. Ov2640 datasheet. https://www.uctronics.com/download/cam_module/OV2640DS.pdf. Accessed on 2020-05.

[17] Panasonic. Out-of-stock detection. https://panasonic.net/cns/invc/osd/. Accessed on 2020-04.

[18] PyTorch. Pytorch. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html. Accessed on 2020-04.

[19] N. Santos. Esp32 arduino: Base64 enconding. https://techtutorialsx.com/2017/12/09/esp32-arduino-base64-enconding. Accessed on 2020-05.

[20] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis, and Machine Vision*. CL Engineering, 2007.

[21] ST. Ldf33dt datasheet. https://datasheet.octopart.com/LDF33DT-TR-STMicroelectronics-datasheet-21772407.pdf. Accessed on 2020-05.

[22] Stackoverflow. Coding community. https://stackoverflow.com. Accessed on 2020-05.

[23] StockVue. An iot solution for real-time inventory management. http://stockvue.net/White-Paper.pdf. Accessed on 2020-04.

[24] thisbejim. Pyrebase source code. https://github.com/thisbejim/Pyrebase. Accessed on 2020-05.

[25] TutorialEdge.net. Python multiprocessing tutorial. https://tutorialedge.net/python/python-multiprocessing-tutorial. Accessed on 2020-04.

[26] Wikipedia. Esp32. https://en.wikipedia.org/wiki/ESP32. Accessed on 2020-05.

[27] S. Williams, J. P. Gonzalez, and S. Skaff. Multiple camera system for inventory tracking. https://patents.google.com/patent/WO2018005369A1/en. Accessed on 2020-04.

[28] K. Zimmermann, T. Petricek, and T. Azayev. Ctu vision for robotics course. https://moodle.fel.cvut.cz/course/view.php?id=2674. Accessed on 2020-05.

Appendix B

CD content

- SmartInventory
  - java - java files
  - res - xml files and static files
- main.py
- settings.py
- dataset.py
- networks.py
- task.py
- firebase_client.py
- record.py
- utils.py
- image_processing.py
- requirements.txt