Ministry of Higher Education and Scientific Research

University of Tunis El Manar

Faculty of Sciences of Tunis

Department of Information Technology

2nd year of Computer Engineering Degree - End of Year Project Report

EMBEDDED COIN RECOGNITION SYSTEM

Authored by

Dhouha GAOUD Khalil OUALI

Proposed and supervised by

Prof. Kamel ECHAIEB

UY 2022-2023

ACKNOWLEDGEMENTS

We thank professor Kamel ECHAIEB for his guidance and supervision in the making of this work.

We are grateful to our esteemed professors at the Faculty of Sciences of Tunis, for the knowledge and education they have imparted upon us.

And we thank our beloved families and friends for their continued help and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Improved chip manufacturing has led to the rise of embedded systems, which are now found in a variety of products and systems. These tiny computing devices have revolutionized automation and optimization, thereby increasing efficiency and reducing costs. As demand for interconnected and intelligent devices continues to grow, the future possibilities of this technology are limitless.

The rise of machine learning and image recognition has been another major technological advancement in recent years, enabling algorithms to classify images with high accuracy, which has led to a wide range of applications in various industries. As this technology continues to evolve, it has the potential to drastically change how we interact with machines and the world around us.

The combination of the two fronts has vast potential applications, improving efficiency, accuracy, and safety in a myriad of fields. As these technologies continue to evolve, they offer exciting possibilities for innovation and advancement, providing unprecedented solutions to old problems.

As part of our Computer Engineering degree, we are fortunate to be exploring both of these technologies and their intersection. In particular, our second-year final project has provided us the opportunity of developing an embedded system for image recognition.

For our project's subject, we picked "Coin Recognition System for the Blind and the Visually Impaired". Our choice was motivated equally by the technical aspect and the humanitarian value; we would get the chance to experiment with an embedded programmable platform, try some machine learning libraries, and hopefully prototype an assistive device for people with disabilities.

This report chronicles our endeavor in analyzing the problem, researching potential technologies, designing a solution, and creating a functional prototype. All the while, we bring to light some of the hurdles we encountered and the steps we took to troubleshoot them.

It is important to note that this report does not go into exhaustive detail concerning machine learning, computer vision and embedded platforms. Although a certain level of understanding was required for each of these topics, a comprehensive discussion lies outside the scope of our project and this report.

CHAPTER I

CONTEXT

1. Introduction

Finance management and monetary transactions are indispensable aspects of day-to-day life. They are particularly difficult, however, for the blind and the visually impaired, as they face challenges in identifying coins and paper bills. This raises their reliance on other people, and harms their privacy. That said, recent advancements in the fields of image recognition and embedded systems may enable the development of new solutions to the problem: a compact and portable coin recognition system.

2. Beneficiaries

The proposed system would mainly benefit the blind and the visually impaired by reducing their dependence and restoring their privacy in financial matters. By extension, the system alleviates the burden placed upon caregivers, family members, and government officials in assisting the target users. A successful solution may also have supplementary uses for automating or facilitating cash exchanges in banks and stores.

3. Available solutions

In the USA, the Treasury Department provides the iBill Talking Bank Note Identifier [1], a compact and portable device which can identify an inserted US currency bill and inform the user of its value in several ways. It has additionally developed mobile apps which can recognize a photographed bill.

Figure 1. The iBill Talking Bank Note Identifier

There are also many other coin and currency recognition apps available on both Android and iOS platforms [2], making use of different software technologies and serving different purposes.

4. Scope

The objective of this project is to create a compact and portable optical recognition system, designed specifically to recognize common Tunisian coins, and calculate the sum of a few coins, and output an audio readout of the sum. For prototyping, we grouped coins by color (Yellows, Silvers, and Fives).

The prototype will be primarily evaluated in terms of speed and accuracy. But, following further development, it should meet several other requirements including portability, compactness, ease of use, etc.

5. Methodology

The suggested system makes use of an embedded hardware platform comprised of an SoC[1] and a number of peripherals and I/O devices. It runs a CNN[2] image recognition model which will be trained on a dataset of labeled images using supervised learning techniques. To develop the system, we first defined its specifications, then researched potential hardware and software solutions, and produced a functional prototype.

6. Conclusion

In summary, the proposed system is a promising solution to the challenges faced by blind and visually impaired individuals in managing their finances and carrying out transactions. By combining the compactness of an embedded computing platform with the power of machine learning, it should offer a reliable, accessible, and convenient solution which enhances independence and improves privacy.

[1] System on a Chip: An integrated circuit combining several key computing components
[2] Convolutional Neural Network: A machine learning algorithm based on nodes and weights

CHAPTER II

DESIGN AND SPECIFICATIONS

1. Introduction

The first stage of the project was to design and model the system, and from there, set a plan of action. This involved defining the specifications of the system, followed by conceptualizing a potential solution and researching hardware platforms and software solutions which would help us conceive it.

2. Technical specifications

After understanding the subject matter, we formulated a technical specifications document *(see appended Technical_Specifications.pdf)* where we defined the following aspects:

a. Objective

The core task that the system is meant to accomplish: Allowing the blind and the visually impaired to recognize coins.

b. Functional and non-functional requirements:

The functionalities which the system must support in order to accomplish the objective (E.g., image capture and processing, input detection, audio output), as well as the qualities that would make the system practical to use (E.g., portability, ability to update the software).

c. Choice of hardware

Potential hardware solutions which would comprise the system. In the original specification, we had intended to use an ESP32-CAM module as the platform for our embedded system. However, due to availability and technical issues, we were not able to obtain a functional module. Instead, we resorted to using a Raspberry Pi 3B Plus board, along with a Raspberry Pi Camera (Model: P5V04A SUNNY). In addition, we determined that we would need a button, a speaker, and a battery for portability (although we ignored this particular aspect during development).

Figure 2.a. ESP32-CAM

Figure 1.b. Raspberry Pi 3 Model B+

d. Choice of software

Potential software solutions which would enable the core functionality of the system: Coin recognition. Given its simplicity, ubiquity and extensibility, we settled on python for the programming language of choice. This allowed us a wide selection of image recognition libraires to choose from, including OpenCV, TensorFlow and Detecto. In the end, we opted to work with Detecto.

Figure 3. Possible Technologies

Detecto is a python package which is built on top of the reputable PyTorch image recognition library. It is meant to greatly simplify the training and usage of a custom PyTorch model. It accomplishes this by making use of a few pretrained models, and providing a number of predefined classes and functions. Thus, Detecto greatly reduces the amount of boilerplate code and technical know-how required to build a custom image recognition model.

3. MBSE[3] description

Having defined the requirements and selected some of the technologies to be used, we proceeded to model our system through three relevant SysML diagrams *(see appended MBSE_Description.pdf)*. For this, we used the StarUML modeling software. During this step, we were yet unsure as to the platform we would be using. So, we chose to refer to it generically as the SoC board.

a. Requirements diagram (req)

In the requirements diagram, we outlined the principal non-functional requirements of the system: performance and portability, as well as their sub-requirements.

b. Sequence diagram (sd)

The sequence diagram details the usage sequence of the system: the user pushes the button which sends an interrupt signal to the SoC. Once the signal is received, a photo is captured from the camera and the coins therein are recognized. Finally, the system outputs audio message to the user through the speaker.

c. Block definition diagram (bdd)

The layout of the hardware components is defined in the block definition diagram. Each of the peripherals: a camera, a speaker, an activation button and a battery, are wired to the SoC Board through the appropriate ports. Each block is attributed a set of relevant properties and operations, and notes indicate which block(s) satisfy each of the already-mentioned requirements.

4. Conclusion

In order to implement the system, we first laid the ground work by defining its specifications, researching key technologies required to engineer it, and modeling it. The next step was to learn and use the chosen technologies, and make them work together.

[3] Model-based systems engineering

CHAPTER III

USING DETECTO

1. Introduction

As mentioned in the previous chapter, we chose to use Detecto because it simplifies training a model using PyTorch, and reduces the boilerplate code involved. This is a log of our attempt to train a Detecto model and run it on the Raspberry Pi 3B Plus.

We used the dataset and the code skeleton from the official Detecto demo on Google Colab [3]. The demo includes a sample dataset including images and labels, and the resulting model can be used to recognize two breeds of dogs –Chihuahuas and Golden Retrievers– in an input image.

2. Set up

a. OS installation and configuration

We used Raspberry Pi Imager[4] to configure and write a Raspbian Bullseye (64-bit) on an SD card. The 64-bit version is necessary to install and run PyTorch [4]. The following steps facilitated the rest of the project:

To make accessing the Pi and transferring files to/from it faster and more convenient, we enabled and used SSH and SCP. The Secure Shell Protocol (SSH) is a means of logging into a computer and controlling it via a remote terminal over the network, while the associated Secure Copy Protocol (SCP) enables secure file transfer under the same settings.

To speed up communication with the Pi, we connected it to a windows machine with an Ethernet cable, and enabled internet connection sharing [5]. This provided a quicker and more reliable connection between the two devices compared to Wi-Fi.

b. Camera stacks

Older versions of the Raspberry Pi OS offered the console commands `raspistill` and `raspivid` as a means of using the Raspberry Pi camera via terminal, as well as the python library `picamera` to control and use it programmatically. These tools use the *legacy* camera stack and have been deprecated on newer versions of the OS.

Raspbian Bullseye, the latest version of the Pi OS which we are using, comes with `libcamera` and `picamera2`, which substitute the old console apps and python library respectively. These utilize the new camera stack [6].

It is possible to re-enable the legacy camera stack and use it. Initially, that is what we did, as most online tutorials and documentation reference the deprecated tools. Finally, however, we chose to use the new one as it is easier to use and more stable.

[4] Raspberry Pi Imager is the official GUI tool offered by the Raspberry foundation to flash SD cards with configured Raspberry Pi OS images.

c. Dependencies

In order to run a Detecto model, we needed to install the Detecto python module and all of its dependencies. PIP (Pip Installs Packages) is the default package manager for python. Normally, it automatically installs all required dependencies for a module. That said, special steps are required to get Detecto running on the Raspberry Pi:

i) **Installing OpenCV:** On the Raspberry Pi in particular, it is recommended to install OpenCV via APT instead of PIP, to avoid long build times [7]. The commands for doing so are included in the Picamera2 manual:
```
sudo apt install -y python3-opencv
sudo apt install -y opencv-data
```

ii) **Installing PyTorch:** We tried installing the default PyTorch packages using `pip install torch torchvision torchaudio`, but there seems to be an issue with them: `import torch` in python crashes with an `Illegal Instruction` error. Installing an older version fixes the issue [8]:
```
pip install torch==1.13.1 torchvision==0.14.1 torchaudio==0.13.1 3
```

iii) **Installing Detecto:** We can finally install Detecto using PIP [9]:
```
pip install Detecto
```

3. Training the model

a. Hardware

Given the limited power and performance of the Raspberry Pi's hardware, it is entirely impractical to train the image recognition model on the device directly. There are two main choices for the training hardware:

Method	**Using Google Colab**	**Locally**
Description	A cloud computing platform with GPU/TPU acceleration available. It allows one to write and run an interactive Python Notebook. [10]	Downloading Detecto and PyTorch and training on a personal computer.
Advantages	• Offloads heavy computation to a free cloud server. • Generally faster than personal hardware	• Available offline. • No time spent uploading datasets and downloading models.
Notes	• The standard class on Google Colab offers a Tesla T4 [11]. A modern mid/high-end GPU (E.g., RTX 3060) may offer better performance on a local machine. • Files can be stored in Google Drive.	If your computer is equipped with a CUDA-capable GPU, it is recommended that you use Conda instead of PIP as it simplifies the installation of PyTorch for use with CUDA [12], which significantly speeds up training.
Results in our case	13m 40s training time (GPU Enabled)	10m 28s training time (RTX 3060 equipped laptop)

Table 1: Training in the cloud vs locally

In our case, we opted to train on local hardware for the cited advantages and performance uplift.

b. Files

Training a Detecto model requires a dataset and a python script [13]. In the official Detecto demo, the dataset is made up of a number of images of dogs found in the images folder, and XML format labels split in two folders: train_labels (used to fit the model) and val_labels (used to validate the model's recognition results after training). The labels reference the image files, and include other information such as the bounding box and class. All three folders must be placed alongside the python training script.

c. Code

Our code for training the model is based on the code from the official Detecto demo on Google Colab, though slightly modified. It is comprised of multiple steps:

i) **Imports:** Importing the necessary python modules

ii) **Conversion:** Converting the labels from XML to CSV format to speed up parsing during training.

iii) **Dataset preparation:** Loading the images and labels for training and validation, as well as augmenting and normalizing the training dataset to improve training

iv) **Model instantiation:** Creating a Detecto model based on a pretrained object recognition model and initializing it for a set of classes.

v) **Training:** Fitting the model to the training dataset over multiple epochs and tracking its performance by recording the losses after each epoch.

vi) **Evaluation:** Plotting the loss function and trying the model on a few sample images from the validation dataset.

vii) **Saving:** Exporting the model for later use in other scripts.

The code can be written in a conventional `.py` script or an interactive python notebook `.ipynb` which is enhances readability and enables the execution of individual blocks of code individually. We chose to use a python notebook.

4. Trial and error

After training the model and running it successfully on the windows machine, it was time to try it on the Raspberry Pi. However, we ran into multiple issues while doing so.

a. 1st try (Failure)

We first trained the model based on *fasterrcnn_resnet50_fpn*, which is the default in Detecto. We tried running the model on the Raspberry Pi multiple times, with two possible outcomes:

- The device runs out of memory and the process is killed by the OS.

```
lceStudent@pi64:~/Documents/detecto $ tail /var/log/syslog
Apr 22 18:15:05 pi64 kernel: [  829.991737] [    893]  1000   893    39255      128    57344       19
Apr 22 18:15:05 pi64 kernel: [  829.991753] [    900]  1000   900    58443      135    73728        0
Apr 22 18:15:05 pi64 kernel: [  829.991768] [    905]  1000   905    77941      204    98304        1
Apr 22 18:15:05 pi64 kernel: [  829.991784] [   1037]  1000  1037    48499     1998   151552        0
Apr 22 18:15:05 pi64 kernel: [  829.991799] [   1059]  1000  1059    40038      154    65536        1
Apr 22 18:15:05 pi64 kernel: [  829.991815] [   1073]  1000  1073    95886     2201   188416        0
Apr 22 18:15:05 pi64 kernel: [  829.991831] [   1079]  1000  1079     1995      406    49152        1
Apr 22 18:15:05 pi64 kernel: [  829.991857] [   1177]  1000  1177   433879   176397  2035712        0
Apr 22 18:15:05 pi64 kernel: [  829.991872] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=/,mems_a
Apr 22 18:15:05 pi64 kernel: [  829.992002] Out of memory: Killed process 1177 (python) total-vm:1735516kB, ano
```

Figure 4. Screenshot of the system log after running the model

- The device completely freezes during detection, and has to be manually reset.

We deduced that this is because the model takes up a lot of RAM as it is not optimized for embedded systems.

b. 2nd try (Failure)

Hoping to optimize the model, we *quantized* it using PyTorch (with each of the three default configs), and tried running it again on the Pi. The result is that the program crashes with a `RuntimeError: Unknown qengine` while loading the model. After researching the issue, we found out that this happens because quantized models require AVX2 instructions to run, which are not supported by the Raspberry Pi's processor [14].

c. 3rd try (Success)

At this point, we decided to try another base model and chose *fasterrcnn_mobilenet_v3_large_fpn*, and we finally obtained a model which works on the Raspberry Pi. Performance is less than ideal: both loading the model and running it require a fair bit of time. Nonetheless, we were happy to obtain a functioning model.

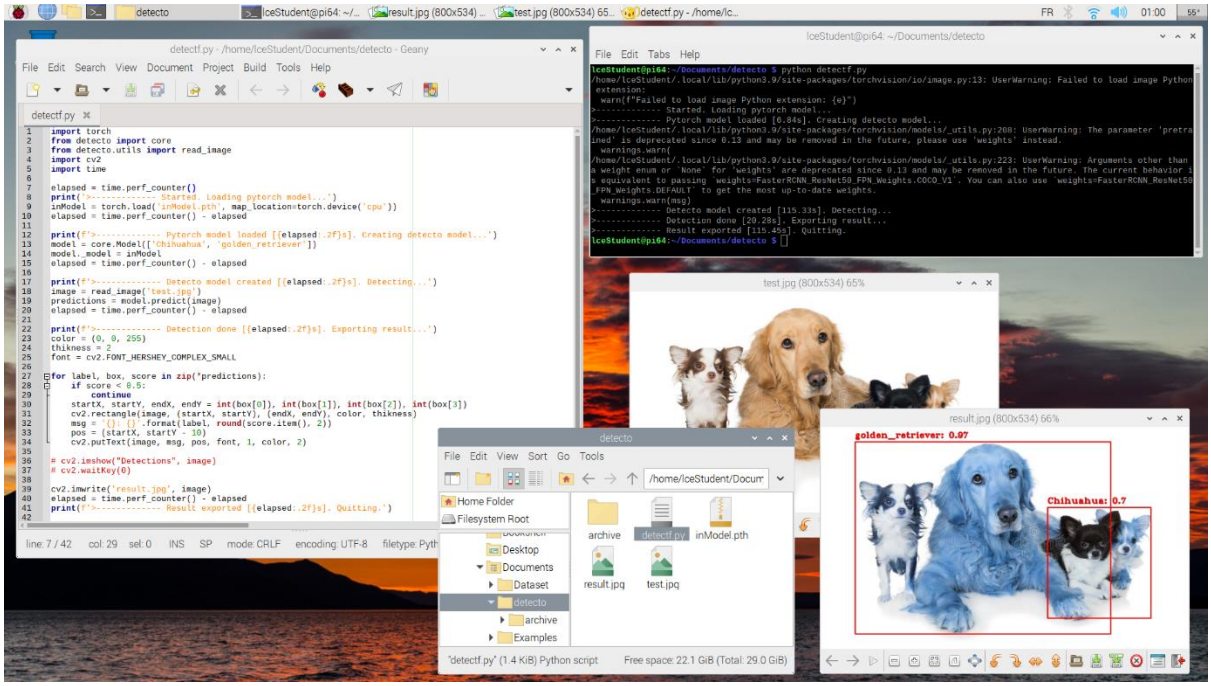

Figure 5. Screenshot of the desktop after running the recognition successfully

Notes:

- For some reason, Detecto could no longer save and load the model correctly once we changed the base model. Instead, we had to save the internal (PyTorch) model, then load it using PyTorch and recreate the Detecto model with it.
- Since the model was trained on a CUDA GPU but must run on a CPU, it is necessary to specify the argument `map_location=torch.device('cpu')` while loading it later.
- The default Detecto function for showing the detection result failed (seemingly due to graphical library issues), and displaying the result using OpenCV causes the device to freeze. Instead, we had to parse the detection result and use OpenCV to export an image with the result. This step takes a significant amount of time.
- Multiple warnings are output by PyTorch while loading the model, indicating the use of deprecated parameters. This is likely because the Detecto package is relatively old and unmaintained. The latest commit to the project on GitHub was on February 9, 2022.

5. Conclusion

In the end, it became evident that Detecto is an outdated solution, and one that is maladapted for embedded systems. Enabling it to work on a Raspberry Pi 3B Plus, while possible, required significant time and effort, and yielded unsatisfactory results. Nonetheless, we opted to continue using the package for multiple reasons including the time constraint, simplicity of code, and uniqueness of the solution.

CHAPTER IV

BUILDING THE DATASET

1. Introduction

Once we became confident of our ability to run a Detecto model on the Raspberry Pi, we moved on to building the dataset to train our custom coin recognition model. Initially, we hoped to find a ready-made dataset that would suit our needs online. However, given that our model was meant to recognize Tunisian coins, we had no luck finding one. Instead, we had to capture the required photos and label them ourselves.

2. Capture

a. Preparation

Given the fragility of the Raspberry Pi camera and the difficulty involved in handling it, we decided to mount the device and the camera at a fixed position, removing the need to hold or move either during capture. This enabled us to collect the images faster.

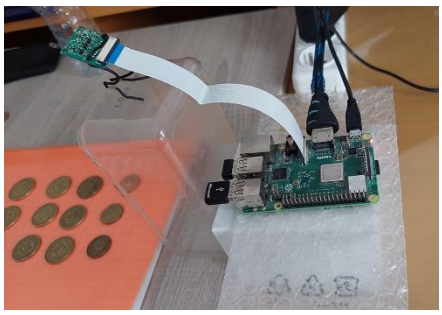

Figure 6.a. Full image capture setup

Figure 3.b. Camera affixed using a cable tie

In order to speed up and facilitate the image capture, we wrote a custom python script which starts a low-resolution preview window and awaits input. It captures a high-resolution numbered photo each time `Enter` is pressed, and closes once `q` is entered.

b. Photos

While the camera was fixed in place, we took a large number of photos (134 in total) each containing 9 or 12 coins. The coins –and by extension, the images– were grouped into 3 categories: Fives (5dt coins), Silvers (500mil, 1dt and 2dt coins) and Yellows (20mil, 100mil and 200mil coins). Each group was captured both face up and face down in multiple lighting conditions (E.g., natural sunlight, indoors lightbulb, flashlight) on multiple backgrounds (E.g., desk surface, black binder, orange notebook cover). This was done in hopes of diversifying the dataset and, consequently, improving the overall accuracy of the final model.

Figure 7. Examples of dataset images

3. Labeling

a. Labeling software

Labeling was done using the LabelImg, an open-source python program which allows us to draw a bounding box around each coin and give it a class [15]. The program then saves an XML file containing the relevant information. As mentioned before, we grouped the coins into Fives (f), Silvers (s) and Yellows (y).

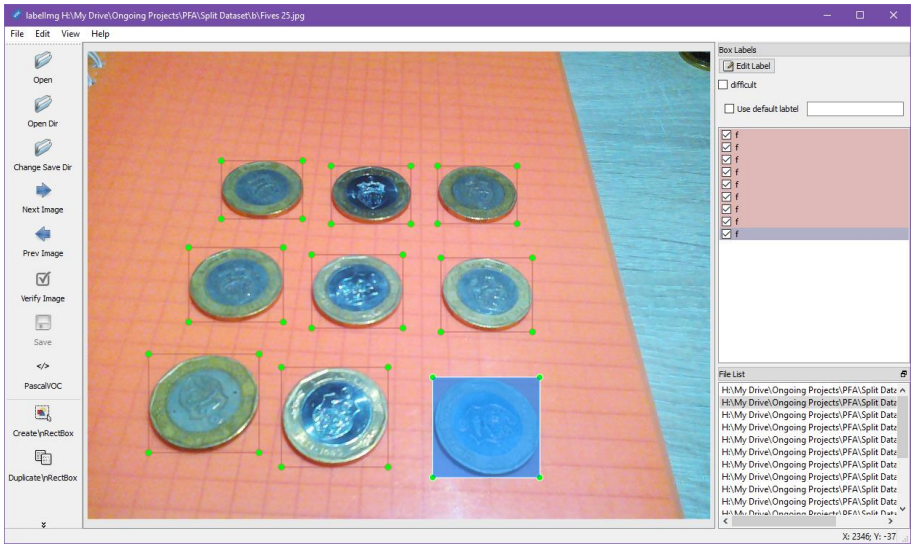

Figure 8. A screenshot of LabelImg

There are indeed other options for labeling software, some more advanced and feature-rich than LabelImg, namely labelme [16] and CVAT [17]. Nonetheless, we chose to use LabelImg as it is simple, popular, and suits our needs adequately.

b. Time saving

To save time on the dataset preparation, we decided to share our captured images with the other pairs working on the same project. Each member of the overall group would label a share of the images. To that end, we wrote another python script which splits the dataset by moving images randomly into subfolders.

Later however, we realized that we could speed up and improve labeling significantly by duplicating the same XML label file and reattributing it to multiple images. This was thanks to the fixed position and angle of the camera. So, we picked the best labeling for each group of similar images and duplicated it for the entire group.

Figure 9. Examples of identically positioned images

4. Hindsight

We were eventually informed of a number of mistakes we had committed while building the dataset.

- We captured many images which proved detrimental to the accuracy of the model, including very dimly lit or dark photos and photos with many reflections. Instead, we should have focused on capturing relatively clear and discernable photos.
- Given that this project is meant to be a proof-of-concept rather than a final functioning product, we should have prototyped with a plain white background. This would have ensured a higher accuracy in specific cases, even though it may have diminished the overall accuracy.
- Initially, we reasoned that grouping the coins by color (Fives, Silvers, Yellows) would make it easier for the model to discern the coins by that criterion. However, labeling the coins in large groups seems to harm accuracy significantly, especially since color alone is not an accurate metric. Instead, we should have classified the coins as specifically as possible (by exact value and face).

5. Conclusion

Building a proper dataset involves a lot of time and effort: taking photos and labeling them appropriately. It is also a critical step in training a custom image recognition model, and must be accomplished properly to ensure adequate performance after training. Images should be varied and clear, and labeling should be very distinct and accurate.

CHAPTER V

TRAINING A CUSTOM MODEL

1. Introduction

Having assembled our custom dataset, it was time to train a custom coin recognition model based on it. Afterwards, we would write the python script which would load the model and use it to recognize a captured photo.

2. Custom training

a. Code

We duplicated the python notebook used to train the dog recognition model *(see III. 3. c. Code)*, only making two key changes:

- `transforms.RandomHorizontalFlip()` is a dataset augmentation function, included in the original Detecto demo, which randomly flips images in the dataset to further diversify it [18]. We removed it because it does not fit our use case.
- The model in the original demo is instantiated with the `'Chihuahua', 'golden_retriever'` classes. We substituted them with our own `'f', 's', 'y'`.

The code is otherwise essentially the same and follows the same steps as the original.

b. Files

In addition to the python notebook, all that is necessary for training is the dataset. We placed the dataset images in the images folder as before. We also split the XML label files into training labels and validation labels (with the majority selected training), and placed them in the respective folders.

3. Further experimentation

We continued to train the model and evaluate it multiple times, changing different parameters in attempts to improve the mode's accuracy. We mainly varied the `batch_size` and the number of `epochs`, and we experimented with the `transforms.random_rotation()` augmentation function.

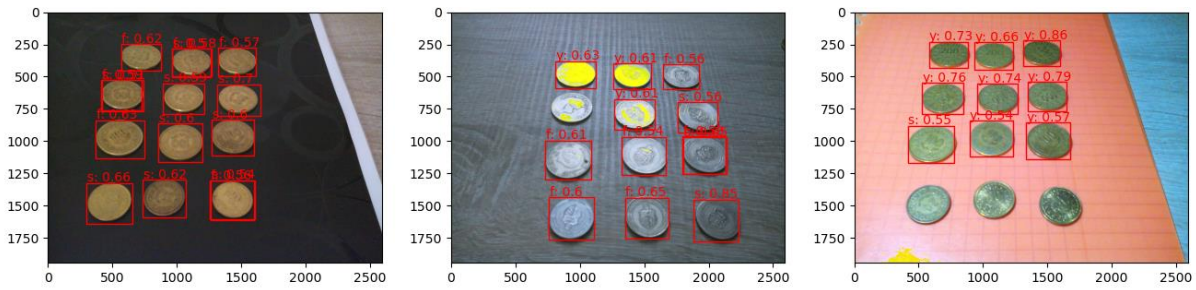

Figure 10. Sample recognition results

Despite a myriad of attempts, we failed to obtain satisfactory results: the model often misses coins, or mistakes them for coins of other classes. In certain cases, the model would perform acceptably during the evaluation phase, but fail once tested on a new photo taken after the fact.

4. Conclusion

Although we were able to train a functional coin recognition model, we would not deploy it for use in a final product, as it is very slow and highly prone to mistakes. Reasons for this likely include:

- The use of Detecto and the *fasterrcnn_mobilenet_v3_large_fpn* base model, which are rather outdated and perhaps offer subpar recognition performance by modern standards. *(See III. 4. Trial and Error, III. 5. Conclusion)*
- The use of a limited dataset with flawed labeling. *(See IV. 4. Hindsight)*
- The lack of knowledge and experience in training neural networks.

CHAPTER VI

ADDITIONAL SOFTWARE

1. Introduction

In addition to the prerequisite modules for image capture and recognition, the final program relies on a number of extra packages to provide the desired functionality (Button input detection and TTS generation), as well as activity and performance logging capabilities.

2. Button input

Although it is possible to use a standard input device, such as a keyboard or mouse click, we determined that the ideal solution would be to detect input from a GPIO connected button. This is made simple by gpiozero, a python interface for easily configuring and using the GPIO pins [19]. It is preinstalled and ready to use on Raspbian Bullseye.

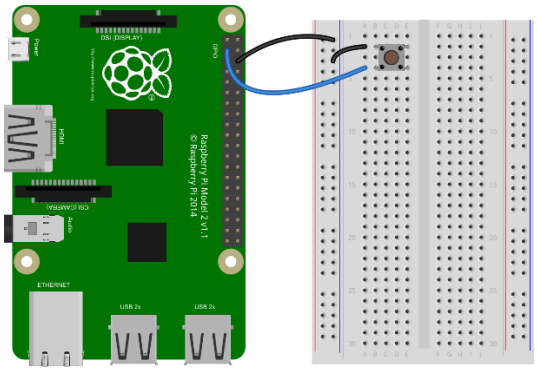

Figure 11. GPIO button wiring example, taken from the official docs

A plain example of a GPIO button wiring and configuration is available in the official documentation of the gpiozero module [20].

3. Text-to-Speech output

As specified in the original design, the system is meant to output audio to alert the user and read out the sum of money captured by the camera upon pressing the button. To achieve this, we researched and tried different options for Text-to-Speech (TTS) synthesis in python.

a. Coqui TTS

Our first resort was the open-source TTS package from Coqui-AI. This is a library which provides pretrained TTS-generation models, as well as utilities to train, configure and customize models [21]. We managed to produce excellent results on the test Windows machine.

Figure 12. The Coqui TTS library logo

Unfortunately, we were unable to use Coqui TTS on the Raspberry Pi 3 as it consumed too much memory and was repeatedly killed by the OS. We tried multiple pretrained models and configurations, but none yielded any results. Additionally, both the library and the models are quite sizeable and take a non-negligible amount of time to download and install on the Raspberry Pi.

b. pyttsx3

The second option was to use pyttsx3, a reputable python TTS package [22]. Unlike Coqui TTS, this library doesn't use AI voice generation. Instead, it invokes a local formant TTS engine: SAPI[5], NSSS[6] or eSpeak. eSpeak is the only supported engine available for installation on Linux.

Figure 13. The pyttsx3 library logo

While pyttsx3 provided acceptable results on Windows, the results we got on the Raspberry Pi were unsatisfactory: the quality of the sound and performance overhead of the packaged led us to abandon this approach.

c. Other python packages

gTTS is a TTS library which calls the Google Translate TTS API to generate the audio [23]. It is an example of a TTS library which requires internet connectivity to function. However, given that our system is meant to be entirely portable and work offline, we avoided such solutions.

d. Directly invoking eSpeak

eSpeak (or eSpeakNG) is a commonly used TTS program available for Linux and other operating systems [24]. It uses formant synthesis which makes it highly compact. It can also be extended with the MBROLA engine, which provides a number of other TTS voices.

As indicated previously, pyttsx3 interfaces with eSpeak on Linux, but the audio quality sounds slightly degraded and there seems to be an overhead. We opted instead to use eSpeak directly by invoking a shell command using the `subprocess` module in python. This produced by far the best results for our purposes.

- eSpeakNG and the libeSpeak voice library must be installed on the Raspberry Pi:
  `sudo apt install espeak-ng-espeak libespeak1`
- MBROLA can be installed for additional voices and languages (E.g., `mbrola-us1`, `mbrola-fr1`) [25]:
  `sudo apt install mbrola <voices>`

4. Logging

Activity and performance logging necessitates several preinstalled python modules:

[5] Microsoft Speech API (SAPI) 5.3 available on Windows
[6] NSSpeechSynthesizer available on macOS

- **datetime:** Provides the current date and time. Used to track start time.
- **time:** Contains the `perf_counter()` function which is the ideal way to measure performance in python, by counting exact CPU execution time.
- **logging:** Allows the logging of information to a local log file, offering some configuration and different levels of logging: Debug, Info, Error, etc.

5. Other modules

- **subprocess:** Allows launching a new subprocess which executes a specific shell command. Used to activate eSpeak as previously mentioned.
- **signal:** Contains the `pause()` function which suspends the python script's execution until reawakened by a specific event – In our case, the button press.

6. Conclusion

A set of software packages and utilities are required to achieve the desired final behavior of the program. Most are pre-installed and ready to use in python on Raspbian Bullseye, others were selectively downloaded and added.

CHAPTER VII

FINAL PROTOTYPE

1. Introduction

The final step of the project was to code the python script, load it on the Raspberry Pi along with the recognition model, and test the final prototype's functionality.

2. Hardware environment

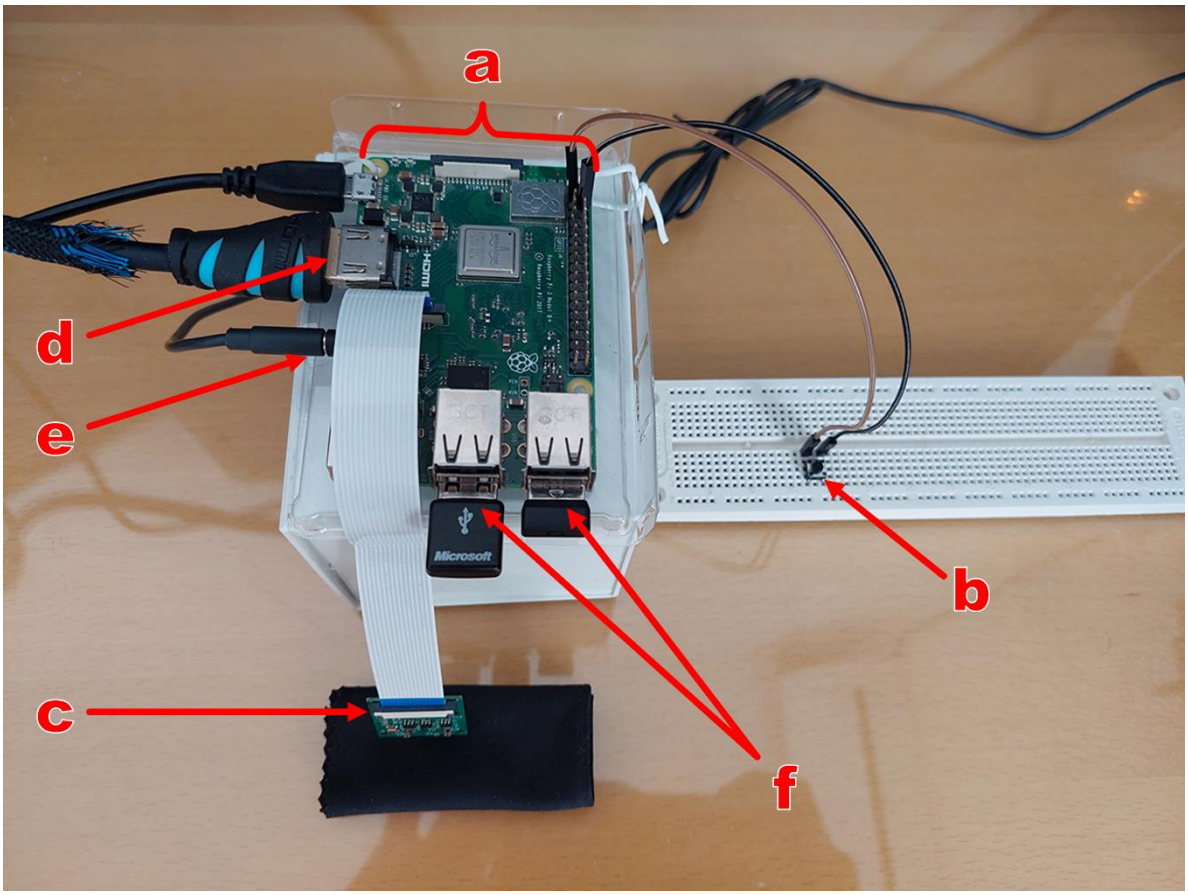

Figure 14. A photo of the hardware setup, with labels for several components/connections

a) **Raspberry Pi 3B Plus**, main board, powered by an AC/DC adaptor.
b) **A button,** wired to the GND pin and GPIO pin nº2 on the Raspberry Pi using 2 wires and a breadboard.
c) **Raspberry Pi Camera,** plugged into the CSI[7] connector of the Raspberry Pi. Model: P5V04A SUNNY.
d) **A display**, connected to the Raspberry Pi via HDMI.
e) **A speaker**, plugged into the audio out jack on the Raspberry Pi.
f) **Standard input peripherals,** a keyboard and mouse plugged into the USB ports.

[7] Camera Serial Interface. Used to connect a supported Raspberry Pi camera.

3. Software environment

As indicated in previous chapters, the final program's functionality depends upon a specific software environment which should be installed and configured on the Raspberry Pi 3B Plus.

a. Operating system

Raspbian Bullseye 64-bit. It can be configured and installed on an SD card using the official Raspberry Pi Imager tool. It includes special packages such as the picamera2 library (for using the Raspberry Pi camera) and gpiozero (for using the GPIO pins on the device).

b. APT packages

- **OpenCV:** A computer vision python library necessary for Detecto and the final script. Should be installed using APT instead of PIP. *(See III. 2. c. Dependencies)*
- **eSpeakNG:** A TTS engine commonly used on Linux. Provides audio output. *(See VI. 3. d. Directly Invoking eSpeak)*

c. PIP packages

- **PyTorch:** A machine learning python library used by Detecto. A specific version is required on the Raspberry Pi 3B Plus. *(See III. 2. c. Dependencies)*
- **Detecto:** Our python library of choice for training and using the image recognition model. *(See III. 2. c. Dependencies)*

4. Files

a. inModel.pth

A custom PyTorch model trained using Detecto, to be loaded and used in the recognition script. The file extension can be either `.pth` or `.pt`. In our case, the file size was about 76MB.

b. detector.py

The final recognition script written in python. It is comprised of several parts *(see III. 3. c. Code, VI. Additional Software)*:

i) Preparing and configuring the logging functionality and the TTS output function.
ii) Importing the packages required for core functionality.
iii) Loading and preparing the model.
iv) Preparing and starting the camera.
v) Preparing the **detection function.**
vi) Configuring the button.
vii) Suspending the process.

The **detection function** includes several steps:

(1) Capturing a photo to an array in memory
(2) Converting the photo to the correct color space
(3) Running the recognition model on the photo
(4) Parsing the recognition result and calculating the sum of the captured coins
(5) Outputting the calculated sum in text and audio format

5. Testing

Upon launching the python script in the console, the program outputs the phrase "Program started. Please wait." via audio. In total, the program takes about 45 seconds to get ready for use. Once ready, the program says "Ready to detect.".

Each time the button is pressed, the program takes a photo and says "Detecting. Please wait.". After the photo is recognized, the program outputs the calculated sum in dinars and millimes and again says "Ready to detect." to indicate it's ready for use. Recognition generally takes about 18 seconds to finish.

Figure 15 . Screenshot of the console after running detector.py

All the while it's running, the program logs its activity in a `log.txt` file. This includes the time and date of startup, each step of the initialization phase, as well as each of the detections. It also records the time taken by each step.

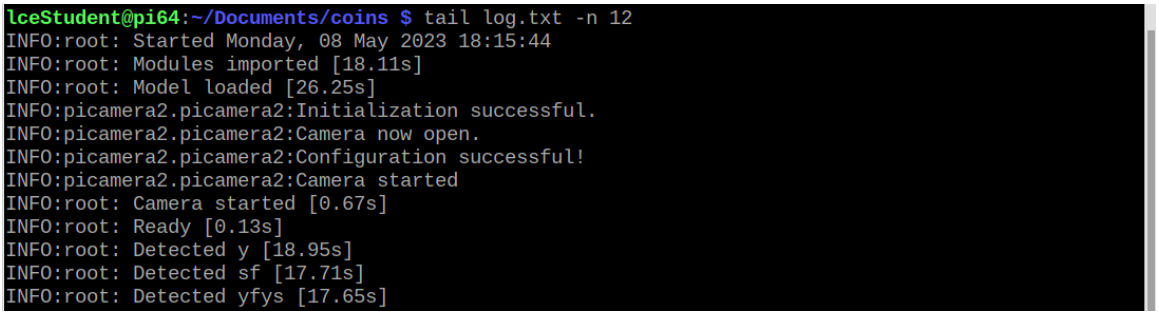

Figure 16. Contents of the log.txt file after running detector.py

6. Conclusion

Though functional, the final program is slow and inaccurate. The most time-consuming steps during execution are importing the modules required for recognition, loading the model, and running the recognition itself.

CONCLUSION AND PROSPECTS

In the course of this end-of-year project, we set out to design and prototype an embedded optical coin recognition system. The device would, among other potential applications, help the blind and the visually impaired to recognize and calculate a sum of money. It is meant to be compact, portable, easy to use and reasonably fast and accurate.

Firstly, we designed the system and determined a few key specifications. Although we had initially planned to use an ESP32-CAM card as the development platform, we settled on using the Raspberry Pi Model 3B Plus due to availability issues. Luckily however, the second device is more powerful and easier to work with. Along with the main board, we would make use of the Raspberry Pi Camera and a number of components and peripherals, notably a button for input detection and a speaker for audio output.

Next, we considered a few options in terms of image recognition software. We chose OpenCV and Detecto, mainly for their supposed simplicity, ease of use, and uniqueness. In practice however, it proved quite difficult to get Detecto running on the Raspberry Pi. Additionally, we needed a TTS software for output, specifically eSpeak. We also made use of some preinstalled packages including picamera2 and gpiozero.

Finally, we put together a prototype running a custom recognition model. Though operational, the obtained product is too slow and inaccurate for real world use.

For lack of time, resources, and knowledge in the fields of computer vision and machine learning, we left several avenues unexplored in our approach to this project.

Principally, it would be interesting to see how well a TensorFlow or TF Lite model would perform compared to our choice of Detecto. It is highly likely that using TF Lite would have been the optimal choice for our use case as it specifically meant for use on mobile and embedded devices [26].

Further, despite its lesser capabilities, we imagine that it would be possible to conceive a similar system based on the ESP32-CAM board. Such a system would likely prove more energy-efficient and portable, given that it wouldn't need an OS to run.

This project has undoubtedly provided us with a great chance to delve into the areas of embedded technology and machine learning, two of the most promising fields of our time. It has also helped us in improving many of our skills including programming, research, documentation, communication, etc.

REFERENCES

[1] AFB (American Foundation for the Blind), "Identifying US Currency," [Online]. Available: https://www.afb.org/blindness-and-low-vision/using-technology/accessible-identification-systems-people-who-are-blind-0.

[2] R. Bulanova, "5 Coin Identification Apps by Picture for Android & iOS | Free apps for Android and iOS," Freeappsforme, 01 06 2022. [Online]. Available: https://freeappsforme.com/coin-identification-apps-by-picture/.

[3] A. Bi, "detecto.ipynb - Colaboratory," [Online]. Available: https://colab.research.google.com/drive/1ISaTV5F-7b4i2QqtjTa7ToDPQ2k8qEe0.

[4] T. Rice, "Real Time Inference on Raspberry Pi 4 (30 fps!) — PyTorch Tutorials 2.0.0+cu117 documentation," The PyTorch Foundation, [Online]. Available: https://pytorch.org/tutorials/intermediate/realtime_rpi.html.

[5] George, "Carbon Stone: Connecting to Pi from Laptop's Ethernet Port," 01 02 2014. [Online]. Available: https://carbonstone.blogspot.com/2014/02/connecting-to-pi-from-laptops-ethernet.html.

[6] Raspberry Pi Ltd, "Raspberry Pi Documentation - Camera software," Raspberry Pi Ltd, [Online]. Available: https://www.raspberrypi.com/documentation/computers/camera_software.html.

[7] Raspberry Pi Ltd, "2.6.1. OpenCV," in *The Picamera2 Library*, Raspberry Pi Ltd, 2023, p. 8.

[8] y. z, "PyTorch 2.0 in Raspberry Pi 4 has error: Illegal instruction - Stack Overflow," 21 04 2023. [Online]. Available: https://stackoverflow.com/q/76073527.

[9] A. Bi, "detecto · PyPI," Python Software Foundation, 02 02 2022. [Online]. Available: https://pypi.org/project/detecto/.

[10] Google LLC, "Google Colab," Google LLC, [Online]. Available: https://research.google.com/colaboratory/faq.html.

[11] "colab system specs.ipynb - Colaboratory," [Online]. Available: https://colab.research.google.com/drive/151805XTDg--dgHb3-AXJCpnWaqRhop_2?usp=sharing.

[12] The PyTorch Foundation, "Start Locally | PyTorch," The PyTorch Foundation, [Online]. Available: https://pytorch.org/get-started/locally/#windows-package-manager.

[13] A. Bi, "Quickstart — Detecto 1.0 documentation," [Online]. Available: https://detecto.readthedocs.io/en/latest/usage/quickstart.html.

[14] A. Veysov, "RuntimeError: Unknown qengine · Issue #142 · snakers4/silero-models," 12 14 2022. [Online]. Available: https://github.com/snakers4/silero-models/issues/142#issuecomment-1096984868.

[15] tzutalin, "labelImg · PyPI," Python Software Foundation, 11 08 2021. [Online]. Available: https://pypi.org/project/labelImg/. [Accessed 09 05 2023].

[16] "wkentaro/labelme: Image Polygonal Annotation with Python (polygon, rectangle, circle, line, point and image-level flag annotation).," [Online]. Available: https://github.com/wkentaro/labelme.

[17] CVAT.ai Corporation., "opencv/cvat: Annotate better with CVAT, the industry-leading data engine for machine learning. Used and trusted by teams at any scale, for data of any scale.," [Online]. Available: https://github.com/opencv/cvat.

[18] The PyTorch Foundation, "Transforming and augmenting images — Torchvision main documentation," The PyTorch Foundation, [Online]. Available: https://pytorch.org/vision/master/transforms.html.

[19] B. Nuttall, "gpiozero — GPIO Zero 1.6.2 Documentation," [Online]. Available: https://gpiozero.readthedocs.io/en/stable/.

[20] B. Nuttall, "2. Basic Recipes — GPIO Zero 1.6.2 Documentation," [Online]. Available: https://gpiozero.readthedocs.io/en/stable/recipes.html#button.

[21] Coqui, "coqui-ai/TTS: 🐸💬 - a deep learning toolkit for Text-to-Speech, battle-tested in research and production," [Online]. Available: https://github.com/coqui-ai/TTS.

[22] N. Bhat, "nateshmbhat/pyttsx3: Offline Text To Speech synthesis for python," GitHub, Inc., [Online]. Available: https://github.com/nateshmbhat/pyttsx3.

[23] P. N. Durette, "pndurette/gTTS: Python library and CLI tool to interface with Google Translate's text-to-speech API," [Online]. Available: https://github.com/pndurette/gTTS.

[24] "espeak-ng/espeak-ng: eSpeak NG is an open source speech synthesizer that supports more than hundred languages and accents.," [Online]. Available: https://github.com/espeak-ng/espeak-ng.

[25] "espeak-ng/mbrola.md at master · espeak-ng/espeak-ng · GitHub," [Online]. Available: https://github.com/espeak-ng/espeak-ng/blob/master/docs/mbrola.md.

[26] Google Brain Team, "TensorFlow Lite | ML for Mobile and Edge Devices," [Online]. Available: https://www.tensorflow.org/lite.

ABSTRACT

This end-of-year project aimed to design and prototype an embedded optical coin recognition system to assist the blind and visually impaired in recognizing and calculating the value of a sum of coins.

The Raspberry Pi Model 3B Plus was chosen as the development platform along with OpenCV and Detecto for image recognition and eSpeak for audio output. Despite some difficulties with software implementation, a prototype was successfully built. However, further improvements are needed to increase speed and accuracy for real-world use.

This project has demonstrated the potential for embedded systems to improve the lives of individuals with disabilities, and may serve as a foundation for future development in this area.

RESUME

Ce projet de fin d'année visait à concevoir et à prototyper un système embarqué de reconnaissance optique de pièces de monnaie afin d'aider les aveugles et les malvoyants à reconnaître et à calculer la valeur d'une somme de pièces.

Le Raspberry Pi Model 3B Plus a été choisi comme plateforme de développement, avec OpenCV et Detecto pour la reconnaissance d'images et eSpeak pour la sortie audio. Malgré quelques difficultés liées à la mise en œuvre du logiciel, un prototype a été construit avec succès. Cependant, d'autres améliorations sont nécessaires pour accroître la vitesse et la précision en vue d'une utilisation pratique dans le monde réel.

Ce projet a démontré le potentiel des systèmes embarqués pour améliorer la vie des personnes handicapées et peut servir de base à de futurs développements dans ce domaine.

ملخص

يهدف مشروع نهاية العام هذا إلى تصميم وإنشاء نموذج أولي لنظام التعرف البصري على العملات المعدنية لمساعدة المكفوفين وضعاف البصر في التعرف على قيمة مجموع العملات وحسابها.

تم اختيار Raspberry Pi Model 3B Plus كمنصة تطوير مع OpenCV و Detecto للتعرف على الصور و eSpeak لإخراج الصوت .على الرغم من بعض الصعوبات في تنفيذ البرنامج ، تم بناء نموذج أولي بنجاح .ومع ذلك ، هناك حاجة إلى مزيد من التحسينات لزيادة السرعة والدقة للاستخدام في العالم الحقيقي.

أظهر هذا المشروع إمكانات الأنظمة المدمجة لتحسين حياة الأفراد ذوي الإعاقة ، ويمكن أن يكون بمثابة أساس للتنمية المستقبلية في هذا المجال.