UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Design and Development of an Autonomous Node for Fauna Monitoring in Areas of Interest

Diploma Thesis

Petros Grammatikou

Supervisor: Christos Antonopoulos

September 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Design and Development of an Autonomous Node for Fauna Monitoring in Areas of Interest

Diploma Thesis

Petros Grammatikou

Supervisor: Christos Antonopoulos

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση και Υλοποίηση Αυτόνομου Κόμβου για την Επιτήρηση Πανίδας σε Περιοχές Ενδιαφέροντος

Διπλωματική Εργασία

Πέτρος Γραμματικού

Επιβλέπων: Χρήστος Αντωνόπουλος

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering University of Thessaly

Member **Georgios Thanos**
Professor, Department of Electrical and Computer Engineering University of Thessaly

Acknowledgements

The completion of this thesis would not have been possible without the support and guidance of my supervisor, Professor Christos Antonopoulos. I am profoundly grateful for his constant availability, his insightful feedback, and the expertise he generously provided for every challenge I faced throughout this research. His mentorship was instrumental in shaping not only this work, but the engineer I have become.

My sincere thanks also go to Professors Spyros Lalis and Fotios Plesas for providing the hardware and infrastructure to develop and test the LoRa Networking and Custom UPS subsystems, which provided invaluable insights during the development phase, even though they were not integrated into the final system.

I would also like to extend my special thanks to Eleni Rigaki, owner of TED3D laboratory in Volos, for sponsoring the 3D printing of the prototype case of the device.

I am also grateful to Professors Nikolaos Bellas and Georgios Thanos for participating in the examination committee of my thesis.

Last but not least, I wish to thank those who have supported me throughout my entire academic journey over these five wonderful years, and will always support me with everything I pursue in life, my family.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Petros Grammatikou

Diploma Thesis

Design and Development of an Autonomous Node for Fauna Monitoring in Areas of Interest

Petros Grammatikou

Abstract

In this Thesis, we present the development of an autonomous solar-powered camera trap for wildlife monitoring. Unlike conventional commercial traps that only capture and store or transfer raw data, this device incorporates artificial intelligence models to perform image recognition and extract information from photographic data without the need for human intervention. The architecture is edge-based, with all computation carried out locally by the device itself. With this approach, no large data transfers to a server node are required to extract information from the images. The device effectively becomes the server, needing only to share the final recognition result, which is no more than a single word. The system is built around a Raspberry Pi 5 System-on-Chip, which manages peripheral sensors such as a PIR (passive infrared) motion sensor and a Night Vision Camera, and executes a custom-trained YOLOv8 model for wildlife detection. Power is supplied by lithium-ion batteries charged through a solar panel and managed by a dedicated solar power management PCB (Printed Cirquit Board). The resulting prototype is a low-cost device that can replace and enhance the functionality of traditional camera traps and was evaluated in the wild under real-life conditions in the foothills of the Paiko mountain.

Keywords:

Wildlife Monitoring, Camera Trap, IoT, Embedded Systems, Edge Computing, Edge AI, Raspberry Pi 5, PIR Sensor, YOLOv8, Solar Power, Real-Time Image Capture on Raspberry Pi 5

Διπλωματική Εργασία

Σχεδίαση και Υλοποίηση Αυτόνομου Κόμβου για την Επιτήρηση Πανίδας σε Περιοχές Ενδιαφέροντος

Πέτρος Γραμματικού

Περίληψη

Στην παρούσα διπλωματική εργασία παρουσιάζεται η ανάπτυξη μιας αυτόνομης, ηλιακά τροφοδοτούμενης κάμερας πεδίου για την παρακολούθηση της άγριας πανίδας. Σε αντίθεση με τις συμβατικές εμπορικές κάμερες πεδίου, οι οποίες απλώς καταγράφουν και αποθηκεύουν ή μεταφέρουν ακατέργαστα δεδομένα, η συγκεκριμένη συσκευή ενσωματώνει μοντέλα τεχνητής νοημοσύνης ώστε να εκτελεί αναγνώριση εικόνας και να εξάγει πληροφορία από τα φωτογραφικά δεδομένα χωρίς την ανάγκη ανθρώπινης παρέμβασης. Η αρχιτεκτονική της είναι βασισμένη στο άκρο του δικτύου με όλη την υπολογιστική επεξεργασία να πραγματοποιείται τοπικά από την ίδια τη συσκευή. Με αυτήν την προσέγγιση, δεν απαιτούνται μεγάλες μεταφορές δεδομένων προς έναν κεντρικό server για την εξαγωγή πληροφοριών από τις εικόνες. Η συσκευή ουσιαστικά μετατρέπεται στον ίδιο τον server, και χρειάζεται να μοιραστεί μόνο το τελικό αποτέλεσμα της αναγνώρισης, το οποίο δεν είναι περισσότερο από μία μόλις λέξη. Το σύστημα βασίζεται στο Raspberry Pi 5 System-on-Chip, το οποίο διαχειρίζεται περιφερειακούς αισθητήρες, όπως έναν αισθητήρα παθητικής υπερύθρης ακτινοβολίας (PIR) και μία κάμερα νυχτερινής όρασης, και εκτελεί ένα ειδικά εκπαιδευμένο μοντέλο YOLOv8 για ανίχνευση άγριων ζώων. Η τροφοδοσία γίνεται από μπαταρίες ιόντων λιθίου που φορτίζονται μέσω ενός ηλιακού πάνελ και ελέγχονται από μία ειδική πλακέτα διαχείρισης ηλιακής ισχύος. Το τελικό πρωτότυπο είναι μια συσκευή χαμηλού κόστους, ικανή να αντικαταστήσει και να βελτιώσει τη λειτουργικότητα των παραδοσιακών καμερών πεδίου, και αξιολογήθηκε σε πραγματικές συνθήκες πεδίου, στούς πρόποδες του βούνου Πάικου.

Λέξεις-κλειδιά:

Παρακολούθηση Άγριας Ζωής, Κάμερα Πεδίου, Διαδίκτυο των Πραγμάτων, Ενσωματωμένα Συστήματα, Υπολογιστική στην Άκρη του Δικτύου, Τεχνητή Νοημοσύνη στην Άκρη

του Δικτύου, Raspberry Pi 5, Αισθητήρας PIR, YOLOv8, Ηλιακή Ενέργεια, Λήψη Εικόνων σε Πραγματικό Χρόνο στο Raspberry Pi 5

Table of contents

List of figures

List of tables

Abbreviations

AI	Artificial Intelligence
SoC	System on Chip
ARM	Advanced RISC Machines
FLOPS	Floating Operarions Per Second
GB	Giga Byte
RAM	Ramdnom Access Memory
RP5	Raspberry Pi 5
V	Volt
A	Ampere
W	Watt
IR	Infrared
LoRa	Long Range
PIR	Passive Infrared
UPS	Uninterupted Power Supply
PCB	Printed Cirquit Board
MPPT	Maximum Power Point Tracking)
DC	Direct Current
LED	Light Emitting Diode
IoT	Internet of Things
Li-ion	Lithium Ion
HC-SR501	Passive Infrared Sensor Model HC-SR501
MP	Mega Pixel
RTC	Real Time Clock
PETG	Polyethylene Terephthalate Glycol
YOLO	You Only Look Once

R-CNN	Recurrent-Convolutional Neural Network
CNN	Convolutional Neural Network
COCO	Common Objects in Context
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
GDDR6	Graphics Double Data Rate Type 6
FP32	Floating Point of size 32 bits
FP16	Floating Point of size 16 bits
INT8	Integer of size 8 bits
AdamW	Adaptive Moment Estimation with Decoupled Weight Decay Regularization
VCC	Voltage Common Collector
GND	Ground
OUT	Output
BC547	transistors name
OS	Operating System
CPU	Central Processing Unit
ROM	Read Only Memory
RISC	Reduced Instructions Set Computer
SD	Secure Digital
FAT32	File Allocation Table 32-bit
BIOS	Basic Input/Output System
UEFI	Unified Extensible Firmware Interface
PC	Personal Computer

Chapter 1

Introduction

Artificial Intelligence is undoubtedly the field driving the technological research and innovation of this generation. Its impact is already noticeable in our everyday lives, and the expectations and speculations about the future of what Big Data might bring to the world are immense. While AI models and datacenters are getting larger and more powerful to meet the increasing demand of Artificial Intelligence services globally, a different need has emerged: to use AI on the edge, in the place where the data are born.

Formed from the convergence of Edge Computing and AI, Edge AI allows data to be processed directly at its source, without any need to connect to the cloud. This approach promises secure, low-latency, real-time AI applications that the world has never seen before and that were previously considered impossible.

1.1 Thesis Objective

This Thesis presents the development process of an Edge AI device: an AI-enhanced camera trap for wildlife monitoring. The functionality of the device differs from the traditional camera traps used by researchers and nature enthusiasts. Instead of simply capturing and storing, or transmitting raw photographic data, it performs image recognition locally to extract the species information depicted in the image without the need for external processing or human intervention.

The applications function can be briefly described as: A motion sensor triggers the system to wake from a powered-off state, capture an image, and immediately perform on-device image recognition using a custom trained YOLOv8 model. If a species is detected, the system

transmits the class name along with the date and time via a LoRa message, while also logging data such as uptime, the number of triggers, and captured images. To conserve energy, the system automatically powers off when no motion is detected for a defined period of time, relying on solar panels and batteries for autonomous operation in remote areas without access to grid electricity.

1.1.1 Contributions

The contributions of this Thesis are summarized as follows:

1. A custom hardware design for the specific project
2. Application (software) development
3. Evaluating the performance of various AI models and selecting the fittest
4. Creating a data set and custom training a YOLO model to be deployed
5. Optimizing the operating systems boot process for faster capture when it is off
6. Deploying the device on-field and evaluating its operation

1.2 Thesis Structure

The rest of the Thesis is structured as follows: Chapter 2 discusses Hardware Design. Chapter 3 is about optimizing the environment used to host the application. Chapter 4 shows the process of customizing a YOLO model to the application's needs. Chapter 5 discusses the evaluation of the system by testing it in a real-life scenario in the wild and suggests future expansions and alternatives that emerged along the way that were not deployed.

Chapter 2

Hardware Design

The basic principle of camera traps is that they capture images only when movement is perceived in the area. In our system, movement is monitored by a PIR sensor, and when present, a motion signal is sent from the motion sensor to the Raspberry Pi 5, which uses the camera to capture an image. The contents of the image are then recognized locally on the RP5. In order to save power, the device can shut itself down when no movement is perceived for a determined period of time and is able to boot again when another motion signal is received. The system is powered by a 30W solar panel integrated with a solar power manager and a mini Raspberry Pi 5 UPS, with cumulative battery life of 50 Ah at 5V. The systems general architecture and the connections between it's components are shown below in Figure 2.1.

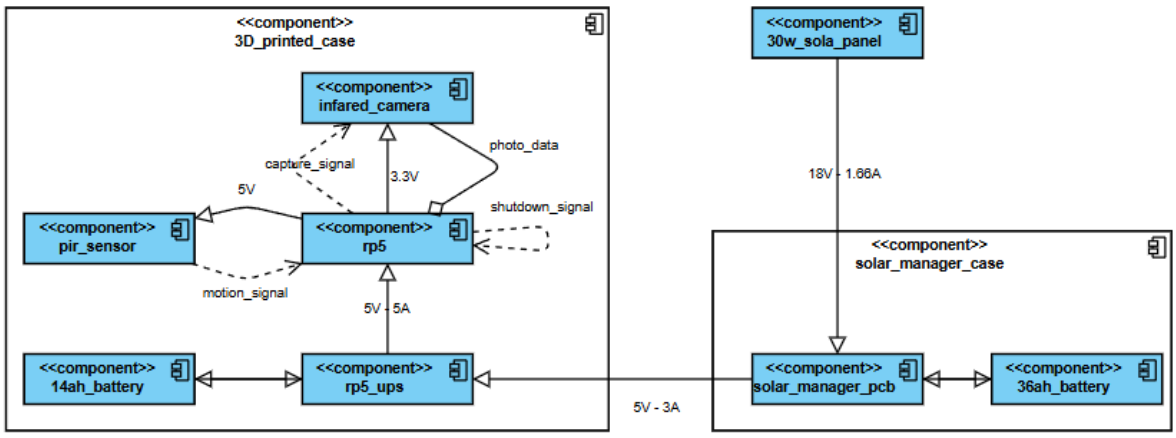


Figure 2.1: System Architecture and Connections Diagram

Straight lines with arrow ends represent power buses and the current's direction, while dashed lines represent logic signal lines. The straight line with a rhombus end represents a data transfer line.

2.0.1 Raspberry Pi 5

The starting point of designing such a system is to figure out the computing hardware capable of carrying out all the processing along with managing the peripherals needed. Unlike monitoring movement and capturing photos, Edge AI, and especially image recognition, requires significant computing power, so a microcontroller is not an option for such a task. What we needed was a single-board computer to process image data and recognize objects in real time at the Edge. Raspberry Pi 5 and its hardware components are shown in Figure 2.2

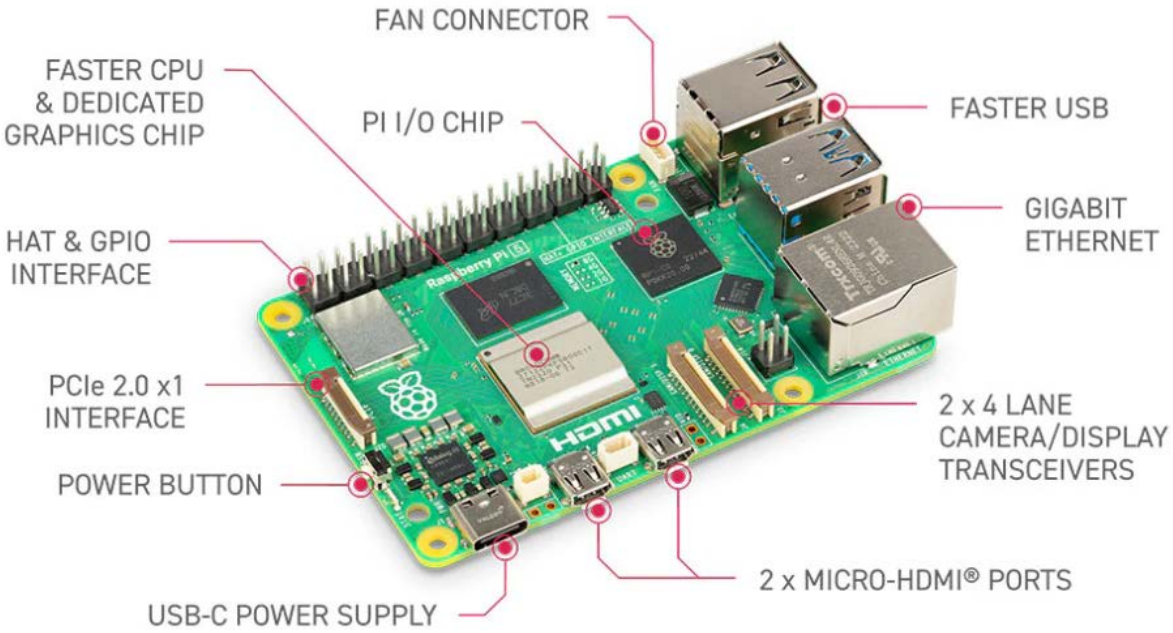


Figure 2.2: Raspberry Pi 5 [1]

The Raspberry Pi 5, the latest and most powerful SoC in the Raspberry series, features the Broadcom BCM2712 quad-core ARM Cortex-A76 processor at 2.4GHz, capable of reaching up to 10 GFLOPS per core. This makes it up to three times faster than the previous generation, and the 4GB RAM model was considered a very good fit for the needs of such a project. However, heavy computation comes with high energy consumption. Thus, the device could not rely solely on battery power to operate for extended periods, and energy-harvesting from the environment was considered mandatory. To resolve this issue, we used a solar panel and a solar power management module.

Power Consumption Calculations

The RP5's nominal consumption from the manufacturer is stated to be 5 V×5 A = 25 W, the IR camera's is more that 1 W, and the LoRa transceiver is also expected to be at 2 W, making the theoretical power consumption approximately 29 W while the PIR sensor's consumption is so low that it is not measured in the calculations. In practice, the measured consumption of the entire system appeared to be almost all the time lower than 15 W, and the RP5 itself was rarely observed to consume a current of 3 A.

Although knowing the system's instantaneous consumption is crucial, daily power consumption cannot be easily predicted because it heavily relies on a factor that cannot be simulated or measured to be factored into the calculation, that is, animal activity. With the device optimized for energy consumption, by turning on only when a warm object movement is perceived by a PIR sensor, much randomness is introduced to the calculation. The randomness of animal activity, along with another uncertainty source that comes from the solar cells' performance, which heavily relies on the weather, makes modeling the device's consumption accurately almost impossible without real life measurements and data.

2.0.2 Solar Panel And Energy Management

Solar Panel

With all these considerations, we decided to make the final calculations for choosing the solar panel according to a worst-case scenario approach. The goal was to ensure that the energy production would be sufficient to power the device for a full day, even in harsh weather conditions. The simulation of the photovoltaic performance was made using real data from panels similar to the candidate ones to be bought, by reproducing their measured performance curve using a Gaussian bell formula, with the formula parameters modeling the real parameters of the panels and the weather conditions the measurements were made under. Our calculation methodology is similar to the one in [12].

The core function modeling the instantaneous power output $P(t)$ of the panel at time t is given by Equation 2.1:

$$P(t) = \eta \cdot P_{\text{max}} \cdot \exp\left(-\frac{(t - t_{\text{peak}})^2}{2\sigma^2}\right) \tag{2.1}$$

where:

- P_{max} is the panel's maximum rated power output (in Watts),

- t_{peak} is the hour of the day at which the sun is at its zenith (resulting in peak production),

- σ controls the width (or the spread) of the Gaussian curve, representing the duration of sunlight,

- η is a degradation or failure factor accounting for panel aging, dirt, and other inefficiencies.

The total daily energy yield E_{total} (in Watt-hours) can be calculated by integrating the instantaneous power output function $P(t)$ (Equation 2.1) over the period of daylight, from the start time t_0 to the end time t_1, as defined in Equation 2.2.

$$E_{\text{total}} = \int_{t_0}^{t_1} P(t)dt = \eta \cdot P_{\text{max}} \int_{t_0}^{t_1} \exp\left(-\frac{(t - t_{\text{peak}})^2}{2\sigma^2}\right) dt \tag{2.2}$$

The final energy output in Amp-hours (Ah), used for comparison with battery capacity, is obtained by dividing the result of Equation 2.2 by the nominal voltage of the system V_{sys} = 5 V.

$$\text{Energy}_{\text{Ah}} = \frac{E_{\text{total}}}{V_{\text{sys}}} \tag{2.3}$$

This simulation, implemented in Python, allowed for the comparison of daily energy production under various scenarios with the known energy consumption of the application. The most accurate simulation achieved simulates a completely sunny summer days production and curve. The produced curves are shown in Figure 2.3

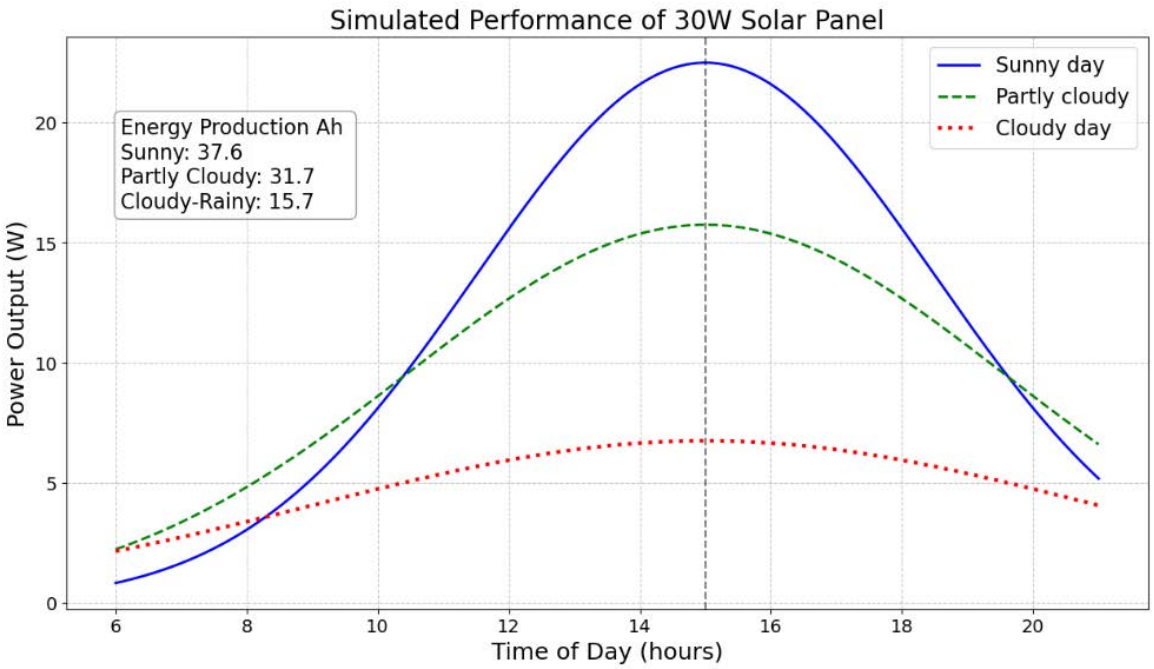


Figure 2.3: Solar Power Daily Production Simulation

These calculations indicated that a 30 W solar panel would be sufficient to cover the device's needs for a day's hours of full use, so the 30W 650x350mm Rolinger GD-1030 Monocrystalline [2] Solar Panel was picked. Because the cell consists of a single crystal, the electrons that generate an electric current have more space to move, making it more efficient than a polycrystalline one.

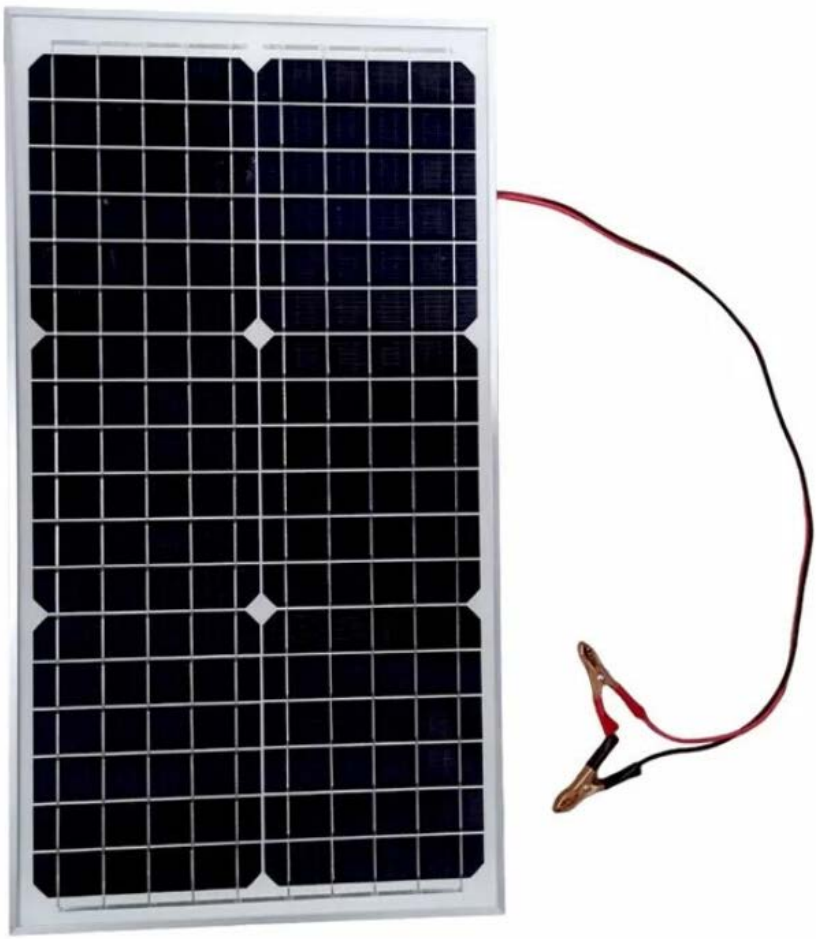


Figure 2.4: 30W 650x350mm Solar Panel [2]

2.0.3 Solar Power Management Module

To manage and be able to use the 18 V output of the solar panel to charge the batteries until full and power up the UPS simultaneously, we used Waveshare's Solar Power Manager Module (D) [13]. This PCB supports MPPT (Maximum Power Point Tracking) function, maximizing the efficiency of the solar panel, and a solar panel/Type-C power adapter for battery charging, which was used to power up the UPS. It supports all panels with nominal voltage between 6 V 24 V through self-adaptive input voltage via DC-002 jack or screw terminal, with input anti-reverse protection, multi-LED indicators for monitoring the status of the solar panel and batteries, and multi-protection circuits: over-charge/over-discharge/overheat/over-current. It is a stable and safe choice for constant energy flow.

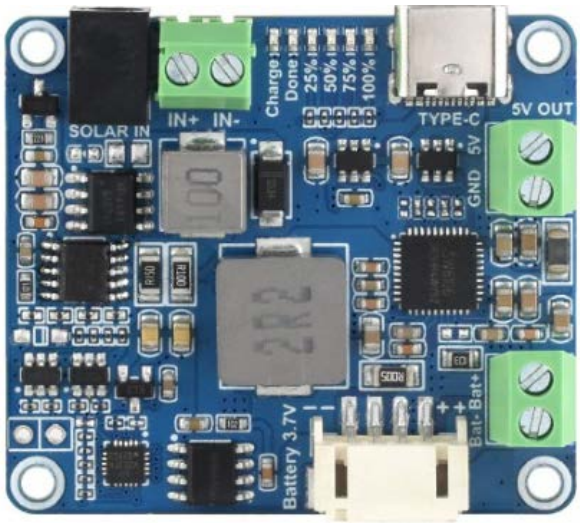


Figure 2.5: Waveshare's Solar Power Manager Module [3]

2.0.4 UPS

Implementing A UPS Functionality With A Power Bank

The term UPS stands for Uninterruptible Power Supply and its most usual case of use is to provide emergency power to a device when the main power source fails. In this case where the power source is the sun, power failures are a certainty, and to make an isolated off-grid compute module function, building or using a UPS is not a choice, but an absolute necessity. Modern power banks tend to have more and more UPS-like features. They offer constant voltage (most of the time) and are capable of pass-through charging. Despite these useful perks, the functionality they provide is not enough to be used as UPSes as they appear to have two major flaws.

The first of them originates from their smart design which in our case does not match the UPS functionality, and is the auto shutdown feature. Power banks, once charged, need to stay fully charged until they are used to charge a device. The current leakage on their own circuit board is enough to discharge a power bank to a noticeable extent for the user, so to solve this problem, a shutdown mechanism was introduced to trigger when tiny currents are detected. Although it is such a useful feature for charging devices, it can be a killer for edge IoT devices like ours that need to run on low power for extended periods of time with the tiny currents that the PIR Sensor draws.

This issue can be readily resolved by connecting a resistor in parallel with the device to simulate a load and maintain the power bank's active state. However, this approach has

the drawback of continuously discharging the power bank with a small current only to keep it operational. A more efficient alternative involves using specialized hardware, such as a Powerbank Keep Alive adapter. This compact electronic device requests a small current pulse at regular intervals (e.g. every 5 or 10 seconds), thereby maintaining the active status of the power bank with minimal energy consumption [14].

The second flaw is that at the moment when pass-through charging is activated or deactivated from an external source, the power supply is cut out for a short period of time, creating a step-down pulse. This power gap is enough to shut down and instantly reboot the system if it is on or simply boot it from sleep when it is off. The reason why the pass-through functionality is not working perfectly in a power bank is that there is no need for it to charge a device that is being kept up by its own battery. The solution to making it work perfectly is simply to connect a capacitor in parallel with our load to fill the voltage gap by discharging itself into the load at that tiny bit of time when no voltage from the power bank is present.

To determine the required capacitance, we connected in the device and the power bank in parallel with a capacitor bank totaling $3 \times 470\mu$F. When an external power source was connected to the power bank, an oscilloscope measurement revealed a voltage drop of $\Delta V = 2$V that occurred over a duration of $t = 140$ms before the power was restored by the power bank. A completely identical pulse with the one depicted by the oscilloscope measurement is shown in the Figure 2.6 below.

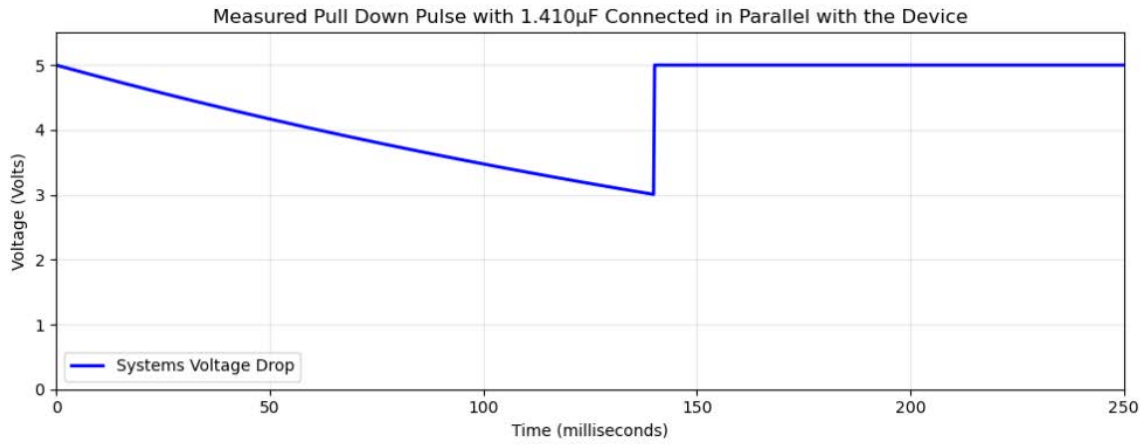


Figure 2.6: Step-down Pulse Created by Pass-through Charging the Power Bank

Using the mathematical expression describing the voltage decay across a capacitor:

$$V(t) = V_0 \cdot \exp\left(-\frac{t}{RC}\right),$$

where V_0 is the initial voltage and $V(t)$ is the voltage after time t, the system is modeled. The equivalent Thévenin resistance R that represents the RP5 was calculated as:

$$R = -\frac{t}{\ln\left(\frac{V}{V_0}\right) \cdot C},$$

which yielded a value of $R = 195\Omega$.

Finally, to compute the necessary capacitance for the desired maximum voltage drop, the formula was solved for C:

$$C = -\frac{t}{\ln\left(\frac{V}{V_0}\right) \cdot R}.$$

Given that the device is characterized by a Thévenin resistance of $R = 195\Omega$ and that the Raspberry Pi 5 can tolerate a maximum voltage drop of 0.2V, the minimum capacitance required to mitigate the power interruption was calculated to be $C = 17.6$mF. Of course, it is preferable to use a larger capacitor to ensure reliable and constant operation of the system without interruptions. Therefore, a 0.68F and a 1F capacitor were simulated, with both successfully flattening the voltage drop pulse and maintaining the supply voltage above 4.99V. The whole system arrangement is shown below in Figure 2.7.

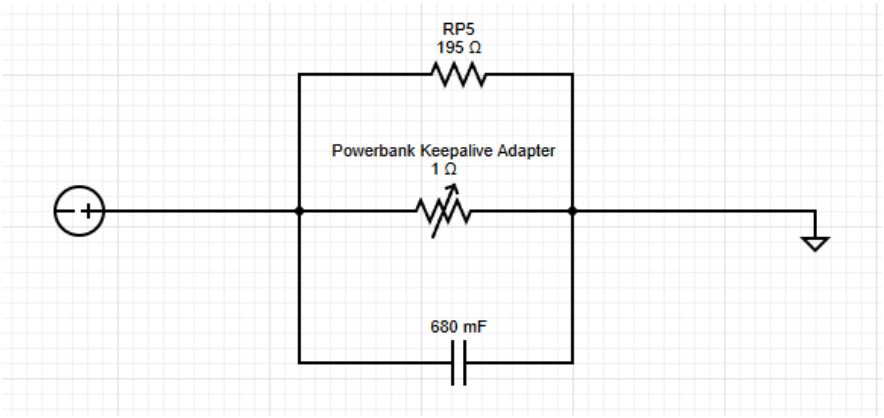


Figure 2.7: Powerbank UPS Components Diagram

Using a UPS PCB

The other more industrial option is buying a normal Raspberry Pi UPS PCB with the big drawback of the low power capacitance that 18650 batteries provide. The way to overcome this is by using the power bank's huge Li-ion 3.7V battery with the solar power manager by 'unsoldering' the power bank electronics from it. At this point, it is important to state the reasons that along with the solar power manager module, a mini UPS was also used, when

all the batteries could have been connected to the solar power manager, and the RP5 could be fed with power straight from the 5V output of the solar power manager, and this is the max current it can provide.

The device was perceived to draw more than 3A in rare but existent cases, and with the RP5 producing power warnings when discovering that 5V - 5A cannot be supplied to it by the power source, a 3A input 5A output UPS interconnecting the solar power manager module's output with the RP5's input, amplifying the max current from 3A to 5A and making everything work in their nominal max values if needed, was considered the best and most clean choice.

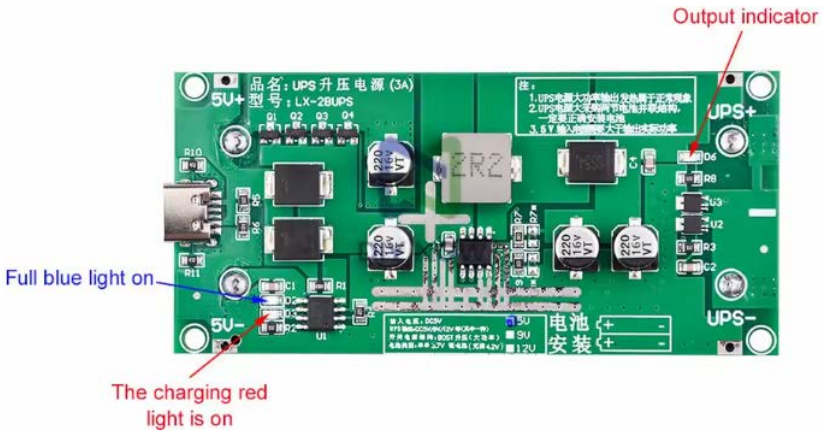


Figure 2.8: 5V - 3A In — 5V - 5A Out UPS [4]

2.0.5 PIR Sensor

An HC-SR501 PIR sensor [11] is a low-power motion detection device that reacts to changes in infrared radiation emitted by warm objects such as living beings. This specific module was chosen because it is very easy to find and use while providing the appropriate functionality. The PIR detection was used not only to capture a photo when the device is already on, but also to boot the system through a transistor that triggers the boot button when it is off. This provided the advantage of being able to shut down the device when movement was not seen for some minutes, saving enormous amounts of energy. The only drawback of booting and capturing when movement is seen is that it can be several seconds slower than just capturing a photo when the system is on, which is instantaneous. This topic will be discussed further later in this thesis in the scope of Operating System Manipulation for the best application performance Chapter 4.

Figure 2.9: HC-SR501 PIR Sensor [5]

2.0.6 Camera

The camera requirements were good resolution and the ability to capture at night. That is why an infrared camera was chosen, to provide the opportunity of capturing and recognizing wildlife at night, at the time when animals do move the most. More specifically, we used the Raspberry Pi Camera Module 5MP 70° - Night Vision camera [6].

Figure 2.10: Raspberry Pi Camera Module 5MP 70° - Night Vision[6]

2.0.7 RTC

Tracking time when there is no internet connection is very important for correct and accurate recordings. The system needs to maintain the time when it is shut down. Thankfully, the RP5 provides us with an embedded real-time clock and pins to use a common CR2032 3V battery with a specially designed battery box.

Figure 2.11: Raspberry Pi RTC Battery Box [7]

2.0.8 Packaging

The device packaging needed to be both water-and dust-proof because in the wild, a plethora of different environmental conditions such as rain will surely be faced. We designed a new case from scratch, to ensure that we would be able to fit all the components in a way that they are protected and functional, with all the right angles and mounting points for the PIR sensor and the camera. The packaging was designed using Rhino 8 CAD software [15] and the printing was kindly sponsored by TED3D [16] laboratory in Volos, where by the package was implemented using the Craftbot 2 3D Printer [17] and PETG (Polyethylene Terephthalate Glycol) filament material. The rendered design is shown in Figure 2.12

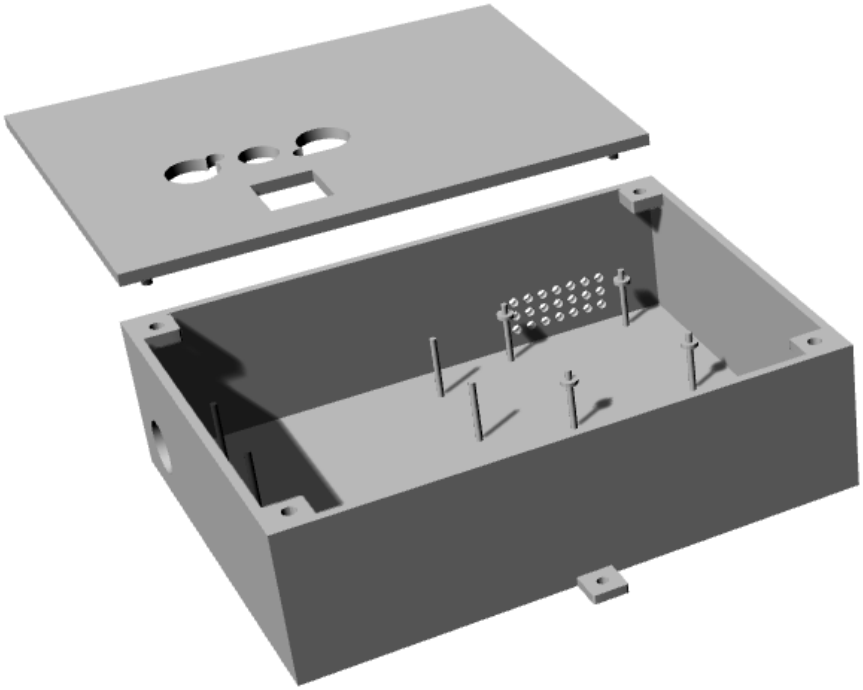


Figure 2.12: Case Rendered in Rhino8 CAD Software

Components like the RP5, the infrared camera and the solar power manager module produce a lot of heat and, when enclosed, especially in the hot summer months where no heat dissipation is provided by the external environment, may cause overheating of the RP5.

One potential solution would be to design the enclosure itself as a heat sink, ensuring that the heat-producing components maintain direct contact with it. However, this option was not feasible for us, as molding a metal enclosure would significantly increase both complexity and cost. Instead, two alternatives were considered: installing a fan module to actively cool the device or separating the components into two distinct enclosures. The latter approach was deemed more efficient, as it avoids the energy overhead of continuously running a fan, so the power electronics and the batteries which produce the most heat as they function throughout the whole day were put in a separate box. The infrared camera was also intentionally not mounted right above the RP5 due to heat interference reasons.

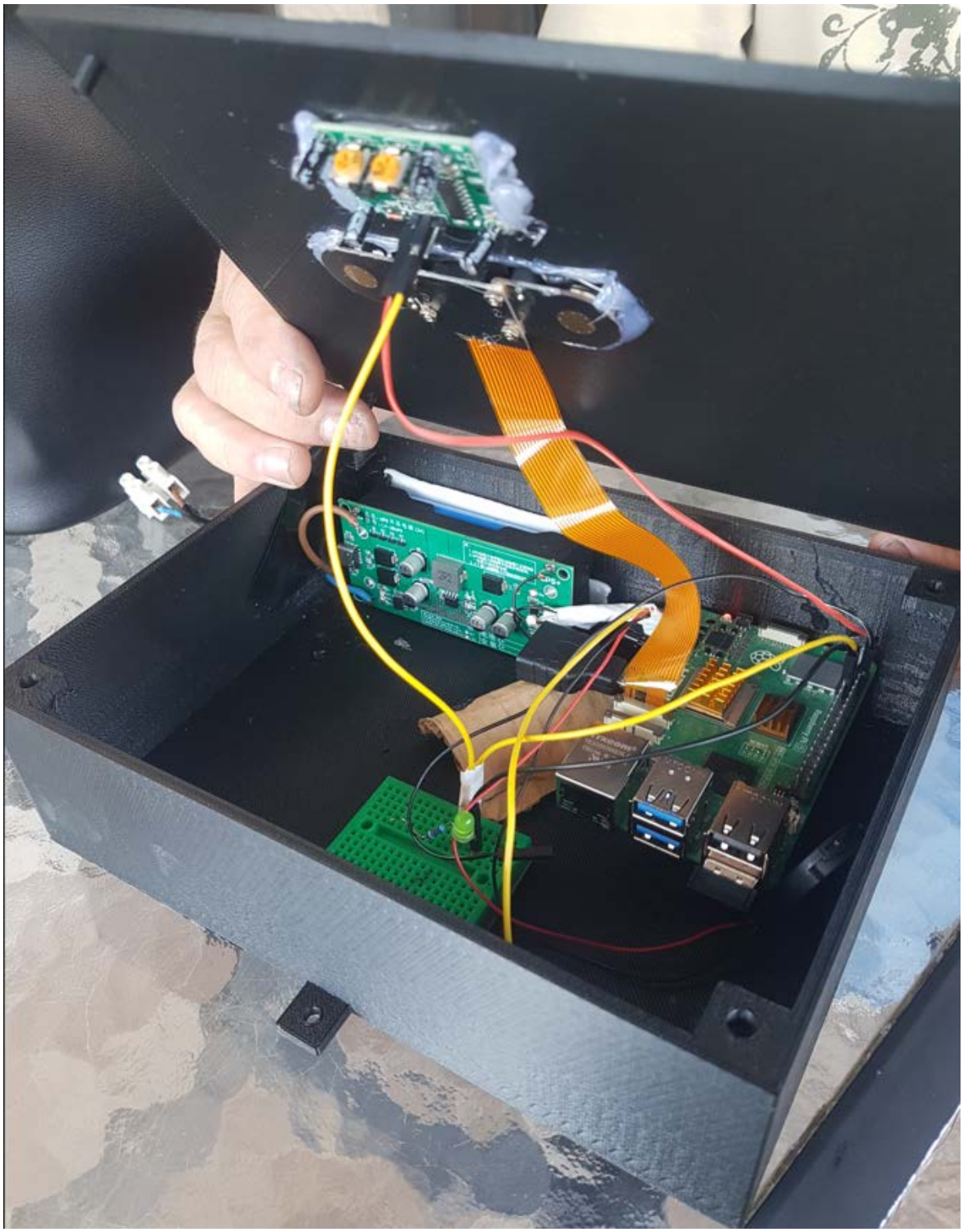


Figure 2.13: RP5 Case Components Arrangement

Figure 2.14: All System Components First Test

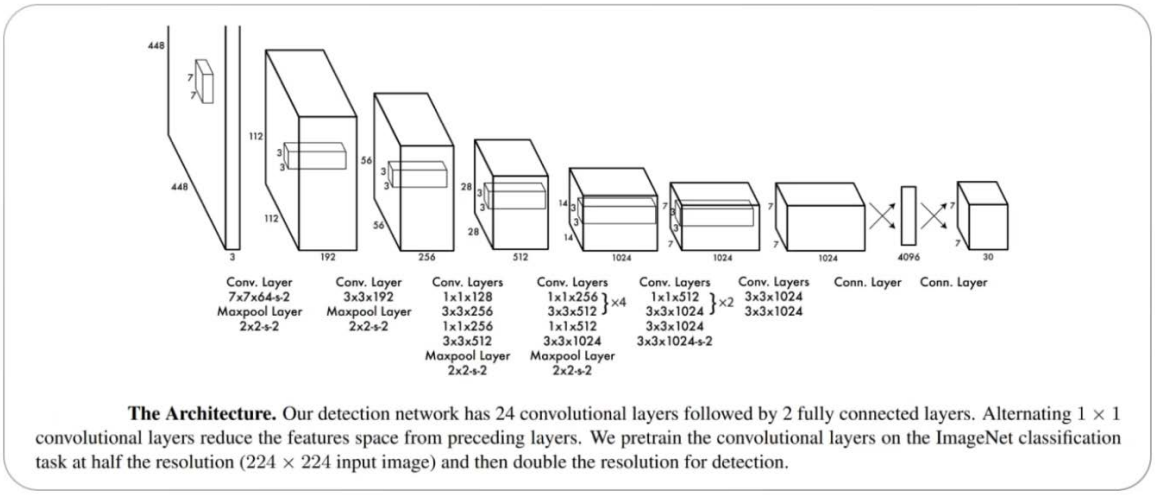
Chapter 3

AI Image Recognition with YOLO

3.1 YOLO Model Selection

3.1.1 YOLO General Architecture And Principles

The first YOLO (You Only Look Once) model [18] for object detection was introduced in 2015 by Joseph Redmon and Ali Farhadi at the University of Washington. Unlike other traditional image recognition architectures of the R-CNN family, by using a single Deep Convolutional Neural Network shown in Figure 3.1, it had the groundbreaking feature of using a one-stage process (named exactly by this feature) for region proposal and classification instead of the traditional two-stage pipeline.

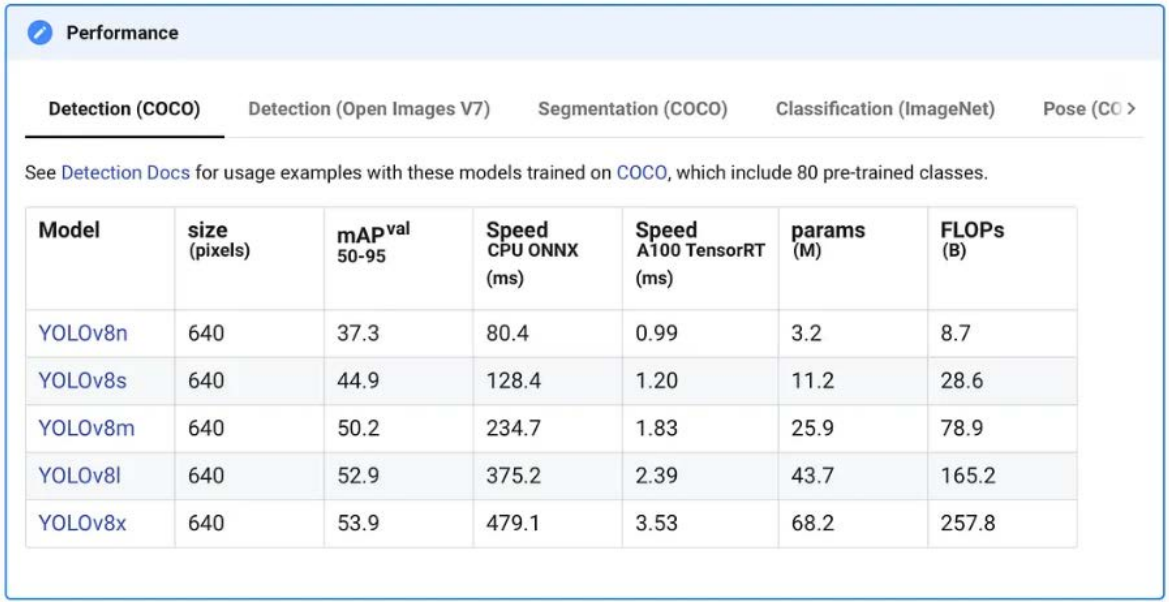


The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

Figure 3.1: Initial YOLO Neural Network Architecture [8]

Although single-shot object detection generally turned out to be less accurate than two-

shot, the computational power needed is far less, making a vast variety of real-time image recognition applications feasible. That is the main reason why a model of the YOLO family was chosen. Figure 3.2 presents the performance of all models of the YOLOv8 Family tested on COCO (Common Objects in Context) dataset.



Performance

Detection (COCO) | Detection (Open Images V7) | Segmentation (COCO) | Classification (ImageNet) | Pose (CO >

See Detection Docs for usage examples with these models trained on COCO, which include 80 pre-trained classes.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figure 3.2: YOLOv8 Family of Models Performance on COCO Dataset [9]

3.1.2 YOLOv8 nano

Over the years of development, along with the complexity of the architecture, both the accuracy and the speed of the YOLO models increased significantly, and many variations of each model with tweaked parameters and sizes, sacrificing accuracy for speed and vice versa were introduced. One of the most successful and widely used is the state-of-the-art model YOLOv8[19].

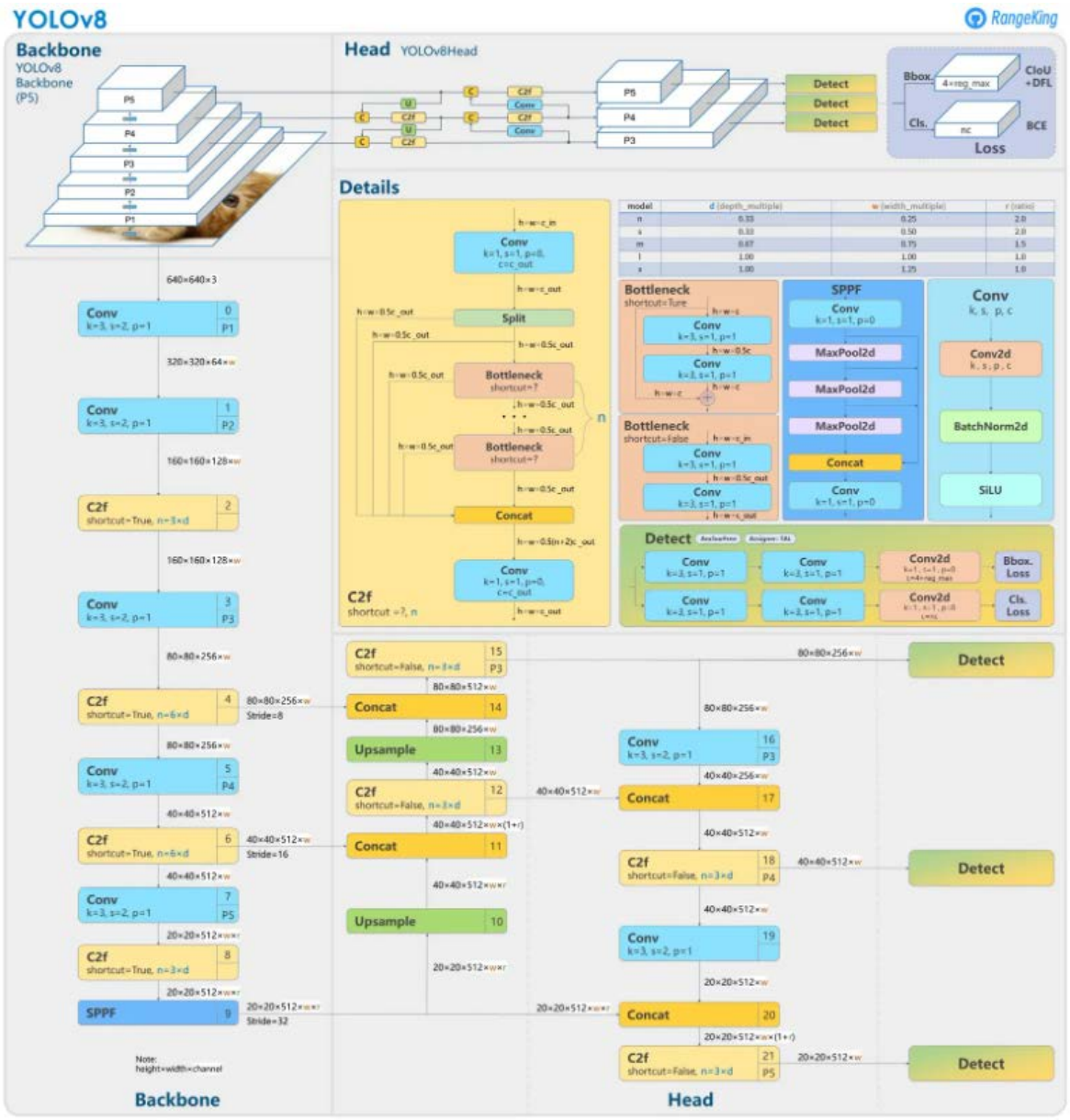


Figure 3.3: YOLOv8 Architecture made by RangeKing on Github [10]

The YOLOv8 nano is a stripped-down version of the huge and complex original v8 model optimized for the tiny computational power of edge devices. With only 3.2 million parameters, the inference time when running on the RP5 is approximately 0.4 seconds on images sized 640x480 without an accelerator or any additional computational hardware, it was considered the best among its big brother candidates to be used with an edge device to solve a time-constrained problem. [20]. The measurements of each model's inference speed on RP5 are shown in Table 3.1.

Table 3.1: YOLOv8 Inference Times Measured On Raspberry Pi 5

YOLOv8 Model	640x480 (s)	640x640 (s)
nano (n)	0.4	0.5
small (s)	1.0	1.5
medium (m)	2.4	3.6
large (l)	4.9	7.6
x-large (x)	7.5	10.6

3.1.3 Data Gathering

Although training and fine-tuning an image recognition model is very complex and often comes down to trial-and-error methods, when many surprising results may go against our intuition and expectations, there are some general rules to be used as a guide for deciding certain things. The number of images in the dataset needed to train a reliable model is of extreme importance.The greater the variability in the images that the model must handle, the more training data it requires. This relationship is not linear. A small increase in potential variability might demand a significantly larger dataset for the model to maintain its reliability. The variability between the inference data a model might hold is decided by the total variability that might occur due to aspects like image size, the changes in environment, the light and colors of the images, the angle, the possible different objects (detectable or not) that might appear in an image, and the size of the objects depicted. For example, a model trained to recognize objects passing on a conveyor belt in a factory will face extremely low variability between the inference images, as all the environmental conditions and the objects that might be seen are predefined, and an extremely small number of images might be enough to reach maximum performance. Whereas a model used to recognize objects from pictures taken from a drone might need an enormously large amount of images for training. Although our model will not face the extreme variability of a drone model, it is surely closer to it than to the conveyor belt one. So, the number of photos needed for training must be large. Predicting is hard, so for better chances of success it was made almost as big as possible [21].

Data were collected primarily from online sources of datasets such as Kaggle[22] and Roboflow[23] Universe. For each photo, the bounding box and the name of the class depicted need to be specified. This can be done by writing directly in the metadata files or with more

user-friendly GUI tools such as the environment provided by Roboflow, which is the easiest and fastest way to do it. The final version of the dataset consists of 8470 images and contains six different classes. More statistics concerning the dataset are shown in Figure 3.4.

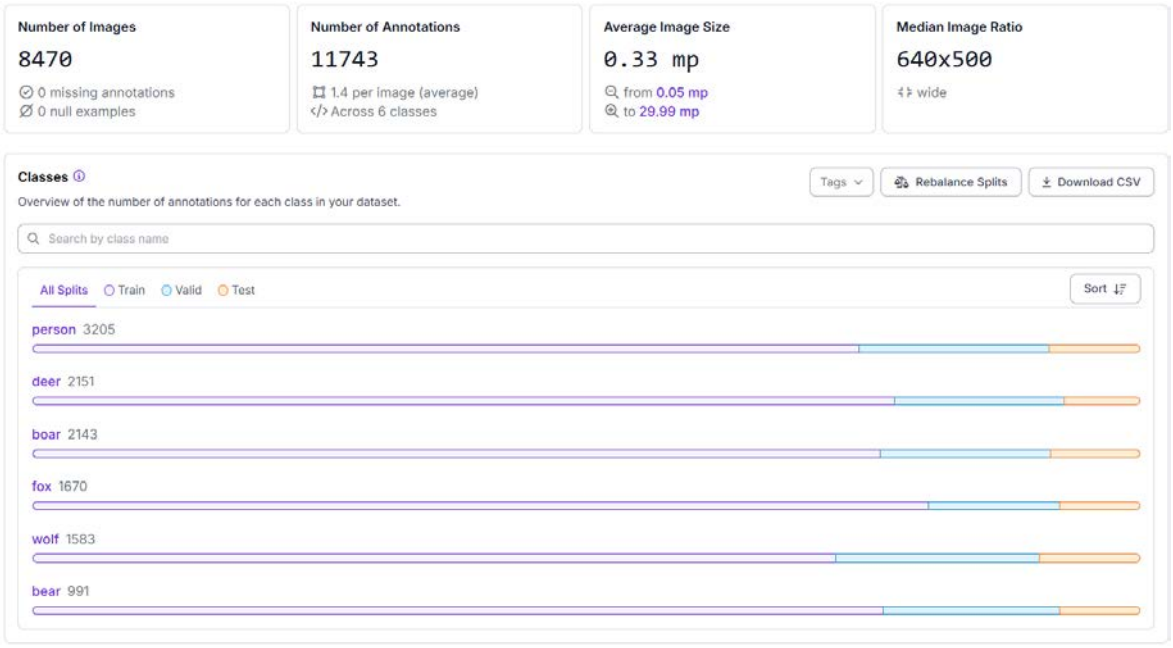


Figure 3.4: Dataset Statistics

3.1.4 Image Size Dilemmas

Many parameters need to be considered when running inference with a YOLO model. The first question to be answered is image size. YOLOv8 models run the inference on maximum sized images of 640×640. This means that the original larger images taken by high-quality cameras are resized before being fed to the model. But most cameras are not square, so what happens with the aspect ratio of the camera? YOLO adapts to that, too, by resizing its image to the aspect ratio of the camera instead of making it square and introducing distortion to the image. For example, in our application, for a 2592×1944 pixel photo with an aspect ratio of 4:3, the YOLO model will run its inference on a 640×480 photo to match the aspect ratio of the original image. Here is a trade-off: either the model itself will reduce the photos' quality (like in the above paradigm) or the camera's resolution must be changed to be square and let the model use max quality for inference but inevitably reduce the physical space of detection being seen by the camera, resulting in more possible misses of animals that might reach the area and will not be captured by the device. An easy answer would be to apply image processing tricks before running the inference by taking one wide picture, converting it into two

square 1944×1944 images, and applying inference on both. This way both the captured area as well as the inference photo's quality are being maximized. But there is more to consider than inference when deciding sizes, as training is equally important (if not more) to inference when the questions of image size emerge [24].

To train a model, it is better to resize all images to a specific size. The general rule is that the model is better to infer the same size photos as the ones it was trained on, so the sizing decision can be highly affected by the available data themselves and how resizing affects them. The final decision was made according to the data collected and considering the performance of the model during inference. With the median aspect ratio on the dataset photos being 1.28 which is really close to the camera's 1.33 and with the tiny model needing on average 0.5 seconds to infer one image (so in the case of making two square 640x640 images, the inference would be around 1 second) the best option was considered to use 640×480 in training and inference, sacrificing some pixels of resolution to get as little resizing distortion as possible in training, as well as the fastest inference possible, at 0.4 seconds.

3.1.5 Training

Preprocessing and Data Manipulation

As stated above, in the preprocessing step, all images were resized to 640x480. To further reduce possible distortions that might appear, especially on smaller images being scaled up, instead of stretching the image, padding with black-colored pixels was applied to get photos to scale up. This also might help the model recognize objects that are far from the camera and appear small in the image. In many applications, when the object size in pixels to be presented to the model is kind of known, this size can be the metric to decide both the image size and the dataset's size [21] [24]. In this application, the scenarios are many and unpredictable, so the strategy was to go as big as possible on the data, trying to cover as many scenarios as possible. Another preprocessing addition was to simulate night vision photos by making 15 percent of the images grayscale to cover the shortage of night vision data. Finally, the training of the model was carried out using Google Colab [25], leveraging a Tesla T4 GPU [26]. The Tesla T4 is a professional-grade graphics card based on NVIDIA's Turing architecture, featuring 16 GB of GDDR6 memory and 2,560 CUDA cores. It is equipped with specialized Tensor Cores that support accelerated mixed-precision calculations. This allows T4 to achieve a theoretical peak performance of 8.1 TFLOPS for FP32 (single-precision), 65 TFLOPS for FP16 (half-

precision), 130 TOPS for INT8, and 260 TOPS for INT4 operations.

Optimizer Algorithm

The gradient descent variation optimizer we chose was AdamW [27] because of its implementation of Decoupled Weight Decay compared to the original Adam optimizer. In the original Adam, weight decay is added to the loss function and becomes part of the gradient, meaning it is subsequently scaled by the adaptive learning rate. This coupling weakens its regularization effect. AdamW rectifies this by decoupling the weight decay term, applying it directly to the weights after the adaptive update. This ensures consistent and effective regularization, which typically leads to better model generalization and was a key reason for its selection in this training configuration [28].

Training Epochs

The final model was selected after a comparative analysis of training durations. While models trained for 100 or 150 epochs demonstrated superior performance on the test set, they exhibited significant overfitting when applied to real-world, unseen data, presented to it from the device. These longer-trained models generated frequent false positives, misclassifying natural patterns such as tree branches and clouds as deers and boars.

This divergence is attributed to two main factors. First, there is a high degree of similarity between the training and test datasets. Although the dataset is large, the sources from which the data were provided were limited. Many images were captured from similar viewpoints and under comparable conditions. This not only caused the model to specialize in the specific locations and conditions shown but also created a significant overlap that compromised the independence of the test set. Second, the dataset lacks a sufficient number of negative example—images without any detectable target class. This absence introduces a significant bias, training the model to always expect an object. Consequently, the model learns to prioritize detection (even a false one) over correctly identifying the absence of a target, as this strategy yields higher accuracy on the specific dataset.

As a result, the overtrained model's generalization capability was limited, causing it to perceive "ghosts" in the false patterns of non-living objects within genuinely novel data. Thus, after testing many different numbers of training epochs, a 60-epoch model was chosen as the most appropriate one, offering superior robustness and a reduced tendency to overfit.

Batch Size

Lastly, the model was trained with a batch size of 16. The batch size defines the number of training examples processed before the model's internal weights are updated using the calculated average gradient across all examples in the batch.

Larger batch sizes generally provide a more stable and direct convergence path, but carry a higher risk of overfitting, as the model may memorize the training data. In contrast, smaller batches introduce helpful noise into the learning process, which often results in a more generalized model that performs better on unseen data, although with a less stable training trajectory.

Batching also provides computational efficiency that significantly reduces the training time, letting researchers experiment and test different models within a reasonable time window. On a GPU, the examples within a batch can be processed in parallel (provided they fit in its memory), drastically accelerating the training process. In this case, a batch size of 16 reduced the training time from approximately 9.5 minutes of a batch size of 1 (where weights are updated after every training photo) to just 2.15 minutes per epoch (a 4.41x speedup). This batch size was selected as it was considered an optimal compromise because it leveraged the GPU's parallel processing to reduce the training time, while remaining small enough to promote the model's ability to generalize to new data, as opposed to over-specializing on the training set.

Training Results And Diagrams

The training procedure was conducted for a total of 100 epochs, with model checkpoints saved every 10 epochs. As stated previously, the final model selection was made in a practical manner, prioritizing performance on real-world data, with more emphasis on minimizing false positives. The reason this metric was chosen to be minimized was about minimizing false notifications, where a notification was designed to be triggered by a class recognition. Even if the false positives minimization could come at a cost of more false negatives, provided that animal activity is more probable to cause many triggers than one (as animals rarely move running at full speed) a detection is almost certain that it will be made on at least one of the triggers.The training statistics presented here in Figure 3.5 further support the choice of a less epochs model.

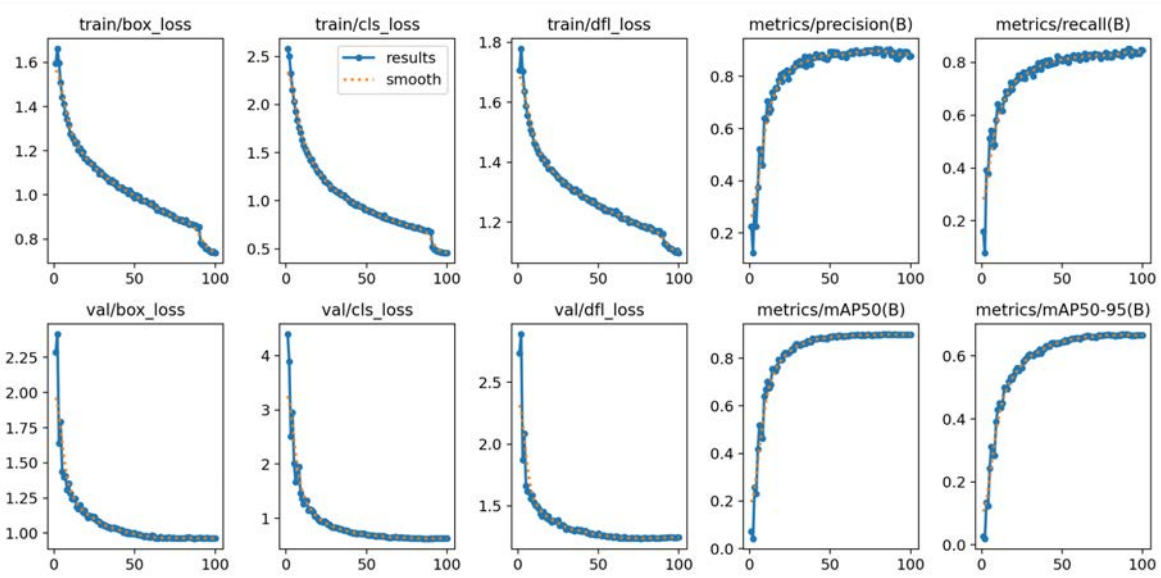


Figure 3.5: YOLOv8-nano model Training curves for 100 epochs

Evaluation Metics

`Precision`: Fraction of predicted boxes that are correct. Indicator of False Positives.

$$\text{Precision} = \frac{TP}{TP + FP}$$

`Recall`: Fraction of true objects that were detected. Indicator of False Negatives.

$$\text{Recall} = \frac{TP}{TP + FN}$$

`Average Precision (AP)`: Area under the precision–recall curve $p(r)$, providing a single value that encapsulates the model's precision and recall performance.

$$AP = \int_0^1 p(r)\, dr$$

`mAP50`: Mean Average Precision across all classes with Intersection over Union (IoU) threshold set at 0.5. [29], [30]

$$\text{mAP}_{50} = \frac{1}{N} \sum_{i=1}^{N} AP_i \quad \text{with IoU} \geq 0.5$$

mAP50-90: Mean Average Precision across all classes, averaged over IoU thresholds from 0.50 to 0.95 with a step of 0.05.

$$\text{mAP}_{50:95} = \frac{1}{10N} \sum_{t=0.50}^{0.95} \sum_{i=1}^{N} AP_i(t)$$

IoU: Intersection over Union, measuring the overlap between predicted and ground-truth bounding boxes. [31]

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Loss Functions

The Loss Function of YOLOv8 is calculated from the summation of three different losses shown in the diagrams.

box_loss: Localization loss for bounding boxes, measuring error in predicted box location, size, and overlap.

cls_loss: Classification loss, measuring error in assigning the correct class to detected objects.

dfl_loss (Distributed Focal Loss): Extension of focal loss that distributes weights across multiple object scales and classes, improving learning from difficult examples and handling class imbalance.

A convergence and subsequent flattening of the loss and evaluation curves can be observed after approximately 50 epochs across all metrics presented, with the more important one in this case being Precision as the primary goal is to minimize False Positive detections. While on training losses a sudden drop can be perceived around epoch 90, the corresponding validation metrics did not exhibit equivalent behavior. Instead, a divergence between training and validation curves appears there, indicating the onset of overfitting. This behavior can be interpreted as the model beginning to memorize patterns specific to the training dataset, leading to false recognitions and justifying the false recognitions of background patterns as animals in the real-world tests made.

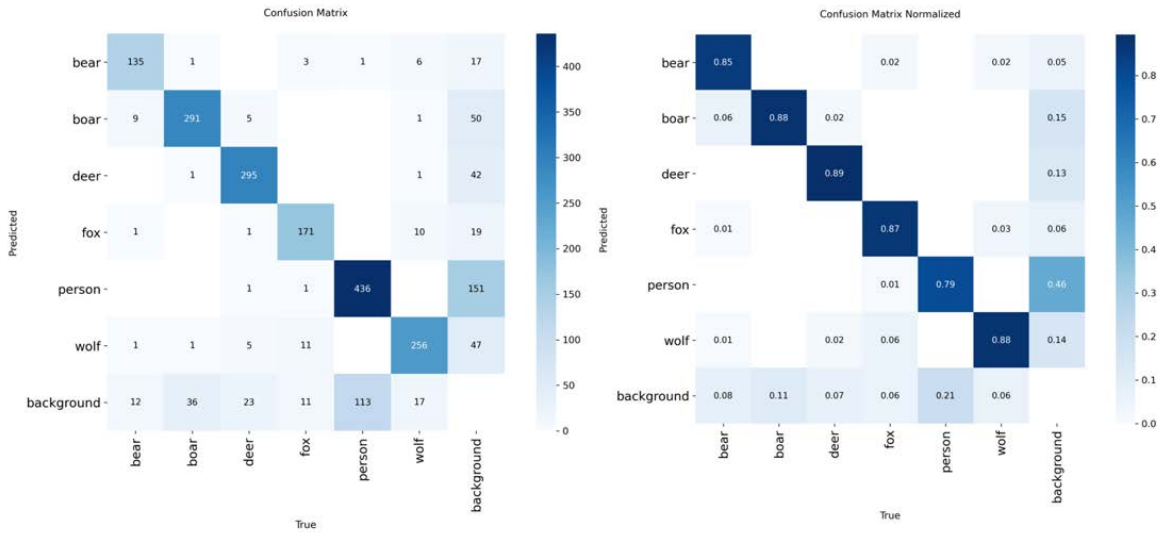


Figure 3.6: Confusion Matrices of the YOLOv8-nano Model on the Test Set

The overfitting problem can also be seen in the confusion matrices of the 100-epoch model of Figure 3.6. Particularly in the background ”class,” where the false detections amount to 320 (sum of the background column) compared to 201 false negatives (sum of the background row). The fact that most mistakes appear in the ”person” class is completely normal here, not only because of the imbalance in the number of person images compared to other classes, but also because more challenging photos of people were deliberately included in the dataset.

Chapter 4

Sensor Interfacing And Operating System Changes

4.1 PIR and PiCamera Usage with Python

4.1.1 PIR Sensor Set Up

The HC-SR501 provides two potentiometers to control its sensitivity and time delay (shown in Figure 4.1). The sensitivity value spectrum ranges from nearly zero to up to 7 meters of detection and was set at a midpoint to avoid false triggers. The time delay was set to the lowest value possible, which is 1 second, because there is absolutely no need for a longer triggered state time for the software to notice the trigger.

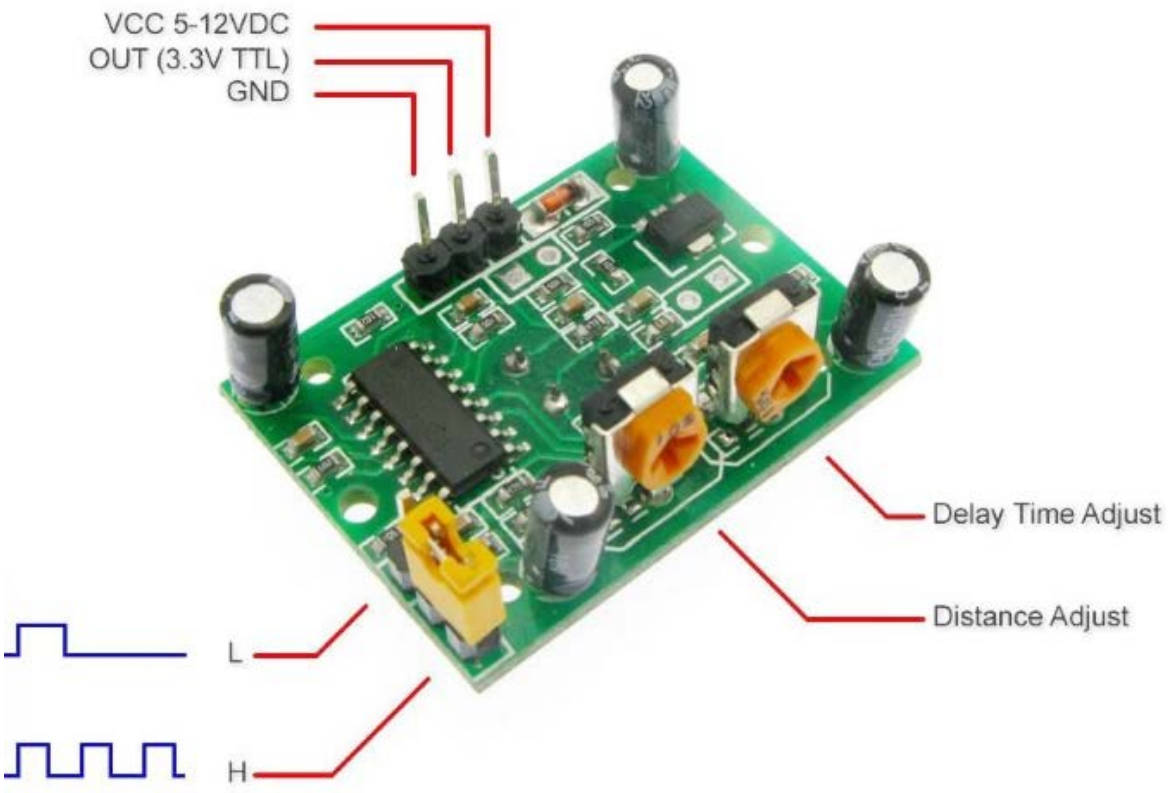


Figure 4.1: PIR Sensor PCB [11]

For the PIR Sensor to be used by the RP5, its VCC, GND and OUT pins need to be connected with jumper wires to the RP5 Pins 4, 6 and 7 respectively. The PIR output was also connected to a BC547 transistor's [32] base in series with a 9.9 kΩ resistor. The transistor's Collector and Emitter legs were soldered onto the open power button pins that the RPi provides and simulate the push of the power button every time the PIR Sensor is triggered by motion. That ensures that the RP5 would boot from shutdown every time motion is perceived, while the 1 second high signal is safely ignored by the system when it is up and running on a headless Raspberry Pi OS (disclaimer: this does not happen in the full desktop OS where one push pops up the Boot Menu and a second push shuts the system down and needs to be configured manually to be ignored). The only case where the transistor power button is not ignored is when its signal stays high for more than 5 seconds. This type of signal does trigger a hardware-controlled force shutdown of the system, and thus, there is a rare case where if movement is perceived constantly for about 4+ seconds the system will mistakenly shut itself down. This case can be prevented by using the PIR Sensor's Trigger Selection Jumper (or pin soldering if no jumper is provided on the sensor) to choose whether the output stays high on re-trigger or not, and change its default behavior to the desired one. To use the PIR sensor

with code, we used Python's `gpiozero` [33] library.

4.1.2 Picamera Setup

The Infrared Camera we used for this project does not have an IR-Cut filter lens, so the day images were cleaned up manually with software interventions by adjusting the camera's color gains to get cleaner and more natural colored images instead of the pinkish daytime and bluish nighttime photos it produced by default. We also did some manual focusing to the camera's lens.

Connecting the camera with the RP5 is extremely easy to do by connecting the camera cable to the RP5's CSI (Camera Serial Interface) port. The camera is automatically recognized and driven by the system using the built-in `libcamera` [34] library in the Raspbian OS. To use the camera with code, we used Python's `picamera2` [35] library.

4.2 OS Manipulation and Application Set Up

4.2.1 Raspberry Pi's Shutdown To Running An App Sequence

At first, we need make clear that the Raspberry Pi board does not support a low-power sleep state like most conventional computers do. The system is either fully ON or completely OFF. Therefore, every time a movement trigger is received after a long period of inactivity, the system must perform a full boot from a powered-off state before it can take a picture. This means that before the Python application can be loaded with all its dependencies (which also requires an amount of time), the time-consuming boot sequence must be carried out first.

The initial measured booting time can be shown with `systemd-analyze`: `Startup finished in 2.034s(kernel)+13.530s(userspace)=15.564s`. The Python application itself, along with necessary firmware operations not measured by the system measurement above as they take place before the kernel is even booted, needs at least 5 more seconds each. With the movement-to-capture latency being around 25 seconds, optimization for a faster boot and picture capture is absolutely mandatory. At such long durations, the chances of missing a fast-moving animal are arguably high. These boot times were the only factor that put RP5 in doubt against a microcontroller solution (this trade-off and alternate arrangements will be discussed in Chapter 5).

The journey from a powered-off state to capturing an image consists of a sequence of distinct phases, with some of them being prone to optimization. The phases are as follows:

Note that on the first 3 bootloader stages, the ARM CPU is held on reset and the VideoCore GPU is carrying all the work.

1. **1st Stage Bootloader:**
   Code stored in the ROM of the SoC is executed on a tiny RISC core embeded inside the GPU. It initializes the SD card interface and mounts the FAT32 boot partition so it can be read by the next stage.

2. **2nd Stage Bootloader:**
   The `bootcode.bin` file is loaded from the SD card into the GPU's cache. This code enables the system RAM and subsequently loads the `start.elf` file.

3. **3rd Stage Bootloader:**
   The `start.elf` file (which contains the GPU firmware and drivers) does the heaviest work. It reads the `config.txt` file to configure the hardware accordingly, it does load the Device Tree from the `bcm2712-rpi-5-b.dtb` file, and then reads `cmdline.txt` for kernel parameters. Finally, it releases the reset on the ARM CPU and starts the `kernel_2712.img` with the correct command-line arguments. From this point and on, the CPU is "in charge".

4. **Kernel Initialization:**
   The Linux kernel is booted. It initializes the system's hardware based on its compiled-in drivers and the device tree, and sets up memory management.

5. **Init System:** System services are started to configure the environment according to system settings.

6. **Application Startup:**
   The main application is launched. The Python interpreter loads, imports its required libraries, and initializes any model needed for the application.

4.2.2 config.txt and cmdline.txt

As established previously, the Raspberry Pi firmware does not utilize a traditional BIOS or UEFI like conventional PCs. Instead, the initialization sequence is managed by the GPU

(VideoCore). The GPU reads the `config.txt` [36] configuration file to configure the hardware before the ARM CPU and main operating system are initialized. This is handled by the third-stage bootloader `start.elf` file, which also contains the GPU driver. While optimizations are possible to be made at this or even lower firmware levels, they are far more complex, require reverse engineering of binary files given by the manufacturer, and were not implemented or tested for this project.

Subsequently, the GPU reads the `cmdline.txt` [37] file, which contains kernel parameters. Its contents are passed to the Linux kernel, which is then executed by the ARM CPU. After the kernel boots, the operating system's init system (e.g., `systemd`) takes over.

Both files can be found and edited in the `/boot/firmware/` directory of Raspberry Pi OS or directly on the SD card when accessed from another device (which is extremely useful for debugging).

Within `config.txt`, we added the following lines in the `[all]` section, indicating that these settings apply to all Raspberry Pi models that boot from this SD card:

- **Disable splash screen** from the boot process:
  `disable_splash=1`
- **Set boot delay to zero**: The delay relationship is defined as:
  *delay = 1000 * boot_delay + boot_delay_ms*, so both were set to zero:
  `boot_delay=0`
  `boot_delay_ms=0`
- **Disable unused hardware interfaces**:
  `dtparam=audio=off`
  `dtparam=uart0=off`
  `dtparam=uart1=off`
- **Remove OS checks**:
  `os_check=0`
  `bootloader_update=0`
- **Disable Bluetooth and WiFi**:
  `dtoverlay=disable-bt`
  `dtoverlay=disable-wifi`

- **Overclock the system**:

  `force_turbo=1`

  This disables dynamic clocking, forcing all frequencies and voltages to stay high. This option requires a reliable power supply.

Furthermore, in `cmdline.txt`, we addded `quiet` argument to suppress non-critical kernel messages during boot. These modifications saved about 4 more seconds:

`Startup finished in 1.194s(kernel)+10.349s(userspace)=11.544s`

4.2.3 Raspberry Pi OS

Raspberry Pi develops and maintains Raspberry Pi OS (formerly Raspbian) [38], a Linux-based operating system and kernel optimized for the Raspberry Pi's ARM architecture, its peripherals, and its GPIO pins. It supports both 64-bit and 32-bit systems and is the recommended starting point for new users. The lightweight, headless version, Raspberry Pi OS Lite, was chosen for this project. Its minimal footprint, requiring only approximately 0.4 GB of storage, offers both ease of use and the potential for a boot time fast enough to capture passing animals.

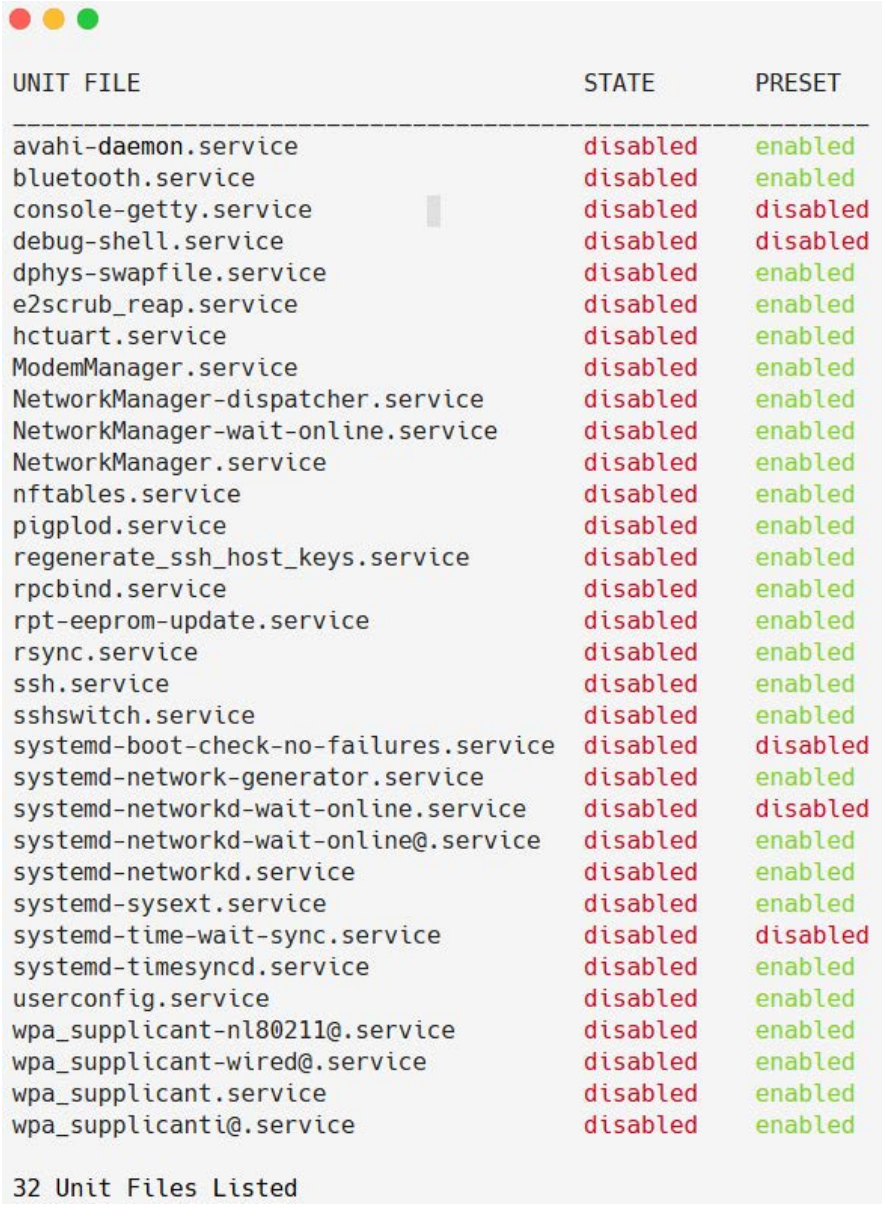
With the altered configuration files, the Linux kernel itself boots in approximately one second. The majority of the total boot time is consumed by the userspace initialization managed by the init system, `systemd`. Following the optimization principle of prioritizing the most time-consuming components, rearranging system services was necessary.

4.2.4 Removing Unnecessary Services

In one sentence, a Service is a Linux program that runs in the background. These programs, also called daemons, are most often launched at boot by the system's init system, which in Raspberry Pi OS is `systemd` [39], and run by the system independently of any user login. Any standard Linux distribution like the Pi OS is designed as a general-purpose system, preconfigured with a wide array of services to support a broad spectrum of use cases, from desktop environments to network administration and multimedia. However, a specialized edge application that performs a single, specific task does not require the majority of these background processes. For example, services related to Bluetooth, Wi-Fi, audio, and networking protocols not only consume system resources but also delay the boot process,

which is of extreme importance in this case. By identifying and disabling the unneeded services, the system's boot sequence is simplified, and the overall startup time can be significantly reduced.

This process needs to be done iteratively by checking the `systemd-analyze blame` report, identifying the most time-consuming ones, researching and deciding whether they are crucial for the system and the application or not, and then according to that disabling them or keeping them. Optimizing the boot sequence leads to a smaller off-to-application latency. For example, most of the time the most time-consuming service on a system is `NetworkManager-wait-online.service`, responsible for ensuring a safe internet connection is established by `NetworkManager.service`, can take from 5 to 15 seconds and can be disabled with `sudo systemctl disable NetworkManager.service` (along with the other networking services too) when no internet connection is needed. After completing this process, the unnecessary services that were disabled can be displayed with: `systemctl list-unit-files –type=service –state=disabled` and are shown in Figure 4.2 below.



```
UNIT FILE                                  STATE      PRESET
------------------------------------------------------------
avahi-daemon.service                       disabled   enabled
bluetooth.service                          disabled   enabled
console-getty.service                      disabled   disabled
debug-shell.service                        disabled   disabled
dphys-swapfile.service                     disabled   enabled
e2scrub_reap.service                       disabled   enabled
hctuart.service                            disabled   enabled
ModemManager.service                       disabled   enabled
NetworkManager-dispatcher.service          disabled   enabled
NetworkManager-wait-online.service         disabled   enabled
NetworkManager.service                     disabled   enabled
nftables.service                           disabled   enabled
pigplod.service                            disabled   enabled
regenerate_ssh_host_keys.service           disabled   enabled
rpcbind.service                            disabled   enabled
rpt-eeprom-update.service                  disabled   enabled
rsync.service                              disabled   enabled
ssh.service                                disabled   enabled
sshswitch.service                          disabled   enabled
systemd-boot-check-no-failures.service     disabled   disabled
systemd-network-generator.service          disabled   enabled
systemd-networkd-wait-online.service       disabled   disabled
systemd-networkd-wait-online@.service      disabled   enabled
systemd-networkd.service                   disabled   enabled
systemd-sysext.service                     disabled   enabled
systemd-time-wait-sync.service             disabled   disabled
systemd-timesyncd.service                  disabled   enabled
userconfig.service                         disabled   enabled
wpa_supplicant-nl80211@.service            disabled   enabled
wpa_supplicant-wired@.service              disabled   enabled
wpa_supplicant.service                     disabled   enabled
wpa_supplicanti@.service                   disabled   enabled

32 Unit Files Listed
```

Figure 4.2: Disabled Services Screenshot

By systematically disabling these services, the system's boot sequence was significantly simplified, and the overall startup time gained about 8 more seconds:

`Startup finished in 1.086s(kernel)+1.873s(userspace)=2.959s`

4.2.5 Making the App a Linux Service

Configuring an application as a service provides a standardized mechanism to control its lifecycle. Services can be configured to start, stop, or restart automatically on failure and can be queried for their status easily, which is invaluable for debugging. Systemd provides a mechanism to create and configure services by creating a `service_name.service` file

in the `/etc/systemd/system/` directory. The syntax of this file is based on key-value pairs organized under specific section headers. The most critical sections and directives are as follows:

`[Unit]` Section for metadata and dependencies, more important to programmers than the system itself.

- `Description`: A human-readable name and a short description of the service's functionality.
- `DefaultDependencies`: A boolean value (yes/no) that controls whether systemd adds implicit default dependencies for the unit to coexist in harmony with the system. It is generally safe to let this setting remain at its default value 'yes'.
- `After`: The units listed in this directive will be started before starting the current unit.

`[Service]` Section defining how the service runs.

- `Type`: Configures the process startup type, which is crucial for systemd to track the main process correctly. The common types are:
  - `simple`: (Default) systemd assumes the main process is the one started by `ExecStart`.
  - `forking`: The process forks a child and then the parent exits. `systemd` expects the child to become the main service process.
  - `oneshot`: Used for scripts that do a single job and then exit. Often combined with `RemainAfterExit=yes`.
  - `idle`: Similar to `simple`, but the execution is delayed until all active jobs are dispatched.
- `User`: The privilege level under which the process runs.
- `WorkingDirectory`: The path to the working directory.
- `ExecStart`: The absolute path and arguments of the command to execute.
- `Restart`: Policy for automatic restart.

`[Install]` Section used to define the behavior of a unit if it is enabled or disabled. For a unit to start at boot, it needs to be enabled.

- **`WantedBy`**: The target unit that will ”want” this service.

After creating or modifying a **`service_name.service`** file, to inform **`systemd`** of the new configuration, the command **`sudo systemctl daemon-reload`** must be executed. To automatically start the service on the next boot, the service must be enabled with **`sudo systemctl enable service-name.service`**. Although this mandatory setting provides automation, it does not provide any speed improvement, and the capture time remains high.

Optimizing the application code itself by rearranging the Python code to run the capture as fast as possible from the time it was launched could reduce some time. Instead of using the standard syntax where all needed libraries are imported in the first lines of code, importing only the essential libraries for capturing to initiate the capture immediately, and then importing the rest after the capture has completed, can save some crucial seconds. However, a much better option to save time is to avoid using the Python application at all for the first capture and instead use another service that can run much faster and earlier in the boot process than the slow Python application.

Therefore, we implemented the application functionality by using two services, one created to do a sprint, and one to run a marathon:

- The **`Type=oneshot`** **`fast_cap.service`**, which runs a **`libcamera-still`** command implemented with c++ code to perform the first capture as fast as possible and save it to the **`/tmp`** directory.

- The **`Type=simple`** **`app.service`** was modified to start later, apply inference on the photo left behind by the **`fast_cap.service`**, and then proceed to its normal operation: repeatedly waiting for movement, capturing in real-time, inferring the captured image, and shutting down the device, if no movement is detected for a specified time period.

With this architecture, with **`fast_cap.service`** running as early as possible and configured with **`After=local-fs-pre.target`** and **`WantedBy=sysinit.target`**, the capture is performed almost simultaneously with the finalization of the boot stage. This

also allows `app.service` to start much later in the boot process,and being configured with `After=systemd-remount-fs.service` and `WantedBy=multi-user.target`.

The `fast_cap.service` also used `DefaultDependencies=no` which, in general, makes the service "a bad citizen" of the system. This way, the service can be configured to start earlier without the default dependencies in the boot sequence, with the risk of failure rising. It is also not killed gracefully by a system signal during shutdown but instead it is stopped abruptly by power loss to the chip. For more complex services like `app.service`, this can cause problems such as leftover corrupted files and uncleaned temporary files. In a worst-case scenario, for a service that acquires locks, it could even lead to a deadlock if the service is killed while in a critical section.

However, the `fast_cap.service` writes a file only once, acquires no locks, and is safely configured to stop after the capture. It is therefore safe to iteratively place it earlier and earlier in the boot sequence until it starts to fail to achieve the fastest capture possible. In every other case, using `DefaultDependencies=no` is a non-recommended option. This optimization includes no improvement in the boot times but only in the capture time. So, its impact can not be shown by the `systemd-analyze blame` command result:

`Startup finished in 966ms(kernel)+1.734s(userspace)=2.700s`

but it can be shown by profiling the whole boot sequence and producing a plot like the one shown in Figure 4.3 with:

`systemd-alalyze plot > plot.svg`

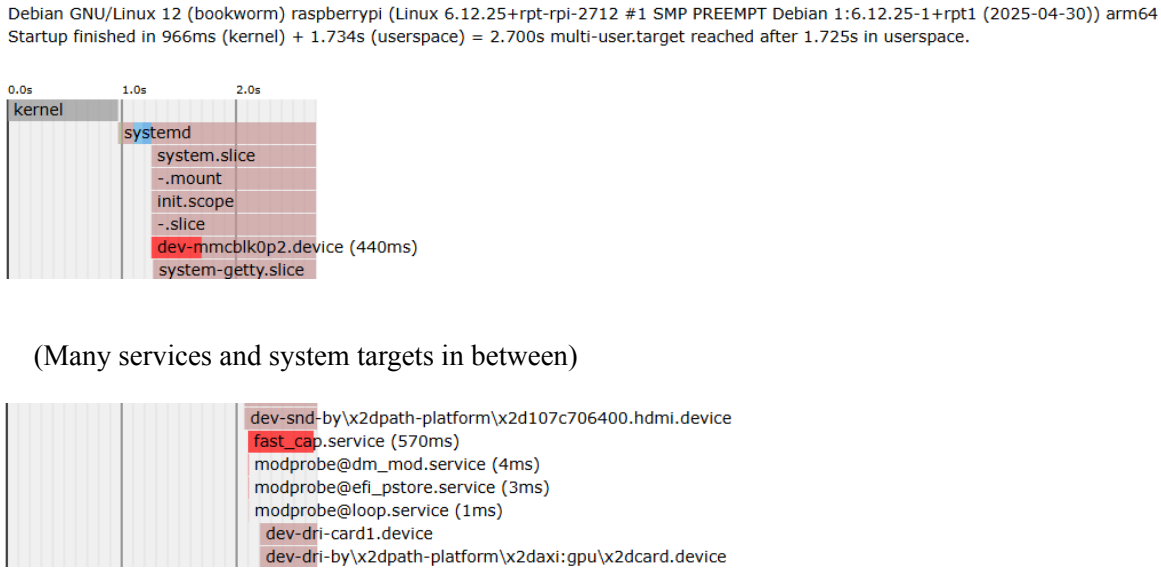


Figure 4.3: Systemd Boot Process Visualization

4.2.6 Time Optimization Results

After all these optimizations, the trigger-to-capture latency was reduced to a more feasible value of around 8 seconds. Throughout the whole Chapter we discussed the optimization steps done and the mechanism behind each of them. The cumulative results are depicted in Figure 4.4 where the time measurements of kernel boot time, userspace time, cumulative time of kernel and userspace, and finally the most meaningful to the application, Capture Time from a power off state, which cumulates all the previous times as long as the Firmware Boot Time and the Python application's time is shown.

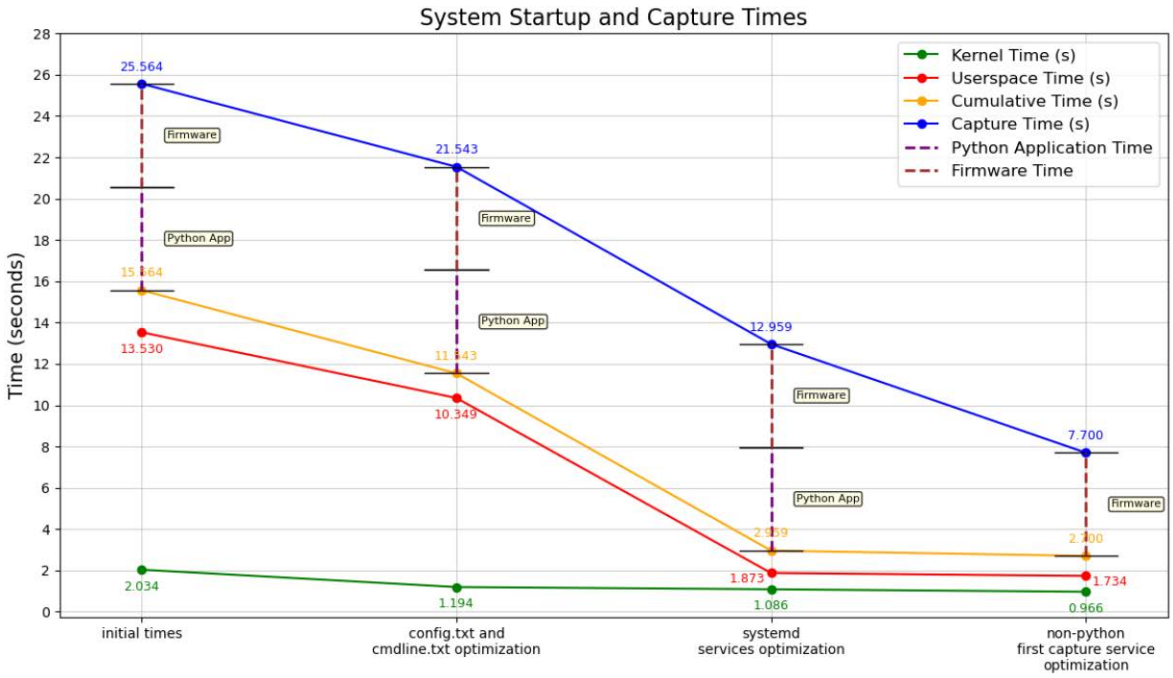


Figure 4.4: Optimization Steps Time Measurements Diagram

Scanning the diagram from left to right, we can observe which time component is improved by each optimization step. The modifications in `config.txt` and `cmdline.txt`, discussed in Section 4.2.2, affect both the kernel boot time and userspace initialization time. The kernel boot time is reduced from 2.034s to 1.194s, while nearly 3s are saved in userspace initialization. The optimization of the `systemd` initialization system, presented in Section 4.2.4, has a substantial impact on userspace time, reducing it by more than fivefold — from over 10s to less than 2s — without affecting kernel time, as expected. Finally, the third optimization described in Section 4.2.5 does not influence kernel or userspace times but effectively eliminates the Python application's latency by introducing a c++ service that performs the capture as early as possible in the userspace's boot process.

Chapter 5

Evaluation and Future Expansions

5.1 Testing

The testing of the system was conducted between the 17th and 20th of August 2025 in a remote location known for its increased activity of wild boars shown with the heart sign in Figure 5.1.

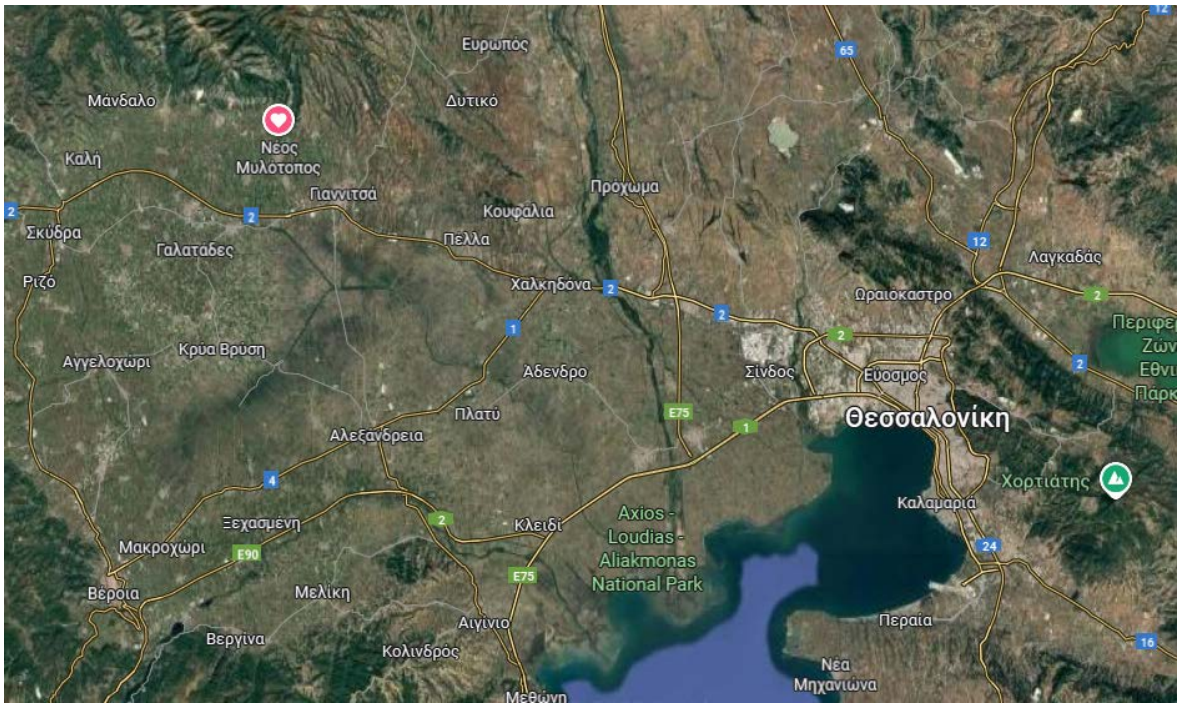


Figure 5.1: Off Grid Testing Location

We mounted the camera case and the solar power controller case in nearby trees at a height of about 120 cm above the ground, with the camera looking slightly downward. The

solar panel was mounted about 5 meters away from them, higher than on a tree about 200 cm above the dense bushes, so it could be directly hit by the sunlight. In addition, we set the `conf` confidence threshold value for a detection label to be assigned to 0.4 (instead of the default value 0.25) to further reduce the possibility of false positive detections.

Figure 5.2: Device And Solar Power Manager

Figure 5.3: Solar Panel

Although the first day was cloudy, and that could affect the battery, no signs of wildlife were monitored and the battery remained very close to 100%. During the second night, a herd of wild boars passed by the area twice and was captured and recognized by the device. Also, some people's photos were taken by the device right after the mounting time and a day later when we paid a visit to ensure everything was working properly. Data for species other than boars and people are not available yet, so the evaluation is only concerning them.

- **Person class:** 15/15 photos correctly detected, with 8 taken during daytime and 7 during nighttime.

- **Boar class:** 3/20 photos correctly detected. The device was triggered twice during the night. At 00:23, it captured 14 photos, with all 3 detections made on the clearest ones,

correctly identifying animal activity in the area. At 04:57 it captured 6 more photos, but no detections were made, with the photos being much darker and blurrier than the previous ones but with the boars still recognizable in the images by the human eye.

- **False detections:** 0 false detections or false positives, which is a very desirable outcome.

After all, the off-to-capture time, amounted to nearly 8 seconds (which is far from instantaneous) was proven to be sufficient as it did not result in missed detections of passing by animals. This result is attributed to the moving pattern of boars that passed by which seems to be more chaotic than linear. In addition, the fact that they move in herds increases the detectable objects, making the capture easier.

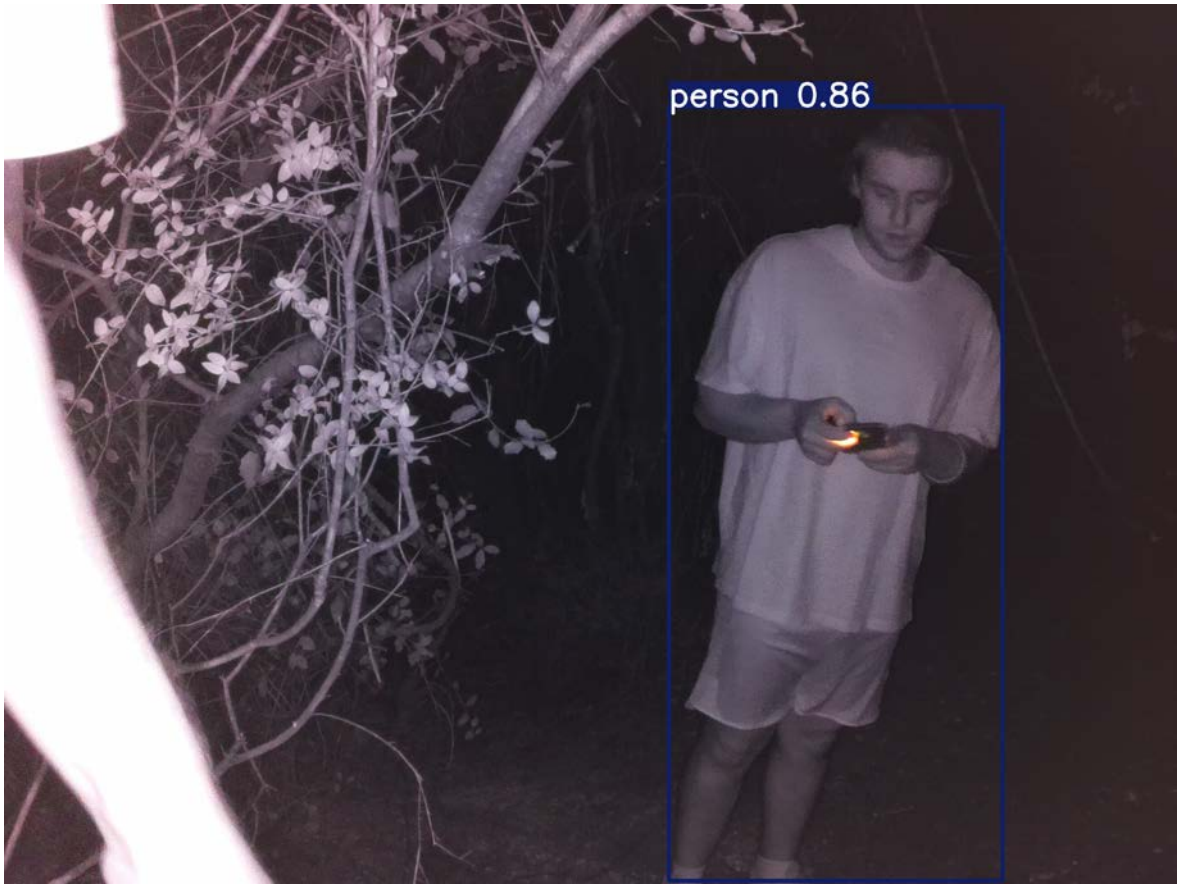


Figure 5.4: Person Detection Example

Figure 5.5: Boar Detection Example

During this period, the device booted itself from sleep 7 times, received a movement trigger 105 times, and stayed awake for 58 minutes. With such a short working time, it is no wonder that the device managed to keep its batteries fully charged for almost all the time. It is obvious that in these conditions the system could function with much fewer resources, with a much smaller battery and a weaker and cheaper solar panel. Although this is true, it is important to consider that the system was designed according to a worst-case scenario of sunlight and wildlife activity and to be mounted and used for much longer periods of time than the time of testing, with additional power consumption from the LoRa networking which was not implemented due to issues discussed in the upcoming Section 5.2.1.

Figure 5.6: Failed Detection Example

5.2 Future Expansions

5.2.1 LoRa Networking

The most important and necessary future expansion is the integration of LoRa networking into the system, which was part of the initial design but was not fully implemented during the development period. Although an attempt was made to develop a peer-to-peer communication protocol using two iU880A long-range USB Adapters [40] provided by the university, no long-range communication was ultimately added to the system. These modules have been discontinued since 2016, and, due to the lack of open-source support, no way was found to utilize them either with code or even through GUI tools. The plan is to replace them with newer, open-source LoRa modules to implement reliable peer-to-peer communication.

5.2.2 YOLO Model Retraining

Another important direction for future work is the continuous improvement of the YOLOv8 model through retraining. The device itself is the means of collecting more image data for training, which are more representative and share more similarities with those used during inference than the data collected from public sources on the internet. So, the next generation of models produced is expected to be more accurate as well.

5.2.3 Firmware Level Boot Optimization

As stated in Chapter 3, the not yet implemented optimization of the boot process, and also the hardest one, is optimizing the boot sequence in boot stages 2 and 3 by reverse engineering `bootcode.bin`, `start.elf`, and `bcm2712-rpi-5-b.dtb`, or by testing other versions of them from older or different configurations and measuring whether any improvement is perceived, provided that the system still boots correctly with them. The upper threshold of this optimization is expected to be around 5 seconds of time saved at most. In the best-case scenario, the system would achieve a final capture time from power-off of around 3 seconds, which would be a great result since it is close to the expected microcontroller capture time from sleep.

5.2.4 Custom Minimal OS Attempts

During the boot-time optimization, for a more lightweight and application-specific Operating System, the creation of a minimal Linux system using Buildroot [41] was attempted. Buildroot is an Embedded Systems tool that automates the process of generating a cross-compiled root filesystem, toolchain, and most importantly a custom Linux kernel, allowing developers to include only the components strictly required by the application. This resulted in a tailored distribution that is smaller and theoretically faster to boot compared to general-purpose operating systems such as the Raspberry Pi OS.

However, even though the kernel and the overall system produced were smaller (295 MB), boot performance did not improve significantly compared to the optimized Raspberry Pi OS (423 MB) setup, which achieved a kernel boot time of 0.966 sec. Furthermore, no differences were observed at the firmware level of booting, which remained the slowest part of the process, as it relied on the same files as the original Pi OS for the lower boot stages.

So, the little-to-no boot-time gains, along with the difficulties that emerged during development, mainly related to the integration of the OV5647 camera driver and the `libcamera` library, as well as the lack of a convenient initialization system, like `systemd`, for managing services and customizing the startup sequence, led to abandoning the custom built kernel and Buildroot-based system in favor of the more reliable and well supported, but still very prone to optimization Raspberry Pi OS. The same problems with libraries and compatibility, even with the sensor libraries, were also presented with the tiniest Linux distribution of tinyCore, piCore (only 40MB).

After all, this optimization might only become meaningful once the firmware boot times are reduced, since the milliseconds saved at the OS level are negligible compared to the 5–6 second overhead introduced by the firmware. Adapting the application to this new custom system is worth it, only if the firmware overhead is reduced first.

5.2.5 Using Different Hardware

In the developer world, it is common practice that when the software needs to become more sophisticated, a problem can sometimes be solved by using better or just different hardware. This philosophy is the basis for the suggestions presented in this section.

Microcontrollers

Instead of optimizing the firmware, the init system, or even the application itself, we could have just used a microcontroller and a compatible camera (the PIR sensor chosen is already compatible) to achieve fast capture. The Raspberry Pi 5 would then serve only for its computational power, as microcontrollers cannot run inference of YOLO models of this size on high quality photographs in feasible times. ESP32 is such a microcontroller and with an estimated boot time between 0.3 and 0.9 seconds [42] it is expected to be able to capture from a sleep state with latency less than 2 seconds. The complexity here is transferred to writing the capture code for a different less powerful device and making the devices communicate and share files while working in a synchronous fashion.

More Power

The off-to-capture latency problem would not exist if the system never turned off. This either requires connecting to grid electricity which in our case is impossible, or ”simulating” a connection to the grid. A system of this scale could be implemented using larger solar panels and batteries to ensure 24/7 operation without energy drains, even for long periods of cloudy whether. However, this energy-intensive approach falls outside the philosophy of system optimization and significantly increases the cost of the final product, so it was never an option.

Bibliography

[1] Raspberry pi 5 vs raspberry pi 4. https://kitronik.co.uk/blogs/resources/the-differences-between-raspberry-pi-4-model-b-raspberry-pi-5.

[2] 30 w solar panel. https://www.skroutz.gr/s/38737709/Rolinger-GD-1030-Monokrystalliko-Fotovoltaiko-Panel-30W-650x350mm.html.

[3] Solar power manager tachnical drawing. https://files.waveshare.com/wiki/Solar-Power-Manager-(D)/Solar_Power_Manager_D_Schematic.pdf.

[4] Mini ups used. https://www.aliexpress.com/item/1005005245619513.html.

[5] Hc-sr501 pir sensor. https://grobotronics.com/pir-sensor-module.html.

[6] Raspberry pi camera module 5mp 70° - night vision. https://grobotronics.com/raspberry-pi-camera-module-5mp-70-night-vision.html.

[7] Raspberry pi 5 battery box for it's rtc. https://www.aliexpress.com/item/1005006285770674.html.

[8] YOLO models documanetation. https://medium.com/@elvenkim1/yolov8-nano-vs-yolov8-large-4f21324baa38.

[9] YOLOv8 Overview. https://docs.ultralytics.com/models/yolov8/.

[10] Yolov8: A complete guide. https://viso.ai/deep-learning/yolov8-guide/.

[11] HC-SR501 PIR Motion Sensor Module Datasheet. https://smarthallroad.com/product/hc-sr501-pir-motion-sensor-module-in-pakistan.

[12] Simplified model to approach the theoretical clear skysolar pv generation curve through a gaussianapproximation. http://researchgate.net/publication/350347002_Simplified_Model_to_Approach_the_Theoretical_Clear_Sky_Solar_PV_Generation_Curve_through_a_Gaussian_Approximation#pf2.

[13] Solar power manager from waveshare's wiki. https://www.waveshare.com/wiki/Solar_Power_Manager_(D).

[14] Powerbank keep alive adapter by pihut. https://thepihut.com/products/power-bank-keepalive-assembled.

[15] Rhino cad software. https://www.rhino3d.com/.

[16] Raspberry pi 5 battery box for it's rtc. https://ted3d.gr/.

[17] Craftbot 2 3d printer. https://www.bhphotovideo.com/c/product/1362119-REG/craftbot_pr_999_002_2_3d_printer_may.html.

[18] Yolo models timeline. https://sandar-ali.medium.com/brief-timeline-of-yolo-models-from-v1-to-v8-7d383140afe2.

[19] YOLOV8 documanetation. https://yolov8.org/yolov8-documentation/.

[20] YOLOv8 nano vs large. https://medium.com/@elvenkim1/yolov8-nano-vs-yolov8-large-4f21324baa38.

[21] YOLO Darknet Guide. https://www.ccoderun.ca/programming/yolo_faq/.

[22] kaggle home page. https://www.kaggle.com/.

[23] Roboflow home page. https://universe.roboflow.com/.

[24] You might be resizing your images incorrectly. https://blog.roboflow.com/you-might-be-resizing-your-images-incorrectly/.

[25] Google colab home page. https://colab.google/.

[26] Nvidias tesla t4 gpu specs. https://www.techpowerup.com/gpu-specs/tesla-t4.c3316.

[27] Documentation of adamw optimizer algorithm. https://optimization.cbe.cornell.edu/index.php?title=AdamW.

[28] Adam vs. adamw: Understanding weight decay and its impact on model performance. https://yassin01.medium.com/adam-vs-adamw-understanding-weight-decay-and-its-impact-on-model-performance-b7414f0af8a1.

[29] map explained. https://learnopencv.com/mean-average-precision-map-object-detection-model-evaluation-metric/.

[30] Yolov8: A complete guide. https://blog.roboflow.com/mean-average-precision/.

[31] Performance metrics article. https://docs.ultralytics.com/guides/yolo-performance-metrics/.

[32] Bc547 transistor documentation. https://www.alldatasheet.com/datasheet-pdf/view/11551/ONSEMI/BC547.html.

[33] gpiozero library documanetation. https://gpiozero.readthedocs.io/en/stable/.

[34] libcamera library documanetation. https://libcamera.org/docs.html.

[35] picamera2 library documanetation. https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf.

[36] config.txt documanetation. https://www.raspberrypi.com/documentation/computers/config_txt.html#what-is-config-txt.

[37] cmdline.txt documanetation. https://www.raspberrypi.com/documentation/computers/config_txt.html#cmdline.

[38] Raspberry Pi OS documanetation. https://www.raspberrypi.com/software/operating-systems/.

[39] systemd init system documanetation. https://systemd.io/.

[40] Lora device we tried to develop peer-to-peer communication. https://web.archive.org/web/20161202173933/http://www.wireless-solutions.de/products/discontinued-products/iu880a-usb.html.

[41] Buildroot tool to generate embedded linux systems. https://buildroot.org/.

[42] Esp32 stated boot time measurement from a reddit user. https://www.reddit.com/r/esp32/comments/dk0vu1/esp32_boot_time/.