Artificial Vision for Humans

João Gaspar Ramôa Gomes

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Luís Filipe Barbosa de Almeida Alexandre
Co-orientador: Prof. Doutor Sandra Isabel Pinto Mogo

junho de 2020

Dedicatória

Dedico esta dissertação a todos os invisuais, para que a sociedade inclusiva seja, cada vez mais, uma realidade flagrante.

Agradecimentos

A conclusão e realização desta dissertação de mestrado contou com inúmeros incentivos e encorajamentos que, sem os quais, a realização da mesma seria impossível.

Em primento lugar, agradeço ao Professor Doutor Luís Alexandre por todos os contributos que fizeram com que este trabalho fosse possível. Agradeço, também, cada uma das suas palavras, carregadas de conhecimento, pois, não só me ajudaram no trabalho, como também contribuiram para o meu crescimento pessoal. Sem o Professor, jamais este projeto teria sido possível. Obrigado Professor Luís Alexandre por fazer parte do meu trabalho e por ter contribuido para a minha evolução como ser humano. Estar-lhe-ei eternamente grato.

Imprescindível também é o agradecimento à minha co-orientadora Professora Doutora Sandra Mogo, pelo seu estímulo e pelos contributos que fizeram toda a diferença neste trabalho. Obrigado por me ajudar a compreender que o conhecimento é resultado de várias interfaces e nunca é estanque. Bem-haja Professora Doutora Sandra Mogo.

Aos meus colegas do *SOCIA-LAB* por todo o apoio que me deram no meu trabalho e por criarem um ambiente proporcionador de golpes de asa. São eles, por ordem alfabética, já que por outra não fazia sentido: Abel Zacarias, André Correia, António Gaspar, Bruno Degardin, Bruno Silva, Ehsan Yaghoubi, Nuno Pereira, Nzakiese Mbomgo, Saeid Alirezazadeh e Sérgio Gonçalves.

Um agradecimento muito especial ao Vasco Lopes pelo apoio e motivação constantes.

Agradeço a todos os meus amigos por compreenderem que nem sempre foi possível estar com eles e, mesmo assim, nunca deixaram de me chamar, tendo-me dado sempre alento para continuar.

Aos meus pais, João Castro Gomes e Mónica Ramôa. À minha irmã, Antonieta. Por todos os dias que cheguei tarde a casa, por todas as refeições fora de horas e por todo o tempo que não pude estar convosco. Muito obrigado pelo vosso apoio e por terem acreditado sempre em mim.

Resumo

De acordo com a *Organização Mundial da Saúde* e *A Agência Internacional para a Prevenção da Cegueira* 253 milhões de pessoas são cegas ou têm problemas de visão (2015). 117 milhões têm uma deficiência visual moderada ou grave à distância e 36 milhões são totalmente cegas. Ao longo dos anos, sistemas de navegação portáteis foram desenvolvidos para ajudar pessoas com deficiência visual a navegar no mundo. O sistema de navegação portátil que mais se destacou foi a *white-cane*. Este ainda é o sistema portátil mais usado por pessoas com deficiência visual, uma vez que é bastante acessivel monetáriamente e é sólido. A desvantagem é que fornece apenas informações sobre obstáculos ao nível dos pés e também não é um sistema *hands-free*. Inicialmente, os sistemas portáteis que estavam a ser desenvolvidos focavam-se em ajudar a evitar obstáculos, mas atualmente já não estão limitados a isso. Com o avanço da visão computacional e da inteligência artificial, estes sistemas não são mais restritos à prevenção de obstáculos e são capazes de descrever o mundo, fazer reconhecimento de texto e até mesmo reconhecimento facial. Atualmente, os sistemas de navegação portáteis mais notáveis deste tipo são o *Brain Port Pro Vision* e o *Orcam MyEye system*. Ambos são sistemas *hands-free*. Estes sistemas podem realmente melhorar a qualidade de vida das pessoas com deficiência visual, mas não são acessíveis para todos. Cerca de 89% das pessoas com deficiência visual vivem em países de baixo e médio rendimento. Mesmo a maior parte dos 11% que não vive nestes países não tem acesso a estes sistema de navegação portátil mais recentes.

O objetivo desta dissertação é desenvolver um sistema de navegação portátil que através de algoritmos de visão computacional e processamento de imagem possa ajudar pessoas com deficiência visual a navegar no mundo. Este sistema portátil possui 2 modos, um para solucionar problemas específicos de pessoas com deficiência visual e outro genérico para evitar colisões com obstáculos. Também era um objetivo deste projeto melhorar continuamente este sistema com base em feedback de utilizadores reais, mas devido à pandemia do *COVID-19*, não consegui entregar o meu sistema a nenhum utilizador alvo. O problema específico mais trabalhado nesta dissertação foi o *Problema da Porta*, ou em inglês, *The Door Problem*. Este é, de acordo com as pessoas com deficiência visual e cegas, um problema frequente que geralmente ocorre em ambientes internos onde vivem outras pessoas para além do cego. Outro problema das pessoas com deficiência visual também abordado neste trabalho foi o *Problema nas escadas*, mas devido à raridade da sua ocurrência, foquei-me mais em resolver o problema anterior. Ao fazer uma extensa revisão dos métodos que os sistemas portáteis de navegação mais recentes usam, descobri que os mesmos baseiam-se em algoritmos de visão computacional e processamento de imagem para fornecer ao utilizador informações descritivas acerca do mundo. Também estudei o trabalho do *Ricardo Domingos*, aluno de licenciatura da UBI, sobre, como resolver o *Problema da Porta* num computador desktop. Este trabalho contribuiu como uma linha de base para a realização desta dissertação.

Nesta dissertação desenvolvi dois sistemas portáteis de navegação para ajudar pessoas com deficiência visual a navegar. Um é baseado no sistema *Raspberry Pi 3 B +* e o outro usa o *Jetson Nano* da *Nvidia*. O primeiro sistema foi usado para colectar dados e o outro é o sistema protótipo final que proponho neste trabalho. Este sistema é *hands-free*, não sobreaquece, é leve e pode ser transportado numa simples mochila ou mala. Este protótipo tem dois modos, um que funciona como um sistema de sensor de estacionamento, cujo objectivo é evitar obstáculos e o outro modo foi desenvolvido para resolver o *Problema da Porta*, fornecendo ao utilizador informações sobre o estado da porta (aberta, semi-aberta ou fechada). Neste documento, propus três métodos diferentes para resolver o *Problema da Porta*. Estes métodos usam algoritmos de visão computacional e funcionam no protótipo. O primeiro é baseado em segmentação semântica 2D e classificação de objetos 3D, e consegue detectar a porta e classificá-la. Este método funciona a 3 FPS. O segundo método é uma versão reduzida do anterior. É baseado somente na classificação de objetos 3D e consegue funcionar entre 5 a 6 FPS. O último método é baseado em segmentação semântica, detecção de objeto 2D e classificação de imagem 2D. Este método consegue detectar a porta e classificá-la. Funciona entre 1 a 2 FPS, mas é o melhor método em termos de precisão da classificação da porta. Também proponho nesta dissertação uma base de dados de Portas e Escadas que possui informações 3D e 2D. Este conjunto de dados foi usado para treinar os algoritmos de visão computacional usados nos métodos anteriores propostos para resolver o *Problema da Porta*. Este conjunto de dados está disponível gratuitamente online, com as informações dos conjuntos de treino, teste e validação para fins científicos. Todos os métodos funcionam no protótipo final do sistema portátil em tempo real. O sistema desenvolvido é uma abordagem mais barata para as pessoas com deficiência visual que não têm condições para adquirir os sistemas de navegação portáteis mais atuais. As contribuições deste trabalho são: os dois sistemas de navegação portáteis desenvolvidos, os três métodos desenvolvidos para resolver o *Problema da Porta* e o conjunto de dados criado para o treino dos algoritmos de visão computacional. Este trabalho também pode ser escalado para outras áreas. Os métodos desenvolvidos para detecção e classificação de portas podem ser usados por um robô portátil que trabalha em ambientes internos. O conjunto de dados pode ser usado para comparar resultados e treinar outros modelos de redes neuronais para outras tarefas e sistemas.

Palavras-chave Visão computacional, Classificação de objetos 3D e 2D, Segmentação semântica, Pessoas com deficiência visual, Deteção e Classificação de portas, Câmera 3D, Sistema portátil, Deteção de objetos 2D, Conjunto de dados de imagens 3D e 2D, sistemas de baixo consumo energético, tempo real.

Resumo alargado

De acordo com a *Organização Mundial da Saúde* e *A Agência Internacional para a Prevenção da Cegueira* 253 milhões de pessoas são cegas ou têm problemas de visão (2015). 117 milhões têm uma deficiência visual moderada ou grave à distância e 36 milhões são totalmente cegas. Ao longo dos anos, sistemas de navegação portáteis foram desenvolvidos para ajudar pessoas com deficiência visual a navegar no mundo. O sistema de navegação portátil que mais se destacou foi a *white-cane*. Este ainda é o sistema portátil mais usado por pessoas com deficiência visual, uma vez que é bastante acessivel monetáriamente e é sólido. A desvantagem é que fornece apenas informações sobre obstáculos ao nível dos pés e também não é um sistema *hands-free*. Inicialmente, os sistemas portáteis que estavam a ser desenvolvidos focavam-se em ajudar a evitar obstáculos, mas atualmente já não estão limitados a isso. Com o avanço da visão computacional e da inteligência artificial, estes sistemas não são mais restritos à prevenção de obstáculos e são capazes de descrever o mundo, fazer reconhecimento de texto e até mesmo reconhecimento facial. Atualmente, os sistemas de navegação portáteis mais notáveis deste tipo são o *Brain Port Pro Vision* e o *Orcam MyEye system*. Ambos são sistemas *hands-free*. Estes sistemas podem realmente melhorar a qualidade de vida das pessoas com deficiência visual, mas não são acessíveis para todos. Cerca de 89% das pessoas com deficiência visual vivem em países de baixo e médio rendimento. Mesmo a maior parte dos 11% que não vive nestes países não tem acesso a estes sistema de navegação portátil mais recentes.

O objetivo desta dissertação é desenvolver um sistema de navegação portátil que através de algoritmos de visão computacional e processamento de imagem possa ajudar pessoas com deficiência visual a navegar no mundo. Este sistema portátil possui 2 modos, *Generic Obstacle Mode* e *Door Problem Mode*. O primeiro serve para evitar colisões com obstáculos e o segundo para solucionar problemas específicos de pessoas com deficiência visual como o *Problem da Porta*. Também era um objetivo deste projeto melhorar continuamente este sistema com base em feedback de utilizadores reais, mas devido à pandemia do *COVID-19*, não consegui entregar o meu sistema a nenhum utilizador alvo. O problema específico mais trabalhado nesta dissertação foi o já referido *Problema da Porta*, ou em inglês, *The Door Problem*. Este é, de acordo com as pessoas com deficiência visual e cegas, um dos problemas mais frequentes que geralmente ocorre em ambientes internos onde vivem outras pessoas para além do cego. As pessoas com deficiência visual batem com a testa na esquina da porta se a mesma for deixada entreaberta. Com portas fechadas ou totalmente abertas não há problema mas com portas entre-abertas as pessoas antes de chegarem ao manipulo da porta batem contra a mesma com a cabeça. Outro problema das pessoas com deficiência visual também abordado neste trabalho foi o *Problema nas escadas*, mas devido à raridade da sua ocurrência, foquei-me mais em resolver o problema anterior. Este problema é raro de ocorrer porque só acontece em ambientes desconhecidos e geralmente nestes ambientes os cegos andam acompanhados com as suas *white-cane* e então facilmente poderão detetar escadas, sejam elas a descer

ou a subir à sua frente.

Ao fazer uma revisão dos métodos que os sistemas portáteis de navegação mais recentes usam, descobri que os mesmos se baseiam em algoritmos de visão computacional e processamento de imagem para fornecer ao utilizador informações descritivas acerca do mundo. Também estudei o trabalho do *Ricardo Domingos*, aluno de licenciatura da UBI, sobre, como resolver o *Problema da Porta* num computador desktop. Este trabalho contribuiu como uma linha de base para a realização desta dissertação e foi nele que começei a trabalhar.

Esta dissertação está organizada em 5 capítulos.

O primeiro capítulo diz respeito à introdução da dissertação, bem como à contextualização, objectivos e motivações da mesma. São descritos dois problemas típicos das pessoas invisuais que já foram referidos, o problema das portas e o das escadas. Em cada problema são descritas e apresentadas as situações de perigo e as situações sem riscos. É neste capítulo que está descrita a organização deste documento.

O segundo capítulo é dedicado a conceitos fundamentais utilizados neste projeto e ao estudo de trabalhos relacionados com este. São descritos algoritmos de visão computacional utilizados nesta dissertação, tais como, segmentação semântica, deteção de objetos, classificação de imagens 2D e 3D. Existem 3 tipos de estudo relacionado com o meu trabalho. O primeiro diz respeito aos sistemas de navegação para pessoas com deficiência visual. O segundo diz respeito a todos os métodos para deteção e classificação de portas. O terceiro é o trabalho do *Ricardo Domingos* que como já foi dito, funcionou como um ponto de partida para o meu trabalho.

O terceiro capítulo descreve todo o material utilizado neste projeto, tanto a nível de hardware como de software, visto que este trabalho envolveu estas duas vertentes. É descrito o computador de secretária que utilizei para treinar e testar os métodos de visão computacional assim como os computadores de placa única que utilizei para construir os dois protótipos do sistema portátil. Os computadores que utilizei foram o *Raspberry Pi 3 B+* e o *Jetson Nano*. São também descritos outros componentes dos sistemas portáteis, como a câmara que utilizei para capturar as imagens e a powerbank. Por último, são descritos os dois sistemas de navegação (versão 1 e 2) que desenvolvi assim como o funcionamento do interface de utilizador de cada um.

O quarto capítulo descreve a base de dados criada para treinar os algoritmos de visão computacional para serem usados pelo sistema portátil. É descrito o programa que criei para guardar imagens através do sistema portátil versão 1.0 assim como alguns detalhes do posicionamento da câmara. A Base de dados está dividida em 2 grandes grupos, uma parte com imagens 2d e 3d de portas e a outra parte com imagens de escadas. Para além

disso, a base de dados das portas, como foi mais trabalhada têm ainda sub-divisões dependendo da entrada algoritmo de visão computacional que se quer usar: classificação de imagens 2d e 3d, deteção de objetos e segmentação semântica. É também feita uma comparação da base de dados com conjunto de dados utilizados e desenvolvidos noutros trabalhos relacionados em relação ao número de amostras e ao tipo de dados (2d ou 3d).

O quinto capítulo diz respeito a todo o trabalho experimental e testes que fui fazendo aos sistemas portáteis e aos métodos de deteção e classificação de portas para resolver o problema da porta. Primeiro descrevo a minha implementação do trabalho do Ricardo Domingos assim como as suas vantagens e desvantagens. De seguida descrevo os algoritmos que começei a utilizar para desenvolver o primeiro método para o problema das portas. Todos os problemas e dificuldades porque passei até chegar à proposta dos dois primeiros métodos para resolução do problema das portas são descritos neste capítulo. É descrita a montagem do protótipo do sistema portátil final assim como as instalações de software que precisaram de ser feitas e os sistemas operativos utilizados. São descritos e comparados os 3 métodos que desenvolvi para classificação e deteção de portas.

O último capítulo descreve as contribuições científicas deste trabalho e faz uma análise geral dos 3 métodos desenvolvidos para abordar o problema das portas. As contribuições de cada método e suas vantagens e desvantagens são descritas neste último capítulo. No fim deste capítulo faz-se também uma perspectiva do que ficou por fazer e do trabalho futuro.

Abstract

According to the *World Health Organization* and the *The International Agency for the Prevention of Blindness*, 253 million people are blind or vision impaired (2015). One hundred seventeen million have moderate or severe distance vision impairment, and 36 million are blind. Over the years, portable navigation systems have been developed to help visually impaired people to navigate. The first primary mobile navigation system was the *white-cane*. This is still the most common mobile system used by visually impaired people since it is cheap and reliable. The disadvantage is it just provides obstacle information at the feet-level, and it isn't hands-free. Initially, the portable systems being developed were focused in obstacle avoiding, but these days they are not limited to that. With the advances of computer vision and artificial intelligence, these systems aren't restricted to obstacle avoidance anymore and are capable of describing the world, text recognition and even face recognition. The most notable portable navigation systems of this type nowadays are the *Brain Port Pro Vision* and the *Orcam MyEye system* and both of them are hands-free systems. These systems can improve visually impaired people's life quality, but they are not accessible by everyone. About 89% of vision impaired people live in low and middle-income countries, and the most of the 11% that don't live in these countries don't have access to a portable navigation system like the previous ones.

The goal of this project was to develop a portable navigation system that uses computer vision and image processing algorithms to help visually impaired people to navigate. This compact system has two modes, one for solving specific visually impaired people's problems and the other for generic obstacle avoidance. It was also a goal of this project to continuously improve this system based on the feedback of real users, but due to the pandemic of *SARS-CoV-2 Virus* I couldn't achieve this objective of this work. The specific problem that was more studied in this work was the *Door Problem*. This is, according to visually impaired and blind people, a typical problem that usually occurs in indoor environments shared with other people. Another visually impaired people's problem that was also studied was the *Stairs Problem* but due to its rarity, I focused more on the previous one. By doing an extensive overview of the methods that the newest navigation portable systems were using, I found that they were using computer vision and image processing algorithms to provide descriptive information about the world. I also overview *Ricardo Domingos's* work about solving the *Door Problem* in a desktop computer, that served as a baseline for this work.

I built two portable navigation systems to help visually impaired people to navigate. One is based on the *Raspberry Pi 3 B+* system and the other uses the *Nvidia Jetson Nano*. The first system was used for collecting data, and the other was the final prototype system that I propose in this work. This system is hands-free, it doesn't overheat, is light and can be carried in a simple backpack or suitcase. This prototype system has two modes, one that works as a car parking sensor system which is used for obstacle avoidance and the other is

used to solve the *Door Problem* by providing information about the state of the door (open, semi-open or closed door). So, in this document, I proposed three different methods to solve the *Door Problem*, that use computer vision algorithms and work in the prototype system. The first one is based on 2D semantic segmentation and 3D object classification, it can detect the door and classify it. This method works at 3 FPS. The second method is a small version of the previous one. It is based on 3D object classification, but it works at 5 to 6 FPS. The latter method is based on 2d semantic segmentation, object detection and 2d image classification. It can detect the door, and classify it. This method works at 1 to 2 FPS, but it is the best in terms of door classification accuracy. I also propose a Door dataset and a Stairs dataset that has 3D information and 2d information. This dataset was used to train the computer vision algorithms used in the proposed methods to solve the *Door Problem*. This dataset is freely available online for scientific proposes along with the information of the train, validation, and test sets. All methods work in the final prototype portable system in real-time. The developed system it's a cheaper approach for the visually impaired people that cannot afford the most current portable navigation systems. The contributions of this work are, the two develop mobile navigation systems, the three methods produce for solving the *Door Problem* and the dataset built for training the computer vision algorithms. This work can also be scaled to other areas. The methods developed for door detection and classification can be used by a portable robot that works in indoor environments. The dataset can be used to compare results and to train other neural network models for different tasks and systems.

Keywords

Computer vision, Visually impaired people, 3D object classification, Semantic segmentation, Object classification, Door detection and classification, Object detection, 3D camera, Portable system, 3D image dataset, real-time, low powered devices.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

API Application Programming Interface

AGI Artificial General Intelligence

ML Machine Learning

GPU Graphics Processing Unit

NN Neural Network

CNN Convolutional Neural Network

PCL Point Cloud Library

PCD Point Cloud Data

IoU Intersection over Union

mIoU Mean Interface over Union

LR Learning Rate

BS Batch Size

CVAT Computer Vision Annotation Tool

RGB Red Green Blue

SDK Software development kit

ROI Region of interest

FOV Field of view

FPS Frames per second

Chapter 1

Introduction

In this chapter, a framework is made about visually impaired people in the entire globe. It's also presented the motivations and the goal of this project. Through feedback from identical plans, it's given the typical two problems that visually impaired people usually have in indoor environments.

1.1 Framework

Globally, it is estimated that around 285 million people are visually impaired, and 39 million of those people are blind. This means that 0.5 % of the entire world population is blind, and 4.2 % are visually impaired people. The majority of visually impaired people (more than 80 %) are over the age of 50

According to the *World Health Organization*, visually impaired people are three times more likely to be unemployed, suffer from depression, be involved in a motor vehicle accident and two times more likely to fall.

Most visually impaired people walk and navigate with what's called a **white cane**. There are several variants of it, but basically, a white cane is a device that is used to scan the surroundings for obstacles or orientations marks at the feet level. Unfortunately, this is the only device that visually impaired people typically have to help them navigate but, thanks to computer vision and artificial intelligence, several portable systems were built that help people with this kind of disability to navigate in indoor and outdoor environments.

1.2 Goals

Visually impaired people have several problems navigating in indoor spaces, even in their own homes, where they usually don't also use their white-cane. With the advance of computer vision is possible to increase the life quality of these people, and that is where the objective of this project is inserted.

The goal of this project is to build a portable system that integrates computer vision algorithms as semantic segmentation, object detection, and image classification to help visually impaired people navigate safely in indoor environments. This mobile system has two modes, one for solving specific visually impaired people's problems and the other is a

generic obstacle avoiding system. It's also a goal of this project to continuously improve this system based on the feedback of real users.

1.3 Motivations

The motivation of this project is to improve visually impaired people life quality by building this portable system. To help to create an inclusive and democratic society where everyone, regardless of their physical and mental condition, can have access to a good quality of life. It's to enhance equality and freedom for all people, including the blind.

1.4 Visually impaired people indoor problems

Through the UBI Optics Center and previous projects of the same nature as this project, we know that visually impaired people have two major problems in indoor environments. The *Door Problem* and the *Stairs Problem*. Each of the problems will be explained in the following subsections as well as in which conditions each of the problems happen and why.

1.4.1 Door Problem

The ***Door Problem***, as the name itself implies, is related to doors in indoor environments. Visually impaired people don't have problems with totally closed or open doors but have with semi-open doors, especially with semi-open doors that open inwards. Visually impaired people tend to hit with their heads in the edge of the door when the same is semi-open, and that's the *Door Problem*. Figure 1.1 represents the dangerous and not dangerous cases for the *Door Problem*.

Figure 1.1: Door Problem - dangerous and non-dangerous situations.

But why this happens? Doesn't the most of visually impaired people use a white cane? Yes, most of the visually impaired use a white cane **in outdoor environments** but usually they don't use it in their homes, and because of that, they cannot use it to prevent

these accidents. If this problem happens in their homes, it has an easy solution, the visually impaired people just need to always leave the door open or closed it. That would be the solution if visually impaired people lived alone, which is not the case for most of these people. They don't live alone and can even live in an elderly home and the other people without noticing and being on purpose leave the door semi-open, and the accidents happen.

Summing up, the *Door Problem* usually happens in the visually impaired people house or elderly house and the reasons are the absence of the white cane and because they don't live alone. Even for the visually impaired people who use the white cane, if they just had a portable system that would inform then about the door being semi-open, they could avoid the problem like they would be able to avoid with the white cane, but they keep their hands free.

This was the problem that was most worked on in this project. A big Door dataset and three different methods were developed for approaching this problem. The dataset and the methods built will be explained in later chapters of this thesis.

1.4.2 *Stairs Problem*

The other problem that was informed to me through feedback that visually impaired people usually have in indoor spaces is the ***Stairs Problem***. These problems as the name implies, are related to stairs in indoor spaces. When compared with the previous issue, the *Stairs Problem* is more dangerous, but it is also less frequent. Both upstairs and downstairs are dangerous, as figure 1.2 shows.

Figure 1.2: *Stairs Problem* - dangerous situations.

This problem happens in unknown places of visually impaired people when they are not familiar with the space where they are. Unlike the previous case, this problem doesn't occur in their home because they know every detail of it, so they know the locations of the

stairs if there are any of them in their home. This problem happens because, for some reason, the visually impaired people aren't using their white cane and so they aren't able to avoid this accident, but even with the white cane this accident can happen if they were in a hurry.

1.5 Document Organization

This report is organised in the following way:

- **Fundamental Concepts and Related Work** - In this section are addressed fundamental concepts of computer vision and artificial intelligence that were used in this project as well as several related works.
- **Project Material** - This section treats all the equipment that were used for this project as well as the portable systems.
- **Database** - In this section are described the datasets built to help to solve visually impaired people's problems and how they were built.
- **Experiments and Discussion** - This section treats all the results and experiments done in this project as well as all the problems that I had for building the final prototype portable system.

Chapter 2

Fundamental Concepts and Related Work

In this chapter will be described several fundamental computer vision concepts that were used in this project such as, 2D semantic segmentation, 2D object detection, 3D and 2D object classification, point clouds and others. All related works and systems that I studied in this project will also be covered in this chapter, divided into three sub-sections. One for the already built system that helps visually impaired people to navigate, other for the *Door Problem* approach related works and another for *Ricardo's work*.

2.1 Computer vision concepts used in this project

This section will cover all the computer vision concepts that were used for the software part of the portable system for visually impaired people.

2.1.1 Point Cloud

In this project, I used a 3D camera, (Realsense Model D435) which will be described in the next chapter, and so, 3D information was used in the methods for solving the visually impaired people problems. The camera can capture RGB and 3D information (depth) and returns this 3D data in the form of a 2D grey-scaled image (e.g. $640 * 480$). This image has in total 307200 pixels ($640 * 480 = 307200$), and each of these pixels has its depth value which corresponds to the grey-scaled value.

This 3D information, instead of the grey-scaled image, can be represented in a **point cloud**. A point cloud is a set of points expressed in a three-dimensional coordinate system. For the previous example, this point cloud would have 307200 points. Each of the points can be represented by X, Y and Z coordinates if we are talking an about a no-colour point cloud, but if the point cloud has colour, each point gets more three coordinates, R, G and B which correspond to the RGB colours. For this project, It was only used point clouds with no colour, where the X and Y correspond to the location of the pixel in the 2D grey-scaled image and the Z coordinate corresponds to the depth value. The colour information was also used, not in a point cloud but the form of a 2D image, for semantic segmentation, object detection and the image classification method.

2.1.2 Algorithms used for the *Door/Stairs Problem*

Figure 2.1 represents the base architecture for all the computer vision algorithms that were used on the methods that approached the *Door and Stairs Problem*. For every algorithm, it's represented its output and input. These concepts are essential to understand each technique that was developed for the visually impaired portable system.

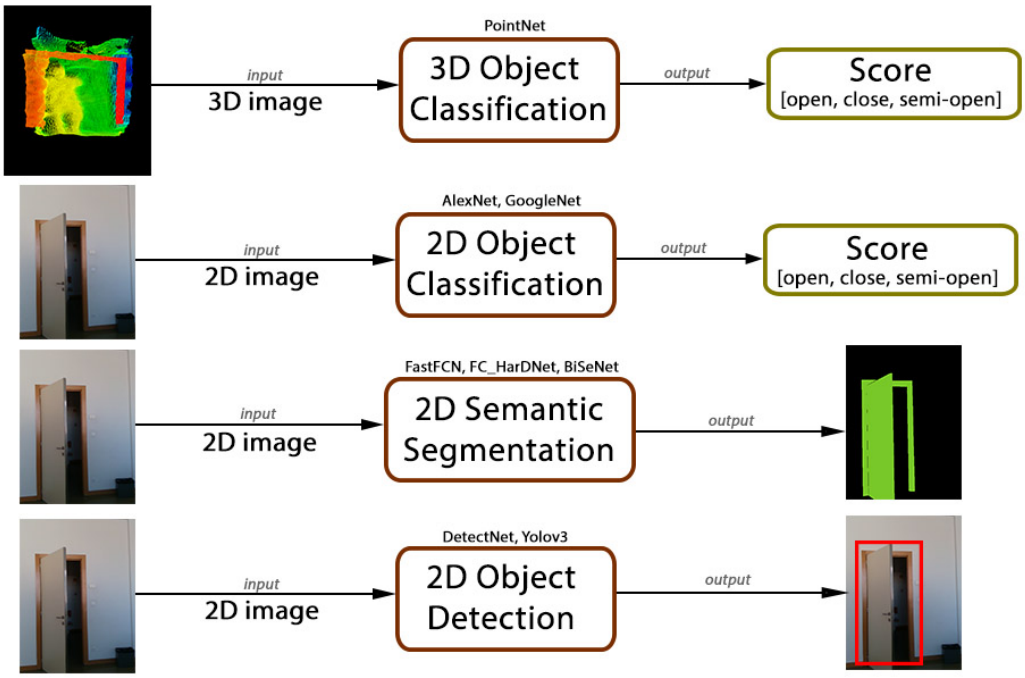

Figure 2.1: Computer Vision algorithms architectures used in this project with inputs and outputs (Examples of *Door Problem*) .

Starting with the first computer vision algorithm in figure 2.1, we have the **3D Object Classification**. As can be seen, the input of this algorithm is a 3D image, which can be represented in a 2D grey-scaled image or a point cloud as it was explained in the previous section. This algorithm receives this 3D image and returns the score value for each class of the problem. For the *Door Problem*, which is the example of the figure, it returns three values between 0 and 1. Each of these values corresponds to the model confidence value of each class, open, close or semi-open. The 3D Object Classification models can use RGB information or not. In the case of this project, it was just used the *PointNet*, [QSMG16] which doesn't use RGB information, it only uses a point cloud with no-colour.

In the second column of figure 2.1, we have the **2D Object Classification**. As the name itself implies, it's very similar to the previous algorithm with only the difference in the input. Instead of using a 3D image, these algorithms use 2D normal images (RGB). The type of output is the same, a score with three values because it's approaching a problem with three classes. The algorithms of 2D Object Classification used in this project were the *GoogleNet*, [SLJ+14] and *AlexNet*, [KSH12].

The third algorithm in figure 2.1 is the **2D Semantic Segmentation**. This algorithm has the same input as the previous one, a 2D RGB Image. This algorithm was used to detect/segment the door, to know its location on the original image which is provided by the camera. The output of these models is a 2D grey-scaled image with the same size as the input image. In the figure the output image is represented in RGB just to be more clear, but the output image is normally a 2D grey-scaled image. For these algorithms, for the *Door Problem*, we have two classes, one corresponds to the door and doorframe, and the other corresponds to all the other objects. The model returns each prediction, and in the output image, the black pixels (value = 0) correspond to "all the other objects" class and the green ones (value = 1 in grey-scaled) corresponds to the pixels of "door-doorframe" class. The algorithms of 2D Semantic Segmentation used in this project were the *FastFCN*, [WZH+19] the *FC-HarDNet*,[CKR+19] and the *BiSeNet*,[YWP+18].

The last computer vision algorithm in figure 2.1 is the **2D Object Detection**. This algorithm is very similar to the *2D Semantic Segmentation* because both of them want to get information about the location of the object, in this case of the door. The *2D Object Detection* uses a 2D image as input and returns all the bounding boxes (red rectangle in the figure), and it's confidence value for each object detected in the image. Normally it just shows the object values with a confidence value superior to 0.7. The bounding box is returned in the form of 4 values, which correspond to the top-left and bottom-right pixels coordinates. The algorithm of *2D Object Detection* used in this thesis were the *DetectNet*, [ATS16] and the *YoloV3*, [RF18].

2.2 Related Work

This section describes all the related work of this project. The related work is divided into three big categories, navigation system for visually impaired people, algorithms for door detection and classification, and a particular UBI student's work.

2.2.1 Navigation systems for visually impaired people

Nowadays, there are already several systems to help visually impaired people to navigate. Some use ultrasonic sensors, and older technologies and others use more current methods, as computer vision algorithms and artificial intelligence. This sub-section will approach each navigation system studied for this project in order from the oldest to the most modern one.

White-Cane (1921)

The most common navigation system that visually impaired people use is the **white-cane**, as it was already mentioned in this report. This tool was invented in 1921 by *James Biggs*. This cane is white for two reasons, to be more visible and to have more impact on other people so they can easily associate that the person that is using it is blind or visually impaired. The advantage of this tool is that it is a simple tool that everyone can afford,

and it is reliable at finding obstacles and possible dangers at the foot level. The disadvantage of the white-cane is that the foot-level is not enough to avoid all the obstacles. For example the barriers at the head-level which aren't at the foot-level, like a tree branch or a hanging sign.

Figure 2.2: White-Cane

Electronic Travel Aids - ETA

The **Electronic Travel Aid** is an electrical obstacle detection device and a form of assisted technology to help visually impaired people to navigate. These devices are more focused on providing obstacle avoidance support than information about the world. Several devices were developed over the years. One example of a sonar-based ETA is the *Bat K Cane*, [HW08]. This unit fits into a standard white cane and radiates ultrasonic waves. The echos from the objects return to the sonar unit and then are converted into a unique sound-based image of the landscaped that gets transmitted to a set of headphones. Other similar examples are the *UltraCane* and the *MiniGuide*. The *UltraCane* is a modified built-in sonar cane and the *MiniGuide* is a hand-held device which makes use of vibrations to provide the visually impaired person with information.

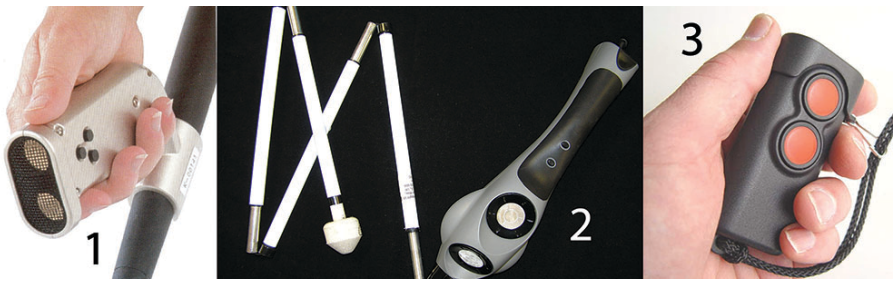

Figure 2.3: Electrical obstacle detection devices (1-*Bat K Sonar Cane*, 2-*UltraCane*, 3-*MiniGuide*)

There were also developed systems that made use not just of ultrasound but position to guide blind and visually impaired people to a nearby destination with obstacle avoidance such as the *GuideCane* and the *NavBelt*.

Figure 2.4: Electrical obstacle detection devices that use ultrasound (1-*NavBelt*, 2-*GuideCane*

With the arrival of global navigation satellite systems, specifically, the GPS (Global Positioning System) new devices that use this system developed such as the *UCSB Personal Guidance System*. This system is a GPS-based portable device, which can lead the user on an outdoor route. This system doesn't provide any obstacle avoidance support, but it was developed to be used as a complement to the white cane where this already support obstacle avoidance.

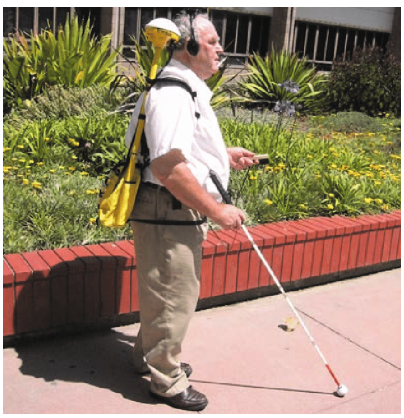

Figure 2.5: *UCSB Personal Guidance System*

Sonar sounds (2000)

Daniel Kish, [TRZ$^+$17], the president of the *Visioneers* which is a Division of World Access For The Blind, and a visually impaired person, uses **sonar sounds** to navigate in the world. This person makes clicks with his tongue, which are flashes of sound that go out and reflect from surfaces all around him. It works just like a bat sonar. The sounds return to him with patterns and pieces of information. Almost every visually impaired person can use this method, but it requires training. The significant advantages of that this method is that, combined with the white-cane it can be beneficial for visually impaired people to obstacle avoiding because it covers bot feet and head-levels.

Figure 2.6: *Daniel Kish*

Systems that use 3D information(2004)

One of the first system to help visually impaired people navigate that used 3D information was the *ENVS* project. This system, instead of using sound to give information about obstacle avoiding it uses haptics sensors, more precisely, special gloves fitted with electrodes for delivering electrical pulses to the fingers. In this project, they make use of a pair of cameras to get the 3D information and present that information to the user through the electrical pulses. The disadvantage is that several persons aren't willing to wear gloves.

Figure 2.7: *ENVS* Project system

Smartphones applications

With more and more powerful mobile phones and better cameras, new applications for visually impaired people were developing for these devices. One example of these systems is the *NavCog* smartphone application. This app was built to establish for indoor navigation for visually impaired people. Still, it can also be used by people who are in an unknown complex indoor place such as a university or an airport. This user interface of this application is 100% sound interface, where the visually impaired select the destination by using voice search, and the app provides turn-by-turn audio feedback.

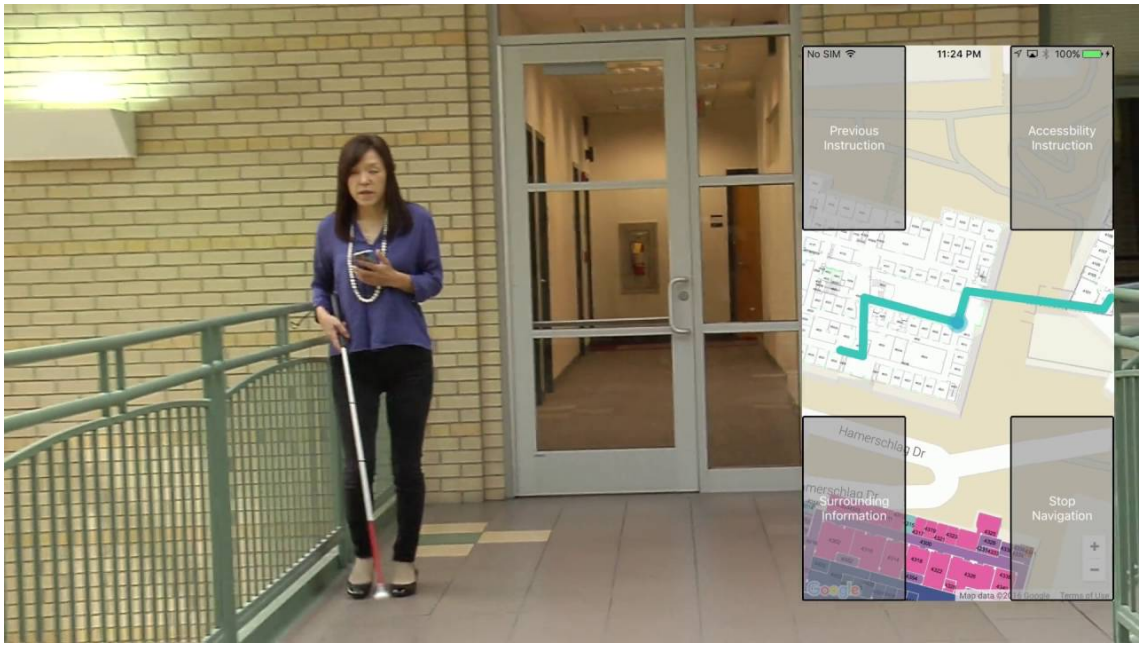

Figure 2.8: *NavCog* application system

Another smartphone application that was developed to help visually impaired people navigate was the *HamsaTouch* application. The *HamsaTouch* is a novel tactile vision substitution system which is composed of a smartphone, photo-transistors and an electro tactile display. The smartphone extracts the edges, and the information is converted into a tactile pattern.

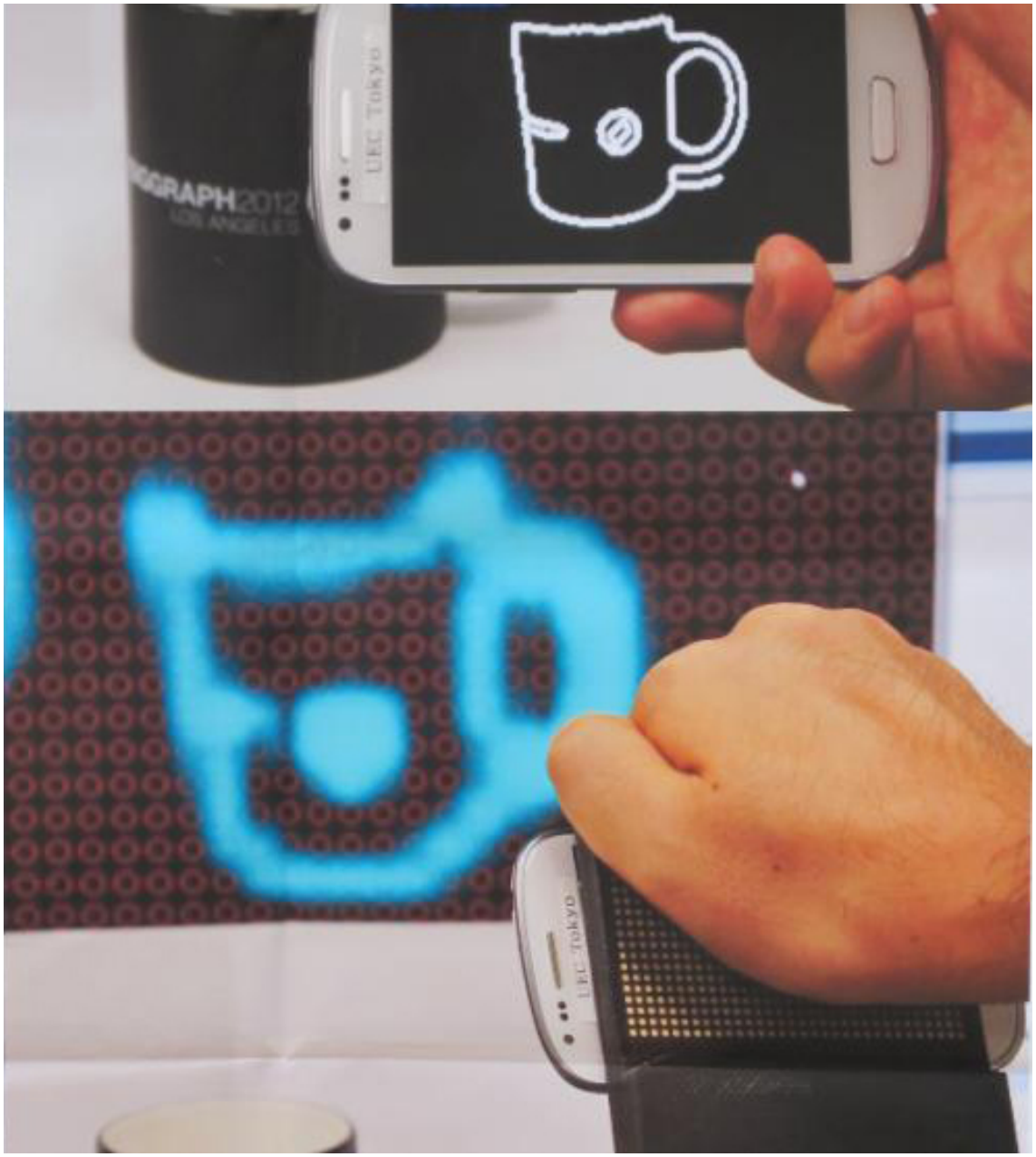

Figure 2.9: *HamsaToush* application system

Systems based on Image Processing (AI)

The most current systems use artificial intelligence and computer vision to help visually impaired people to navigate. There are two types of systems that use AI, portable systems(wearable) and smartphones applications.

For the smartphone applications there is the *Seeing AI* and the *TapTapSee*. These applications describe images from the real world through an audible interface by making use of remote processing resources in a cloud computing server. The *Seeing AI* was developed by *Microsoft* and in addition to image descriptions, it can recognise text, products and persons. The *TapTapSee* is more focused on describing images, but, it is also able to describe small videos.

Figure 2.10: Smartphone applications based in Computer Vision (1-*TapTapSee* and 2-*Seeing AI*)

The other systems that use AI to help navigate visually impaired people are the wearable portable systems. The big advantages of using these systems are the wider sensors field-of-view, discreet system and hands-free solution. Examples of these kind of system are, the *Tyflos*, *Orcam MyEye* and *BrainPort Vision* systems.

The *Tyflos* systems is constituted by a pair of sunglasses with two tine cameras mounted on it, a range sensor, a GPS device, an RFID reader, a microphone, an ear-speaker, a portable computer and a vibration array vest. This device is used to read a text and for navigation purposes.

Figure 2.11: *Tyflos* system

The *BrainPort Vision* is constituted by a small camera, a pair of sunglasses, a controller box that converts the video signal of the camera into an electro tactile signal. These signals stimulation patterns on the surface of the tongue like moving ”bubble-like patterns” as the users of this system described.

Figure 2.12: *BrainPort Vision Pro* system

The *Orcam MyEye* system was created in 2015 and its most recent version in 2017. This last version is capable of text reading, face recognition and product recognition with a user-interface base on hands movements and gestures. One of the most significant advantages of using this system is that it is very portable and can be mounted in a regular pair of glasses.

Figure 2.13: *Orcam MyEye* system

The advantage of using systems based on Image Processing and Artificial Intelligence is that they can provide not only obstacle avoiding information but information about the world. These systems are capable of describing the world and providing that information to the visually impaired person.

2.2.2 Related work (Door classification and detection) *Door Problem*

There are already a vast number of studies that used door detection and classification for robot navigation tasks as moving between rooms, robotic handle grasping and others. Some have used sonar sensors with visual information, [MLRS02], others used only colour and shape information, [CDD03], some have used simple feature extractors, such as [KAY11], [ZB08] and others have used more modern methods like CNN (Convolutional neural networks), [LRA17] and the use of 3D information, [YHZH15], [QPAB16], [MSZW14] and [QGPAB18]. Of course that the most of these studies and systems can be used to help visually impaired people to navigate but since there were no articles that would do door classification and detection with the propose to help visually impaired people I focused on studying the robotic application methods for door detection and classification.

Using visual information and ultrasonic sensors to traverse doors was an approach used in [MLRS02]. The goal was to cross an open door with a certain opening angle using a B21 mobile robot equipped with a CCD camera sensor and 24 sonar sensors. The door traverse

was divided into two sub-tasks, the door identification and the door crossing. The door identification, which was the sub-task of interest for this work, used a vertical *Sobel* filter applied to the grey-scaled image. If there were a column more extensive than 35 pixels in the filtered image, it would mean that the door was in the picture. The sonar sensors were used when the robot approached the door at a distance of 1 meter to confirm if it was a door or not.

In [KAY11], an integrated solution to recognise a door and its knob in an office environment using a humanoid platform is proposed. The goal is for the humanoid to recognise a closed-door and its knob, open the same door and pass through it. To recognise a door, they match the features of the input image with the features of a reference image using the STAR Detector [SDor] as the feature extractor and an on-line randomised tree classifier to match the feature points. If the door is in the scene, the matched feature 3D points are computed and used so that the robot walks towards the door.

The use of colour and shape information can be sufficient for identifying features to detect doors efficiently. The approach in [CDD03] used two neural networks classifiers for recognising specific components of the door. One was trained for recognising the top, left, and the right bar of the door and the other was trained for detecting the corners of the door. A door is detected if at least 3 of these components are recognised and have the proper geometric configuration.

In [LRA17], a method is implemented for detecting doors/cabinets and its knobs for robotic grasping using a 3D Kinect camera. It uses CNN to recognise, identify and segment the region of interest in the image. The CNN used was the *YOLO* Detection System trained with 510 images of doors and 420 of cabinets from the *ImageNet* dataset. After obtaining the Region of interest (ROI), the depth information from the 3D camera is used to get handle point clouds for robot grasping.

Like the previous approach, in [YHZH15], a Kinect sensor is used for door detecting, but, this method uses only depth information. The camera sometimes produces missing points in the depth image, and the algorithm is based in the largest cluster of missing pixels in the depth image. The total number of holes indicates the status of the door (open or semi-open). The main advantage of this method is that it works with low-resolution depth images.

There are methods developed under a 6D-space framework, like [QPAB16], that use both colour (RGB) and geometric information (XYZ) for door detection. For detecting open doors, they identify rectangular point cloud data gaps in the wall planes. The detection of closed doors is based in the discontinuities in the colour domain and in the depth dimension. It also does door classification between open and closed doors. The improved version of this algorithm, [QGPAB18], can even distinguish semi-open doors using the

set of points next to the door to calculate the opening angle. Another improvement in [QGPAB18] was in the dataset, which is larger in size, complexity and variety.

In [MSZW14], a method is proposed that uses 3D information for door detection without using a dependent training-set detection algorithm. Initially, the point cloud containing all the scene, including the door, is pre-possessed using a voxel-grid filter to reduce its density and its normal vectors are calculated. A region growing algorithm based on the pre-calculated normals is used to separate the door plane from the rest of the point cloud, and after that, feature extraction is used to get the edges of the door and the doorknob.

To detect doors 3D cameras or sonar sensors are not required, a simple RGB camera can do the job as in [ZB08], focusing on real-time, low-cost and low-power systems. This work used the *Adaboost* algorithm to combine multiple weak classifiers into a robust classifier. The weak classifiers were based in features such as detecting pairs of vertical lines, detecting the concavity between the wall and the doorframe, texture and colour and others. They built a dataset with 309 door RGB images, 100 for training their algorithm and the rest for testing.

Table 2.1 summarises the previous approaches and related work to detect and classify doors in indoor spaces, categorising each method studied. Although most of the strategies just do door detection and not classification, as I did for the *Door Problem* in this work, they have a similar goal, to provide the robot with the necessary information to move between rooms, and that is the reason why I included them in this work. The first column states whether the method uses 3D information or not. The following three columns indicates the applicability of the method (closed, open or semi-open doors). The last column focus on whether the method works in real-time or not, based on the experimental results of each technique. Four of the methods do not present information regarding their speed and are marked with a ”-”.

Table 2.1: Related work comparison (door detection).

Method	3D	Closed doors	Open doors	Semi-open doors	Real-time
Monasterio [MLRS02]	×	×	✓	×	-
Cicirelli [CDD03]	×	✓	×	✓	×
Kwak [KAY11], Chen, [ZB08]	×	✓	×	×	✓
Llopart [LRA17]	✓	✓	✓	✓	✓
Yuan [YHZH15]	✓	×	✓	✓	-
Quintana [QPAB16]	✓	✓	✓	×	-
Borgsen [MSZW14]	✓	✓	×	×	×
Quintana [QGPAB18]	✓	✓	✓	✓	-
Method A - *Door Problem*	✓	✓	✓	✓	✓
Method B - *Door Problem*	✓	✓	✓	✓	✓
Method C - *Door Problem*	×	✓	✓	✓	✓

2.2.3 Ricardo Domingos's work - *Door Problem* method

Bachelor final project of *Ricardo Domingos* was **Artificial Vision for Blind People**. His work was also to build a portable system that helps impaired visually people day by day, using semantic segmentation and computing vision algorithms. Ricardo's work was important in this project because it had the roots for the construction of the portable system, and it also had information about all the computer vision algorithms that he used. The report also has all the difficulties that he went through and the main problems that visually impaired people have in indoor spaces, *Door Problem* and *Stairs Problem*.

Ricardo's work was focused on solving the *Door Problem* for visually impaired people. Although the goal of his project was to build a prototype system, he didn't build it and simply used a portable computer with a 3D camera. The following figure shows Ricardo's proposal to solve the *Door Problem*.

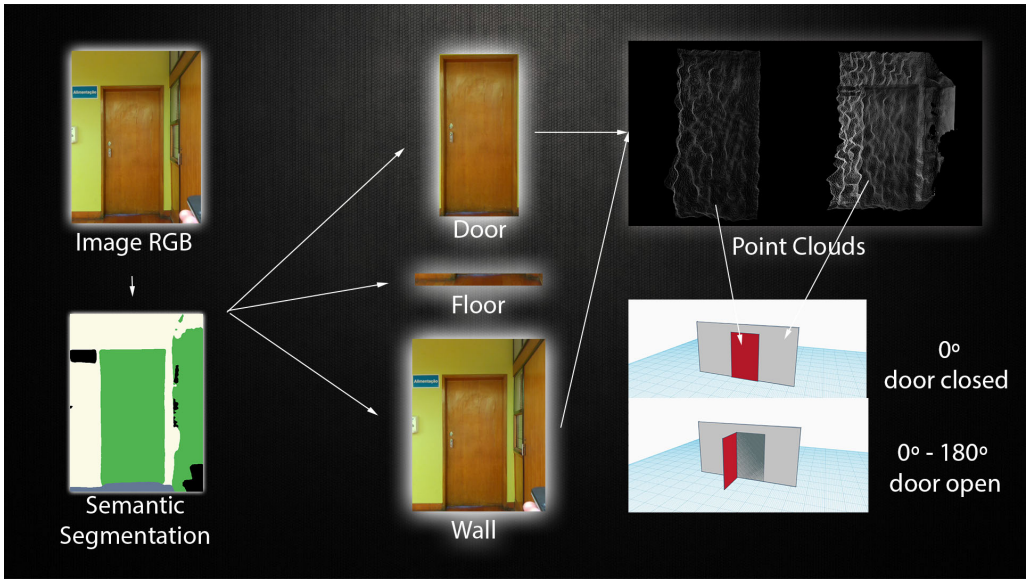

Figure 2.14: Ricardo's proposal to solve the *Door Problem*

According to his proposal, the first step was to get the RGB and Depth image from the 3D camera. The next step was to use semantic segmentation algorithms of 4 different classes, "floor", "wall", "door" and everything else. He was only using these four classes because they are enough for the system to classify if the door is open or closed. Ricardo changed the original colour palette of the ADE20k pre-trained model for semantic segmentation with 150 classes to a palette with only four classes. For the semantic segmentation, it was used the method "context encoding" which has an implementation in PyTorch. After the semantic segmentation, the system would calculate the biggest area for the "wall", "door" and "floor" classes and store the bounding box of those areas. The most significant area in this context its the biggest cluster of pixels of each class. After that, the RGB and depth

images were cropped for each class according to the bounding box. Then, the point clouds are built using the cropped depth and RGB images. For each class, we have a point cloud. To classify if the door is open or closed it was calculated the plan of each point cloud, and after that, it was estimated the angle between the plane of the wall and the plane of the door. If the angle between those two planes were 0º degrees or near that, the system would classify the door as closed. If the angle were bigger than 0º degrees, the system would classify the door as open.

Further details of Ricardo Domingos's work will be addressed in later sections of this thesis as well as how I used his work as a baseline for this project.

Chapter 3

Project Material

In this chapter is described all the hardware and material used indirectly and directly for this project. Each section of this chapter corresponds to specific project material. The equipment for this project was financed by the optic centre and *Socia lab*.

3.1 Lab Desktop Computer

The central computer that was used to perform and run neural networks and computer vision algorithms was the lab desktop computer. This computer was used not just to train the computer vision algorithms but to test them as well to later migrate those algorithms to Jetson Nano since the goal of this project is to perform these algorithms in a low powered and easy to transport device. Several neural networks models were trained and validated in this computer, and if they couldn't make inference in real-time on the desktop computer for sure, they wouldn't run in real-time in *Jetson Nano*.

3.1.1 Description and characteristics

The desktop computer has *Pop!_OS* 18.04 LTS (Linux) installed with 15,7 GiB RAM, Processor *AMD Ryzen* 7 2700 eight-core processor * 16 and Graphics *GeForce GTX* 1080 ti (11175 MiB). It has two disks, an SSD Disk with 487 GiB and an HHD Disk with 3.0 TiB to store the dataset and other files.

3.2 Raspberry Pi 3B+

The first single-board computer used in this project was the *Raspberry Pi* 3 model B+. This device was used in the first portable system to help visually impaired people to navigate. It was the most powerful single board computer that we had in the lab at the start of this project. This device had the characteristics to be used in the portable system since it was compact and lightweight.

Later this device was replaced by *Jetson Nano* which will be described in the next section. This computer was never used to run neural networks or computer vision algorithms despite belonging to the portable system version 1.0. It was used instead to build the datasets that were used to train, validate and test the neural networks models and all the computer vision algorithms used in this project.

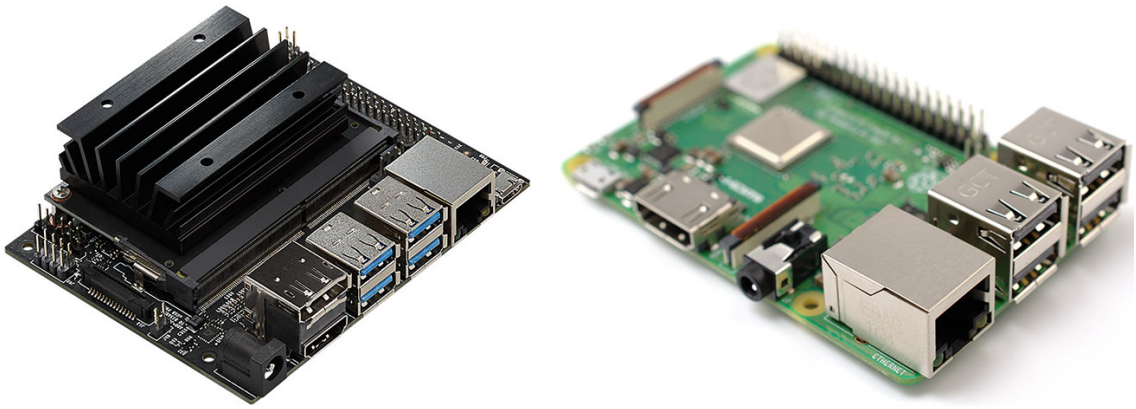

Figure 3.1: Jetson Nano (Left side) and Raspberry Pi 3 Model B+ (right side).

3.2.1 Descriptions and characteristics

The Raspberry Pi 3 Model B+, 3.1, has a RAM 1GB LPDDR2 SDRAM, 1.4GHz 64-bit quad-core processor, a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz and a 5V/2.5A DC power input.

It was used a 16 GiB Micro SDCard XC with the *Raspberry Pi OS* as the operating system.

3.3 Jetson Nano Nvidia

The main single-board computer that was used in this project and in the final version (2.0) prototype portable system was the *Jetson Nano* from *Nvidia*. The big difference between *Jetson Nano* and *Raspberry Pi* is that *Jetson* has as embedded GPU to run computer vision algorithms and neural networks faster. The main disadvantages of this device is that it is a little heavier and bigger in height when compared with the *Raspberry Pi 3 Model B+*.

At the time Socia lab received the *Jetson Nano* it also receive the new version of *Raspberry Pi*, the *Raspberry Pi 4 Model B*. The big reason why I didn't stick with the *Raspberry Pi* franchise and switch to the *Jetson franchise* was because of the embedded GPU of *Jetson Nano*. The *Raspberry Pi 4* also doesn't come with a GPU, and the only way to have one was to use a *USB Coral Pen* or something similar, but it was more comfortable and more intuitive just to use *Jetson Nano*.

Two *Jetson Nano* were used in this project. One was used to test all the computer vision algorithms in the lab, while the other was used to build the portable system and later to get feedback from a real-user, a visually impaired person.

3.3.1 Descriptions and characteristics

Jetson Nano, 3.1 is equipped with a **CPU**, Quad-core ARM A57 @ 1.43 GHz. It has 4 GiB 64-bit LPDDR4 25.6 GB/s of RAM memory and a 128-core Maxwell GPU. For energy supply, it has a USB 2.0 Micro-B and a Barrel Jack port. One of the most outstanding components is Jetson Nano's heatsink. One of the biggest problems of Jetson Nano is that it easily overheats when running neural networks models and using a big percentage of GPU/CPU memory. Later in this report, I will approach this problem and how it was solved (fan installation).

This portable system works in two modes, 5 watts and 10 watts. The default mode is 10 watts, and almost all of the algorithms were tested in this method. In this report, if the Jetson mode isn't referred in a specific experiment, it means that the research was tested in Jetson 10 watts mode.

For the two *Jetson Nanos* it were used two 64 GiB Micro SDCard XC with the Jetpack as the operating system. Initially, both had the Jetpack version 4.2, which was the most recent when I first work with these devices. Then Jetpack version 4.3 came in, and I installed in one of the Jetson this version. Even later in this project, in April, Jetpack version 4.4 was released, with a new version of *TensorRT* and this Jetpack was installed in the Jetson with the Jetpack version 4.2. *TensorRT* will be explained later in this project, but it stands for Tensor Real-Time, and it's a technology that allows running neural networks faster without losing too much precision in the model's output.

3.3.2 Installation

It was developed a Git repository with the big part of the installations in Jetson Nano as well as a guide to those installations and the possible errors. This repository was created with the intention of serving as a backup if I need to install the *Jetpack OS* again or some components, but it also can be used as a guide to new users of *Jetson Nano*. The link to the repository is the following:
`https://github.com/gasparramoa/JetsonNano-CompVision`

This repository isn't restricted to Jetson installations guides, and it also has installations guides for the desktop computer of tools that are used for Jetson as the tool *DIGITS*. This tool will be explained in more detail in the next chapters of the report but is a graphic user interface to train and validate neural network models which, in turn, will be used in *Jetson Nano*. The repository is also one of the ways I used to move files from the desktop computer to *Jetson Nano* and it has all the implementations and methods I develop that worked on *Jetson Nano*.

After installing the specific Jetpack versions for both *Jetson Nanos*, I did several steps and commands (the recommended ones) to prepare Jetson Nano to run computer vision algorithms.

First, I increased the swap memory by 4 GB. This was done because *Jetson Nano* only has 4GB of RAM and neural network models quickly fill up these 4GB of memory.
Several dependencies of deep learning frameworks and libraries were installed:

- *git*
- *cmake*
- *libatlas-base-dev*
- *gfortran*
- *python3-dev*
- *python3-pip*
- *libhdf5-serial-dev*
- *hdf5-tools*
- *numpy*
- *matplotlib*
- *opencv*
- *open3d*
- *torch*
- *torch-vision*
- *setuptools*

It was also necessary to install the *Realsense* tools to work with the 3D *Realsense* camera, D435 which will be explained later in this chapter.
To decrease the memory used for the graphic user interface and since this one would be used for the visually impaired people, the *i3* was installed in *Jetson Nano*. The *i3* is a non-graphic interface, and with it, we are able to load bigger images or have a bigger batch size in the memory.
A big part of the unnecessary start-up programs was removed and disable to increase the seep up of *Jetson Nano's* startup. As I increased the speed up of Jetson's startup, I also added the python-script to the *bashrc*. This allowed the program to start after *Jetson Nano* startup without the need to perform any command or action.

3.3.3 Python libraries version for Jetpack 4.3

This subsection treats the most important python3 libraries that were installed and used in Jetpack version 4.3 for several computer vision algorithms that I tested for Jetson Nano. This information is useful if someone would like to replicate this work or any of the methods I develop with the exact same results. The libraries installed and its versions for the *Jetpack 4.3* were the following:

- **torch** - version 1.0.0

- **torch-vision** - version 0.2.2
- **torch-encoding** - version 1.0.2
- **scipy** - version 1.4.1
- **numpy** - version 1.18.0
- **open3d** - version 0.9.0.0
- **matplotlib** - version 2.1.0
- **jetson-states** - version 1.7.8
- **CUDA** - version 10.0.326
- **TensorRT** - version 6.0.1.10
- **cuDNN** - version 7.6.3.28
- **VisionWorks** - version 1.6.0.500n
- **OpenCV** - version 4.1.1.

3.3.4 Python libraries version for Jetpack 4.4

This subsection, as the previous one, treats the most important python3 libraries that were installed and used in Jetpack version 4.4 for several computer vision algorithms that I tested for Jetson Nano. The libraries installed and its versions for the *Jetpack 4.4* were the following:

- **torch** - version 1.5.0
- **torch-vision** - version 0.2.2
- **scipy** - version 0.19.1
- **numpy** - version 1.13.3
- **jetson-states** - version 2.0.4
- **CUDA** - version 10.2.89
- **TensorRT** - version 7.1.0.16
- **cuDNN** - version 8.0.0.145
- **VisionWorks** - version 1.6.0.501
- **OpenCV** - version 4.1.1

3.4 RealSense 3D camera

The camera used in this project was the *Realsense* camera model D435. This camera is able to capture RGB and depth images. The depth channel with a range up to 10 m. This camera is up to 1280 × 720 active stereo depth resolution and up to 30 fps in the lowest resolutions. The depth channel can go even further and reach 90 fps. It has the dimensions, 90 mm (length), 25 mm (depth), 25 mm (height). It has a USB□C* 3.1 connector, that can also work in 2.0 USB but with limited fps and resolutions as it was used in the Raspberry Pi 3 Model B+.

For the develop dataset and for the visually impaired people, we used the resolution 640 x 480 because it was the most significant resolution that the camera could provide at 30 fps using a 2.0 USB port. The other factor was that if I increased the resolution, the neural network methods inference time would also increase. It's better sometimes to work on lower resolutions but get our methods working in real-time.

This camera was used in the two prototypes portable system that was built for this project, and it was also used to construct the dataset that would train and validate the models for improving the mobile system.

Figure 3.2: 3D Realsense camera Model D435.

The depth channel it reproduces is 2D grey-scaled images with a depth scale equal to 1/16. The RGB image resolution used was 640(width)x480(height) and the depth scale has precisely the same resolution. The depth Field of view (FOV) is 87°±3° × 58°±1° × 95°±3°. The depth channel distance ranges from 0.105 m to 10m.

Using the realsense-viewer, which is available by installing the pyrealsense library and its dependencies, I can provide further details and information about the camera. For the Stereo Module, several parameters can be changed, such as:

- Resolution (From 256 x 144 to 1280 x 800)
- Frame Rate (From 6 to 90)
- Available Streams (Depth, infrared 1 and 2)
- Controls such as Exposure, Gain and Laser Power.
- Depth units
- Post-Processing such as Magnitude and Threshold Filters.

For the RGB Module, there are also parameters that can be changed, such as:

- Resolution (From 320 x 180 to 1920 x 1080)
- Frame Rate (From 6 to 60)
- Color Stream (RGB8, BGR8, Y16)
- Controls such as Brightness, Contrast, Exposure, Saturation and others.
- Post-Processing such as Decimation Filter.

3.5 Power bank 20000 mAh

The power bank used in this project was a dual USB *TECHLINK Recharge power bank* with 20000 mAh. This power bank has one USB Fast-charging capable of enabling a current with 2.4 A. With a voltage of 5 V, with this current, this power bank could provide a power of 12 W, which was more than enough to power up Jetson Nano in 10 W mode.

The dimensions of this power bank are (W) 82 mm (D) 22 mm (H) 160 mm although, this power bank was unmounted for the portable system and its dimensions reduced a little bit in every axis.

3.6 Portable System 1.0

I built two portable systems for helping navigate visually impaired people in indoor environments, version 1.0 and 2.0. The portable system 1.0 its constituted by:

- Raspberry Pi 3 B+ (Single board computer).
- 3D Realsense camera Model D435.
- Power bank TECHLINK Recharge 20000 mAh (power source).
- Smartphone (User-interface).
- In-Ear phones (User-interface).

This portable system was used to build the datasets for the *Door and Stairs Problem*. It was used to construct an image dataset, but it can also save videos. Initial this system was built to be used for the visually impaired people, but as Jetson Nano came in this system was remodelled.

Figure 3.3: Portable System 1.0

For the user-interface, I used a Smartphone which has a hotspot. The Raspberry Pi after the startup it automatically connects to this hotspot. Then, using an SSH application in the Smartphone, the user can communicate with the system. That was how I built the dataset by running the program using SSH communication.

I had a lot of difficulties installing *pyrealsense*(Cross-platform ctypes/Cython wrapper to the librealsense library) on raspberry PI. This library was necessary to use the realsense camera pipeline and get the frames of it through a python script. Unlike the lab desktop computer, I had to install this library manually through the link, `github.com/IntelRealSense/librealsense/blob/master/doc/RaspberryPi3.md`. This last URL is a website specific to install the librealsense in the Raspberry Pi, but it had some errors in it. After several researches, I found the following tutorial, `github.com/IntelRealSense/librealsense/blob/master/doc/installation_raspbian.md`.

One of the dependencies of the librealsense is the opencv library. The installation of this last library is also different in the Raspberry Pi, `https://www.pyimagesearch.com/2017/09/04/raspbian-stretch-install-opencv-3-python-on-your-raspberry-pi/`. Another dependency that was also installed differently than the lab computer was the protocol-buffer library:

`osdevlab.blogspot/how-to-install-google-protocol-buffers`.

If we want to use the *librealsense* it's necessary, in the *Raspberry Pi* system, to have the python executable program in the same directory where is located the *realsense.so* to be able to import this directory.

It's important to say that the *Raspberry Pi 3 B+* is not going to be the final single-board computer to be used in the portable system. After installing all the libraries and its dependencies, the system stayed with only 2.5Gb free disk space. It's still necessary to install several neural network benchmarks and implementations via Tensorflow or PyTorch. The system must have also some free disk space available to store the RGB and depth information. One solution to this problem would be just to use a bigger SD-Card but them *Jetson Nano* came in and so, a new version of this system was built taking into account all of the previous problems.

3.7 Portable System 2.0

In the second version of the portable system, several things changed. This system is now constituted by the following components:

- Jetson Nano (Single board computer)-
- 3D Realsense camera model D435
- Power bank TECHLINK Recharge 20000 mAh (power source).
- Hand (User-interface).
- In-Ear phones (User-interface).
- Box that contains all of these components.

3.7.1 System characteristics

The version of this system was built focusing more on the user-interface with the visually impaired people and on the methods to help their navigation in indoor spaces. This portable system, unlike the previous, has several characteristics that are fundamental so it can be used by visually impaired people everywhere. It must have a long-lasting battery to last at least one full day in operation. It must be light and small to be carried everywhere. And finally, it must not overheat. To solve this problem, this system also has a fan installed in the box cover of the portable system, but further details about this will be discussed in the next sections.

Figure 3.4: Portable System 2.0

Figure 3.5: Portable System's Limitations

In addition to these features, this version of the system, unlike the previous one, is easier to use by a visually impaired system. As it can be seen in figure 3.6, the box of the portable system only has two components outside, in the surface of the box, a power on/off button and a micro USB port to charge the power bank.

With this just two components, this system is much easier to be used by visually impaired people and beyond that, it is easily transportable because of its weight and size. For example, this system can be transported by using a normal backpack. The camera is also easily mounted in a backpack thanks to the new mount system based on the *GO PRO* cameras system. The Realsense 3D camera has a universal screw hole that also works in the *GO PRO* camera accessories. I mount a small system *GO PRO* accessory that allows the camera to be in the correct rotation position and it can be mounted on the shoulder strap of a backpack.

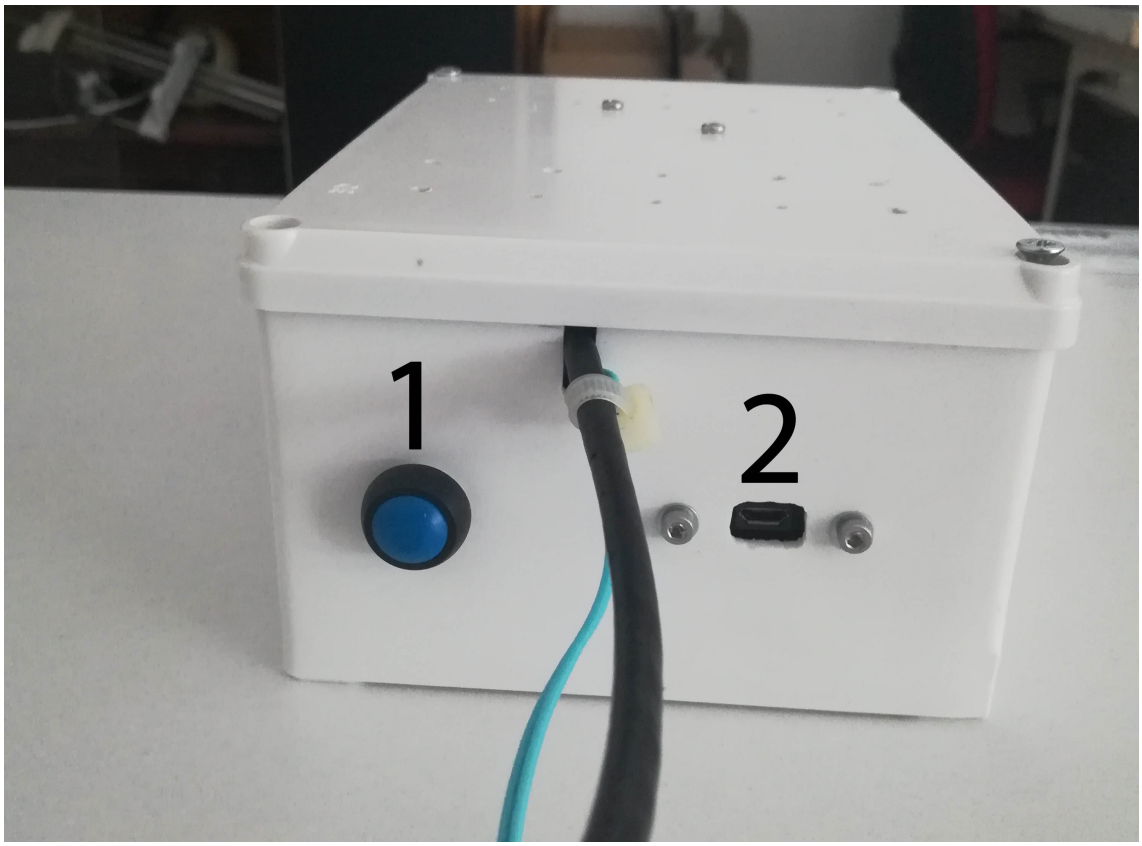

Figure 3.6: Portable System Simplicity, 1 corresponds to Power on/off button and 2 corresponds to the micro USB port for charging the power bank

Initially, the power bank was providing power to the *Jetson Nano* via micro USB, but due to energy and current problems, which will be addressed later in this project, now the power bank is connected via the barrel jack of *Jetson Nano* after some modifications and welds.

3.7.2 System Modes

This portable system has 2 modes, the **Generic Obstacle Avoiding** mode and the ***Door Problem*** mode.

Generic Obstacle Avoiding Mode

The goal of this mode is to help the user avoid obstacles at the head and trunk level in indoor but also outdoor environments.

The **Generic Obstacle Avoiding** mode, as the name implies, it's a more generic mode which simply does obstacle avoiding. The goal of this mode is to help the user avoid obstacles at the head and trunk level in the street and unknown places. It's also the mode where the *Stairs Problem approach* will be implemented. It uses the 3D information of the Realsense camera and depending on the distance of the nearest obstacle, it reproduces a beep sound. This mode works in a very similar way as the car parking sensor system when the

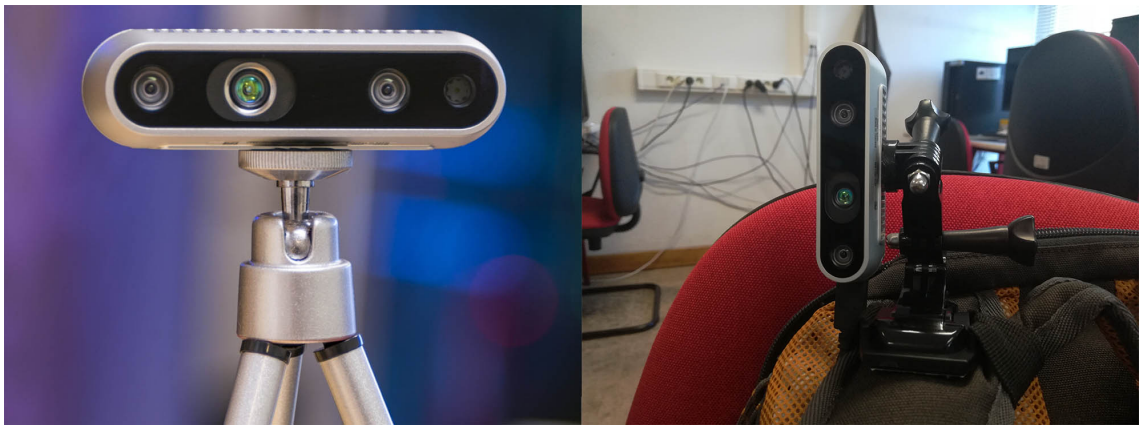

Figure 3.7: Original 3D Realsense camera D435 at the left side and *GO PRO* system with Realsense camera D435 mounted on the backpack's should tap.

user has an obstacle at his left side, a beep sound will be produced in the left in-ear phone, and the same happens on the opposite side.

Initially, I built a prototype method for this mode that uses the z-axis from the 3D camera to produce a specific sound. The smallest the distance between the user and the obstacle, the louder the sound the system produces. This method uses multi-threads to divide the 3D data of the camera into left and right zone of the image with the objective to work in stereo. If an object is positioned farther to the left, the system will reproduce the sound louder on the left headphone.

From this point, *Sérgio Gonçalves*, finalist student of computer engineering, is going to improve this system by reproducing sounds in the 3D matrix that represents the depth data from the 3D camera. The prototype system that I built for this mode will also be further explained later on this project since this chapter is more focused on hardware components and not in software.

***Door Problem* Mode**

This is the mode where I invested more time and, as the name implies is a mode-specific for solving the *Door Problem*. Three different methods were developed for solving the *Door Problem*, method A, B and C. Method A uses 2D *Semantic Segmentation* and 3D *Object Classification*. Method B just uses 3D *Object Classification* and Method C uses 2D *Object Detection / Semantic Segmentation* and 2D *Object Classification*. This 3 methods will be approached later on this project.

This mode also uses sound to give information to the visually impaired person. After each frame that is processed by each method, a sound is played to inform if the door is open, closed or semi-open. This will also be explained later on this project.

3.7.3 User-interface

The user-interface of this portable system is different from the previous portable system version. It's simpler and easier because the user doesn't need to use the smartphone it just needs to use his hands. To power on the system, the user must simply press the power on the button placed in the portable system box. A beep sound is reproduced so the visually impaired person can know that it successfully turned on the system. This also happens when the user turns off the system for the same reasons. The default mode that is started when *Jetson* turns on is the **Generic Obstacle Avoiding Mode**. This mode has its own sounds to guide the visually impaired people as it was already said previously. To switch to the other mode, the visually impaired people simply need to put his hand in front of the camera during 1 second, and it will automatically change to the **Door Problem Mode**. A sound is also played when the user switches between modes. To switch back to the first mode, the user will just need to put his head again in front of the camera.

Chapter 4

DataSet

In this chapter are described the datasets created for this project and how they were built. Two different datasets/databases were created, each one for help solving one type of problem, *Door Problem* and *Stairs Problem*. These datasets were build to be used in computer vision algorithms and to train neural network models. In its turn, the neural network models were then used to solve the *Door and Stairs Problem*.

Several images of doors and stairs and its surroundings were captured with different textures and sizes. Some of these images have obstacles that obstruct and hide part of the door and stairs such as, chairs, tables, furniture and even persons. The goal was to create a more generic and realistic real-world dataset. I also changed the pose to get different perspectives of the same door and stairs. The images captured are from Universidade da Beira Interior (UBI), public places (Piscina Municipal da Covilhã) and people's houses.

This chapter is organised as follows:

- **System to capture data for building the Dataset** - In this section is explained the script used to capture the images to build the dataset as well as some camera details, the after process and the errors that I got in the process of the dataset creation.
- **System to build semantic segmentation and object detection datasets (CVAT)** - In this section is explained the system used to create the semantic segmentation and the object detection versions datasets by using the original datasets.
- **Door Dataset** - In this section has described the dataset built for the *Door Problem* as well as its sub-datasets and the list of neural networks models that used this dataset.
- **Stairs Dataset** - In this section has described the dataset built for the *Stairs Problem*.
- **Dataset Comparison with Related Work** - In this section, I compare the Door dataset with the related work datasets in terms of RGB/3D Coverage and in the number of samples.

4.1 System to capture data for building the Dataset

The system used to build the datasets of this project was the *Prototype System 1.0*. The main component of this system, as it was already referenced, is the single board computer *Raspberry Pi 3 Model B+*. The only reason why *Jetson Nano* wasn't used here instead of the *Raspberry* was because *Jetson* didn't have arrived at the lab at the time I started to build the Door and Stairs datasets.

It was develop a Python (*2.6 version*) script that would save information from the 3D Realsense camera, "*save_img.py*". The python libraries used for this program were, the *pyrealsense2*, *numpy*, *opencv* and *time* library.

4.1.1 Python script

First, it was created the *realsense pipeline* and configuration where it was set to enable the camera to stream image with 640(height)*480(width) in colour (BGR) channel and in the depth channel. In the configuration, it was also configured the depth channel to have a depth scale equal to 1/16. The depth image is 2D grey-scaled image with the same size as the colour channel images, but each pixel value corresponds to the distance between the object in that pixel and the camera. The depth scale equal to 1/16 means that one meter in real-life is 16 (pixel value) in the depth image. For example, if a pixel in the depth image has a value equal to 32, it means that the object of that pixel is (32/16 = 2) 2 m away from the camera. After the configuration of the realsense camera channels, it was configured the files to write the images (colour and depth). Every time the program captures a frame, that frame is divided in depth and colour channels, and each channel is converted to a *numpy* array. For the depth image it was also used the colormap *COLORMAP_JET* from OpenCV with a *alpha* equal to 0.03. These values were chosen because they were the default values for getting the depth channel with this colourmap.

To save the image, the user simply would need to press a key in the keyboard to save the current frame in the realsense pipeline. Later, as the I built the prototype system version 1.0, instead of the keyboard, I used the smartphone communicated via SSH to *Raspberry Pi*. With the smartphone, I could simulate the input of the keyboard to save the images, and it was more practical as well since I didn't have to transport a keyboard to save images.

Later on, the script *"save_img.py"* was modified, specifically in the input to save the images. Instead of having just one input, it was added 5 types of input with the goal to label the images by saving them to a specific folder (open-doors, closed-doors, semi-open-doors, up-stairs and down-stairs).

- input key **o**: Open door folder.
- input key **c**: Closed door folder.
- input key **s**: Semi-Open door folder.

- input key **u**: Up stairs folder.
- input key **d**: Down stairs door folder.
- input key **n**: Normal image folder.

Each folder (each class) was also divided into two folders, *color* and *depth*, where the colour channel and depth channel images were saved respectively. The colour and depth channel images have the same name, which is the time and date the images were taken. Later on, this project, the name of these images was changed, and the images were sorted with the goal to start building the test, validation and training sub-sets.

4.1.2 Camera Detail

The camera used to capture the frames was a 3D Realsense camera. This camera has a horizontal viewing angle (86 degrees) higher than the vertical viewing angle (57 degrees). We rotated the camera 90 degrees to switch the angles with the purpose of including all the door area and the stairs area in the image, 4.1. The camera was placed 135cm above the floor.

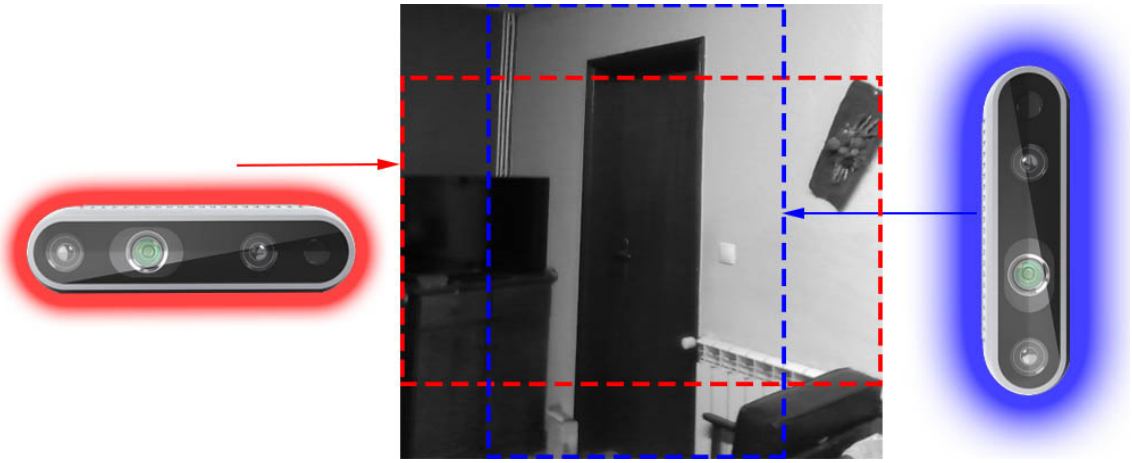

Figure 4.1: Difference between using the 3D Realsense camera in the original position and 90 degrees rotated.

The input image size defined in the Realsense configuration in the previous script is 640(width) * 480(height). But, as the camera was rotated 90 degrees, the images were rotated and have now the dimensions 480(width) * 640(height).

4.1.3 After Process - Dataset

With the system to capture the frames of the Realsense and saved the 2D and 3D images in folders according to the class of the folders what's left to do is to organise those images and apply filters to those images.

I created a folder in my desktop computer where I saved all the data and images of the dataset. The folder was first divided into *Door and Stairs Problem*. In the *Door Problem* I have three folders, one for each class, open, closed and semi-open and in the *Stairs Problem* I have two folders, up-stairs and down-stairs). En each folder, I have two folders, one for colour images and the other for depth images. For the 2D Image classification model the only thing left to do is to just use the colour images and ignore the depth images but to use the 3D Object Classification PointNet [QSMG16] there is another process that is needed to be done.

The input of the PointNet model are points sets which are represented in *.pts* files. Each file corresponds to a 3D image or Point cloud, and each row of the file corresponds to a point. Each row has three values (columns) which correspond to the three axes, x, y and z in the 3D space.

I developed a small script that cycle through all depth images and using the *Open3D* library, [ZPK18], I convert the 3D grey-scaled image into a point cloud data image.
With this format we can view point clouds by using the *Open3D* library or the *PCL*, Point Cloud Library viewer tool, [RC11]. After I got the ".pcd" (point cloud data) files, I used the *Open3D* library again to cycle every point of the point cloud and wrote the points in the ".pts" file.

4.1.4 Errors in the 3D information

Later on this project, after training the *PointNet* with this dataset I didn't get any great results, the mean test accuracy was very near 0.33 (around 0.37) which means that the network model wasn't learning at all. Even after trying different parameters, the results were always weak, and because of that, I decided to verify if all the images were well labelled and if there was no problem with them.

I developed a script that would cycle through all images and using the *Opencv* library, [Bra00], and the *Open3D* library, I was able to reproduce and view the point cloud of each image. After using this script, I came to the conclusion that several 3D images (point clouds) were damaged probably due to the camera lens being dirty or something similar. As it was said previously, the size of the images is 480(width) * 640(height), this means that each image has 307200 pixels (640*480), so each point cloud has 307200 points because each pixel corresponds to a point in the point cloud. The problem was that of some the point clouds (3D images) didn't have 307200 points but had only 1000 points, and that was what was wrong and incorrect. These images were excluded from the dataset, and the results increased.

4.2 System to label semantic segmentation and object detection datasets (CVAT)

The previous system was the system built to save and capture images and data for building the dataset using the 3D Realsense camera and the Raspberry Pi but these systems only allowed to label images for tasks as 2D and 3D Image/Object Classification. It was used other computer vision algorithms in addition to the previous ones to solve and approach the *Door and Stairs Problem* as the 2D semantic segmentation and 2D object detection.

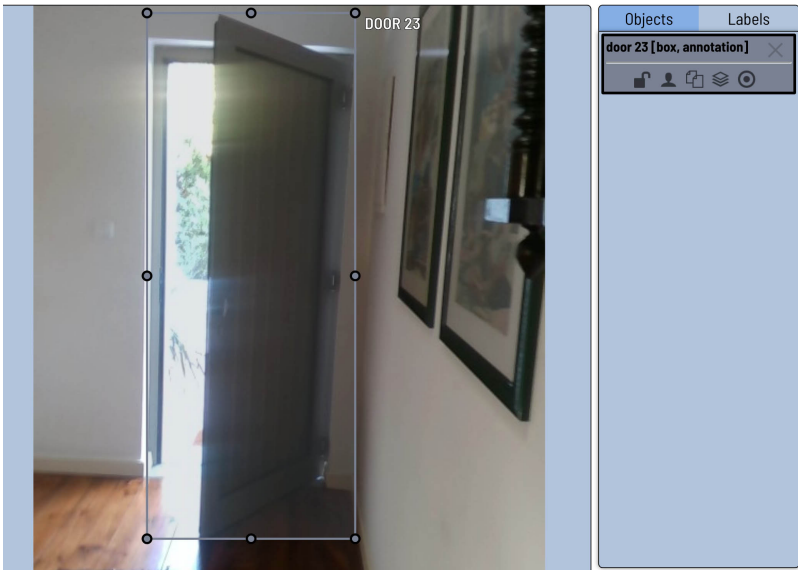

Figure 4.2: Example of CVAT using the box as the annotation tool.

To label the dataset for this computer vision algorithms, I used the *CVAT*, [opeon], which stands for Computer Vision Annotation Tool. This tool is built from the OpenCV library, and it allows to label semantic segmentation datasets by using polygons and polylines, but it can also be used to label object detection datasets by using boxes. One of the big advantages of using this tool is that it can export to several formats:

- CVAT XML 1.1 for videos
- CVAT XML 1.1 for images
- PASCAL VOC ZIP 1.0
- YOLO ZIP 1.0
- COCO JSON 1.0
- MASK ZIP 1.0
- TFRecord ZIP 1.0

The format that was used to export the labelled datasets was the MASK format because in this format we get a mask of the semantic segmentation with the same size as the original image. For object detection, the format used was the YOLO format, because, several object detection methods that were studied and tested in this project used this format. This tool was installed using its repository, *https://github.com/opencv/cvat* and it runs in the *localhost*, port 8080. To start annotating, a job needs to be created. I created two jobs, one for the validation set and the other for the train set of the semantic segmentation and object detection dataset. For object detection, the tool used to annotate was the *Box* and in the semantic segmentation, the tool used was the *Polygon*. Figure 4.2 represents an example of using the CVAT and using the box as the annotation tool to label an image for an object detection algorithm.

4.3 Door Dataset - Version 1.0

This dataset was built using a 3D Realsense camera with the portable system which allows me to save images from several places as it was mentioned before. The places where the samples were taken from are the following:

- Universidade da Beira Interior (UBI)
  - Canteen
  - Laboratory
  - Corridors
  - Classrooms
- Three private houses
- Piscina Municipal da Covilhã

The *Door Dataset - Version 1.0* is divided in 3 sub-datasets. Each sub-dataset is specified for one computer vision task. Those tasks are Image classification, semantic segmentation and object detection. In the following sub-sections, the motivations, specifications and characteristics of each sub-dataset are described. After the description of each part of the Door dataset, it's listed all the neural network models that used this dataset and which part of it was used.

This dataset is freely available online through the link, `https://github.com/gasparramoa/DoorDetect-Class-Dataset`. In this link, the user can view a simple description of the dataset as well as the descriptions of each sub-dataset. I provide the Intrinsic matrix (pixels) values of the camera that were used to capture the images, and it is also provided how the dataset is structured and organised.

4.3.1 Door Classification (3D and RGB) sub-dataset

This is the original dataset built for solving the *Door Problem*. The motivation was to use the 3D and RGB information to create point clouds or point sets, which, in turn, would be used in the 3D object classification *PointNet*. The 3D parcel of this sub-dataset was used in the first two methods for solving the *Door Problem* but it wasn't used in the third method, *Method C*. This last method just uses the RGB / 2D information.

This dataset has **1206** 2D(RGB) door images. 588 open doors, 468 closed doors and 150 semi-open doors. It also has the corresponding 3D component of these images; in other words, this dataset also has 1206 3D door images which correspond to the 3D component of the RGB image.
The test (60 samples) and validation (60 samples) set contain each by 20 samples of each class (open, closed and semi-open doors). The training set(1086 samples) is constituted by the remaining images (110 semi-open, 548 open and 428 closed doors).
The image size is equal to **480(width) * 640(height)** pixels. Both the 2D and the depth images have this size, but the depth images are in grayscale with a depth scale equal to 1/16. For example, if one pixel has the value 16, it is 1 meter away from the camera.

There is also a "cropped" version of this sub-dataset. This version is exactly equal to the previous one with the exception that the images are cropped according to the door and door frame localisation. This is to simulate the result obtain if it was using a door object detection method or semantic segmentation in the original images.

Figure 4.3: Door Classification (3D and RGB) sub-dataset with original and cropped versions.

The biggest problem of this dataset is the fact that it is not stratified. The difference between the number of samples in each class is not small. The test and validation set is stratified but as if was said previous the train set isn't. There are more closed and open doors than semi-open doors.

Although there were older versions of this dataset, from now on this project, this sub-dataset will be called **Door Class. 3D-RGB Dataset-version 1.0**.

4.3.2 Door Semantic Segmentation sub-dataset

The motivation to built this dataset was to use it for training semantic segmentation models to segment doors and door frames in *Method A* and *Method C* for the *Door Problem* which will be described later.

This sub-dataset has 240 labelled door images for semantic segmentation and the corresponding 240 original RGB door images. The RGB door images of this sub-dataset came from the previous sub-dataset. The labelled images were annotated using the *Computer Vision Annotation Tool (CVAT)*.

The images are divided into a test set (40 samples) and in train set (200 samples). The image size is equal to 480(width)*640(height). The labelled images are in grey-scale, where the pixel values vary from 1 to 2. If the pixel value is 1, it means that the pixel corresponds to the class ”don’t care” and if the pixel value is 2, the pixel corresponds to the class ”door” and ”doorframe”.

The weakest point of this dataset is its size due to the time it takes to annotate the images. Even if I used a tool to annotate the images, as it was used the CVAT, I have to draw several polygons for each image, and it’s a tiresome and repetitive task.

Although there were older versions of this dataset, from now on this project, this sub-dataset will be denoted as **Door Sem. Seg. Dataset-version 1.0**.

Figure 4.4: Door Sem. Seg. Dataset-version 1.0 with original and labelled images

4.3.3 Door Object Detection sub-dataset

This sub-dataset was built to detect door and doorframes on 2D images in *Method C* for the *Door Problem* which will be described and explained in a later section of this report.

It is composed by 120 annotated door and door frames images from the *Door Class.(3D-RGB) Dataset-version 1.0*, 149 annotated door and doorframes images from the *DoorDetect Dataset* and 144 images without doors from the *COCO Dataset*. The images without doors are used to count the number of False Positives in each method tested. Further details will be explained later in this report.

The test set has 60 images, and the training set has 353 images. The image size is equal to 480 * 640, and they are RGB images. The annotation files have four numbers which are the x and y coordinates of the top-left corner and the bottom-right corner of the bounding boxes. The images were annotated using the *Computer Vision Annotation Tool*.

The weakest point of this dataset is also its size due to the time it takes to annotate the images exactly for the same reason as the previous sub-dataset.

Although there were older versions of this dataset, from now on this project, this sub-dataset will be denoted as **Door Detection Dataset-version 1.0**.

4.3.4 List of Neural Network Models that used this dataset

Door Class. 3D-RGB Dataset-version 1.0

- *PointNet* (Method A & B)
- *GoogleNet* (Method C)
- *AlexNet* (Method C)

Door Sem. Seg. Dataset-version 1.0

- *FC-HarDNet* (Method A)
- *FastFCN* (Method A)
- *SegNet* (Method C)
- *BiSeNet* (Method C)

Door Detection Dataset-version 1.0

- *DetectNet* (Method C)

4.4 Stairs Dataset - Version 1.0

There was also built a dataset to approach the *Stairs Problem*, which wasn't the real focus of this project. The focus of this project was to solve the *Door Problem* since it happens more often to visually impaired people. The reason for the *Door Problem* happening more than the *Stairs Problem* is because the *Stairs Problem* only happens when visually impaired people are in unknown indoor places without their white canes which are a rare case, but it happens. The *Door Problem* happens when the visually impaired people are in their houses but, because they live together, the other people can without any intention leave the door semi-open and then the accident happens.

Either way, the Stairs dataset was built but, unlike the Door Dataset, it doesn't have sub-datasets to specific computer vision tasks as the semantic segmentation and the object detection tasks.

This dataset has 48 2D-RGB labelled images of stairs from our University, *Universidade da Beira Interior*. As the Door Dataset, it also has the 3D component of these 48 images in 48 grey-scaled images with the same size, 480(width) and 640(height). The images are annotated into two classes, Stair-up and Stair-down. It has 17 images of downstairs and 31 images of upstairs.

4.5 DataSet Comparison with Related Work

The **Door Dataset - Version 1.0** was compared with the related work, specifically with the door classification/detection related work methods. The sub-dataset that was compared was the 2D-3D Door classification.

Table 4.1: Door Dataset - version 1.0 comparison with related work.

DataSet	3D	RGB	Number of samples
Chen [ZB08]	×	✓	309
Llopart [LRA17]	×	✓	510
Quintana [QGPAB18]	✓	×	35
Ours	✓	✓	**1206**

Table 4.1 compares the **Door Dataset - version 1.0** with the datasets built-in related works. From table 4.1 we can conclude that **Door Dataset - version 1.0** has more samples than the other datasets and has RGB and Depth images which none of the related work databases has.

The dataset develop isn't just bigger than the related work datasets. It has several doors from at least six different locations. While the others 3 datasets have doors in a very controlled environment except for the *Llopart* dataset, [LRA17], which used the *ImageNet* dataset.

Chapter 5

Tests and Experiments

This chapter will address all the experiments and tests that I did to build a portable system (software and hardware). It will also discuss all the problems that I had in each process and how I solved them.

5.1 Ricardo's work

I started to read and implement Ricardo Domingos's bachelor final project method to solve the ***Door Problem***. Ricardo's work was like a baseline to my project, and that's the reason why I started to study his work, his problems and implement it.

5.1.1 Ricardo's work problems

Ricardo was having problems with noisy point clouds and with double doors. The point clouds captured by the 3D camera Realsense D435 were noisy. To solve this problem, Ricardo tried to get the depth information from other 3D cameras, like the Kinnect from Microsoft XBOX, but the problem remained. Because of the noise, the point clouds had swings that interfered with the calculations of the planes of the point clouds, and the obtained angle between the two planes wouldn't be the real angle between the door and the wall. The second biggest problem that Ricardo had was getting the correct crop image in double doors. As he was calculating the biggest area of each class in the semantic segmentation output, if we have a double door, where one door is open, and the other is closed, we should say that the path is clear for the visually impaired people to walk by because one door is open, but the system would say that the door was closed. This was happening because the biggest area calculated of the class door would be the closed door and not the open door.

5.1.2 Implementation of Ricardo's work

As an initial goal, I try to replicate his work in the lab computer. I implemented the same semantic segmentation algorithm C*ontext Encoding*, [ZDS+18], to detect the door in the input RGB image, using the same model pre-trained in the ade20k dataset. This model was the fully convolutional network *EncNet*, with *ResNet_101* as the backbone network. I used the benchmark *Context Encoding*, implemented in PyTorch for the semantic segmentation. I worked with Ricardo's code so I could see if the camera Realsense was working correctly and also if I was getting the same results as his.

I implemented Ricardo's work until the calculation of the point cloud planes because I notice one big problem that his proposal had to solve the *Door Problem*. In most cases, using the angle between the door and the wall to determine if the door was closed or open works, but there are cases that it doesn't work, the corner cases. For better understating of this problem, the following figure is presented.

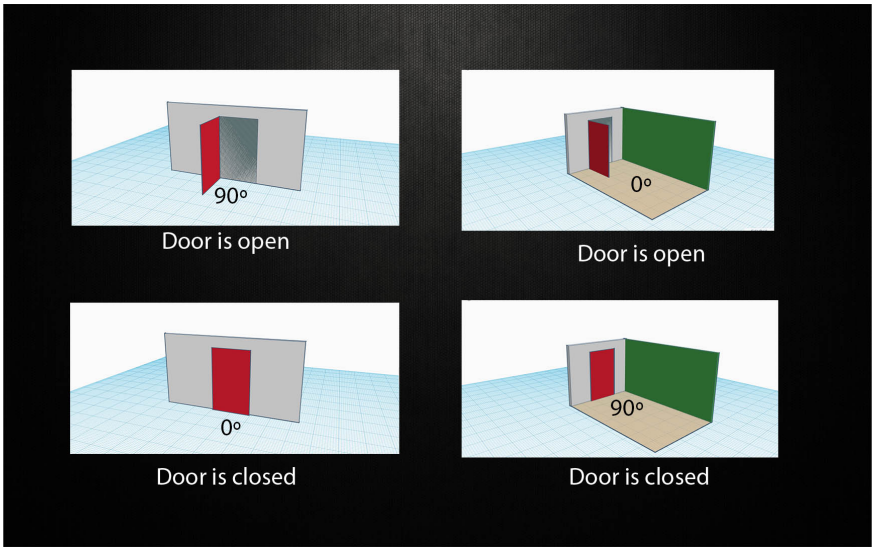

Figure 5.1: Problem in Ricardo's proposal for solving the *Door Problem*.

As we can see in the figure, the two pictures in the left represent the standard cases. When the angle between the wall and the door is 90 degrees, it means that the door is open and when that angle is 0 degrees, it means that it's closed. But on the right side of the figure, it's represented the corner situation where the scene is constituted by two walls and one door. The green wall represents the biggest area calculated in the semantic segmentation of the class wall, which means that the angle calculated will be between the door plane and the green wall plane. In the right top figure, for example, the angle between the wall and the door is 0 degrees which, according to Ricardo's proposal would mean that the door was closed, but we can see clearly that it isn't. The inverse situation happens in the right bottom figure, where the door is clearly opened, but, according to Ricardo's method, it would say that it was closed.

Another big problem of Ricardo's work for solving the *Door Problem* wasn't in the method itself but in his view of the *Door Problem*. As it was already said previously in this report, visually impaired people don't have problems either with open doors not closed door but with semi-open doors. The method for solving the *Door Problem* shouldn't be monitoring whether the door is open or closed but rather if the door is semi-open or not. Initially, I was approaching the *Door Problem* in the same way as Ricardo, where I just concerned if the door was closed or not, but later I changed my approach. I started to divide the detect door into three classes, open, semi-open and closed, and I stick in this approach for the rest of this project.

5.1.3 Semantic Segmentation - Context-Encoding PyTorch

I begin with the algorithm for the 2D semantic segmentation. I started to explore the benchmark that Ricardo used in his project, the *PyTorch Context Encoding*. Firstly, I installed this benchmark and all its dependencies in the lab desktop computer manually. An important detail was that the version of *Torch* that it was required, was the version 1.0.0 and only that version worked for this benchmark. I follow the following tutorial, `hangzhang/PyTorch-Encoding/experiments/segmentation`. I ran the script to build the dataset *ADE20K*, [ZZP+17], which was the same dataset that Ricardo used for the pre-trained model in the semantic segmentation. As I mentioned previously, I used the network *EncNet*, with the backbone *ResNet101* and the weights pre-trained on the *ADE20K* dataset. I tested the *demo.py* script, and everything ran fine, without any problems and errors.

5.1.4 Conclusion

In the final version of Ricardo's *Door Problem* method, the depth information wasn't used, derived from the problems that Ricardo had with the point clouds. Following all these problems, I understand that Ricardo's proposal couldn't solve 100% the *Door Problem* but it was the baseline and the roots for my proposal. Summing up, I implemented the use of semantic segmentation to calculate the biggest area of each class, and I also implemented the crop of both RGB and depth image and the creation of the point clouds of each class using the *PCL* toolkit, *png2pcd*. The *PCL*, which stands for Point Cloud Library is a framework for 2D/3D Image and point cloud processing. This framework is open-source software, and it contains numerous state-of-the-art algorithms, including surface reconstruction, filtering, registration, model fitting, segmentation, and others. After implementing Ricardo's work in the lab desktop computer, the next step was to implement it in a portable system for visually impaired people and also start to explore some related works to find other approaches and methods to solve the *Door Problem*.

5.2 Use of 3D object classification models to solve the *Door Problem*

As mentioned before, Ricardo's proposal couldn't solve *Door Problem* because of all the problems previously mentioned. My initial proposal was, instead of comparing the plane between the biggest area of the door and the wall, I used neural networks to do the classification. Since I have access to both RGB and depth information, the idea is to use neural networks that use this type of information and not just RGB images as the typical neural networks do. Before I started to explore 3D object classification models and algorithms, I built a mini dataset to then test these models.

5.2.1 Mini-DataSet

I started by creating a small/prototype dataset using the **prototype portable system version 1.0**(Raspberry PI Model 3B+). Once again, it's important to refer that these datasets had just two classes, open and close doors. This was my first approach to the *Door Problem*. Only later in this project, I realised that I should use another class.

I went to several places to take photos of doors in indoor spaces. If the door was open and one human could fit in that door, assuming he would walk in a straight line, that would mean that the door was **open**. If the person couldn't fit the door, even if it was semi-open, I would assume that the door was **closed**. The main objective in the *Door Problem* isn't to see if the door is open or closed, but to see if we can pass through it or not. In the middle term situation, when the door is half-open(semi-open), for us, not visually impaired people, the door is open, but for the visually impaired, the door is closed. The door is closed for visually impaired people because it is still an obstacle and they will get hit if they walk in a straight line. Once again, this approach was changed later to just had one class for semi-open doors.

I saved not only the traditional RGB image but its depth(3D) component. Every photo taken has one RGB image and one depth image. This way, I can use both pieces of information for object classification and could try different computer vision algorithms to solve this problem.
This *Mini-Dataset* was the beginning of *Door Classification (3D and RGB) sub-dataset - Version 1.0*. Of course that the labels of this mini-dataset were just open and close doors, but they were later changed to 3 classes, open, semi-open, and close doors.

5.2.2 PointNet

The first 3D object classification neural network that I started exploring was the *PointNet*. The goal was to use a method that uses both RGB plus depth information to classify if the door is open or closed or if the stair is up or down and other problems that visually impaired people have. I studied the PointNet because it is a foundation network that uses depth information as input and several networks derive from it. The original *PointNet* doesn't use RGB information for object classification. The original model is also capable of 3D part semantic and 3D semantic segmentation, but these approaches weren't used since they weren't in real-time. It uses only depth information, namely, an array that represents the coordinates of all points in one point cloud without RGB information. The number of rows in this array is the number of points of the input point cloud, and it always has three columns that represent the three coordinates of the 3D space, the x-axis, the y-axis, and the z-axis as it was already explained in the *Dataset* chapter, 4, of this report.

The PointNet was one of the first neural networks that used point clouds as input. They can be used to object classification, semantic segmentation, part object segmentation, and others. The main advantage of this network is that they don't need any type of voxelisation of the input point cloud. This will decrease the time of training the train set because the data is not so bulky as it would be if the point cloud was converted to a 3D voxel grid.

To implement and explore the PointNet, I used the following repository, `github.com/fxia22/pointnet.pytorch`, that is an implementation of PointNet in PyTorch. I tested with the same dataset that they used in the paper, the *ShapeNet*, [CFG+15] dataset, and everything was working correctly except for the Test and Val set. They were using the Test set during the training of the network when they should be using the Val set. I change this mistake simply using the value set in during the training and only using the test set after training the network as it should be.

With the prototype dataset that I built earlier(*Mini-Dataset*), I could now use it in this network to test both the network and the dataset. I needed to convert the depth image to a Numpy array with the file format *.pts* which represents the coordinates of the points in the point cloud. This was the file format used by *PointNet* and is just another representation of 3 data. These files needed to be placed in different folders dependently on their class, in my case, two classes, two folders, (open and closed).

It was also necessary to create the *.seg* files although these didn't need to be used in the object classification task and only in the object segmentation task, which I didn't use in this project. These files represent the sub-3D parts of an object in the input point cloud. In other words, they are a complementing file to the *.pts* files. Each point set or point cloud belongs to a class, but that class can be divided into sub-class. For example, the door can have the sub-classes, doorframe, door handle, and the rest of the door. The *seg* files have information about the sub-classes, which pixel belongs to each sub-class. These files aren't needed for the 3D object classification, but in this implementation of PointNet, they are required anyway due to some malfunction error in the code. The objects in the mini-dataset weren't divided into subparts, so I simply stipulated that there was only, for each object, one subpart of the same.

5.2.3 Dataset for PointNet

After getting an acceptable quantity of samples from the camera, I started to organise the dataset in the same format as the *ShapeNet* dataset. I did this because it was necessary if I wanted to use my Mini-dataset in the implementation of PointNet in PyTorch.

The following list is the structure of the dataset directory without any filtering and data augmentation.

/DataSet

- open
  - color (157 samples)
  - depth (157 samples)
- closed
  - color (731 samples)
  - depth (731 samples)

I was comparing my prototype dataset with *ShapeNet* dataset. One point cloud of the mini-dataset was much larger (307200 points) than the samples of the *ShapeNet* dataset (mean 2000 points).

One solution to this problem was to, instead of using the entire point cloud (307200 points) I only used the point cloud interest zone to classify if the door was open or closed. Basically, the point cloud interest zone is all the pixels that belong to the door itself in the original image. If I use only the points of the interest zone, the point cloud size will decrease.

But how did I got the interest zone of the point cloud? For that, I use 2D Semantic Segmentation (Context-Encoding). The advantage is that I will reduce the size of my samples that will enter as input to *PointNet* without losing any important information to distinguish between the open and closed doors. I only need to show the region around the door(door and doorframe) instead of showing all the regions that the camera 3D Realsense captured. I used the benchmark *Context-Encoding* in PyTorch, the same that Ricardo used in his work for the 2D semantic segmentation. I used the *EncNet* with the network backbone *Resnet101* pre-trained in the *ADE20K* dataset. For each RGB image, I generated a semantic segmentation image where I was only looking at the door class. The result of the 2D semantic segmentation was stored in the *DataSeg* directory with the following structure:

/*DataSeg*

- open (157 samples)
- closed (731 samples)

The *DataSet* directory is similar to the *Mini-Dataset*, the only different is that it only contains RGB images and those images are the output of the semantic segmentation *Context Encoding* previously mentioned.

Using the output of the 2D semantic segmentation, I built the *"DataSet-Slim"*. This dataset is equal to the original DataSet, but the RGB and depth images are cropped according to the result of the semantic segmentation.

One problem was that the semantic segmentation didn't work for all the samples, and because of that, several images in the *DataSet-Slim* weren't properly cropped or weren't cropped at all.

To solve this problem, I built a simple script to filter all the images in the *DataSet-Slim* directory. For example, if the door wasn't visible in the cropped region, that sample would be discarded and not used. This script requires a real user to filter all the images. It was built because it makes the process of filtering more efficient and we don't have the problem of eliminating the wrong RGB image, or it's depth component.

There was also develop another filtering script that, without the need of a real user, would filter the images for other situations. These situations were the situations where the crop of the image was too small. In other words, the semantic segmentation didn't almost detect any door object in the image. In this case, the sample wasn't used, and it was discarded. Another filter of this script that was used was in the height and width correlation of the image. If the height of the cropped image was smaller than the width, that image would be discarded because the height of the doors is bigger than the width. After applying all the filters, the structure of the *DataSet-Slim* was the following:

/DataSet-Slim

- open
  - colour (59 samples)
  - depth (59 samples)
- closed
  - colour (495 samples)
  - depth (495 samples)

If this dataset is compared with the *Mini-DataSet* for *PointNet*, after filtering the images, the dataset lost $(157 - 59 = 98)$ 98 images of open doors and $(731 - 495 = 236)$ 236 closed doors images. In total, the *DataSet-Slim* has 334 less images them the original one for *PointNet*, *Mini-DataSet*.

It was necessary to separate each class set in train, test, and Val sets. To do that I used the library *split_folders* (*Python 3*) which already does the data division. This library is also capable of using oversampling to one class if the dataset isn't stratified, which was the case. After applying all the filters previously mentioned I had almost 10 times more

samples of the ”closed class” than ”open class”. For each class I used 10 samples for testing, 10 samples for validation and the rest(475 samples) for training. Only the training set data has suffered oversampling. The oversampling is simply the creation of others copies in the class set that has less samples. It wasn't data augmentation. I created two new dataset directories one related to the other, *DataRGB3* and *DataDepth3*. The first one is the division of train, test and val of the RGB images of the dataset, with the following structure:

/*DataRGB3*

- test
  - closed (10 samples)
  - open (10 samples)
- train
  - closed (475 samples)
  - open (475 samples)
- val
  - closed (10 samples)
  - open (10 samples)

The second has the same structure, but instead of having the RGB images, it has its depth component.

5.2.4 Data augmentation for dataset for PointNet

All these sub-datasets that are being referred weren't used in the final version of the dataset as the same was already described in the *Dataset* chapter. It's true that these datasets are the prototype of the final version since several images of them were used to build this last version. It's important to say that none of the sub-datasets of the final version doesn't have data augmentation. It is up to the user to use data augmentation or not.

I did data augmentation after I generated the dataset *DataRGB3* with these three partitions, train, test, and Val. The test dataset didn't suffer data augmentation because I wanted to use real images and real-life scenarios.

At this moment, I used 2 data augmentation techniques. First I used Horizontal flip, that increased by two times the validation set size ($10 * 2 = 20$) and the train set size ($475 * 2 = 950$). After flipping the images I used an angle rotation between -25 and 25 degrees every 5 degrees ($-25, -20, -15, -10, -5, 0, 5, 10, 15, 20, 25$). This increased by eleven times the size of the validation set ($20 * 11 = 220$) and the size of the train set($950 * 11 = 10450$).

After getting a bigger dataset thanks to data augmentation I converted the RGB images with the depth images into point cloud data(.pcd) using the *PCL* tool, *png2pcd*. Once I got all the *.pcd* files I convert this into *.pts* which was the format required to use the implementation of the *PointNet*. The *.seg* files aforementioned were also created although they are not needed in the object classification task as it was already said. The only parameter still missing was the *.json* files. These were the files that contained the information about the sets (train, test, Val) that each sample belongs to. After all these processes, the structure of the dataset I built for PointNet was the following:

/Dataset-PointNet

- closed
  - points (10680 samples)
  - points_label (10680 samples)
- open
  - points (10680 samples)
  - points_label (10680 samples)
- train_test_split
  - shuffled_test_file_list.json
  - shuffled_train_file_list.json
  - shuffled_val_file_list.json

10680 samples because it aggregates all the 3 sets, train, test and val ($10450 + 220 + 10 = 10680$).

5.2.5 *PointNet* implementation results

After building my own prototype dataset with only images and depth information of closed and open doors I started to test the *PointNet*. I run the script for training the network to distinguish between ”open-door” and ”closed-door” with the following parameters:

- batch size = 20
- number of epoch = 7
- number of points = 2500
- learning rate = 0.001

After training the model, I tested it with the evaluation script, and the accuracy was only **0.55**, which means the network nearly learned anything about distinguish the two classes. The value **0.55** means that the network classified well 11 samples in 20 samples in the test set, which could easily be pure luck since there are only two classes (0.5).

I researched a little more about the implementation of PointNet and how the training was being done, and I found out that the problem was in the number of points used to training and testing. The parameter, **number of points**, defines the number of points that will be randomly selected in each sample to train the model. The default value for this parameter is **2500**, which in the case of the *ShapeNet* makes sense since the point cloud samples have on average 2000 points. In the dataset that I built, the point cloud samples have **100000 points** and the standard deviation can go between 500 points to 200000 points. Taking into account this, I trained the model with the number of points parameter equal to **10000**. The ideal case would be to use 100000 points or even 300000 points, but the memory capacity of the GPU couldn't take it, so I used only 10000 points.
This time, the parameter for training were:

- batch size = 20
- number of epoch = 5
- number of points = 10000
- learning rate = 0.001

I saved the weights of the network (model) in every epoch so I could evaluate each model and see if the network learned anything epoch by epoch. In the evaluation, I used the same batch size and the same number of points. I only evaluate the models after the training. The following table shows the results that I got for evaluating each model. I evaluated five times each model and then calculated the mean and standard deviation for both the loss and the accuracy value.

Evaluation	Mean accuracy	Mean loss
1 Epoch	0.69±0.02	0.6945±0.007
2 Epoch	0.75±0	0.6161±0.006
3 Epoch	0.72±0.02	0.6398±0.007
4 Epoch	0.79±0.02	**0.5728±0.007**
5 Epoch	**0.80±0**	0.5884±0.012

Table 5.1: Evaluation results on 5 models from Pointnet trained in my own PointNet dataset

I can conclude that the number of points greatly influences the results of the evaluation of the network. The dataset that I built has a lot of points (100000 on average), and it's important to use more points for training the network. Recalling that each point cloud has 100000 points in average and not 307200 because it was cropped according to the result of the semantic segmentation method previously mentioned. With only 2500 points, the network couldn't learn anything because they only represented on average 2% of the entire point cloud.

5.3 First proposal to solve The *Door Problem*

From the previous section, I merged all the main steps and built my first proposal to solve The *Door Problem*. This proposal was proposed in the prototype portable system version 1.0, and it's based on 2D Semantic Segmentation and 3D object classification. First, the camera gets the 2D and 3D images/frames. The second step is to use 2D semantic segmentation. In this case, it was used the *Context-Encoding* benchmark with the *EncNet*. Then, the biggest door area in the image was calculated using the same approach as Ricardo Domingos used in his method. This method was to use the function *FindContours* of *opencv* library using a masking threshold between the RGB values of the door in the semantic segmentation output. Then I got the bounding box in the output of *FindCountours* using the function *boundingRect* of *opencv*. The original depth image was cropped based on this bounding box. The cropped depth image was then converted into a *.pts* file, which is the input of the *PointNet* and them the *PointNet* returns the score value which consists of three values(one for each class) between 0 and 1.

The red rectangle boxes represent parts of this proposal that could be improved. For the *Semantic Segmentation* I could use another algorithm or neural network to do it. The same goes for the *PointNet* rectangle, I could use other 3D Object Classification model instead of the *PointNet*. Figure 5.2 represents the proposal previously described.

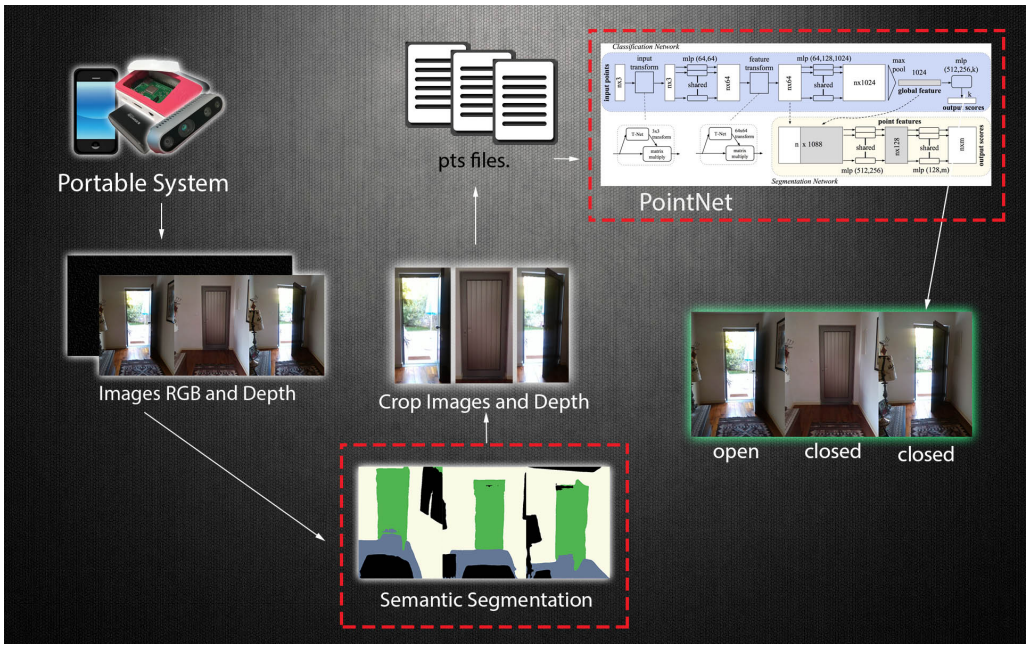

Figure 5.2: First proposal to solve the *Door Problem*

This proposal is very similar to the first method for solving the *Door Problem* that I develop (*Method A*). This method will be covered later in this report. The main difference is in the semantic segmentation model used and in the labelling of the dataset. This first proposal just uses two classes(open and closed), and *Method A* uses 3 (open, closed, and semi-open door).

5.3.1 Problems with the dataset

The dataset I built for solving the *Door Problem* for visually impaired people is still incomplete and has several problems. It contains point clouds with a lot of difference in terms of the number of points. There are point clouds with almost no information, and so they are useless to the model. There are also point clouds with extra information that will make the system slow, and that extra information isn't necessary to distinguish between the closed and open doors. Although the dataset already has 10.000 samples with data augmentation, it's still not enough because it needs more different kinds of doors so the model can generalise better.

5.3.2 Problems with the semantic segmentation

The problems with this proposal are not only due to the database. There were several samples that were being wasted because of the semantic segmentation. The problem was with the cases when the door was open. In these situations, the 2D semantic segmentation couldn't segment/detect very well the door jamb. One idea was to use the semantic segmentation algorithm to detect the door frame and not the door. There are cases when the door is wide open, 180 degrees, and the segmentation result instead of being the door and the door frame, is just the door itself which is open of course, but with that context, it looks like its closed, and it will induce errors to the network. The following picture 5.3 represents the previous situation:

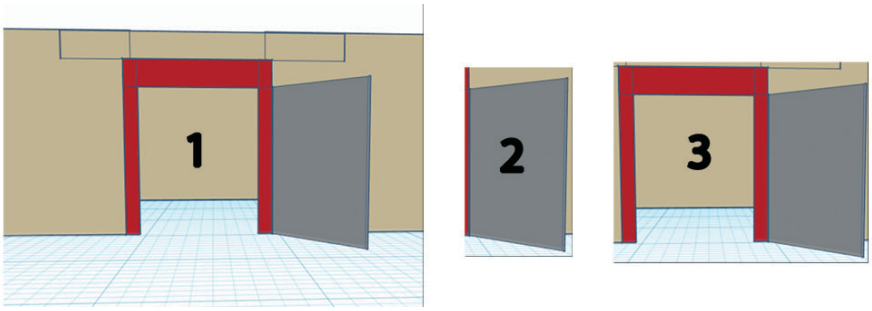

Figure 5.3: Semantic Segmentation problem in the first proposal. (1-Represents the image captured by the camera, 2-Semantic Segmentation output and 3-Expected Semantic Segmentation output)

The *ADE20K* dataset has several classes and one of them is the *doorframe* or *doorcase* which is the ideal class to use in this proposal. The problem is that the *doorframe* is not actually a class but a sub-class and the model that I was using for the semantic segmentation, the *Context Encoding*, uses only the 150 main classes of *ADE20K* (in the default ADE20K data-loader of the semantic segmentation model) and *doorframe* isn't included. Another problem with the repository of the implementation of the Context Encoding is that the training of the model is restricted to multi-GPU. The lab computer that I am using has only one GPU so I couldn't train the model and change it.

5.4 FastFCN semantic segmentation

As I couldn't train the model, I decided to explore other semantic segmentation algorithms with the conditions of having a *ADE20K* data-loader and single GPU training. The best three methods for the semantic segmentation in the *ADE20K* at the moment were the *PSPNet*, [ZSQ+16], the *Context Encoding* and the *FastFCN*, [WZH+19] which also uses the *EncNet* just as the Context Encoding method uses. In these three methods, the only implementation that allowed single-GPU training was the *FastFCN*. The FastFCN method was a good choice because it's a modification of the previous method that I was working with, the Context Encoding, so it's implementation was similar and simple.

Unfortunately this method also only uses the 150 main classes of the *ADE20K* dataset so I couldn't simply use the class *doorframe* because it wasn't labelled. In fact, the implementations of the best three methods for the semantic segmentation in the *ADE20K* dataset only use the 150 main classes in the *ADE20K* dataset.

I downloaded the entire *ADE20K* dataset, with all the classes and sub-classes and I filtered it so it would only have the images with doorframes and stairs. I used the class *stairs* because of the other problem that I got information that visually impaired people usually have, the *Stairs Problem*. The annotations in the original *ADE20K* dataset didn't bring any information about which value corresponds to which class. I had to use a specific mask that the *ADE20K* used for the annotations, and after that, by trial and error, I discover which value in the mask corresponded to the class/sub-class *doorframe* and to the class *stair*.

Instead of having one dataset with images and annotations of 150 classes, I have now one dataset for the semantic segmentation with only two classes. The annotations are grayscale images where the value of the pixel match to one of these two classes (doorframe and stair) or no-class(everything else).

- class ”doorframe” - value 1
- class ”stairs” - value 2
- no-class - value 0

After filtering the original dataset *ADE20K*, the dataset became with 132 samples in the validation set and 1133 samples in the train set. It was also necessary to make some change some in the files, *ade20k.py, option.py*, so the new dataset could work, and the network could start training.

The resulting dataset had the following structure:

/ADE20K-Modified-DoorFrame-Stairs

- annotations
  - training (1133 samples)
  - validation (132 samples)
- images
  - training (1133 samples)
  - validation (132 samples)
- objectInfo150.txt
- sceneCategories.txt

In the training set (1133 samples), there were 255 samples for the class *door(and doorframe)* and 878 samples for the class *stairs*. In the validation set (132 samples), there were 30 samples for the class *door/doorframe* and 102 samples for the class *stairs*.

5.4.1 Training *FastFCN* for semantic segmentation with *doorframe* and *stair* classes

With the resulting dataset, I started to train the *FastFCN* Semantic Segmentation mode for just the two classes that needed to be segmented for the *Door Problem* and *Stairs Problem*. I trained the *FastFCN (EncNet)* during 50 epochs, with batch size equal to 6 and the backbone used was the *ResNet101*. At the end of training, the validation pixel accuracy was equal to **0.984** and the mean intersection over union(*IoU*) was **0.962**. I used the pixel accuracy and the intersection over union (*IoU*) as the evaluation metrics. The intersection over union is the area of superposition between the predicted segmentation and the ground truth divided by the area of the union of these last two. The mean IoU is to the mean of the Intersection over union for each class. The pixel accuracy is the per cent of pixels in the input image that are classified correctly. The results looked good, but when I tested in the test set, I could see that there was something that wasn't working correctly.

In figure 5.4, in the prediction(right side), the blue value is correlated with the class *doorframe* and the green value with the class *stairs*. The network almost didn't learn anything and the problem, in my opinion, might be because there are two classes, but the prediction should predict three different values, 1 if is a doorframe, 2 if is a stair and 3 if is neither of them. The strategy here was to add one more class that I called *no-class*, and it will have assigned the value 3 instead of the value 0, which was mentioned earlier. I did this because I noticed that every annotation done previously by the authors of the repository had one class value to each pixel. I thought that the value 0 was for class *no-class* but I was wrong, in fact, the value 0 wasn't assigned to any pixel in any annotation.

Figure 5.4: Prediction of FastFCN in 1 image of the test set from the *ADE20K* dataset using only 2 classes, *doorframe* and *stairs*.

After this modification, I trained the model again, with the same arguments, the only thing different was this last modification. At the end of the training, the validation pixel accuracy was **0.970**, and the mean intersection over union was **0.556**. It may seem that the *IoU* was worse, but in this problem, I had three classes, being the last one assigned to everything that wasn't a door or stair. In the previous training, I had only two classes, where the pixels that didn't belong to nether this two classes had the value 0 assigns and the *IoU* wasn't calculate taking into account the pixels with the value 0, and that's why it's value was bigger in the previous case. The results are a little better but are far from good. In the most part of the images, the class 3, *no-class*, is predicted in almost every pixel by the model.

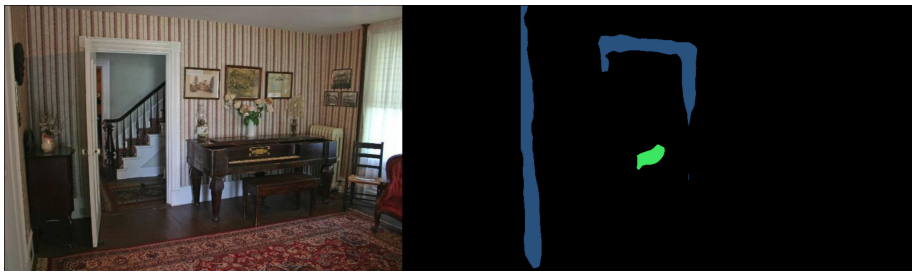

Figure 5.5: Prediction of FastFCN in 1 image of the test set from the *ADE20K* dataset using 3 classes, *doorframe*, *stairs* and *no-class*

In figure 5.5, in the prediction, the blue value is correlated with the class *doorframe*, the green value with the class *stairs* and the black value with the class *no class*. The problem is that some images and annotations are too complex, especially in samples with stairs. Some of these complex cases in the portable system for visually people will never occur so, I deleted all the samples that contained the class *stairs* and start to focus only on the *Door Problem*.

5.4.2 Training the *FastFCN EncNet* with only 2 classes, *doorframe* and *no-class*

I trained the *FastFCN EncNet* with 2 classes, *doorframe* and *no-class* instead of the previous 3 classes. This way, I was removing the complex cases(*stairs* samples), and I focused first on solving the *Door Problem* for visually impaired people. After removing these cases, my dataset had 255 samples in the training set and 30 samples in the validation set. In the best epoch I got a validation pixel accuracy equal to **0.960** and the mean *IoU* was **0.702** when in the previous test it was just **0.556**.

5.4.3 Improve in the dataset for the first Proposal to solve the *Door Problem*

After seeing I was still not getting the expected results, I improved both the semantic segmentation dataset (*ADE20K*) and 3D object classification Mini-dataset.

In the last test I did, I had 255 samples in the training set and 30 samples in the validation set. To improve the semantic segmentation algorithm, I increased the dataset of doorframes. As I already had several images of doors, taken for building the dataset for object classification, I used some of those images to build the segmentation dataset. To label the images, I used the *CVAT* (Computer Vision Annotation Tool), which was simple to use. The user only needs to draw the polygons in the image. One advantage of using this tool is it has one format to export the annotation that is very similar to the default format of the *ADE20K* dataset and so, I could easily convert it to the correct format.

Previously, in the Mini-dataset, I had 731 images of closed doors, 157 images of open doors. This dataset also had two images of downstairs and 31 images of upstairs for the *Stairs Problem*. As I was focused on the *Door Problem*, I didn't increase too much the *upstairs/downstairs* dataset. After using the prototype portable system version 1.0 to take more pictures of doors, my dataset was now constituted by 1096 images of closed doors, 734 images of open doors, 12 images of downstairs, and 33 images of upstairs.

Once again I used the tool *split-folder* to split my dataset into test and train sets. A strong point of the new dataset was that the number of samples of closed doors was no longer five times bigger than the number of samples of open doors. With the improvement, the number of samples of closed doors wasn't even two times bigger than the number of samples of open doors. With the tool, *split-folder* I oversampling the number of samples of open doors to be 1096 instead of 734. After the split, the dataset had 1992 images in the training set and 200 images for testing the model.

What was missing was label this new Mini-dataset so it could be used on the *FastFCN* model. As I was annotating the images I came to the conclusion that it would take too much time annotating all the images (1992 + 200), so I stopped annotating, and instead

of using the full dataset I created a small version of it. I came to this conclusion because in the entire version of the dataset I have ten pictures of the same door, but with different rotations, perspectives and illumination and they will be used in the 3D object classification model, but for the semantic segmentation, one picture or two per type of door was enough to for the model to be able to generalise well. So I reduce the dataset manually and in the final, I had 200 samples in the train set and 40 samples in the test set.

5.5 Door 2D Semantic Segmentation

The idea of the 2D semantic segmentation is to detect the objects. In the first proposal for solving the *Door Problem*, this method was used to reduce the point cloud that will be used in the 3D object classification (PointNet) model and to just send the necessary information to distinguish between the open and close door.

5.5.1 Using only *doorframe* class in semantic segmentation

Initially, I was doing 2D Door semantic segmentation as Ricardo did in his work, but later on, I saw that doing only door segmentation would induce the model to bad results since, in some situations, the information of the door may not be enough to classify its opening. To solve this problem I proposed the *doorframe* instead of the *door* but, as I was labelling the previous dataset in the *CVAT*, I came to the conclusion that there also are situations that using the doorframe only wouldn't work. The following situation shows the problems of only using the *doorframe* class:

Figure 5.6: Semantic Segmentation problem of using just the *doorframe* class. (1-Represents the input image, 2-Semantic Segmentation output prediction, 3-Expected Semantic Segmentation output)

As it can be seen in the figure, 5.6, the doorframe is divided by the door into two annotations. There are two-door frames annotations for just one door. If we apply the proposed algorithm, the first step would be to calculate the biggest area of all of the door frames in the picture and them draw a bounding box, and crop the image following that bounding box. But in this case, the door frame is divided into two parts, so only the biggest part will be considered, and the image will be cropped following only that part of the doorframe resulting in the wrong cropped image expected.

5.5.2 Using *doorframe* and *door* class in semantic segmentation

One solution for the previous problem would be to instead of segmenting only the doorframe, segment both doorframe, and door, and considered them as one class. In the previous example (5.6) if I used as one class both doorframe and the door, the predicted output of the semantic segmentation model would be very similar to the expected.

5.5.3 Evaluation of the possible semantic segmentation strategies

To see which strategy was the best to use, I evaluated all the semantic segmentation *FastFCN* different strategies, comparing the training times and the predictions using exactly the same parameters in all the strategies. The strategy with the most correct cropped images in the test set would be the best to use because it would mean that it can generalise better and crop more pictures correctly than the others. An image was correctly cropped when it keeps only the interest zone in it. If the cropped image didn't have enough information to see if the door was open or closed or if the image had overload information, that prediction would be considered as a bad prediction.

The difference between the strategies that were evaluated was which classes were labelled in the dataset. The after-process was the same for all the strategies. The model *FastFCN* was trained with only two classes, the class that defines the strategy (just door, just doorframe, door, and doorframe) and the class that represents all the other objects (no-class). After this, the model was evaluated with the same test set (40 images) for all the strategies, and the predictions were saved. For each prediction (40) it was calculated the biggest area of the class that defines the strategy, and then, the bounding box was drawn around that area. The original RGB image was cropped according to the bounding box. After this step, filters (image or the width is too small) were applied to the cropped image. The final step was to compare the resulting cropped images of each strategy.

3 strategies were considered:

- Labelling **only** the **door** class in the dataset
- Labelling **only** the **doorframe** class in the dataset
- Labelling **both door** and **doorframe** as one class in the dataset

The *FastFCN(EncNet)* model was trained separately for each one of the strategies during 50 epochs. All the parameters of the training were the same, batch size equal to 5, the backbone network was the *ResNet101*, and the learning rate was the default value (0.003125). After training each model with the different datasets, the model was evaluated in the 40 samples of the test set. The results were the following:

Strategy	Train time	Mean accuracy	mIoU
Only door labelled	22:27 min	0.9573	0.8622
Only door frame labelled	23:20 min	**0.9650**	0.7918
Door and door frame labelled	22:24 min	0.9474	**0.8700**

Table 5.2: Evaluation results on *EncNet FastFCN* with 3 different strategies

As it can be seen in table 5.2, all training times were more or less the same for all strategies. The difference was in the mean pixel accuracy and in the mean interface over the union. Normally, in semantic segmentation, the mean *IoU* is taken into account more than the mean pixel accuracy. This is true because the pixel accuracy only represents the number of corrected guessed classes in each pixel while the interface over union is the area of overlap between the predicted segmentation and the ground truth divided by the area of union between the predicted segmentation and the ground truth. According to the results, the best strategy was to use both door and doorframe labelled images as expected because it had the biggest mIoU value although the strategy that only uses the door had also a similar mIoU value.

The next step was to compare each model/strategy cropped image prediction in the 40 images of the test set. This second test was made because the most important aspect is the number of corrected cropped images that each method can predict. The goal was to get the biggest amount of corrected cropped images to use in the 3D object classification network because they will generate smaller point clouds then the original images would generate and have all the information that the network needs to distinguish between the open and closed door. To test this, I just compared all the cropped prediction images and saw manually if they had the necessary information for the door classification. The results of this test were the following:

Strategy	Corrected cropped images
Only door labelled	18 / 40
Only door frame labelled	29 / 40
Door and door frame labelled	38 / 40

Table 5.3: Corrected cropped images on *EncNet FastFCN* with 3 different strategies.

Although the strategy of only labelling the door class had good results in the evaluation of the model, which means, the model can segment very well the door, it was the weakest in the number of corrected cropped images. This strategy was the original one, but it's quite simple to understand why it wasn't very good at getting the corrected cropped images. In images where the door was open, sometimes the door itself was almost occluded, and so the algorithm wouldn't detect it. I concluded with this test that the best strategy at this moment was to label both the *door* and *doorframe* classes as one class.

5.6 PointNet - (3D Object Classification)

As the 2D Semantic Segmentation strategy and the *Mini-dataset* had been improved, I focused on the 3D object classification models. At this moment of the project, the main goal was to solve the *Door Problem* and build as fast as I could one prototype system to work to get feedback from a real user.

The *PointNet* was the first and only 3D Object Classification model that was tested in this project. With the previous improvements, the dataset was no longer unbalanced in terms of the number of samples in each class, 1992 images for the train set(996 open doors and 996 closed doors), 100 images for the test set(50 open doors and 50 closed doors) and 100 images for the validation set(50 open doors and 50 closed doors).

Using the 2D Semantic Segmentation model *EncNet FastFCN* trained in the images with both *door* and *doorframes* labelled as one class, I could reduce the size of the images to the interest zone. Although the semantic segmentation has improved (previous section), it still wasn't as good as I wanted it to be. There were scenarios where only the door is detected, and the doorframe wasn't. If the door, in this scenario, was open, it would still mislead the network because the resulting cropped image would only have information around the door, making it look like the door was closed when it wasn't. After the semantic segmentation, the dataset will always be smaller because the semantic segmentation algorithm will fail in detecting doors in some cases, normally the most difficult ones. The size of the dataset at the moment before the use of semantic segmentation was 724,7 MB. After that, the size of the dataset decrease to 308,7 MB. The dataset, due to the segmentation, had 1032 samples of closed doors and 565 samples of open doors. Using *up sampling*, I increased the number of samples of open doors to 1032, starting to have in total 2064 samples. This set was divided in the train[1864](932 open doors plus 932 closed doors), test[100](50 open doors plus 50 closed doors), and val[100](50 open doors plus 50 closed doors).

I followed the same process when I built earlier the first dataset to test the PointNet. I converted the RGB and Depth images to the file format *.pcd* and them to the file format *.pts*. As I was converting the images to the point cloud data format, I noticed that some point clouds had noise in the depth axis in terms of missing points. This was probably due to the lens being dirty at the moment the image was photographed. The only solution to this problem was to analyse the point cloud manually, one by one, and see if it was noisy/corrupted or not.

I trained the model with the previous 2064 samples during 50 epochs and using the batch size equal to 10. The learning rate was 0.001, and the number of points parameter was equal to 10000.

As I was testing the PointNet, I came to the conclusion that if I wanted to build a prototype system before Christmas, I had to implement a program that would put all the modules explored and implemented until now together to be able to get feedback from a real user.

5.7 Prototype Program

I built the first **Prototype Program**. This script waits for the user to interact with *("press enter button")*. This script uses the segmentation algorithm *FastFCN* to detect doors in front of it using the information from the 3D camera RealSense D435. If there was any door in front of the user, the program would use the 3D object recognition algorithm PointNet to predict if the door was open or closed and inform the visually impaired user through sound.

The first step in the program was to get the information (RGB and Depth) from the 3D camera Realsense D435. After getting these, both the RGB and the Depth are rotated 90 degrees since the camera is 90 degrees rotated as it was already explained in the *Project Material Section*.

After rotating the image, the semantic segmentation algorithm FastFCN would detect if there were doors in it. After getting the resulting image of the semantic segmentation the biggest area of the class *door_doorframe* was calculated. If the biggest area wasn't big enough, it would mean that the algorithm barely detects any door and it would conclude that there weren't any doors in front of the user. If the semantic segmentation detected a door, the 3D object detection algorithm PointNet would predict if the door was open or closed using the depth information.

5.7.1 Problem - Real-Time

The biggest problem wasn't if the semantic segmentation algorithm or the 3D object detection algorithm didn't always predict the correct results but the inference time of these algorithms. If the results don't arrive at the user in real-time, it doesn't matter if they are correct because the visually impaired people can't wait and he could already have an accident. To process one frame and predict if the door was open or closed, the program took 16/17 seconds which was really bad because the ideal case was to process at least 1 or 2 frames per second. In other words, the program should take no more than 1 second per frame.

To solve this problem, I looked into the program code and tried to see which instruction/process could be improved in terms of speed, and I found one big mistake that the program had. Both the model of the semantic segmentation algorithm and the model of the 3D Object Recognition were loaded in every frame without being needy. Also, the *data loaders* weren't necessary because I was just testing one frame at the time instead of a big

data set. After doing those modifications, the total script inference time reduced 8 seconds, which was great but far from the expected time. Another aspect that could reduce the program's time was to remove all file creation that the program was doing. I change to the program to simply use variables without the need to save files. With this modification, the total inference time also reduced about 8 seconds. The creation of the *.pts* file for the PointNet model was taking to much time, and it was this creation that influenced more the total inference time. The following table shows the results of all the modifications I did to get to the final results:

Program modification	***MSI* time per frame**	***MSI* time in FPS**
Original program	16.0 seconds	0.0625
Removal of models loading	8.0 seconds	0.125
Removal of files creation	0.2 seconds	5

Table 5.4: Mean script inference times(MSI time) per frame and in frame per second in the desktop computer after all the modifications in the prototype program. ()

5.8 *PointNet* Tests without Semantic Segmentation

This section will approach all the PointNet tests without the use of the semantic segmentation model. In other words, in these tests, the most important was to see the mean accuracy and inference time of the 3D object classification method. The main goal here was to see the difference between using the entire(original) image captured by the camera and using the cropped image obtained through the output of semantic segmentation. In other words, the goal here was to see if it was possible to classify the point clouds correctly without the need of using cropped point clouds with the objective also to decreased the mean inference time.

5.8.1 *PointNet* with original size point clouds

The first test was to simply train the *PointNet* using the previously created Mini-dataset of *open* and *closed* doors with the original sized point clouds.

Dataset used

For this test, as I said before I used the point cloud and images with the original size, 640 height, and 480 widths of the dataset I had at that time. This dataset had only 2 classes, *open door* with 734 samples and *closed door* with 1096 samples. As it was built software to check if the RGB images were clean or blurry, it was also built another software to check the Depth information and to my surprise, I had a lot of 3D images with noise. This was probably due to the lens being dirty and also due to some of the places I got samples didn't have the best illumination. After filtering these images, I got 615 samples of *closed doors* and 479 samples of *open doors*, in total I lost almost half of my original dataset about 736(481 + 255) samples out of 1830.

Using data augmentation(angle rotation and horizontal flips), I increased the number of samples of *open doors* to match the number of samples of *closed doors*, 615. I used 50 samples of each class for testing, 50 samples of each class for validation, and 515 samples of each class for training. In total, I had 1230 samples for test, validation, and training.

Train parameters

I trained the model during **20 epochs** with a batch size equal to 20 with a *number of points*(*PointNet*) equal to 10000. There were 1030 samples for training which would give 51 iterations in training with the size of the batch (20). After the training, in each epoch, it was calculated the loss and accuracy in the validation set. As the training wasn't deterministic, the model was trained three times(Iterations), and after that, it was calculated the mean accuracy and loss between those three iterations for the test, validation, and train set.

Results

It's important to say that, each iteration consists in 20 epoch of training and validation plus testing in the test set after the 20 epochs. It was also calculated the average time of each script iteration to be later compared to other approaches.

The following table 5.5 shows all the results in this test:

	Mean Loss			**Mean Accuracy**			
Iteration	Train	Val	Test	Train	Val	Test	Iteration time (seconds)
1º Iteration	0.6243	0.5957	0.6047	0.6600	0.6375	0.6800	16307
2º Iteration	0.6197	0.6160	0.6937	0.6700	0.6690	0.6700	16314
3ª Iteration	0.6148	0.6003	0.6054	0.6750	0.6645	0.5000	16272
Mean	0.6196	0.6040	0.6346	0.6683	0.6570	0.6167	16298

Table 5.5: Results in training and testing the PointNet with the Custom Filtered Dataset with the original sized images.

Results analysis

As can be seen in the table, each iteration of training with 20 epochs takes more or less 16000 seconds which wasn't too much in the machine learning area, but there are several ways to reduce it and still have good results. Each epoch took around 15 minutes. The results after 20 epochs of training weren't good because the mean accuracy in the test set was **0.6167** which, for a problem with two classes isn't good enough(0.5 if random). As was mention before, probably this was due to the network choose only 10000 points out of 300000 points randomly, which only represented 3% of the original point cloud. It was the same as representing a 2D Image with 300000 pixels with only 10000 random pixels of those 300000, sometimes we can clearly classify the object and in other cases, we can't. The training wouldn't happen in the portable system for visually impaired people, but the prediction would and it was important to have the highest number of predictions possible in 1 second. The iteration time is directly correlated with the inference time.

5.8.2 *PointNet* with voxelized grid original sized point clouds

In this test, I trained the PointNet with voxelised point clouds. Voxelised point clouds are point clouds that have fewer points that their original size (down-sampling). This was great because in this work it was really important that the system was in real-time and with point clouds that represent the same information as the original ones but are lighter would help the system to predict must faster. Voxel down-sampling uses a regular voxel grid to create a uniformly down-sampled point cloud using the original point cloud.

Dataset used

The dataset used in this test was exactly the same as the previous one with the modification of the point clouds to voxel down-sampling. In the previous test, each point cloud had 307200 points (640 * 480), but with the voxelisation, now each point cloud (sample) has around 10000 points. Although the number of points of each point cloud had been reduced, the point clouds still represent the depth information greatly with less memory. The dataset with the original sized 1230 samples/point clouds had almost **18GB** of memory. With the voxelisation, the 1230 samples/point clouds have less than **1GB** of memory. To build the voxel grid point clouds I used the function *voxel_down_sample* of the *Open3D* library, [ZPK18].

Train parameters

Regarding the train parameters, I also trained the model during 20 epochs with the batch size 20 and the number of points equal to 10000. In the voxel down-sampling, I used the **voxel size equal to 0.00001**. This value was chosen by trial and error until I get enough voxelisation to represent the point cloud in more or less 10000 points.

Results

The following table 5.6 shows all the results in this test:

	Mean Loss			**Mean Accuracy**			
Iteration	Train	Val	Test	Train	Val	Test	Iteration time (seconds)
1º Iteration	0.5888	0.6112	0.6998	0.7042	0.6590	0.6900	982
2º Iteration	0.5782	0.6067	0.4503	0.7122	0.6905	0.7500	988
3ª Iteration	0.5919	0.5685	0.4077	0.7043	0.7145	0.7000	987
Mean	0.5863	0.5955	0.5193	0.7069	0.6880	0.7133	986

Table 5.6: Results in testing the PointNet with the Custom Filtered Dataset with the voxel down-sampled, original-sized point clouds.

PointNet with or without voxelization

Comparing the results of using the Pointnet with the original sized point clouds, 5.5 with the results of using the Pointnet with the voxel down-sampling, 5.6 we can see that, in the last approach, the mean iteration time is shorter as the mean loss in the train, validation and test set. The mean accuracy in the train, validation, and test set is bigger in the ap-

proach that uses voxel down-sampling. Merging the results of both approaches we can see more clearly which one is more suitable for the portable system 5.7.

	Mean Loss			Mean Accuracy			
Approach	Train	Val	Test	Train	Val	Test	Mean IT (sec)
Original dataset	0.6196	0.6040	0.6346	0.6683	0.6570	0.6167	16298
Voxel down-sampled dataset	0.5863	0.5955	0.5193	0.7069	0.6880	0.7133	986

Table 5.7: Mean loss, accuracy and iteration time values between using the Pointnet with the original sized point cloud and with voxel down-sampled point clouds. *IT* stands for iteration time.

It's important to say that initially, I was saving only the mean values in each iteration of the train and validation set. Later on, I changed my method of presenting the results to save the model/epoch with the best accuracy value in the validation set and the accuracy value in the train set in that epoch. After all the epochs in one iteration, I used the model with the best validation accuracy to test in the test set. The following table 5.8 represents the comparison between using voxel down-sampling in the original dataset and not using it with this modification.

	Mean Loss			Mean Accuracy			
Approach	Train	Val	Test	Train	Val	Test	Mean IT (sec)
Original dataset	0.5898	0.4966	0.5443	0.6873	0.7567	0.7333	15419
Voxel down-sampled dataset	0.5115	0.3995	0.5553	0.7673	0.8067	0.7400	954

Table 5.8: Mean results of using the best model of each iteration between using the Pointnet with the original sized point cloud and using voxel down-sampled point clouds. *IT* stands for iteration time.

Conclusions between using voxelization point clouds

Analysing the table 5.8, I concluded that using voxel down-sampled point clouds will give better results, the main accuracy in the validation and test set is much better in the approach that uses voxel down-sampled in comparison with the approach that doesn't use it. The iteration time is also a lot smaller in the voxel down-sampled approach.

The voxel down-sampled approach seems to be the best in terms of better accuracy because the point clouds have about 10000 points and the random parameter in the pointnet model, **number of points**, will select exactly those 10000 points in contradiction to the other approach where each point cloud have around 300000 points and sometimes the network wouldn't select the best 10000 points that better represent the point cloud. The biggest problem of this approach is the **time that takes to convert a normal point cloud** into a voxel down-sampled point cloud. It's important to say that I built the dataset of normal point cloud and the dataset of voxel down-sampled point clouds before running the pointnet, which means that the iteration time doesn't have into account the time that it takes to downsampling the point clouds.

In the portable system, the voxel downsampling takes more time than the actual prediction in the pointnet, what the system improves in the accuracy decreases in the time. The time it takes to predict one point cloud in the pointnet method is around 0.1 seconds in the *Jetson Nano* portable system. The time it takes to down-sampled the point cloud before using it in the pointnet is around 0.4 seconds using the method of the open 3d library. Because of this, I opt to use

5.8.3 Train Pointnet with cropped point clouds

I tested the difference between using the original point clouds with the voxel down-sampled point clouds, but the main goal of these tests was to see if the semantic segmentation was really necessary to predict if the door was open or closed and other problems that the visually impaired have in indoor spaces and this system might solve. Recalling the first proposal to solve the *Door Problem*, 5.3, after doing the semantic segmentation to know the location in the RGB image of the door, the depth image is cropped and used in the pointnet to classify if the door was open or closed. In this test, I used the same filtered dataset, but instead of using the original point clouds, I cropped the point clouds according to the location of the door in the RGB image with the objective to replicate the semantic segmentation section.

Dataset used

The dataset used in this test was the same as the previous one, but instead of having the original point clouds, the point clouds are cropped according to the location of the door in those point clouds. 1030 samples for training, 100 for testing, and 100 for validation. The original dataset had almost 18 GB of memory while the cropped dataset had around 9 GB of memory.

Train parameters

The parameters of this test were the same as the two previous one, 5.8.1, 5.8.2 , 3 iterations, 20 epochs for training with the batch size 20 and the number of points equal to 10000.

Results

The following table 5.9 shows the results for this test:

	Mean Loss			**Mean Accuracy**			
Iteration	Train	Val	Test	Train	Val	Test	Iteration time (seconds)
Iteration 1	0.5136	0.2713	0.6603	0.7696	0.8600	0.7000	7635
Iteration 2	0.5568	0.4399	0.6420	0.7294	0.8200	0.5600	7671
Iteration 3	0.5389	0.4519	0.4479	0.7392	0.8400	0.6900	7688
Mean	0.5364	0.3877	0.5834	0.7461	0.8400	0.6500	7665

Table 5.9: Results of using the best model of each iteration using the Pointnet with cropped point clouds

Results analysis

Analysing the results, I concluded that the mean iteration time was smaller for cropped point clouds as expected. Although the point clouds only had information about the door, the accuracy-test results should be bigger in this approach but, compared with the initial approach, with the original point clouds, the accuracy was smaller.

5.8.4 Merge of all the approaches

Due to the increase in the number of tables, it was better to summarise all the tests in the point net and merge all the results in one table for better analysis. It was also tested, despite the three approaches, an approach that uses cropped point cloud with voxel down-sampled with the same parameters as the previous tests. The following table summarises all the results in the pointnet:

	Mean Loss			**Mean Accuracy**			
Approach	Train	Val	Test	Train	Val	Test	Mean Iteration time (sec)
Original dataset	0.5898	0.4966	0.5443	0.6873	0.7567	0.7333	15419
Voxel down-sampled original dataset	0.5115	0.3995	0.5553	0.7673	0.8067	**0.7400**	954
Cropped dataset	0.5364	0.3877	0.5834	0.7461	0.8400	0.6500	7665
Voxel down-sampled cropped dataset	0.5309	0.5034	0.4632	0.7555	0.7967	**0.7400**	**492**

Table 5.10: Summary of all the best models results in each approach for the Pointnet 3d object classification

Analysing the summary table it can be seen that there was a draw in terms of mean test accuracy using the best validation accuracy model between the approach that uses voxel down-sampled point clouds and the approach that also uses voxel down-sampling but with the cropped point clouds. As I said before, the voxel down-sampled approach is faster to train but is slower to predict and inference when compared to the non-voxel down-sampled approach.

Another interesting result was to compare the original approach with the cropped dataset approach. The cropped point cloud simulates the result that the semantic segmentation model sent to the pointnet and as it can be seen, it was better to give to the network all the information around the door than just the door itself. This was a very important result because in the first proposal to solve the *Door Problem* I proposed to use semantic segmentation to know the location of the door, use it to crop the depth image and use only that crop point cloud to classify and solve the problem but now that section can be discarded. The accuracy of the test set in the original dataset approach is bigger than the accuracy in the cropped dataset approach.

I conclude with these tests that the semantic segmentation wasn't needed for door classification; the simple use of the pointnet can solve the problem. The semantic segmentation can be used to detect the location of the door and to inform the visually impaired people that same location, for example, if it is at his right size or left size. One big problem of using the entire point cloud to classify instead of using just the area around the door is when the situation has more than one door. One door can be closed, and the other can be open, for example, what should the algorithm return? Another big problem would be when there isn't any door. Should be created a class (*No-Door*) for this situation? Well, since the *Door Problem* happens in locations where the person knows the locations of the doors, he could only use the program when he is facing the door.

5.9 Testing in Jetson Nano

I started to test the program and the algorithms in the single board computer *Jetson Nano*. The Raspberry Pi 4 doesn't have an integrated GPU, and it was necessary to use an *Edge TPU* accelerator to run neural networks and all the machine learning algorithms in real-time.

5.9.1 Installations

I installed the necessary libraries and packages to run the prototype program except for the semantic segmentation algorithm *FastFCN* repository. After several searches, I came to the conclusion that this package wasn't compatible with the Jetson Nano system because at the moment no one had installed the package successfully in any kind of these devices.

To replace the semantic segmentation algorithm I used the *Fully Convolutional HarDNet*, [CKR+19], which is an implementation based on the Harmonic *DenseNet*, a low memory traffic network. I chose this method because it was one of the fastest in terms of FPS, and it had already the *ADE20K* data loader implemented, which has the same structure as the dataset that I built. This algorithm was installed in the Jetson Nano without any problem.

At this point, the strategy or method to solve the *Door Problem* wasn't fixed, so a new version of the prototype program was created. This new version of the prototype program, instead of using both semantic segmentation and 3d object classification to predict if the door was open or closed, only uses the 3d object classification. The goal was to see the difference in fps between the two versions of the program and also to see if the method runs in real-time in the Jetson Nano which is much less powerful than the lab computer.

Although the use of only the Pointnet to classify the opening of the door was better than using both semantic segmentation and Pointnet it has its disadvantages. The use of only the Pointnet only works in environments where the visually impaired person knows the

place and the locations of the doors. If this prototype program was used in an environment that wasn't known by the person, it wouldn't work very well because the program couldn't tell where the door was. If there was more than one door in the scene, the program would only predict as if there was only one door, and its answer would always be incorrect because the program must answer for each door.

Because of the previous reasons, I didn't discard the semantic segmentation approach completely since it could still be used to improve the program to solve the *Door Problem* and it would also be crucial to other tasks that the portable system would solve.

The following section presents a quantitative evaluation of the results of testing the programs in the *Jetson Nano*.

5.10 Testing the program between different versions of Jetpack

For this project there were assign two *Jetson Nano* and at the time this project was being made the *Jetson Nano* Developer Kit, *Jetpack* release a new version with improvements in the OS, *TensorRT*, *cuDNN*, *CUDA* and others improvements, the Jetpack "4.3".

Two versions of the *Jetpack* were tested, the old version 4.2 and the newest version 4.3. (As it was already said in the Project Material Section, later I installed the *Jetpack* version 4.4 in one of the *Jetson's* while the other remained with the version 4.3) Two versions of the prototype program to solve the *Door Problem* were tested. The version ***B*** it's the fastest version that only uses 3D object classification to classify if the door is open or closed. The second version, ***A***, uses both semantic segmentation to crop the original point cloud and them it uses the *PointNet* to classify.

One of the biggest problems of the single board computers was their heat dissipation limitations. These devices normally overheat very quickly, and the system throttles. Because of this, I tested the script with a Fan and without it, after running the script for 15 minutes to simulate the overheat the system. Further tests about the Jetson Nano temperature will be approached in this section.

The following table represents the tests in the Jetson Nano between the two versions of the *Jetpack*, between having the *Jetson* cold or hot and between the two versions of the prototype script to solve the *Door Problem*.

Table 5.11: Results in testing two different *Jetpack* versions in two programs with and without fan in terms of time per frame prediction.(Program version A uses Semantic segmentation and Pointnet and version B only uses the Pointnet to predict)

Jetpack version	Program version	Fan	Mean time per frame(sec)	Standard deviation (sec)
4.2	A	×	0.4476	0.0040
4.2	A	✓	0.4447	0.0039
4.2	B	×	0.1543	0.0010
4.2	B	✓	0.1567	0.0028
4.3	A	×	0.4264	0.0136
4.3	A	✓	0.4211	0.0052
4.3	B	×	0.1563	0.0020
4.3	B	✓	0.1569	0.0033

After analysing the results of 5.11 it was clear that the use of a Fan to cool the system so it wouldn't throttle didn't have almost any effect in the performance of the programs. Although the temperature rises and the systems get hot, the run of the prototype system was independent of that and kept the same frames per second as it was running when the system was cold which was excellent taking into account that the user will use the system very often and it will eventually overheat.

Comparing the two versions of the program, the system gets at least five frames per second with the **B version** and gets at least 2 frames per second with the **A version**. Taking into account this fact, I considered that the best method that I had to distinguish between just open and closed doors was program version *B*, just use the *PointNet*.

Finally, comparing the two different *Jetpack* versions, they don't have any difference in the mean times per frame except for the version **A** of the program. In the newest *Jetpack,* version the script **A** is faster and the mean time per frame is smaller.

5.11 First prototype portable system for real-user

After all the testings in the single board computer, *Jetson Nano*, and after building one final prototype approach to solve the *Door Problem*, the final step was to prepare the system to be used by a real-user, a visually impaired person so I could get feedback.

5.11.1 Speed up the Jetson Nano start up

Before I concerned about the architecture of the portable system, there were still some improvements that could be done in the *Jetson Nano*.

The boot time of the *Jetson Nano* could be faster and that was important because the system would get shut down when the user wasn't using it, and it must be fast to boot when the user needs it.

The original boot time of the *Jetson Nano* with the *Jetpack* version 4.3 was **36.00** seconds. The best way to reduce the boot time was to disable startup programs that wouldn't be used in the program to help the visually impaired people. After removing by trial and error startup services like the *gdm* service, the *networkd* service, the *ubuntu-fan* service, the *snapd* service among others, the boot time was reduced to **27.50** seconds.

5.11.2 Auto start Program after boot

Another improvement that was made was to auto start the script right after the system boots. This was also very important because the user didn't have to concern about starting the program because the program would start after the *Jetson Nano* boots.

To do this, firstly I enable the auto-login by changing the configuration file so it wouldn't be necessary to login in manually. To auto-execute the script after the boot I added in the *bashrc* file the execution command to run the script and then changed the startup configuration file to start the terminal automatically after the system boots.

5.11.3 Improved approach - *Semi-open* class

As was said, the prototype approach, after testing different methods was to simply use the *PointNet* to classify between *open* and *closed doors*, just two classes to simplify the problem. The main goal of the *Door Problem* was to prevent the visually impaired people from hitting with their heads in the edge of the door. The most dangerous case is when the door is semi-open or semi-closed. In this case, the model that I built in the *PointNet* would say that the door was closed because all of the images with the door semi-open were labelled as closed doors.

To solve this issue, I add a third class to the model, the ***semi-open*** class to distinguish between totally closed doors and semi-closed or semi-open doors. With this improved approach, the system would give more information about the position of the door, and the user will be aware of the door was *semi-open*, which was the most dangerous and important case.

5.11.4 Add Sound

Sound is the best way to communicate with visually impaired people. I added to the script *bip* sounds when the same starts, so the user knows that the system is loading the model weights and getting ready to start predicting.

I also added sounds when the system predicts *open, closed, and semi-open* doors as it was already said in previous sections of this report. These sounds are generated using the ***Google* text to speech library** but they won't be played in the script every time the program inference an input point cloud. As the best approach of the moment could make at least five inferences per second the sound could only be played after every five frames, and, the answer would be the mean inference of the five frames. For example, if three frames would say that the door was open and two would say that the door was semi-open, the final answer would be that the door was opened.

5.11.5 Building of the prototype portable system version 2.0

The original idea was to have a portable system constituted by:

- a single board computer (*Raspberry Pi* or *Jetson Nano*).
- a power bank to power the system.
- a 3D camera .
- some in-Ear phones.

These were the original items that would be part of the portable system, but the biggest problem was how to merge these items and fuse them in one single portable system simple enough to visually impaired people use it.

In all the previous four components, the component which was more complicated to fuse was the *Jetson Nano* because the same was very fragile and it had to be cover by some kind of box. Because of this, I started building the system using as a base the single board computer.

In figure 5.7, we can see that there are four holes to use screws in each corner of the *Jetson Nano*. Using these four holes as base a box was built using a 3D printer that could fit the

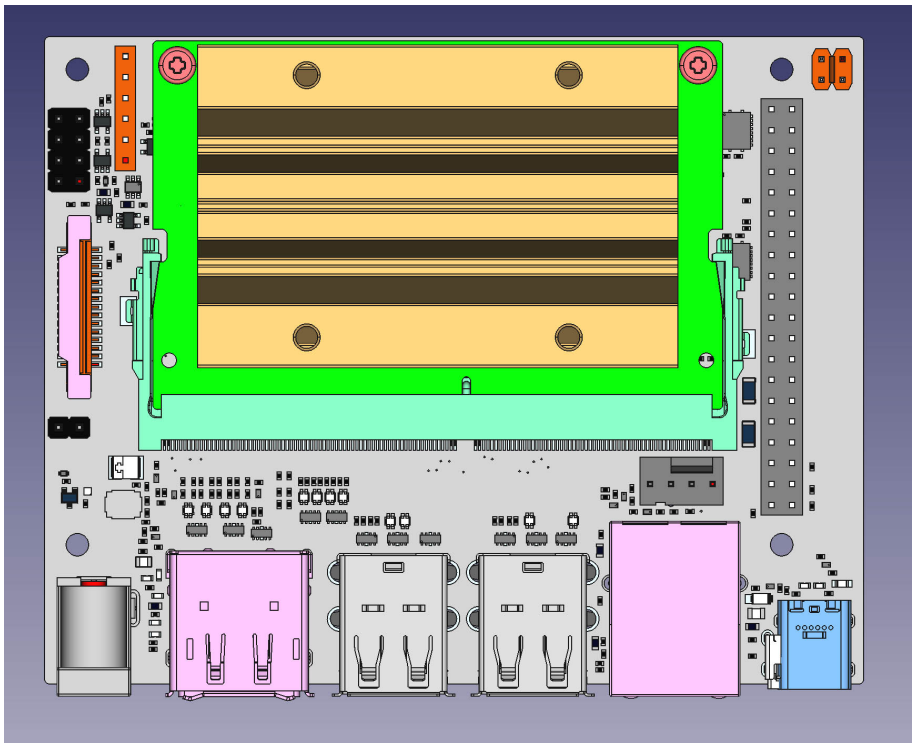

Figure 5.7: Jetson Nano top view from [Nvi19].

Jetson Nano using screws. Also, inside of this box are the power bank and all the cables. The box has one gap to let the cables pass through, namely the camera cable and the hear hook headphones cable. It also has one USB entry to charge the power bank. It's only one entry to be simpler to the visually impaired user.

The biggest problem of this system would be how to turn on and off the *Jetson Nano* without damaging it, because by default, the *Jetson* doesn't have any power on/off button. It's possible to power off the *Jetson Nano* without the need to unplug the power cable using the J40 pins. The pin 7 and pin 8 in the J40 disable the auto power on and the pins 1 and 2 initiate power-on if auto power-on is disabled. Using a button connected with the pins 1 and 2, we can turn on and turn off the *Jetson Nano*. The single issue about this method is with the power bank, as it turns off after not receiving any signal from the *Jetson Nano* for 20 seconds. If the user turns off the *Jetson Nano*, after 20 seconds the power bank will turn off, and after that, it's impossible to turn on the *Jetson* without first turning on the power bank using its button. The solution was to use a single button that turns on the power bank and the *Jetson Nano* and turns off only the *Jetson Nano* because after 20 seconds, the power bank will turn off automatically. This way, the system will turn on without any problem and the user can turn off without damaging the SD card because the *Jetson* turns off before the power bank. The biggest advantage is to save energy because both the *Jetson* and the power bank don't need to be always on.

5.12 Generic Obstacle Avoiding Mode

I built two modes for the **prototype portable system version 2.0**. One mode more correlated with just the *Door Problem* which I called the *Door Problem Mode* and the other more related to obstacle avoiding which I called *Generic Obstacle Avoiding Mode*.

Unlike the *Door Problem Mode*, the *Generic Obstacle Avoiding Mode* is a more general mode that will help the visually impaired people to navigate by informing them of the object's distance in the environment. This was done by using a similar approach to how the parking car sensors work. Using the depth information of the 3D *RealSense camera*, it's possible to inform the user if there are obstacles within a specific threshold.

I built a program for this mode which would get the depth information of the *Realsense camera* in the form of a matrix with the size 640 x 480. This matrix was divided by lines, so it was possible to see if the obstacles were closer to the left side or on the right side. *Threads* were used to speed up the process to go through each element/pixel of the matrix to see its depth value. In total, the program uses eight threads for each frame/matrix. Each thread represents a region of the point cloud. In the figure, 5.8 is represented the base operation of this Mode, but instead of using eight threads in the figure I just represent four threads, 2 for the left side and 2 for the right side.

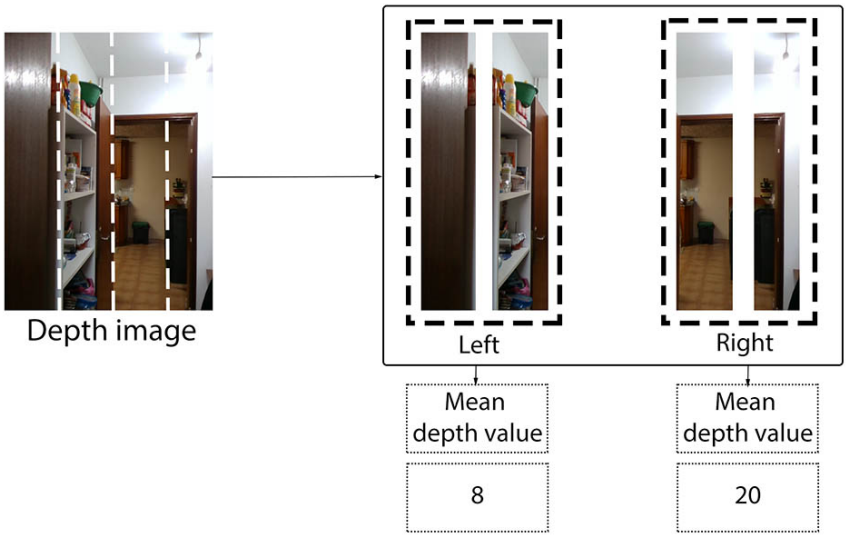

Figure 5.8: Operation of *Generic Obstacle Avoiding Mode* - Depth image is divided in columns and for each column the mean depth value is calculated.

The first lines of the matrix normally would represent the top region of the image, but as the camera is rotated 90 degrees, the first lines represent the left side of the image, and the last lines of the matrix represent the right side of the image. In other words, the first four threads correspond to the left region of the image, and the last 4 represent the right region. As the matrix has 480 lines, each thread processes 60($480/8 = 60$) lines. For

each thread, the mean value of the depth information between the 60 lines of that thread is calculated. After having the mean value for all the eight threads the mean value of the first four threads which will correspond to the left side and the mean value of the last four threads corresponding to the right side was calculated.

According to these mean depth values, the sound would get higher if those values were small (meaning that the obstacles were closer). The sound will get lower as the distance of the obstacles gets higher. This Mode works in stereo, in the way that the sound gets higher in the left headphone if the mean value of the left side is smaller and the same goes for the right side. This way, the visually impaired person can have an idea of the environment around him and avoid obstacle collision in unknown places.

This mode is always in a loop doing the previous process. If the mean values of both sides are smaller, meaning that the distance between the obstacle is smaller the sound will get faster exactly like the *beep* sounds of the parking car sensors systems. If there aren't obstacles in a certain threshold, no sound will be played, so the user can relax since constantly hearing *beep* sounds may be tiring.

This system/mode may seem useless because, usually, when the visually impaired people are walking in the street they use a cane to help them navigate and avoid obstacles, but they can't avoid all of them. The most dangerous obstacles are the obstacles at the same level as a person's head, for example, a tree branch or a fallen signal. As the cane is usually used in the ground region, these kinds of obstacles won't be detected by the cane, and the person would collide with them. The biggest advantages of the *Generic Obstacle Avoiding Mode* is that these obstacles will get detected, because the camera can cover both the ground and the level head at the same time, allowing the user to avoid all the dangerous obstacles as it is represented in figure 5.9.

Figure 5.9: Advantage of using the *Generic Obstacle Avoiding Mode*(On the middle image the user collides with the fallen tree since the white-cane doesn't work at the head-level. On the right image, the user uses the portable system and the same informs him about the nearby obstacle).

From this point of the project, *Sérgio Gonçalves*, finalist student of computer engineering, improved this system by reproducing sounds in a 3D matrix that represents the depth data from the 3d camera.

5.13 Power Bank Issues

The power bank used for the prototype version 2.0 was a *Techlink* power bank, 20000mAh, Dual USB with 2.4A fast charging as it was already described in the Project Material section. The official power supply considerations for *Jetson Nano* recommend micro-USB power supplies with **5V-2.5A**. *Jetson Nano* runs in two modes, 10 watts mode and 5 watts mode. Until now, all our tests and experiments were tested with the 10 Watt mode, which is the most power-full one allowing the system to work with four cores instead of only 2(5 watts mode).

After merging the two developed modes(*Door Problem Mode* and *Generic Obstacle Avoiding Mode*) in one script, I came to the conclusion that the power bank wasn't supplying enough power to the *Jetson Nano* in 10 watts mode. The minimum power supply for *Jetson Nano* is 2.0 A, but with the 3D camera and the in-earphones, the power needed is bigger(>2.0 A). Although the power bank used didn't have the capability to 2.5 A it had 2.4 A, which is very similar and should have been enough to power up all the systems. Due to this problem, we develop several tests and experiments to actually see if the power bank had the necessary power to power up the system or not. We also compared with two other power supplies, the *Raspberry Pi Universal Power Supply - 5V 2.5A* and a *Barrel Jack Power Supply - 5V 5A*.

Due to some system malfunction, the power-bank couldn't provide enough power to the *Jetson Nano* even without running any program and any connected devices. The *Jetson Nano* turned off after 5 seconds when using the power-bank to powered it. I even try to change the *Jetson Nano* mode to 5 watts, but I got the same effect. Using a USB voltage tester, I analysed the voltage, current, and power that was being provided from the power-bank to the *Jetson Nano*. I also analysed these measurements for the aforementioned power supplies.

Table 5.12 treats the results in terms of voltage, current, and power provided to the *Jetson Nano* using different power supplies. I measured the system with and without running the script.

Table 5.12: Voltage, current and power measurements provided to *Jetson Nano* from different power supplies with and without the script running.

Power supply	Script is running	*Jetson* goes down	Voltage (V)	Current (A)	Power (W)	Power factor
Power-bank	×	✓	4.5	0.42	1.63	1
Raspberry micro-USB	×	×	5.0	1.3	4.2	0.64
Raspberry micro-USB	✓	×	5.0	3.8	10.5	0.55
Barrel Jack	×	×	5.0	1.9	4.2	0.44
Barrel Jack	✓	×	5.0	4.8	10.5	0.44

We can confirm that the power-bank previous acquired, due to some system malfunction doesn't provide the current (0.42 A) that it should (2.40 A). It can only provide 1.63 watts which aren't enough to even start-up the *Jetson Nano* (4.2 watts). If the power-bank had the current it should (2.40 A) with a voltage of 4.5 V and a power factor of 1, it would provide about 10.8 watts, which would be more than enough to power up the *Jetson Nano* while running the script in 10 watts mode, (P(watts) = $1 * 2.4 * 4.5 = 10.8$).

After research, I came to the conclusion that the power bank wasn't supplying enough power due to some defect in cable connection. To solve this, I welded a Barrel Jack cable in the Fast charging USB pins. After that, the *Jetson Nano* never turned off again using the power bank as the power supply. Even if the script was running, the power bank would still be able to power it.

5.14 Method A and B - *Door Problem*

So far, I have been focused on the *Door Problem*. Two approaches were developed for solving this problem, one uses semantic segmentation with 3d object classification, and the other only uses 3d object classification. As was said previously, the first method has higher validation accuracy values but is slower since it adds semantic segmentation when compared to the second method. From now on, I will denote the first method, ***Method A*** *for Door Detection* and the second method, ***Method B*** *for Door Detection*. Figure 5.10 represents method *A* for door detection and figure 5.11 represents method *B*.

5.14.1 Method A - 2D Semantic Segmentation and 3D Object Classification

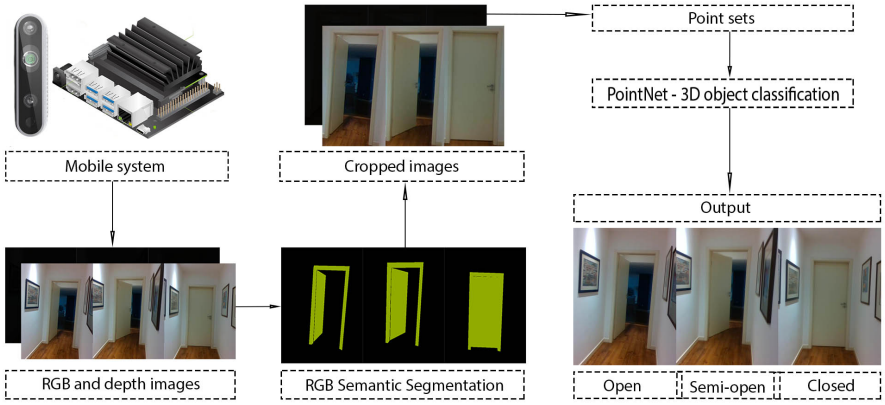

Figure 5.10: Algorithm of Method ***A*** (2D semantic segmentation and 3D object classification).

Method A was the first approach that I developed for solving the *Door Problem*. It's based on 2D Semantic Segmentation and 3D Object Classification. The prototype system version 2.0 captures the RGB and depth information through the camera. The RGB image is used as input for the 2D Semantic Segmentation. The semantic segmentation only uses 2 classes, one for *door/doorframe* and the other for *no-class*. The biggest *door/doorframe* area in the semantic segmentation output is calculated with the goal to obtain a bounding box around that area. Then, the depth image is cropped according to the bounding box location. This depth information is the input of the 3D Object classification, and this network returns three values. Each value corresponds to a class(open, closed, and semi-open). The output class is the highest value of these three. This method works at **3 FPS** in *Jetson Nano*. Each frame inference takes around 0.28 seconds.

5.14.2 Method B - 3D Object Classification

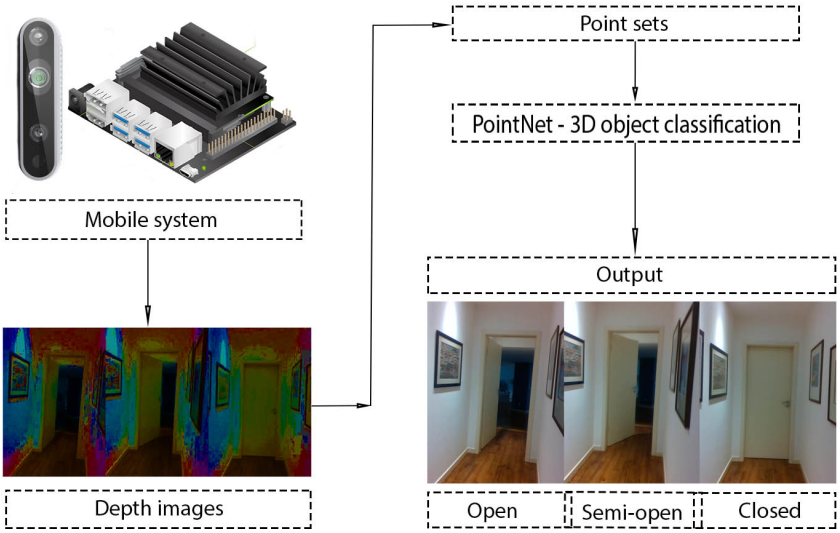

Figure 5.11: Algorithm of Method ***B*** (only 3D object classification).

Method B is very similar to the previous method with just one difference. Instead of sending the cropped depth images to the *PointNet*, this method sends the original size depth images. It doesn't use 2D semantic segmentation, just 3D object classification. The output works exactly in the same way as the previous method output works. This method works at **5/6 FPS** in *Jetson Nano* since it doesn't use the 2D Semantic Segmentation part. Each frame inference takes around 0.15 seconds.

Until now, the *PointNet* and the *semantic segmentation FastFCN* results were made based on the previous approach (2 classes only, open and closed door). These methods were tested again in the final approach of the *Door Problem*, 5.11.3, which uses three classes, open, closed, and semi-open doors.

I compared both methods (*Method A and B for Door Detection*) against each other in real-time scenarios. Two semantic segmentation algorithms used in *Method A* as the *PointNet* were trained in the *desktop lab machine*.

It's important to mention that I didn't change any of the algorithms used in the methods as the *PointNet*, *FastFCN*, and *FC-HarDNet*. I only changed the data loaders and did the necessary configurations to work with the data sets.

The dataset used for these methods was the **Door Dataset - Version 1.0**. This dataset was built using the last filtered dataset with just two classes labelled. This last dataset had 615 closed-door images and 479 open door images. The *Door Dataset - Version 1.0* has 588 closed doors images, 468 open doors and 150 semi-open doors images. At this point in the project, this was the dataset used in all the experiments.

Experiments in *Method A for the Door Problem*

I compared the accuracy and speed of *FastFCN* and *FC-HarDNet* semantic segmentation algorithms in method *A*.

I built a dataset (*Door Semantic Segmentation sub-dataset - version 1.0*) for training the semantic segmentation algorithms that used part of the RGB images from the *Door Classification sub-dataset - version 1.0*. To built this dataset, I used the Computer Vision Annotation Tool (*CVAT*), which allows us to draw polygons in the RGB images that represent one class.

This dataset has 240 grey-scaled images with the size 480 x 640. I used the pixel accuracy and mean intersection over union as the evaluation metrics for these tests. I also compared the training and inference time in the aforementioned desktop computer.

Table 5.13: Comparison between using the *FastFCN* and the *FC-HarDNet* algorithms in *Method A for Door Detection*.

Method A with	Test pixel accuracy	mIoU	Training time (sec)	Inference time (sec)
FastFCN	0.909	0.808	567	0.515
FC-HarDNet	0.701	0.418	426	0.019

The *FastFCN* algorithm has better results in pixel accuracy and mIoU in the test set when compared with the *FC-HarDNet* algorithm, but the focus in this project was in real-time door classification/detection methods. The *FC-HarDNet* is not as good as the *FastFCN* at door segmentation, but it had a much smaller inference time (was more than 20 times faster) and more importantly, it was compatible with the *Jetson Nano*. Taking this into account, I opted to use the *FC-HarDNet* algorithm in *Method A for the Door Problem*.

Experiments in *Method B for the Door Problem*

For *Method B*, I was concerned about the one parameter of the *PointNet*, the **number of points** that this model randomly selects from the input point set. These tests were done previously in 5.8.4 but with the old dataset, with just two labelled classes.
As the focus was in real-time door classification/detection methods, I built a downsampled version of our dataset for *PointNet* using the voxel downsampling tool from the *Open3D*, [ZPK18] library. As the *PointNet,* randomly selects the number of points in the point clouds, the points selected in the downsampled point cloud will better represent the point cloud because the downsampled cloud has fewer points (30000 on average) compared with the original cloud (307200 on average). The goal here was to see if it was worth it to downsample the point clouds taking into account the time it takes to do it and the improvement in validation accuracy compared with the original point clouds.

Table 5.14: Evaluation of Method B with the original size point clouds in *PointNet* and using downsampled point clouds.

Point cloud size	Mean validation accuracy	*Jetson Nano* inference time(sec)	Downsampling time(sec)
30k	0.428	0.111	0.386
300k	0.417	0.111	-

I trained the *PointNet* during ten epochs with a batch size equal to 20 and K=10000. For each approach, I trained three times and used the best validation accuracy value. I used a voxel size, in the voxel downsampling *Open3D* tool, that produced a proportion of 10 to 1 in the downsampled point cloud.

Table 5.14 represents the difference between using the original size and the downsampled point clouds. The mean validation accuracy was a little better in the downsampled point clouds as expected. The *PointNet* is more likely to select points that represent the point cloud uniformly since these have fewer points than the original size ones. The inference time in *Jetson Nano* was the same for both approaches since the number of points selected was the same (*number of points*=10000) but with the downsampling time, the downsampled approach was almost five times slower than the original one ($(0.111 + 0.386)/0.111 = 4.47$). In view of the above and taking into account that the focus was on real-time methods, I opted to used the original size point clouds and discarded the downsampling for *Method B of the Door Problem*.

Method A vs Method B for Door Detection

Method A has semantic segmentation that isn't used in *Method B*. Certainly, *Method B* is faster, but the addiction of semantic segmentation removes unnecessary information for the object classification, which could lead to better results in terms of accuracy. I compared both methods with respect to speed and test accuracy. I created another version of the 3D dataset with cropped point clouds that represented the output of the semantic segmentation module from the first method. This dataset was exactly equal to the original in terms of sample numbers, and the distribution in the test, validation, and train set was also the same. I trained the *PointNet* with this new dataset, and I compared the results with the original dataset. This way, I could compare both methods assuming that the semantic segmentation module returned the correct cropped point cloud.

Analysing the results of table 5.15, I came to the conclusion that the addition of semantic segmentation in method A doesn't pay off the time it takes because of the difference in test accuracy. *Method A* takes twice as long when compared to *Method B*. It's the semantic segmentation time plus the inference time of the *PointNet*.

Although I'm removing unnecessary information on the point cloud, I'm also removing information about the door surroundings, which has an important role in helping classifying doors. This was the justification for the small difference in test accuracy between method A and method B.

Table 5.15: Comparison of the methods assuming that the semantic segmentation module is returning the correct output.

Method	Mean test accuracy	*Jetson Nano* inference time(sec)	Segmentation time(sec)
A (after segment.)	0.494	0.111	0.131
B	0.433	0.111	-

5.15 Method *C* - *Door Problem*

From the previous said, it's clear that the developed methods for solving the *Door Problem* are fast and in real-time but the test accuracy wasn't the desired (0.494 for *Method A* assuming the semantic segmentation module segments correctly the doors and 0.433 for *Method B*).

The official creators and developers of *Jetson Nano* released new documentation and tutorials for using computer vision algorithms in real-time specifically for *Jetson Nano*. We can take advantage of the *NVIDIA's TensorRT* accelerator library to perform real-time image classification, object detection, and semantic segmentation in *Jetson*.

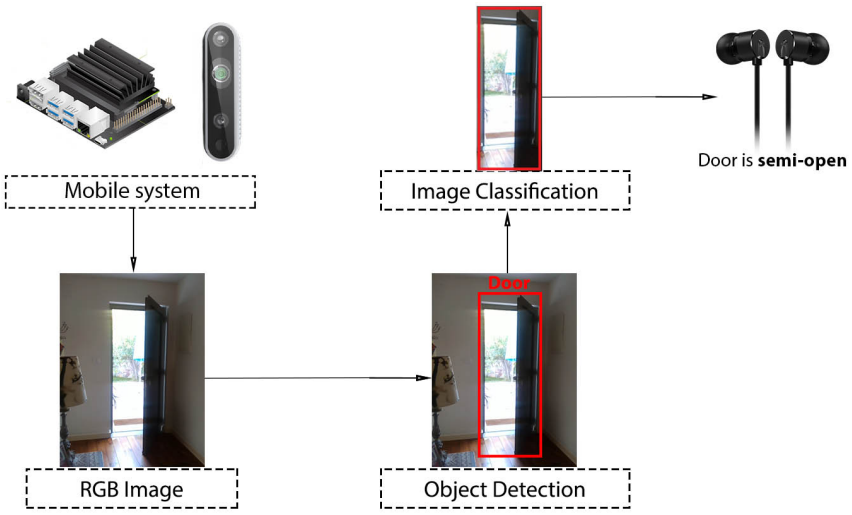

Figure 5.12: Algorithm of Method *C* (2D Object Detection and 2D Image Classification).

The big problem earlier in this project was to perform these algorithms in real-time. With the addition of this documentation for *Jetson* I could use real-time methods as the *AlexNet* and *Detectnet* and implement them for the *Door Problem*. The idea was to use firstly one real-time object detection method to detect the door and crop the image according to the door detection. After getting the RGB cropped image, which now only contains the door, I would use one real-time image classification method to classify the door (open,

closed, or semi-open).
Figure 5.12 represents the algorithm of **Method C** for the *Door Problem*. Using the Realsense camera, the portable system captures the depth and RGB channels. With the RGB Image, this method uses an object detection or semantic segmentation method to get the location of the Door. The RGB image is cropped with the information on the output of the previous step.
With the RGB information, a 2D object classification model is used to classify the cropped image in three different classes. The information is provided to the user via in-Ear phones, with the class of the door and how distance it is from the user. This method works at **7/8 FPS** in *Jetson* if it uses the Object Detection *DetectNet* and it works at **1/2 FPS** if it uses the semantic segmentation approach. These two variants, object detection, and semantic segmentation, will be discussed later in this report chapter.

5.15.1 *Jetson inference* repository

Jetson inference is a repository that helps deploying deep-learning inference networks such as *ImageNet*, [DDS+09] and *DetectNet*, [ATS16] with *TensorRT* and *NIVIDA Jetson*. (The *TensorRT* concept will be approached in the next section) The repository has several tutorials and guides in real-time object detection, image classification and semantic segmentation. It uses the *DIGITS* tool from *NVIDIA* which is a *GUI* for training neural networks. *DIGITS* is used for managing datasets, designing and training neural networks and monitoring the training in real time. This tool was used in this project because it had all the conditions for using the neural network models and apply them to door detection and classification.

The repository was installed in the lab *Jetson* as all the necessary configurations for using it with the *Realsense* camera. The *DIGITS* tool was installed in the lab computer for training the neural networks, and saved its checkpoints. The checkpoints would them be migrated to *Jetson Nano* through *jetson inference* repository.

5.15.2 Object detection with *DetectNet*

After following and completed all the tutorials successfully for object detection in *jetson inference repository* I started to implement object detection networks in the *Door Problem*.

Firstly, I created a small version of the current dataset for object classification. Both the *doors* and *doorframes* were annotated as class *door* with bounding boxes. One hundred twenty images in total were annotated, 60 for testing, and 60 for training. This dataset only had door images and 2 classes, *door* and *dontcare/no-door*. This last one represents all the objects that aren't doors. This dataset was the beginning of the *Door Object Detection sub-dataset - version 1.0*

I used the *DetectNet* model since it was recommended by the *nvidia* developers for real-time object classification in the *Jetson*. *DetectNet* uses the *GoogLeNet* fully-convolutional network (FCN) to perform feature extraction and prediction of object classes and bounding boxes per grid square. There are used two loss functions simultaneously in the training, one to measure the error in predicting the object coverage (*coverage_loss*) and the other the error in object bounding box corners per grid square (*bbox_loss*). To measure the model performance against the validation set it's used the mean Average Precision (*mAP*) metric, the precision (ratio of true positives to true positives plus false positives), and the recall (ratio of true positives to true positives plus true negatives). The intersection over union, which is the ration of overlapping areas of two bounding boxes to the sum of their areas was computed for each predicted bounding box using the ground truth. The predicted bounding boxes can be assigned as true positives or false positives depending on the ground truth bounding box and coverage value. Using a *IoU* threshold (0.7 by default), the bounding box is designated as false negative or true negative (depending on the predict coverage value) if the ground truth bounding box cannot be paired with the predicted such that the *IoU* does not exceed the threshold.

Experiment 1

For the first experiment, I didn't change the *DetectNet* model. The model was trained with the aforementioned dataset during 1000 epochs with a batch size equal to 5 and an exponential decay learning rate starting at 2.5e-05. The model learned nothing until the 400th epoch where the *mAP*, precision, and recall values started to grow. In epoch 10000th, the model had a *mAP* of 0.1077, precision of 0.1700 and recall of 0.5846.

Experiment 2

The results weren't the expected, and I did research on how to increase these values. The *DetectNet*, as default, uses data augmentation, which isn't good for all datasets as we know. I changed the model to use the original images without the data augmentation and trained the network with the same parameters as the previous training. In the final epoch, the *mAP* was 0.0128, precision was 0.0459 and recall was 0.2105. With this, I concluded that the data augmentation in our dataset, contrary to what I thought, improved the model precision.

Experiment 3

In the next experiment, although the data augmentation increased the precision and the other metrics, I kept it down and increased the training dataset instead with door images from other datasets. I used the *DoorDetect Dataset*, [ATS19], which has 149 samples of door images labelled and is freely available online. With the addition of these images, our training set increased from 60 samples to 209 (60 + 149 = 209). The parameters for

training the model remain the same as the previous tests, 10000 epochs, batch size equal to 5, and learning rate starting at 2.5e-05 with exponential decay. In the last epoch, comparing with the previous experiment, the *mAP* increased to 0.0194, precision increased to 0.0583, and recall decreased to 0.1739.

Experiment 4

The *DetectNet* by default uses *data augmentation*, namely, shifts on the images, image rotation, image scale, hue image rotation and image desaturation. In the previous two experiments, I removed all these augmentations to train the model with the dataset only and see the difference. In this experiment, *data augmentation* was added again, such as image rotation, hue image rotation, and image desaturation. The precision value was too low, which means that the model was getting a lot of false positives. In other words, the model was detecting almost every object like a door. I added several images (from *COCO* dataset, [LMB+14]) without any *doors* or *doorframes* to the dataset with the goal to reduce the number of false positives. In total, I added 144 images not containing doors which increased the training set from 209 to 353 $(209 + 144 = 353)$. Another addition that was made in this experiment was in the input image size. The *DetectNet* uses images with size 640*640 by default, and I resized all of the images to that size as well. In the last epoch, the *mAP* was 0.2618, the precision was 0.4432, and the recall was 0.5819. These results were much better than all the results I got from the previous experiments.

Table 5.16 compares all the previous four experiments on *DIGITS* in terms of data augmentation, training set size, precision, recall, *Jetson Nano* inference time and training time.

Table 5.16: Comparison of object detection experiments in *DIGITS* in terms of data augmentation, training set size, validation precision, validation recall and training time.

Experiment	Data Aug.	Training set size	Precision	Recall	Jetson inference time	Training time (hours)
1	✓	60	0.170	0.585	10 FPS	5
2	×	60	0.049	0.211	10 FPS	4
3	×	209	0.058	0.173	10 FPS	13
4	✓	353	**0.440**	0.582	10 FPS	27

Analysing the table, I can conclude that training with data augmentation, for the *Door Problem*, leads to better results in terms of precision and recall. The precision value in the first three experiments was too low even though the recall value wasn't. This was happening because the training dataset only had door images, leading the network to always classify each detected object like a door. Adding other images that didn't contain doors, helped to avoid this problem since the network could learn that the objects in those im-

ages weren't doors. The validation set (60 samples) remains the same to compare all the experiments fairly in *DIGITS*.

5.15.3 Image classification with *AlexNet* and *GoogleNet*

The Door detection module will return one or more image for each *door/doorframe* detected. Each cropped image is going to be classified as *open*, *closed* or *semi open* using a image classification model as the *AlexNet* or *GoogleNet*.

I used the sub-dataset *2D Door Classification* of the *Door Dataset - version 1.0*. This dataset has 1086 images for training(548 open doors, 428 closed doors, and 110 semi-open doors), 60 images(20 of each class) for validation, and 60 images(20 of each class) for testing. Recalling that these images only contain the doors and doorframes for simulating the output cropped image of the object detection model. These images were resized using the function *resize* of the *opencv* to 480x640 with the goal to keep more or less the doors aspect ratio.

Experiment 1

The *AlexNet* neural network was used in the first experiment of door classification in *DIGITS*. The model was trained during 100 epochs using a learning rate equal to 0.02 with a step-down policy and with a batch size equal to 32. The epoch with a bigger accuracy validation value was the 100th epoch with validation accuracy of 90.625, train loss of 0.369, and validation loss of 0.375.

Experiment 2

As the *AlexNet* got good results (validation accuracy greater than 90%), I tested the *GoogleNet* as it was also already implemented in *Caffe* and it worked in *DIGITS* directly without the need to install any additional library. The model, as the previous experiment, was trained during 100 epoch with the same learning rate, the same policy, and with a batch size equal to 16. The higher value of validation accuracy was 64.06, and it was reached in epoch 70th with a train loss of 0.786 and validation loss of 0.837.

Experiment 3

The *AlexNet* model uses data augmentation by cropping the original image, which in this case was 480x640 to a 227x227 image. The input image for *AlexNet* is a 227x227 image. Due to this random factor, the dataset was changed, and using the *resize* function of *opencv* library the images were resized from 480x640 to 227x277 to ensure that the crop would contain all the door information. The other training parameters were the same as the parameters in experiment 1. The best epoch was the 32nd, with a validation accuracy of 96.875, validation loss of 0.145, and a train loss of 0.052.

Experiment 4

As the previous experiment returned the best results, I used the 227x227 images for training and validation again. The other parameters were equal with the exception of the batch size and the learning rate. The default batch size for training the *AlexNet* was 128. In this experiment, this parameter was changed to 6, and the learning rate was also reduced to 0.001. As the batch size was smaller, the number of iterations per epoch increased, and for its consequent the training time. The model reached the 100.00 validation accuracy in epoch 13 and 30 with validation loss equal to 0.026 and train loss equal to 0.006.

Experiment 5

In this experiment the *GoogleNet* was used again but 224x224 images were used instead. As the *AlexNet*, the *GoogleNet* uses data augmentation by cropping the input image in 224x224 sizes. We changed the original dataset used in experiment 1 and 2 using the *resize* function of *opencv* to resize the images from 480x640 to 224x224. The other parameters remain the same as in experiment 2. In epoch 65 the validation accuracy was 90.00 with validation loss of 0.700 and train loss equal to 0.002.

Experiment 6

In experiment 4, by reducing the batch size and reducing the learning rate, the model got better validation accuracy, but the training time was longer. In this experiment, the batch size was reduced from 32, which is the default value of *GoogleNet*, to 6. The learning rate was also reduced to 0.001, as it was in experiment 4. I used the 224x224 images dataset with these training parameters. The best validation accuracy was 93.33 in epoch 34 with validation loss equal to 0.179 and training loss equal to 0.049.

Table 5.17 compares all the previous six experiments on *DIGITS* for image classification in terms of the neural network used; train set batch size, input image size, validation accuracy in the test set, training time and the inference time in *Jetson*.

Table 5.17: Comparison of image classification experiments in *DIGITS* in terms of neural network used, batch size, input images size, best validation precision, validation loss, train loss and training time.

Experiment	Neural Network	Batch size train set	Input Images size	Accuracy(Test)	Jetson inference time	Training time (sec)
1	*AlexNet*	128(default)	480x640	56.67	55 FPS	194
2	*GoogleNet*	32(default)	480x640	36.67	65 FPS	339
3	*AlexNet*	128(default)	227x227	95.00	55 FPS	188
4	*AlexNet*	6	227x227	**98.33**	55 FPS	499
5	*GoogleNet*	32(default)	224x224	91.67	65 FPS	342
6	*GoogleNet*	6	224x224	93.33	65 FPS	636

From the table, it is clear that the *AlexNet* neural network is the most suitable for door classification in the Door dataset compared to the *GoogleNet*. The test accuracy values of *AlexNet* in experiments 3 and 4 are higher than the values obtained by using the *GoogleNet* in experiments 5 and 6.

5.15.4 Development of *Method C*

This section describes the build and the implementation process of the *Method C* for door detection and classification.

In this method, I only used 2D information for door detection and classification, and the 3D was used for providing extra information as it was aforementioned. The *jetson-inference* repository provides all the tools and frameworks to do 2D object detection and image classification, but it doesn't provide examples with these two algorithms together.

First, I used the *DetectNet* for object detection using the model of the best validation epoch in *Experiment 4* in 5.15.2. *Jetson-inference* provides two python scripts for the test and the use of our models. The *detectnet-camera.py*, that, as the name implies, it uses the trained model and provides a window with the objects detected in real-time in the RGB camera channel. The *detectnet-console.py* is a script that also uses a specific trained model but just returns the detected objects of one input image. I used this last one script, but instead of returning the image with the detected objects and writing it in the system, I cropped the image according to the detected bounding box coordinates. I also change the input of the script, instead of using just one image, I am providing it with the RGB channel of the Realsense camera in 60 frames per second. I added the image classification network after the object detection, using the best validation trained model (AlexNet) in *Experiment 4* in 5.15.3. The input of the image classification network is the cropped image according to the detected bounding box. The output of the image classification is the door classification (open, closed, or semi-open).

5.15.5 Speed Evaluation of *Method C*

After implementing the 2D part of *Method C* for door detection and classification, I evaluated *method C* speed in *Jetson Nano* to compare it later with the other developed methods.

To test the speed of this method, the detect time, classification time, and total script time were counted. Each of these times was counted 100 times, and it was calculated the average of each. As it was said several times in this document, *Jetson Nano* has two modes, the 5 watts mode, and the 10 watts mode. I also tested the speed of *Method C* in both of these modes with the goal of saving energy of the power-bank. In the experiments done in 5.15.3 I used two different image classification networks, *AlexNet* and *GoogleNet*. Both of these networks were also tested in terms of speed (inference time) in the *Jetson Nano*.

Table 5.18 summarises the speed tests in the*Jetson Nano* using its two different modes (5 and 10 watts), using the *DetectNet* as the object detection network and using the *AlexNet* and *GoogleNet* as the image classification networks.

Table 5.18: *Jetson Nano* inference time in 5 and 10 watts mode of *Method C*.

Jetson Mode	Object Detection NN	Image Classification NN	Obj. Detect. Inference time(s)	Img. Class. Inference time(s)	Total Inference time(s)
5 watts	*DetectNet*	*AlexNet*	0.1356	0.0231	0.1698
10 watts	*DetectNet*	*AlexNet*	0.0993	0.0193	0.1255
5 watts	*DetectNet*	*GoogleNet*	0.1355	0.0201	0.1697
10 watts	*DetectNet*	*GoogleNet*	0.0965	0.0173	0.1202

As it can be seen in table 5.18, the total inference time isn't the sum of the object detection inference time with the image classification inference time. In the total inference time, it is also taken into account the time that it takes to crop the image after the object detection, the resize operation after it, and other crucial pre-processing methods. The *GoogleNet* is faster than *AlexNet* in the *Jetson* although, the last one provided better accuracy values (5.15.3). It can also be seen a significant difference between using the two modes of *Jetson* in the total inference time.

In short, if the *Jetson* is in 5 watts mode it can perform the *Method C (with DetectNet)* in 5/6 FPS (frames per second), which is more or less the same speed that the *Jetson* performs *Method B* in 10 watts mode. In other words, *Method C* in 5 watts can be as fastest as *Method B* is in 10 watts,. *Method C* has also better test accuracy values considering that the object detection detects the door. If the *Jetson* is in 10 watts mode it can perform *Method C (with DetectNet)* at 8 FPS.

5.15.6 Power-bank Duration in *Method C*

In 5.13, the power-bank for the system (*Techlink* 20000mAh with 2.4A fast charging) was tested using a USB voltage meter. I measured the voltage, the current and the power of the energy provided by the power-bank to the *Jetson Nano*. Instead of the 2.4A of current, the power-bank was only providing energy with a current of 0.42A and the same could handle the *Jetson Nano*. The portable system would shut down after a few seconds (5 / 10 seconds) since the provided energy wasn't enough (1.63W).

This was happening because of the cable that I was using to power the *Jetson* was too weak and couldn't provide the original amount of current (2.4A) that it was supported by the power-bank. This cable was switched with a barrel jack cable powerful enough to power up the *Jetson Nano*.

As the power-bank was now working, it was possible to calculate its duration while performing *Method C* for door detection and classification. This was really a piece of important information since it concerns the viability of the system and how long it can be used.

For testing the real power duration of the power-bank, the same was recharged and used to power up the *Jetson Nano* in 5 watts mode while running *Method C (with DetectNet)* for door detection. The power-bank provided energy to the system for **9** hours and **42** minutes which is a good time since the *Method C* is the method that gets more out of the GPU and uses it at its maximum because it takes advantage of *TensorRT*, designed specifically for running these computer vision algorithms in the *Jetson Nano*.

In other words, the power-bank duration was tested while running only the *Door Problem Mode* using *Method C*. In a real case scenario, the visually impaired person would switch between the *Door Problem Mode* and the *Generic Obstacle Avoiding Mode*. This last mode in terms of power consumption uses less energy which means that the power-bank can at least provide energy to the system for 9 hours and 42 minutes, but that isn't its limit.

5.16 Temperature Experiments in *Method C*

Jetson Nano is a single board computer that has only a heat-sink to prevent system throttle. If the temperature of the system gets too high, the portable system will shutdown. It is really important to regulate and monitor the portable system temperature to prevent overheating of the same. The goal of these experiments was to reduce the temperature of the portable system or at least reduce the time it takes until overheats and the system starts to throttle.

It was measured the temperature in the CPU and GPU of the *Jetson Nano* using the information of the thermal sensors located in zone 1 and 2 in *Jetson*. It was also measured the temperature inside the portable system box (Power-bank and *Jetson*) using a pressure and temperature sensor (*BMP280*). This sensor is connected to the *Jetson* by the *J41* pins.

5.16.1 Experiment 1 - Open Box

For the first experiment, the temperature was measured with the box cover open while running the *Method C* for door detection and classification (*Descriptor Mode*). The box, CPU, and GPU temperatures were monitored for 30 minutes. Figure 5.13 represents the aforementioned experiment.

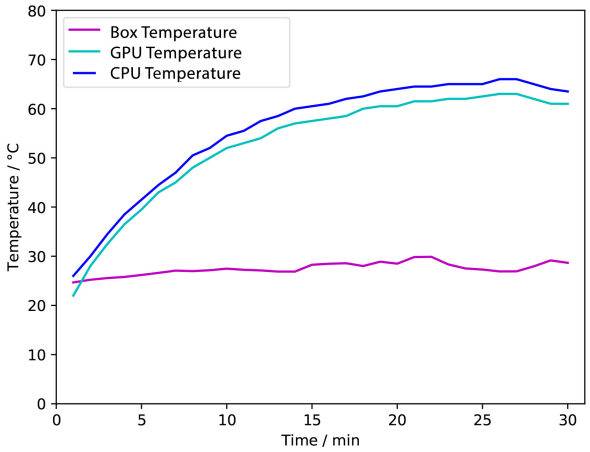

Figure 5.13: Temperature experiment 1, portable system with box cover open.

The box temperature didn't change much since the box cover was open. Its maximum value in this experiment was 29 °C. CPU and GPU temperatures varied with the time much more than the box temperature. They keep increasing over time and only stabilised on the 25 minutes mark. Their maximum value in this experiment was around 65 °C, more than twice the box temperature.

5.16.2 Experiment 2 - Closed Box

The difference between this experiment and the previous one is that in this experiment, the box cover is closed as it should be when the visually impaired people use the portable system. Figure 5.14 represents temperature experiment 2.

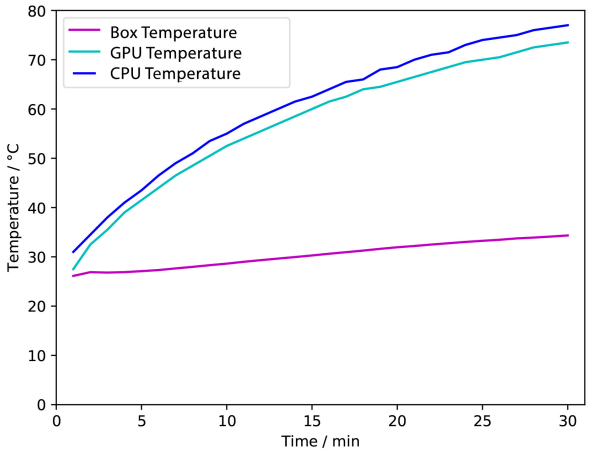

Figure 5.14: Temperature experiment 2, portable system with box cover closed.

Unlike the previous experiment, the box temperature didn't stabilise, and on the 30 minutes mark, it was still showing signs that it could increase even more. The maximum value for the box temperature was around 32.5 °C. The most worrying values were the CPU and GPU temperature values which reached 77.0 and 72.0 °C, respectively. As the box temperature, the CPU and GPU temperature didn't stabilise and were still showing signs that they could increase even more.

5.16.3 Experiment 3 - Decrease Box Temperature

In the previous experiment, the box, GPU and CPU temperatures didn't stabilise and reached very high values. In this experiment, the temperature variation of the CPU, GPU, and the box was tested again but for 1 hour. It was tested with this duration to ensure that the temperature stabilises. Figure 5.15 represents the temperature variation over time with the original box of the portable system.

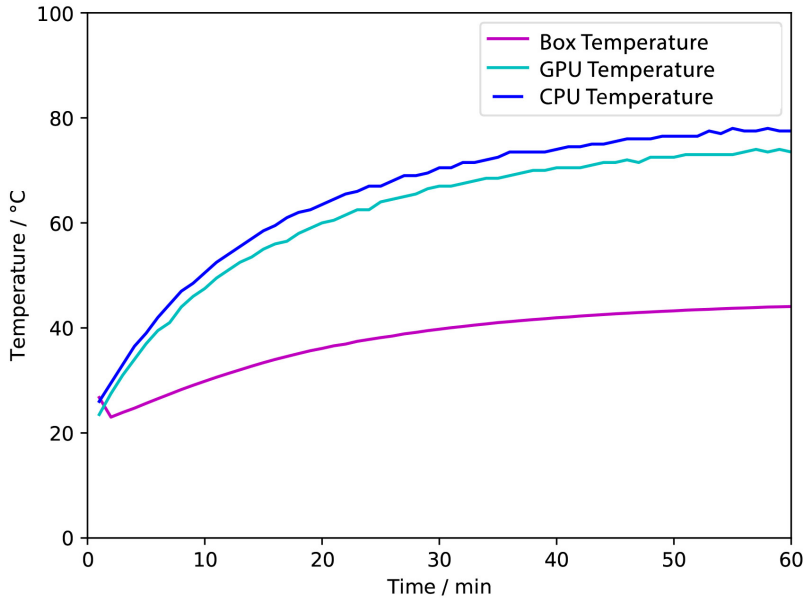

Figure 5.15: Temperature variation over 1 hour in experiment 3, portable system with box cover closed.

Running the program in 1 hour instead of 30 minutes allows the temperature to stabilise. In 1 hour of program time, the GPU temperature is 77.5 °C, the CPU temperature is 73.5 °C, and the Box temperature (inside the box) is 49.5 °C. These temperature values are too high, and in addition to making the system throttle, the visually impaired user can be hurt.

To solve this problem, 14 extra holes were made in the box cover. Originally, the box cover had six ventilation holes, but these holes were not enough according to this experiment. The difference between the original box cover and the actual box cover can be seen in figure 5.16.

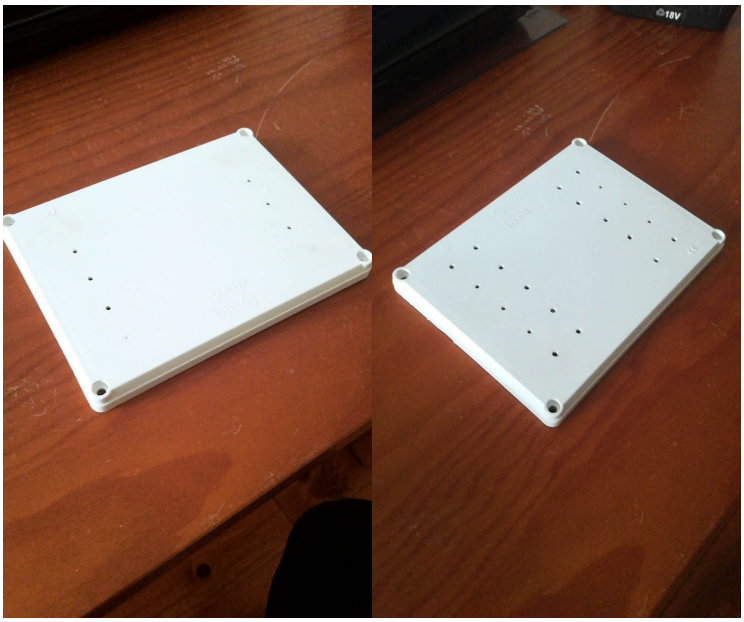

Figure 5.16: Difference between the portable system's original box cover (left side) and the portable system's new box cover (right side).

After drilling the holes in the box, I compared the difference between the original and the new box cover in terms of temperature variation. Figure 5.17 treats the results of the temperature variation for 1 hour of the original box cover and the new box cover.

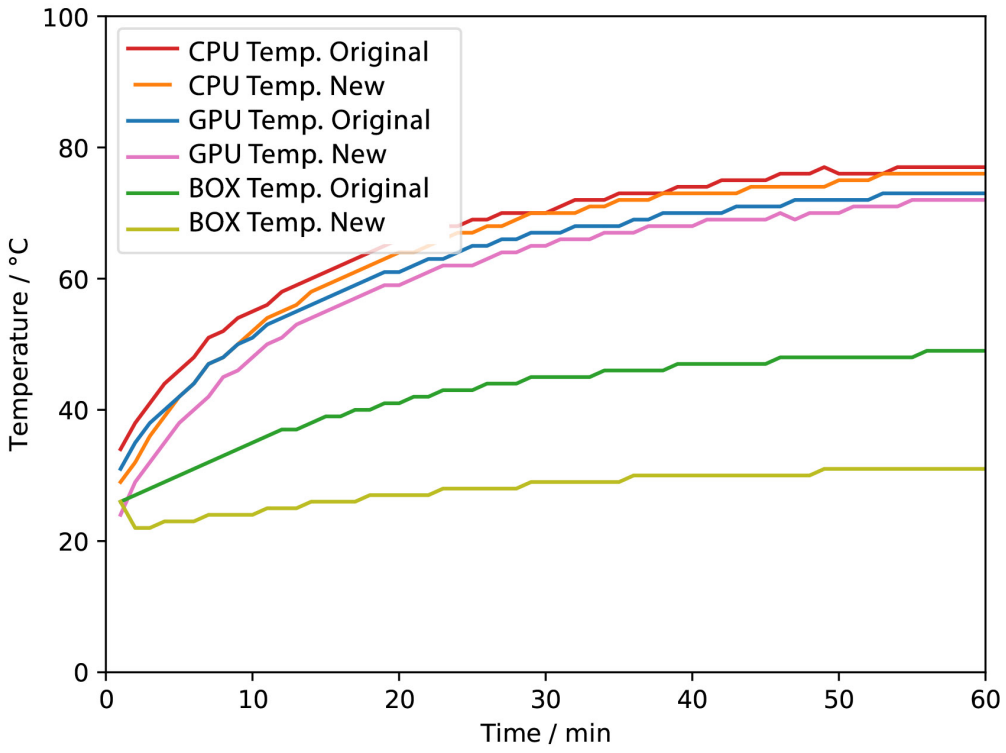

Figure 5.17: Temperature variation over 1 hour with the original portable system's box cover and with the new portable system's box cover.

As it can be seen in figure 5.17, the addition of the new holes in the box cover allows the air to circulate more, consequently allowing the system not to overheat as much as it was with the original box cover. With the new box cover, after the program has been running for 1 hour, the GPU temperature is 72.5 °C when it was 73.5 °C with the old box cover, the CPU temperature is 76.5 °C when it was 77.5 °C and the Box temperature was 31.5 °C when it was 49.5 °C. The biggest difference in temperature was in the Box temperature (dropped 18.0 °C).

Although we got already good results, the air circulation of the portable system can still be improved, consequently decreasing its box, CPU, and GPU temperatures. To decrease even more these temperatures and increase the air circulation, I drilled 16 holes on the sides of the box as it can be seen in figure 5.18. Eight holes on each side, equally distant from each other

Figure 5.18: Difference between the mobile system box before this experiment (left side) and during this experiment, with new 16 holes (right side).

Once again, I tested for 1 hour, the box, CPU, and GPU temperatures variation before and after these new 16 holes in the sides of the portable system's box. It's important to mention that these temperature experiments were done on Jetson Nano in 5W mode. Figure 5.19 represents these results. The mobile system has now a total of 36 holes, 20 on the box cover, and 16 on the sides of the box. I will call this new version of the mobile system, **mobile system 36-holes**, and the previous version will be named **mobile system 20-holes** because it only had 20 holes (on the box cover).

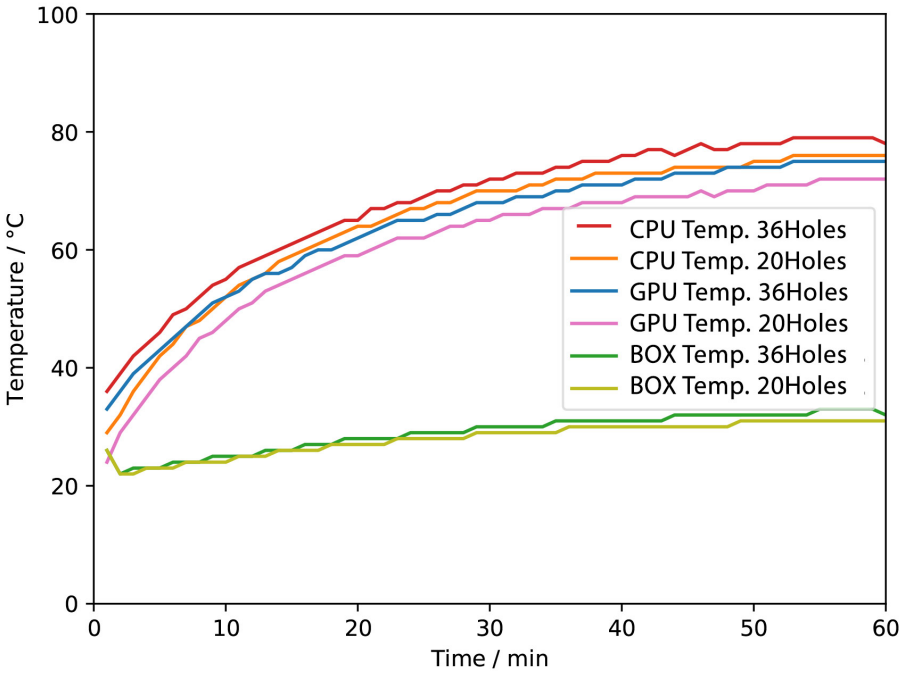

Figure 5.19: Temperature variation over 1 hour with the 20-holes mobile system version and with the 36-holes version.

The results weren't expected. The addition of the holes decreased none of the evaluated temperatures; in fact, it did the opposite. The mean temperature values increased with the new version of the portable system box. The main reason for getting worse results probably has to do with the initial temperatures values. It is remarkable that if the initial temperature values were the same in both systems, the difference in temperature values would be smaller. We can conclude that the addition of these new 16 holes didn't pay an important role to decrease the mean temperature values with the goal to avoid CPU/GPU throttle.

5.16.4 Experiment 4 - Add a fan

We successfully decreased the portable system temperature(CPU, GPU, and Box) from the previous experiments, but it is not yet the intended result. Even after drilling more holes on the cover and sides of the box, the air circulation is poor and almost nonexistent.

To increase the air circulation inside the box, beyond the holes, I used a fan. This fan was mounted in the box cover. The goal was to mount this fan on top of Jetson Nano heatsink, but the box height wasn't big enough so, it was mounted over the Jetson(in the box cover) but not over its heatsink. It can be seen in figure 5.20 how the fan was mounted on the box cover. The fan is small (30(L)x30(W)x10(H)mm) since we are limited by the available space inside the box.

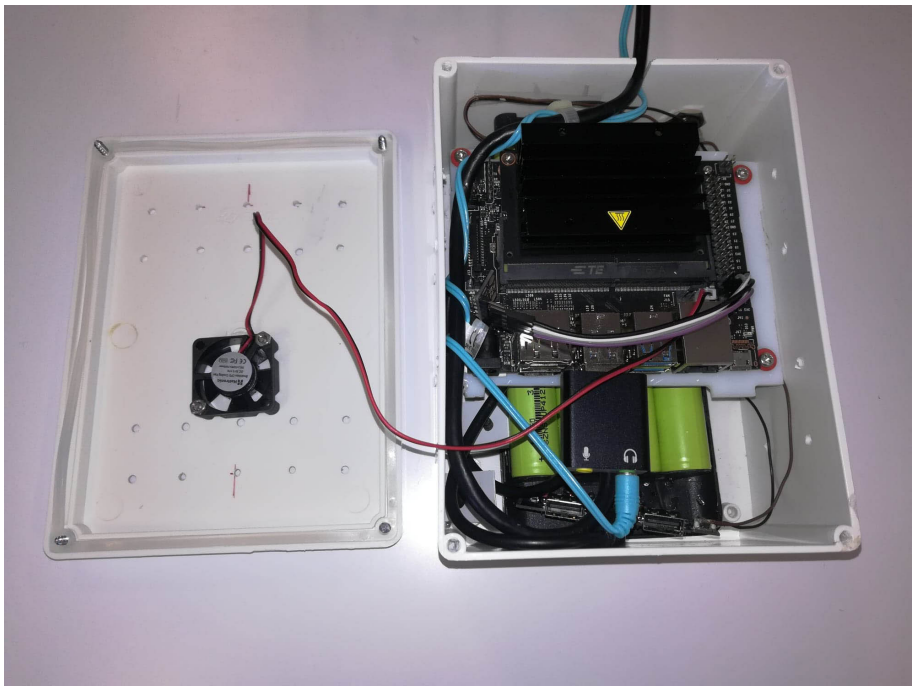

Figure 5.20: Mounted fan in the portable system box.

With the fan mounted, the temperature experiments were repeated for 1hour. As in the previous tests, it was measured as the CPU, GPU, and box temperatures. Figure 5.21 treats the results of temperature variation for 1 hour of the portable system with and without the fan. The fan in this test was always working at 100% speed during all the experiment.

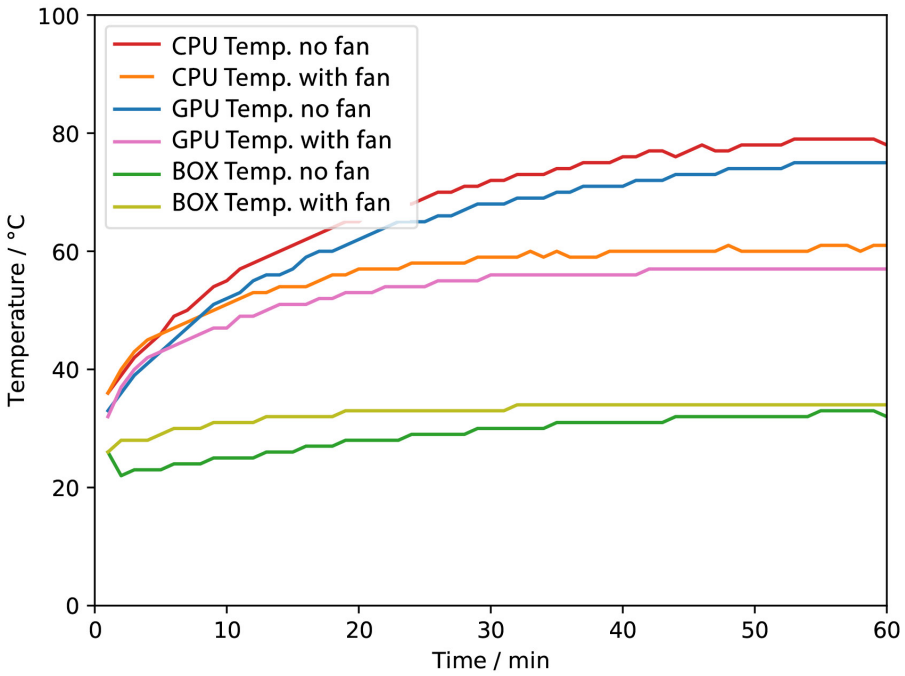

Figure 5.21: Temperature variation over 1 hour with and without the fan on the portable system.

With the addition of the fan, the mean temperature values of the portable system decreased as can be seen in figure 5.21. After the program has been running for 1 hour, the GPU temperature is 57.0 °C when it was 72.5 °C(no fan), the CPU temperature is 61.0 °C when it was 76.5 °C, and the Box temperature is 34.5 °C when it was 32.5 °C. The only temperature that didn't decrease with the fan addiction was the box temperature, but that was probably due to the time I took to put the box cover and isolate the portable system. In the previous experiment, I took longer, and that's why the box temperature was lower in the first minutes when compared with the temperature with the fan. Another factor that may influence this increase in the box temperature could be the heat that the fan reproduced behind it and the hot air that is leaving the *Jetson Nano* heatsink.

5.16.5 Resume of all experiments

Table 5.19 treats all the temperature experiments and compares them in terms of CPU, GPU, and Box temperature after the program been running for 30minutes and 1 hour.

Table 5.19: Comparison of the portable system temperature (GPU, CPU and Box) values after the script of method C been running for 1 hour with the state evolution of the portable system (With or without box cover, fan and number of holes on the portable system).

Box cover	Nº of holes	Fan	T(°C) GPU 30min	T(°C) CPU 30min	T(°C) Amb 30min	T(°C) GPU 1h	T(°C) CPU 1h	T(°C) Amb 1h
×	6	×	63.0	66.0	29.0	×	×	×
✓	6	×	72.0	77.0	32.5	73.5	77.5	49.5
✓	20	×	65.5	70.0	29.5	72.5	76.5	31.5
✓	36	×	68.5	72.0	30.5	75.0	79.5	33.0
✓	36	✓	**56.0**	**59.0**	33.5	**57.0**	**61.0**	34.5

From table 5.19 it can be concluded that the addition of the fan reduced the GPU and CPU temperature of Jetson Nano considerably. The box temperature didn't decrease, and the reason was already explained in the previous sub-section. It can also be concluded that it's better to use a box cover with a fan than not using a box cover at all. The GPU and CPU temperature values of the first temperature experiment are bigger than the GPU and CPU temperature values with the box cover and the fan.

5.17 Improve Door Detection/Segmentation for *Method C*

This section treats all the experiments and improvements on the object detection/semantic segmentation module of *Method C* for the *Door Problem*.

5.17.1 Improve *DetectNet*

From the experiments in Object Detection with *DetectNet*, 5.15.2 and the experiments in Image Classification with *AlexNet* and *GoogleNet* I concluded that the module that needs more improvement is the object detection module. The big difficulty was the door detection and localisation.

The big problem with the object detection was that the system usually detected objects that weren't doors, as doors. I called these cases *False Positives*.

Figure 5.22 represents what are *False Negatives*, *False Positives* and *True Positives* in object detection.

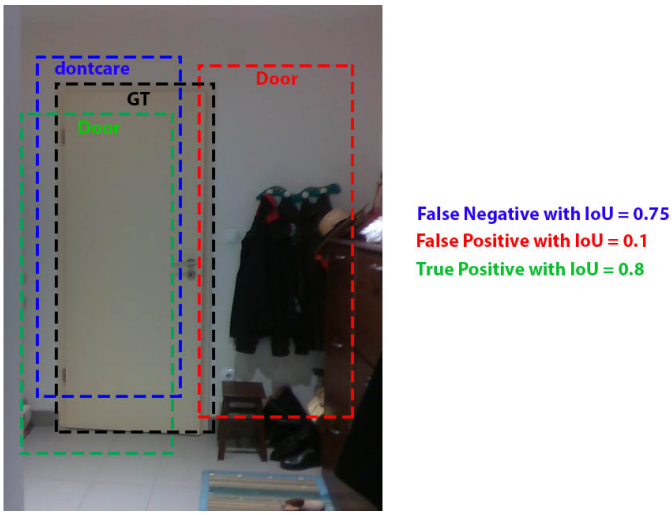

Figure 5.22: Example of False Positive, False Negative and True Positive in *DetectNet*.(GT stands for *Ground True*)

The metric *Recall* is the ratio between the True Positives with True Positives plus False Negatives. The metric *Precision* is the ratio between True Positives with True Positives plus False Positives. Following this, if the *Recall* is small, it means that the system is predicting a lot of False Negative cases and if the *Precision* is small, it means that the number of False Positives is high.

From the experiments in 5.15.2 the *Recall* was high, meaning that the number of False Negatives was low but, the *Precision* wasn't, meaning that the model was predicting many False Positives. Taking this into account, the main focus was to decrease the number of False Positives to increase the *Precision*.

A strategy to infer that the object that the system is predicting isn't a door is to annotate the dataset with the class "*dontcare*". Until now, the dataset only had the annotations of the doors. To solve this, I built a script that randomly writes annotations of bounding box outside the already annotated door bounding boxes area. The goal of this strategy was to train the system to classify all the other objects in the class *dontcare*. After annotating the modified dataset for door detection (353 for training and 60 for testing) I trained the *DetectNet* with the same parameters as the last experiment (Experiment 4) in 5.15.2 since it was the best experiment in terms of results. Figure 5.20 treats the experiment results and compares them with the results of Experiment 4 in 5.15.2.

Table 5.20: Comparison of the *DetectNet* model with the annotations of the class *"dontcare"* and without them in terms of Precision, Recall and Training time.

"dontcare" Anno.	Data Aug.	Training set size	Precision	Recall	Jetson inference time	Training time (hours)
×	✓	200	0.440	0.582	10 FPS	27
✓	✓	200	0.423	0.530	10 FPS	27

As it can be seen in table 5.20, the results were not expected. In fact, the results were worse than the previous experiment that didn't have the class "*dontcare*".I came to the conclusion that the class "*dontcare*" didn't influence the evaluation of the model. I other words, the model was only concerned about the *door* class. The results were worse but not so different from the previous ones. The *DetectNet*, as already said, uses Data Augmentation and the reason why the results of Precision and Recall in both the experiments were slightly different was probably because of the Data Augmentation randomness.

5.17.2 *Object Detection* limitations in *jetson-inference*

We can use *jetson-inference* repository with the *DIGITS* platform to train object detection models that are compatible with the*Jetson Nano* system. The issue here was that it was only possible to train the *DetectNet* and although I was getting good results in terms of speed (58□ FPS), the precision remained very low (0.440). The *DetectNet* model is the model that the *NVIDIA developers* provided for object detection in *Jetson Nano* but it's used to detect smaller objects in big pictures such as detecting cars from a satellite image. The objects(doors) that I am trying to detect in the **Door Detection Dataset - Version1** occupy a big part of the image. They are large objects while the objects which the *DetectNet* was built to detect are small.

5.17.3 *Semantic Segmentation* in *jetson-inference*

Since I wasn't getting good results in door detection and the only available neural network for door detection was the *DetectNet* which wasn't the most suitable for the *Door Problem* I decided to explore semantic segmentation in the *jetson-inference*. According to *jetson-inference*, using the SUN RGB Dataset (which is a semantic segmentation dataset of indoor spaces) with 640x512 image size they reached 17 FPS in Jetson Nano with 65.1 % accuracy. Since the images of the **Door Semantic Segmentation Sub-Dataset - version 1.0** are 640x480 pixels, the FPS on *Jetson Nano* would be more then 17 (640x512 is bigger than 640x480). For the semantic segmentation, the *jetson-inference* provides one neural network compatible with Jetson Nano, the *SegNet*. Similarly to the *DetectNet* in Object Classification, the *SegNet* can be trained using the *DIGITS* platform with my dataset and the *jetson-inference* also provides pre-trained weights (*FC AlexNet*).

I did the tutorial of semantic segmentation in *jetson-inference*, which consists of training the *SegNet* with the *NVIDIA-AERIAL* Dataset (2 classes only, sky and land). This is a toy problem, and the network reached very high accuracy values in the first epoch (98.385%). One problem of *DIGITS* was that it didn't provide any evaluation metrics (like mean intersection over union) other than accuracy.

I trained the *SegNet* with the *Door Semantic Segmentation Dataset - version 1.0*, 190 images for training and 10 for validation. Like the tutorial training, the network reached high accuracy values in the first epoch(81.4%), but it didn't exceed those values in the following epochs. In the beginning, I thought that these were really great results, but when I tested it on Jetson Nano, I found the opposite. The network was inferring that the entire 640*480 image was the door when it wasn't, 5.23.

Figure 5.23: Difference between the original input image and the output of *SegNet* trained in Door Sem. Seg Dataset(Version 1).

The network could run at **2** FPS in Jetson Nano, but it wasn't really doing door segmentation even after other training parameters where changed in the network training, as the learning rate and the policy to decrease it throughout the training. On the one hand, we have a network that works in real-time in a low powered device, but on the other hand, that network can not do door segmentation.

5.17.4 Convert models to *TensorRT*

At this time of the development of this project, I already had more information about *Tensor Real Time*(TensorRT). As the developers themselves describe, *"NVIDIA TensorRT™ is an Software development kit (SDK) for high-performance deep learning inference. It*

includes a deep learning inference optimiser and runtime that delivers low latency and high-throughput for deep learning inference applications.". TensorRT models run faster without using significant reductions in accuracy and precision and are what the *jetson-inference* models are based on. All of their models are TensorRT models which are the most suitable for these kinds of low powered systems.

It is possible to even convert a Torch or Tensorflow model to the TensorRT model using the corresponding library to do it. But on the other hand, these tools and libraries still have a lot of problems with compatibility since Torch, Tensorflow, and the other deep learning development frameworks are constantly being updated. At the time I was developing this project, there were very few tutorials and information for converting models into TensorRT models.

Using TensorRT was the solution for the previous problem because we can achieve high accuracy values in real-time in low powered devices.

5.17.5 Semantic Segmentation - *TorchSeg*

I had worked with semantic segmentation in *PyTorch* in *Method A* for the *Door Problem* but the methods were either fast with little accuracy or very accurate but slow (*FC-HarDNet* and *FastFCN*). Another problem was that it was very difficult to make these models compatible with Jetson Nano and a big part of them wouldn't work on it.

I explored several benchmark repositories for real-time semantic segmentation algorithms implemented in PyTorch since it was the framework for deep learning development that I was more comfortable with. The benchmark repository that I found more suitable was the *TorchSeg* repository. It supports real-time semantic segmentation networks as the *PSPNet*, [ZSQ+16] and the *BiSeNet*, [YWP+18]. It also supports network training and inference. I installed this repository in my lab computer and started to explore the *BiseNet* model since this repository provided a *BiseNet* network with a *ResNet18* which is also used in the *jetson-inference* models. This model was trained in the *ictyscapes*, [COR+16], dataset but I trained it using the provided pre-trained weights in the *Door Semantic Segmentation Dataset - version 1.0*. The model and the trained weights can be saved every epoch.

After training for a few minutes(20min), I made the inference on the test set and printed the image results, and I conclude that this network was already giving better results than the *SegNet* in terms of accuracy because it was already segmenting the door.

5.17.6 *Torch* to *TensorRT*

I had the model/snapshot with the weights of the last epoch of training. The next step would be to convert the *BiSeNet* and these weights to a *TensorRT* model. I explore several approaches to do it.

The approached that seemed the simplest was to use the library *torch2trt* but it wasn't. I installed the library with the repository without any problem, but the function simply couldn't convert our Torch model into a TensorRT model. I came to the conclusion that the *BiSeNet* model couldn't be converted to a TensorRT model using this library.

The other approached that seemed more difficult, was to convert the *Torch* model to a *ONNX* model and then convert this *ONNX* model to a *TensorRT* model. *ONNX* stands for *"Open Neural Network Exchange"*, and it's an open format built to represent machine learning models. This approached, although it seemed harder, at first sight, was the technique that was being used usually to convert models to *TensorRT*.

Convert to *ONNX*

To convert the *Torch* model to *ONNX* format I used the function *torch.onnx.export()* from *PyTorch*. The arguments for this function are the input and output layers names of the network, the model itself, a dummy input, and *opset version*. The *opset version* is the version of the *ONNX* sub-module. Later versions support more current networks while the first versions support older networks. After some trial and error, the supported *ONNX opset version* that worked for my case was the 11, which is the second most current version.

Convert to *TensorRT*

Once we got the *ONNX* model we can use the tool *Netron* to view or model. *Netron* is a viewer for neural network, deep learning and machine learning models that supports *ONNX* models. The converted model, *BiSeNet* with *ResNet18* as an input of shape *float32[1,3,640,480]* and output of shape *float32[1,3,80,60]*. It returns a smaller output because this network does a crop in the input image and this is how it was design. After this, we simply do a *opencv* interpolation and get an output image with the same size as the input.

The conversion to *TensorRT* and the installation of *TensorRT* in the lab computer were more complex than the conversion to *ONNX*. I tried several tutorials to install *TensorRT*,

- Medium - Accelerate PyTorch Model With TensorRT via ONNX
- Medium - Installation Guide of TensorRT for Yolov3
- GitHub - NVIDIA TensorRT

The solution was to use the official instructions/guide from *NVIDIA*, `https://docs.nvidia.com/deeplearning/sdk/tensorrt-install-guide/index.html`. After several errors in paths and missing libraries, I successfully installed *TensorRT 7.0* which is the most

recent version for desktop computers. This version was installed because it is the only version that's compatible with the *opset version* 11 of *ONNX*.
I used the tool *onnx-tensorrt* to convert the *ONNX* model to *TensorRT* model. This tool simply requires the *ONNX* model as the argument. I converted the model successfully but to make sure that, in fact, the model was in *TensorRT* the mean inference time was calculated in both of the models, *TensorRT* and the original *PyTorch* model in the lab computer. I also compared the resulting images of both networks, as can be seen in figure 5.24.

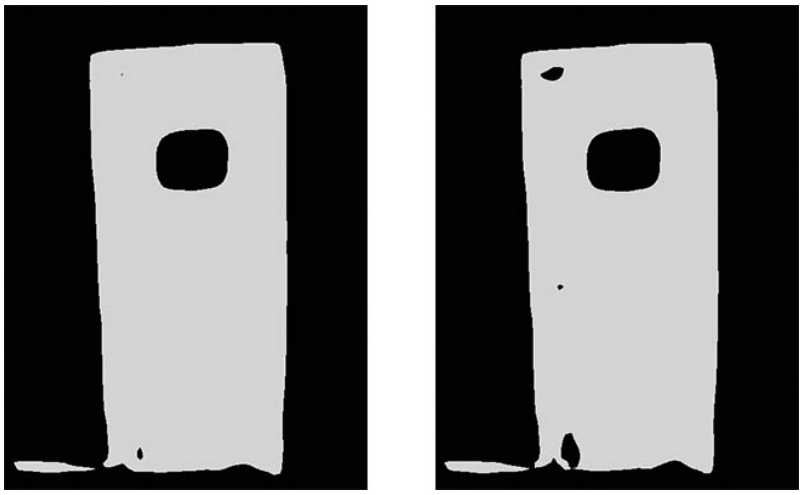

Figure 5.24: Outputs of both *Torch* and *TensorRT BiSeNet* models with the same input door image. *Torch* on the left side and *TensorRT* on the right side.

In figure 5.24, we can clearly see that the difference in the output of both models is very small. The output of the models may be similar, but the inference time isn't. The mean inference time in the *Torch* model was 0.02829 seconds, which corresponds to around **35 FPS**. The mean inference time in the *TensorRT* model with the same input was 0.01901 seconds, which corresponds to **52-53 FPS**. We gained 17-18 frames per second when we used the *TensorRT* model in the lab computer.

Figure 5.25 represents all the tested methods to convert the *Torch BiSeNet* model to a *TensorRT* model. The last method of the figure was the method chosen to convert the model.

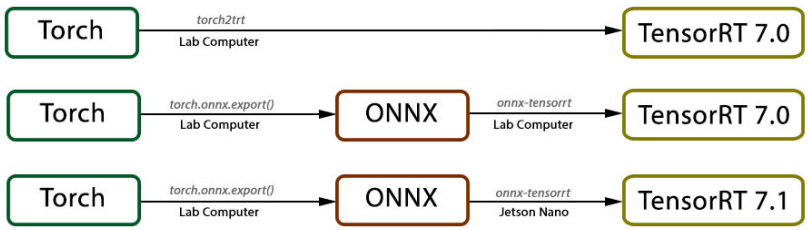

Figure 5.25: Tested methods to convert a *Torch* model to a *TensorRT* model. Arrows represent conversions. Text above the arrow refers to the conversion method and text below the arrow refers where the conversion was done.

5.17.7 *TensorRT* in Jetson Nano

TensorRT is installed by default on the *NVIDIA Jetpack SDK*. I tried to do the inference in *Jetson Nano* with the *BiSeNet TensorRT* model but I was not successful due to incompatibility of *TensorRT* versions. The *JetPack* I had installed in *Jetson Nano* was the *Jetpack 4.3* which has *TensorRT 6.0* and the model I created in the lab computer was created using *TensorRT 7.0* and different *TensorRT* versions aren't compatible.

At the time I was having this issue, a new version of *Jetpack* was released, *Jetpack 4.4* which supported *TensorRT 7.1*. I installed this new SDK in *Jetson Nano* but when I tried to make the inference, I still got the same error of incompatibility of different *TensorRT*. I thought that *TensorRT 7.0* was compatible with the *TensorRT 7.1* from *Jetson* but it wasn't. One simple solution would be to update the desktop computer *TensorRT* version from "7.0" to "7.1" but version "7.0" was the last release for desktop computers, so it wasn't possible to update it.

The solution was to convert the *ONNX BiSeNet* model to *TensorRT* in the Jetson Nano. I installed the same tool that was used to do the conversion, *onnx-tensorrt* in Jetson Nano, and I successfully converted the model. Finally, I was able to make the inference on Jetson with the *TensorRT* model.

The *TensorRT BiSeNet* in Jetson Nano (10W mode) takes 45 seconds to make the inference of 100 images. The mean inference time of this model in Jetson is **0.40 seconds**. With just this inference time, we can conclude that this approach will work at only 1 FPS, taking into account the image process and the image classification time. Compared to previous methods for door detecting this method seems slow, even in a *TensoRT* model but in fact, it isn't. The mean inference time of the *SegNet* model trained in *DIGITS* is also round **0.40 seconds** in Jetson Nano (10W mode). The *SegNet* model, which is model design by *jetson-inference* to run fast on Jetson and is also a *TensorRT* model, is as fast as the *TensorRT BiSeNet* model. Beyond that, the *SegNet* model can't detect doors while the *BiSeNet* is already detecting doors with just a few epochs of training.

5.17.8 Training and Evaluating of the *BiSeNet* model

The *BiSeNet* torch model was trained in the lab computer using the *Door Semantic Segmentation Dataset - version 1.0*. 200 images for training, 20 for validating and 20 for testing. The evaluation metric used was the mean intersection over union (*mIoU*).

Firstly, I trained the model for 400 epochs (around 50 minutes), with a batch size equal to 4 and a learning rate equal to 1e-2. I use a *ResNet18* model pre-trained in *Cityscapes* dataset. Every ten epoch, the model weights were saved, and the mean train and validation IoU were calculated. Figure 5.26 represents the mean train and validation intersection over union throughout the training epochs.

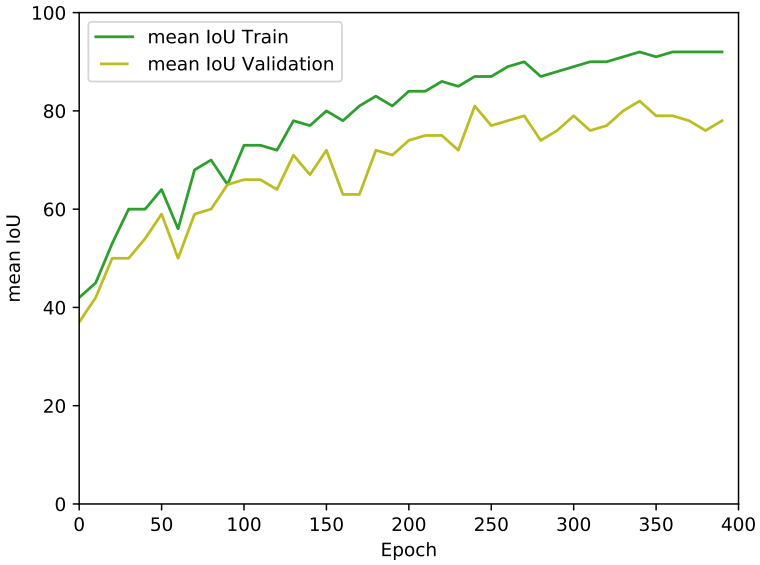

Figure 5.26: Mean train and validation intersection over union during 400 training epochs.

I was expecting to overfit of the network, but there isn't any evidence in the figure. The max mean validation intersection over union was in the 350th epoch, but I needed to train the model for more epochs to see if this value had its maximum in epoch 350 or if it could still grow in the following epochs. The goal here was to obtain the maximum validation IoU value, and I didn't have enough epochs to conclude that this was the highest value. Due to the previous, I repeated the process and trained the *BiSeNet* for 1000 epochs. Figure 5.27 represents the mean train and validation intersection over union throughout 1000 training epochs.

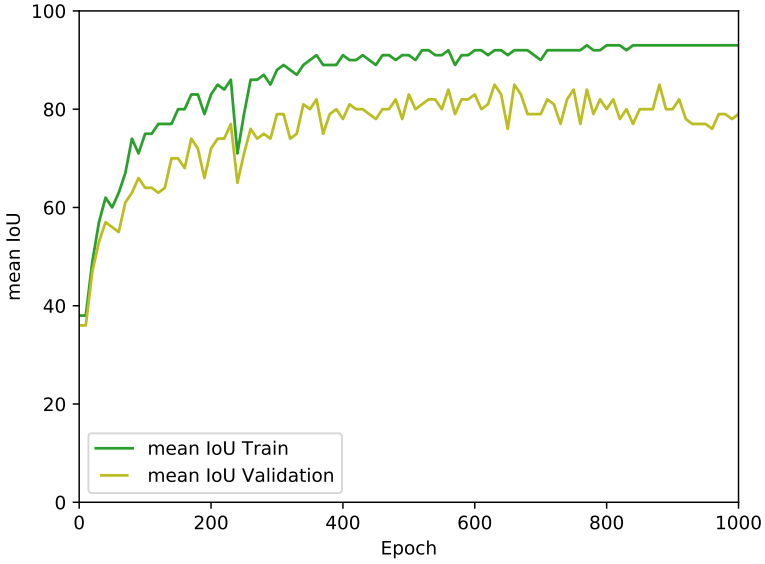

Figure 5.27: Mean train and validation intersection over union during 1000 training epochs.

From figure 5.27 it's clear that the model over fitted around epoch 900 with a mean train *IoU* equal to **93.238** and a mean validation IoU equal to **85.005**. The model was tested in the test set using the weights of epoch 900 and the mean test *IoU* was **82.227**.

5.17.9 Testing all approaches for Door Detection/Segmentation

Until now, I implemented and tested 3 different approaches for Door Detection/Segmentation on Method C. The *DetectNet* for door detection, the *SegNet* for door semantic segmentation and the *BiSeNet* also for door semantic segmentation. To evaluate and compare an object detection method with a semantic segmentation method I simply compare the output of the model, and if the same allows cropping the door correctly for door classification, I count as a correct output. The goal of these methods, (door detection or door semantic segmentation) is to detect door contours or borders in the image and provide the necessary information to the image classification model.

The object detection models already output a bounding box with the location of the object, but the semantic segmentation models do not. The strategy here was to first detect the biggest door cluster in the output of the semantic segmentation models and use the smallest and the biggest x and y values to build the bounding box and crop the image. Filters of *Dilation* followed by *Erosion* (Closing filters) (*opencv*) were also used on the output of the semantic segmentation. If the biggest/maximum door cluster wasn't bigger than 30000 pixels or if the door *width* was not bigger than 150 pixels that door cluster was forgone. These filters were the same that were used in **Method A** for Door detection and classification.

The fact that it's required to use these filters after the semantic segmentation outputs is a disadvantage compared with the object detection version because they consume time while time is precious since we are building a real-time method. The output of the *BiSeNet* is an 80*60 image, and so, this image needs to be resized to a 640*480 image, and this resizes operation takes time as well. So, for this semantic segmentation method, even more, time is required. Figure 5.28 represents the door detection/segmentation process of **Method C** and the difference between using the object detection method, *DetectNet* and the semantic segmentation method, *BiSeNet*.

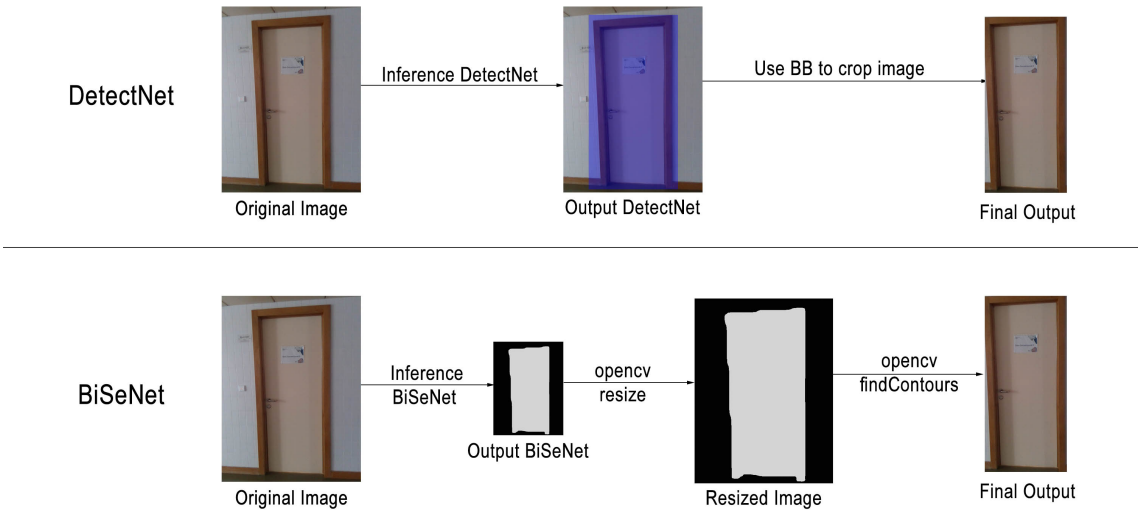

Figure 5.28: Difference in operations and filters between using the semantic segmentation *BiSeNet* and the object detection *DetectNet* in the process of door detection/segmentation in Method C.

As it can be seen in figure 5.28, the semantic segmentation method *BiSeNet* involves several more processes after the inference itself, and that's one of the biggest disadvantages of using semantic segmentation for the door detection/segmentation.

Too further analyse the advantages and disadvantages of using the *DetectNet*, the *SegNet* or the *BiSeNet*, all of these methods were tested in terms of inference speed and precision. Twenty door images were used to represent the positive cases, and 20 images with no doors were used to represent the negative cases. The positive case is when there is a door in the image, and the negative case is when there isn't any door in the image. I used 20 images with no doors because of the *DetectNet*. This model was detecting the doors, but it was also detecting doors when there wasn't any door in the image (False Positive case). With this, I could evaluate both the True and False Positives of each model and compare the results. I analysed each image, and if the method output contained all the necessary information for the image classification network, I would consider that case as a True Positive. It was measured the mean inference time of each method and the post inference time (just for the semantic segmentation approaches) in seconds in *Jetson Nano*. It was also calculated the total time (inference time + post inference) of each method. The total time represents the time, in seconds, that it takes to give the cropped RGB image to the image classification network in Method C.

Table 5.21 represents the evaluation and comparison of *DetecNet*, *SegNet* and *BiSeNet* on Door Detection/Segmentation in terms of number of True Positives, False Positives, the mean inference time, post inference and total time in Jetson Nano.

Table 5.21: Evaluation and Comparison of *DetecNet*, *SegNet* and *BiSeNet* on Door Detection/Segmentation in terms of number of True Positives, number of False Positives, mean inference, post inference and total time in seconds in Jetson Nano.

Method	True Positives	False Positives	Mean Inference time(s)	Post Inference time(s)	Total Inference time(s)
DetectNet	14/20	5	0.130	0	0.130
SegNet	0/20	20	0.400	0.006	0.406
BiSeNet	19/20	2	0.400	0.012	0.412

From table 5.21 we can conclude that the worst out of these 3 methods is the *SegNet*, the default semantic segmentation algorithm in *jetson-inference*. The *SegNet* does not have any True Positives since it always detects the entire image as the door object. Instead of providing only the necessary information, it provides all the original image to the image classification algorithm. It has 20 False Positives (20 negative images) due to the same reason. The post inference time (**0.006s**) is a little smaller when compared with the *BiSeNet* post time because the *SegNet* outputs a 640*480 image without having to resize it. The *BiSeNet* needs to resize the image because it outputs an 80x60 image as it was said previously. *SegNet* output is already a 640*480 image but if the *Image Classification model input image size* is 227*227, the *SegNet* will also need to do resize the image to a 227*227.

The *DetectNet*, default object detection network in *jetson-inference*, has a total inference time, equal to **0.130**s while the *BiSeNet* has a total inference time equal to **0.412**s. The *DetectNet* is more than 3 times faster than the *BiSeNet* (0.412/0.130 = 3.17). But, on the other side, The *BiSeNet* is the method that achieved the best results in terms of number of True Positives and False Positives. With the *BiSeNet* network I couldn't only detect one out of 20 doors in the 20 doors images, while the *DetectNet* failed to detect 6 out of 20 doors. The biggest problem of the *DetectNet* was with the False Positive Cases, and we can see this issue from these results, it detected 5 doors in images without any doors, while the *BiSeNet* detected only 2 doors. To conclude, in terms of speed, the best approach is undoubtedly the *DetectNet* approach but, in terms of precision, the best approach is the *BiSeNet* approach.

Chapter 6

Conclusion

In this chapter all the scientific contributions of this work will be described, one final experiment to conclude which of the developed methods would be the best for the visually impaired people and the future work, or what could still be done in this project.

6.1 Scientific Contribution

In sort, the contributions of this work were:

- One **portable system to help visually impaired people** navigate that is easy to use and transport, lightweight, doesn't overheat, and can still be improved.
- A ***Git* Repository** with all the instructions to prepare a *Jetson Nano* to run neural network models and the developed methods in this project.
- Two **Datasets for 3D and 2D Door and Stairs Classification** labelled, freely available online, and with information about the test, train and validation sets.
- **3 Methods** to solve the visually impaired people ***Door Problem*** that work in real-time in low powered devices.

6.2 *Door Problem* Methods

I develop three methods for solving the *Door Problem* that work in real-time in low powered devices such as the *Jetson Nano*. **Method A** uses 2D Semantic Segmentation to detect the door and uses 3D Object Classification to classify it. **Method B** just classifies the door with a 3D Object Classification method. **Method C** uses 2D Semantic Segmentation to detect the door and uses 2D Image Classification to classify it. Each method has its own advantages and disadvantages but which one is the best to use in the portable system for visually impaired people?

I compared all these three methods in terms of Door Detection, Segmentation Intersection over Union (IoU) and inference time, in terms of Door Classification test accuracy and inference time, and in terms of total method inference time. The following table represents this comparison.

Table 6.1: Comparison of all the *Methods* for the *Door Problem*.

Method	Seg. Network	Seg. mean test IoU	Seg. mean time(s)	Class. Network	Class. test acc.	Class. mean time(s)	Total time(FPS)
A	*FC-HardNet*	0.418	0.131	*PointNet*	0.494	0.111	3
B	×	×	×	*PointNet*	0.433	0.111	5-6
C	*BiSeNet*	0.822	0.412	*AlexNet*	0.983	0.019	1-2

For each method, I used its best algorithms based on the experiments presented in the previous chapter. That's the justification for comparing method A with the ***FC_HarDNet*** algorithm for Semantic Segmentation and method C with the ***BiSeNet*** for Semantic Segmentation and ***AlexNet*** for Door Classification. In other words, I used the best algorithms in each method with the goal to compare each method at its best.

Talking first about the Semantic Segmentation part. Without a doubt that the best method to Detect a Door is *Method C*. Using the *BiSeNet* in the *TensorRT* form, it gives a 0.822 mean test IoU which, when compared with the *FC-HardNet* is very good. *Method B* doesn't detect the door which could be a problem since this method doesn't know if there is any door or not in the scene while the other two know this and only classify the image if there is a detect door in it. The only advantage of *Method A* is its speed, but it's better to have a method that works but takes a little longer than having something that works very fast but fails several times.

Now, talking about the Door Classification part. Once again, without a doubt that *Method C* is the best method. Initially, I thought that it would be better to classify an object with 3D information, but the RGB information, in this case, is much more valuable. With the *AlexNet* classification network, *Method C* got a mean test accuracy equal to **0.983** which is excellent when compared to the other methods.

To finish, the total inference time of each method. Here, *Method C* is the worst and can only work at 1 to maximum 2 frames per second, while the others can work at **3 FPS**, (*Method A*) and **5-6 FPS**, (*Method B*). But isn't 1-2 frames more than enough? If every frame of those frames per second would always be a clear image, without being blurred and if the user walks slowly, this frame rate would be enough. The sound itself that is reproduced each time the Door is detected and classified ("*open door*, *closed door*, *semi-open door*") takes also some time(0.5 seconds) to reproduce. The problem is when a frame captured is blurred and will induce the system to produce wrong classifications, and the person will just have another response in the next second.

Concluding, *Method C* is the last and the best method to solve the *Door Problem* since the precision in detection and classification of the door pays off the time this method takes. It's better to have a method that still is in real-time, and it's capable of providing the cor-

rect information to the visually impaired user than having a method that can provide the information faster but not so correct.

6.3 Future work

For future work, I would have liked to migrate the last method for solving the *Door Problem*, *Method C* to the *Stairs Problem*, which was not so explored in this work. The reason why I focused more on the *Door Problem* and not in the *Stairs Problem* was because of the frequency that the *Door Problem* happens when compared with the *Stairs Problem*. The *Door Problem* usually happens when a visually impaired person lives in a shared house, and so it can happen a lot of times. The *Stairs Problem* doesn't happen in the visually impaired person's home, it happens instead in unknown indoor places or in places where they already have been to but don't know every corner of the place. The *Door Problem* it's much more frequent than the *Stairs Problem*, and that's the main reason why I focused on building the portable system to solve this problem.

What was also left to be done was to get feedback from a real user. Due to the Pandemic of *SARS-CoV-2 Virus* I wasn't able to lend the portable system to a visually impaired person to test it and give me feedback in return. This feedback would have been a great contribution to this work and the next step to improve the portable system.

Bibliography

[ATS16] Jon Barker Andrew Tao and Sriya Sarathy. Detectnet: Deep neural network for object detection in digits, 2016. 7, 88

[ATS19] Miguel Arduengo, Carme Torras, and Luis Sentis. Robust and adaptive door operation with a mobile manipulator robot, 2019. 89

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 38

[CDD03] Grazia Cicirelli, T. D'Orazio, and Arcangelo Distante. Target recognition by components for mobile robot navigation. *J. Exp. Theor. Artif. Intell.*, 15:281–297, 07 2003. 15, 16, 17

[CFG+15] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015. 49

[CKR+19] Ping Chao, Chao-Yang Kao, Yu-Shan Ruan, Chien-Hsiang Huang, and Youn-Long Lin. Hardnet: A low memory traffic network, 2019. 7, 73

[COR+16] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 106

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. 88

[HW08] Brian Hoyle and Dean Waters. *Mobility AT: The Batcane (UltraCane)*, pages 209–229. Springer London, London, 2008. Available from: `https://doi.org/10.1007/978-1-84628-867-8_6`. 8

[KAY11] N. Kwak, H. Arisumi, and K. Yokoi. Visual recognition of a door and its knob for a humanoid robot. In *2011 IEEE International Conference on Robotics and Automation*, pages 2079–2084, May 2011. 15, 16, 17

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran

Associates, Inc., 2012. Available from: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf. 6

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014. 90

[LRA17] A. Llopart, O. Ravn, and N. A. Andersen. Door and cabinet recognition using convolutional neural nets and real-time method fusion for handle detection and grasping. In *2017 3rd International Conference on Control, Automation and Robotics (ICCAR)*, pages 144–149, April 2017. 15, 16, 17, 44

[MLRS02] Iñaki Monasterio, Elena Lazkano, Inaki Rano, and Basilio Sierra. Learning to traverse doors using visual information. *Mathematics and Computers in Simulation*, 60:347–356, 09 2002. 15, 17

[MSZW14] S. Meyer Zu Borgsen, M. Schöpfer, L. Ziegler, and S. Wachsmuth. Automated door detection with a 3d-sensor. In *2014 Canadian Conference on Computer and Robot Vision*, pages 276–282, May 2014. 15, 17

[Nvi19] Nvidia. Jetson nano developer kit 3d cad step model [online]. 2019. Available from: https://developer.nvidia.com/embedded/downloads. xx, 78

[opeon] openCV, Computer Vision Annotation Tool: A Universal Approach to Data Annotation. Available from: https://github.com/opencv/cvat. 39

[QGPAB18] Blanca Quintana Galera, Samuel Prieto, Antonio Adan, and Frédéric Bosché. Door detection in 3d coloured point clouds of indoor environments. *Automation in Construction*, 85:146–166, 01 2018. 15, 16, 17, 44

[QPAB16] B. Quintana, S. A. Prieto, A. Adán, and F. Bosché. Door detection in 3d colored laser scans for autonomous indoor navigation. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8, Oct 2016. 15, 16, 17

[QSMG16] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016. Available from: http://arxiv.org/abs/1612.00593. 6, 38

[RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011. 38

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018. 7

[SDor] STAR-DETECTOR, Willow Garage Star Detector. Available from: http://pr.willowgarage.com/wiki/Star-Detector. 16

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014. 6

[TRZ+17] Lore Thaler, Galen M. Reich, Xinyu Zhang, Dinghe Wang, Graeme E. Smith, Zeng Tao, Raja Syamsul Azmir Bin. Raja Abdullah, Mikhail Cherniakov, Christopher J. Baker, Daniel Kish, and Michail Antoniou. Mouth-clicks used by blind expert human echolocators – signal description and model based signal synthesis. *PLOS Computational Biology*, 13(8):1–17, 08 2017. Available from: https://doi.org/10.1371/journal.pcbi.1005670. 10

[WZH+19] Huikai Wu, Junge Zhang, Kaiqi Huang, Kongming Liang, and Yizhou Yu. Fastfcn: Rethinking dilated convolution in the backbone for semantic segmentation, 2019. 7, 58

[YHZH15] T. H. Yuan, F. H. Hashim, W. M. D. W. Zaki, and A. B. Huddin. An automated 3d scanning algorithm using depth cameras for door detection. In *2015 International Electronics Symposium (IES)*, pages 58–61, Sep. 2015. 15, 16, 17

[YWP+18] Changqian Yu, Jingbo Wang, Chao Peng, Changxin Gao, Gang Yu, and Nong Sang. Bisenet: Bilateral segmentation network for real-time semantic segmentation, 2018. 7, 106

[ZB08] Zhichao Chen and S. T. Birchfield. Visual detection of lintel-occluded doors from a single image. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, June 2008. 15, 17, 44

[ZDS+18] Hang Zhang, Kristin Dana, Jianping Shi, Zhongyue Zhang, Xiaogang Wang, Ambrish Tyagi, and Amit Agrawal. Context encoding for semantic segmentation, 2018. 45

[ZPK18] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018. 38, 69, 85

[ZSQ+16] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network, 2016. 58, 106

[ZZP+17] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017. 47