EEEE3056 Third Year Projects in Electrical and Electronical Engineering

2020/21 Session

Indoor Autonomous Cruise Vehicle based on UWB and Computer Vision

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

UNIVERSITY OF NOTTINGHAM NINGBO CHINA

Name	:	Tianchen Song
Student ID	:	20124957
Supervisor	:	Dr. Liang Huang
Moderator	:	Dr. Jing Wang

Declaration

I hereby declare that the work described in this report has been done by myself and no portion of the work contained in this report has been submitted in support of any application for any other degree or qualification on this or any other university or institution of learning.

Tianchen Song

Abstract

In this report, the autonomous robotic vehicle for indoor environment that I prototyped will be introduced. UWB modules were used for vehicle localization and A-star search algorithm was realized for global path planning. A deep neural network for collision-free space segmentation was trained and deployed onto the embedded platform with TensorRT acceleration. The robotic vehicle could automatically go to the target position without running into obstacles. It could also communicate with an iPhone through Bluetooth when an emergency stop was necessary. The GitHub repository of this project: SimonSongg/FYP (github.com)

Table of Content

Chapter 1 Introduction

Nowadays autonomous robot is being applied into various fields for different tasks such as food serving in restaurant, cargo transport in factories and package delivery in the city. This kind of autonomy enhances the efficiency of production and even daily lives.

However, although many sub-tasks of autonomous driving such as SLAM, object detection, scene understanding and path planning were greatly developed, it is still challenging to design a robust navigation and obstacle avoidance system that could adapt to plenty of different scenarios.

On the localization side, outdoor autonomous robots are usually guided by GNSS, while for indoor environment, where GNSS is not available, simultaneous localization and mapping (SLAM), BLE and UWB are three main methods for robot localization. SLAM mainly has two variants, which are LIDAR-based SLAM and visual SLAM. The idea of SLAM is to let the machine construct the map on its own so that it could know its exact position in the map just like what human-beings do. On the contrary, BLE and UWB methods reply on pre-installed anchors in the space that the robot will operate.

On the obstacle avoidance side, with the fast development of deep learning, computer vision with sub-tasks such as object detection, semantic segmentation and lane tracking plays a significant role. For simple task like operating in a limited space in indoor environment, applying a series of models in computer vision will be redundant and not computing-efficient. Therefore, low level scene understanding method like collision-free space segmentation might be suitable. Moreover, in order to improve the robustness, several ultrasonic sensors will be used to provide extra environment perception.

In this report, I prototype a robotic vehicle that could navigate itself within a certain indoor area and avoid obstacles in a smooth manner, which mainly use UWB, computer vision and ultrasonic sensors.

Chapter 2 Literature Review

2.1 UWB Localization

Like Bluetooth and Wi-Fi, ultra-wideband is a wireless communication protocol that uses radio waves. A UWB transmitter sends billions of radio pulses per second and a UWB receiver then translates the pulses into data. Since the radio pulses have short duration (nanosecond), it will occupy wide spectrum of frequency. The pulses are sent in low power to avoid interference with other device in band so that it takes 32 to 128 pulses to encode a bit of data [1].

The ultra-wideband positioning system consists of tags and anchors. To locate the tag, several anchors must be installed at places with known coordinates. There are two main positioning methods for UWB, which are Two-way-ranging (TWR) and Time Difference of Arrival (TDOA) [2]. For TWR, the tag will send a packet out, and the anchors that receive the packet will immediately send it back, which is illustrate in Figure 2-1. The distance between the tag and each anchor could be calculated by,

$$\left(\frac{T_{loop} - T_{reply}}{2}\right) \times The\ speed\ of\ light \tag{1}$$

Once the tag has ranged with at least three anchors, the position of the tag could be derived by trilateration, which is illustrated in Figure 2-2.

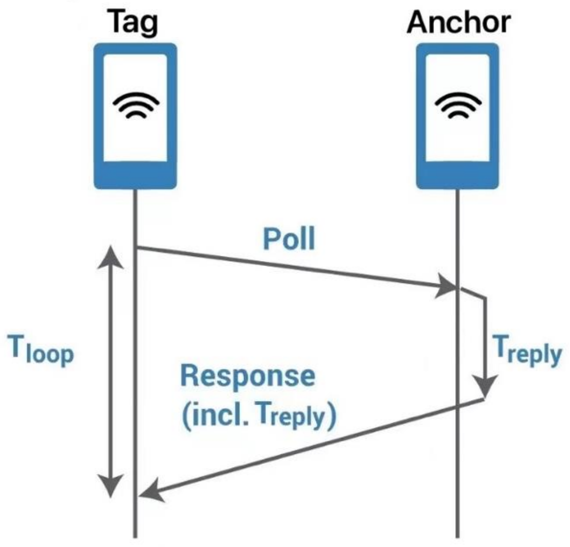


Figure 2-1. The principle of TWR ranging [3]

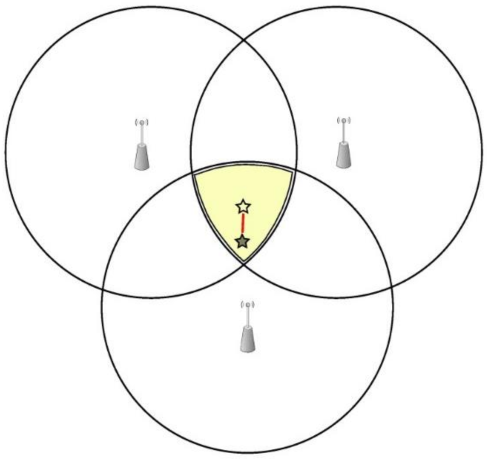


Figure 2-2. Triangulation localization

Assume the coordinates of three anchors are (x_1, y_1), (x_2, y_2) and (x_3, y_3) and the coordinates of the tag are (x_t, y_t). If the distance between the tag and each anchor is known as d_1, d_2 and d_3, the coordinated of the tag could be calculated by solving the equations set

$$(x_1 - x_t)^2 + (y_1 - y_t)^2 = {d_1}^2 \tag{2}$$

$$(x_2 - x_t)^2 + (y_2 - y_t)^2 = {d_2}^2 \tag{3}$$

$$(x_3 - x_t)^2 + (y_3 - y_t)^2 \leq {d_3}^2 \tag{4}$$

For TDOA, all the anchors must be strictly synchronized on timing and connected to a positioning server. The tag will periodically transmit packet, and anchors at different place will receive the signal at different time. The time difference will be sent to the server for calculation. The tag itself will never know its position in this scenario. Therefore, in this project, TWR will be used to localize the tag.

2.2 Deep Learning Free-Space segmentation based on RGB-D image

Machine learning is the systematic study of algorithms and systems that improve the program performance using data [4]. Instead of handcrafted rules, machine learning algorithms could automatically find more complex mapping rules between the input and the output from a batch of data.

Deep leaning is a subset of machine learning, which became popular in the era of data and push the significance of data to a new height. The deep learning model is generally composed of multi-layer artificial neural networks (Figure 2-3), which was initially inspired by the biological architecture of animals' brain.

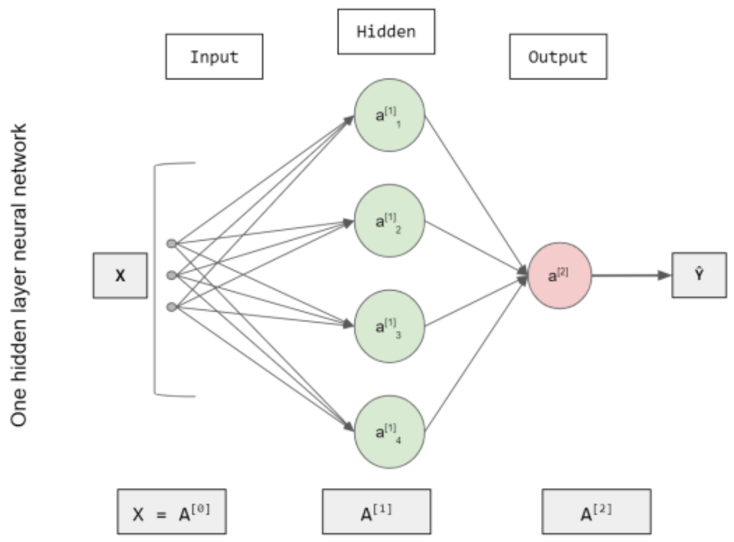


Figure 2-3. A one hidden layer neural network [5]

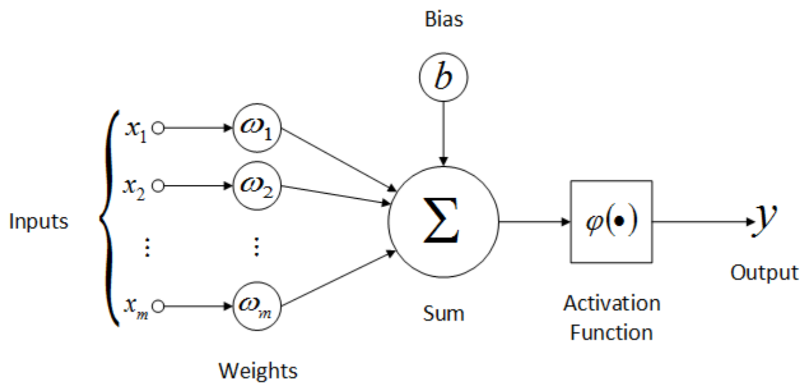


Figure 2-4. The neuron architecture [6]

As illustrated in Figure 2-4, each neuron in the network takes the output of the neurons from the last layer, multiplies each input with a weight and sum up the results and adds a bias on it. The result will pass a non-linear activation function to obtain the final output of the neuron. This process could be expressed as,

$$y = \varphi\left(bias + \sum_{n=1}^{m} \omega_n x_n\right) \tag{5}$$

The reason why an activation function is necessary is that a non-linear system is expected to be more capable to express a complex problem. If a linear activation function or no activation function is used, the whole system will be linear and less flexible. Common activation functions include Sigmoid, tanh, Relu, LeakyRelu etc [7]. The deep neural network is composed of these neurons in a layer-to-layer pattern, so that it could be considered as a very complex non-linear function, which has a huge number of variable parameters.

In the field of computer vision, convolutional neural network is playing an important role. The two main reasons the convolutional neural network is powerful are that it could learn translational invariant features and the spatial hierarchies of patterns. These features of CNN are mainly achieved by two architectures, which are convolutional layer and pooling layer.

- Convolutional layer

  In convolutional layer, convolutional kernels with learnable parameters are used to extract the features in the image. As shown in Figure 2-5, the kernel will slide in the input matrix and the dot product of the kernel and the matrix it covers form a feature map, which indicates the distribution of the pattern in the kernel. Therefore, the kernel is working as a filter to filter out the useful features. In training phase, the values of the convolutional kernels will be gradually optimized to extract the most useful features for the specific batch of training data.

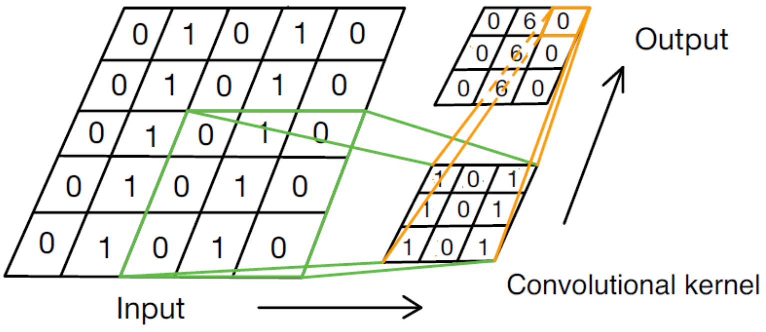


Figure 2-5. The convolution processes

- Pooling layer

  Pooling layers are usually used right after the convolutional layers. Pooling is a down sampling process. For max pooling process shown in Figure 2-6, the maximum number for each quarter remains and generate a new smaller matrix, while the average values are used for average pooling. The function of pooling is to eliminate the details of the feature and concentrate on the distribution of the feature in a larger scale.

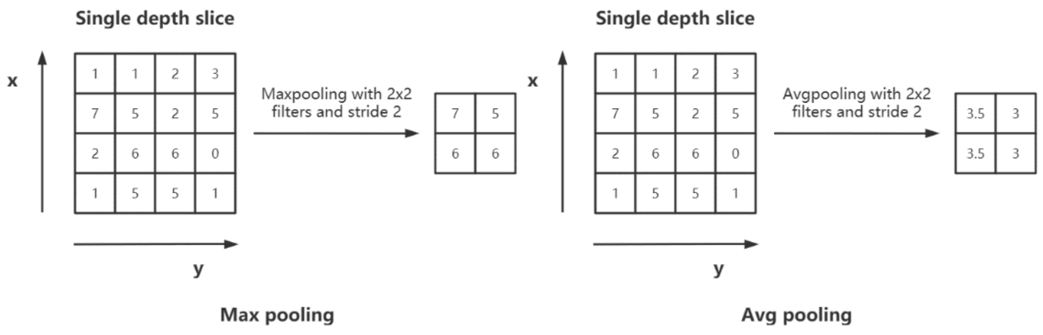


Figure 2-6. Max pooling and average pooling operation

The aforementioned two functional layers are followed by each other for a general CNN. Imagine you are observing a picture of a cat, you firstly notice the features such as eyes, ears and a flurry body, then you may recognize the object in the picture is a cat according to the distribution of the found features in the last step.

Deep learning has achieved better performance in computer vision tasks such as object detection and semantic segmentation compared to classical algorithms. For an autonomous robot, scene understanding is critical for path planning and obstacle avoidance. With the increasing embedded computational power, this task could be perfectly done by deep learning on a mobile embedded system such as NVIDIA Jetson development board.

Collision-free space segmentation could be considered as a low-level scene understanding. Through this approach, the area with and without obstacles could be classified in pixel level. And this task is usually realized by fully convolutional neural network (FCN). In FCN, the input will pass several convolution-pooling blocks for feature extraction. The extracted feature map was deconvoluted to the original size. This architecture could be trained end-to-end to produce a pixel level classification.

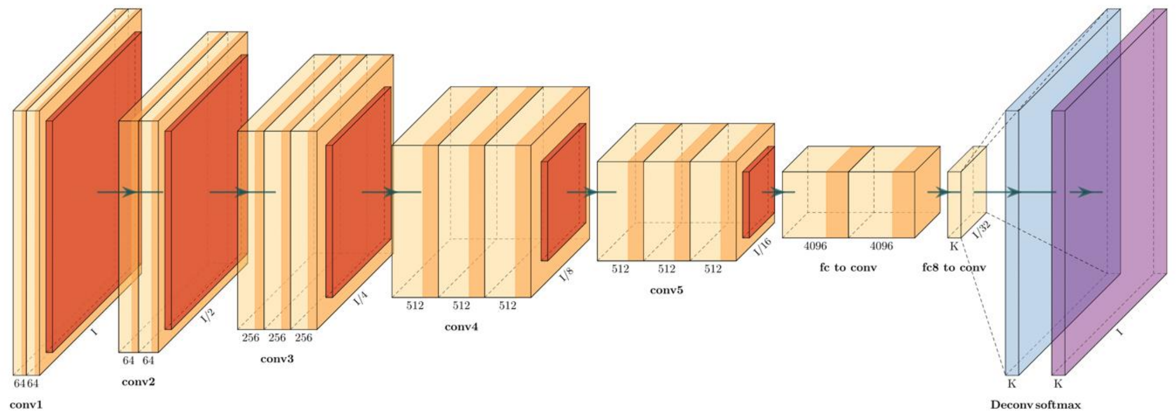


Figure 2-7. Fully convolutional neural network architecture

However, it is still far from good in the segmentation performance by only RGB frame input. Human-beings could perceive the spatial information by observing the environment through two eyes. This ability allows us to tell whether a “pattern” seen is an obstacle or just the texture on certain surface. Machine confronts the same problem. In some environment with complex distributed textures, is it difficult to confirm the existence of obstacles only by the features captured in RGB image. Therefore, depth camera was used to provide complementary spatial information so that the deep learning model could fuse it with RGB image to produce a prediction with higher accuracy. Caner et al. compared the performance of the collision-free space segmentation with and without depth input. The results shown that with the same RGB branch of the network, the accuracy of the prediction with depth information was about 5% higher.

Rui et al. proposed a model for free-space detection [8], which was shown in Figure 2-8. The RGB frame and depth map are input to extract the features by a series layer of convolutional neural network which use ResNet as backbone. The feature maps in two branches are hierarchically concatenated to fuse the information. Then the concatenated feature maps are densely connected to be unsampled to the original resolution.

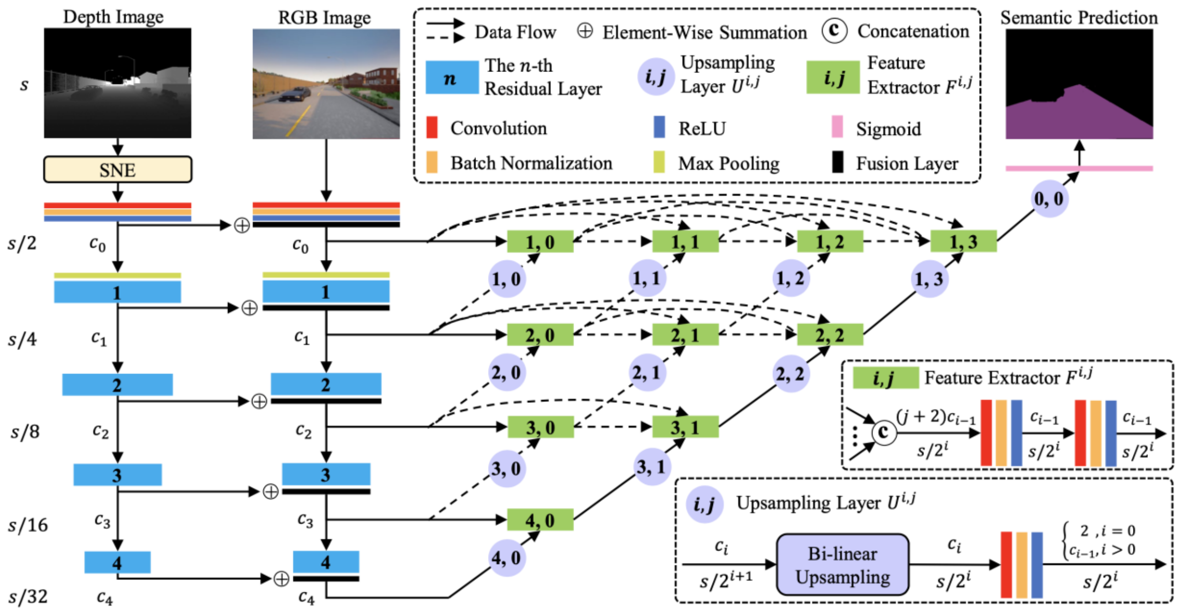


Figure 2-8. The architecture of the segmentation neural network [8]

This model could fuse the information in both RGB image and depth map to obtain a higher accuracy compared with using RGB image or depth map alone. *Rui et al.* also compared the performance quantitively in five indexes, which are Accuracy, Precision, Recall, F-score and IoU. The definitions of the performance assessment indexes were illustrated and explained below.

As shown in Figure 2-9, consider the red area as ground truth and the area rounded by dotted line as the prediction of the deep learning model. There are four kinds of areas, which are true positive, true negative, false positive and false negative. The five indexes were defined by Equation 6-10.

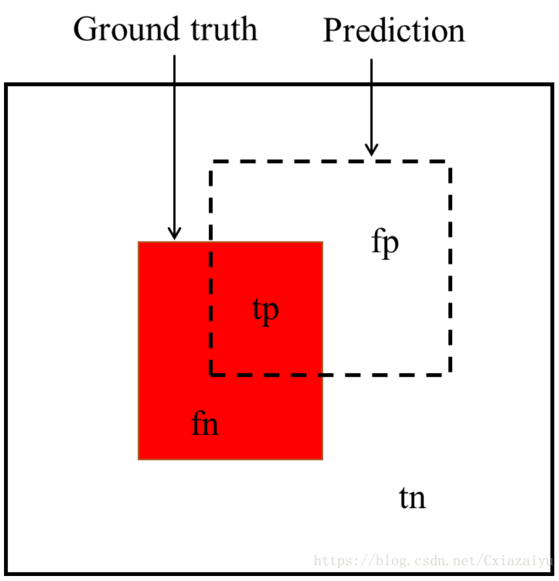


Figure 2-9.

Tp: True Positive

Tn: True Negative

Fp: False Positive (false detection)

Fn: False Negative (miss detection)

$$IoU = \frac{tp}{tp + fp + fn} \tag{6}$$

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn} \tag{7}$$

$$Precision = \frac{tp}{tp + fp} \tag{8}$$

$$Recall = \frac{tp}{tp + fn} \tag{9}$$

$$F - score = (1 + \beta^2)\frac{Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} \tag{10}$$

The results shown in Table 1 demonstrated that fusing RGB and depth information could give a higher accuracy. However, it turned out that using only depth image could even achieve higher accuracy. This phenomenon proved that depth information was essential for free-space segmentation task, which was quite intuitive because obstacles often appear with huge depth difference.

Table 1. Performance comparison (%) [8]

Architecture	Accuracy	Precision	Recall	F-score	IoU
RGB	94	91.9	93.8	92.8	86.6
Depth	96.7	97.6	94.6	96.1	92.4
RGB-Depth	95.1	92.8	95.6	94.2	89.0

2.2.1 TensorRT inference Engine

When we talk about utilizing deep neural network to solve problems, there are usually two stages, which are training and deploying (inference). TensorFlow and PyTorch are two popular frameworks for deep neural network training. However, deploying the network by using TensorRT will profoundly minimize the latency. According to the official document, TensorRT could provide about 5 times speed improvement in ResNet inference.

TensorRT accelerates the deep learning inference mainly from four aspects [9]:

- Layer & Tensor fusion
  TensorRT will do layers and tensor fusion during the optimization, which was illustrated in Figure 2-10. This will significantly reduce the cost of reading and writing the tensor data between layers.

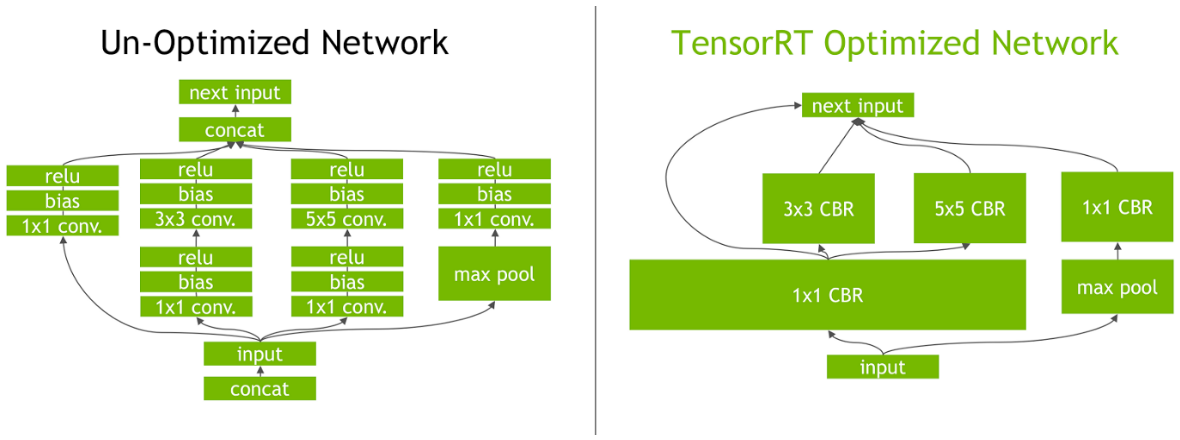


Figure 2-10. The layer fusion

- Mixed Precision

  When we train the neural network, we do not care about the speed but the accuracy. The neural network is trained by slightly changing the weights in the network towards the direction of gradient, which is called Gradient Decent method. For each epoch of training, it is ideal to gradually change the weight slightly and finally reach the state that could precisely express the mapping between the input and the output. Therefore, 32-bit float point (FP32) weights are used to guarantee the high accuracy. However, it is not necessary to use FP32 when we want the network to make inference. Hence, FP16 and even INT8 are used to replace the original FP32 weights in the trained model.

- Kernel Auto-tuning

  NVIDIA integrated some special hardware architecture named tensor core into their latest GPU platform for high performance tensor computing. TensorRT will optimize the model based on the target GPU hardware to fully utilize the general kernels and tensor core on GPU to obtain maximum computing efficiency.

- Dynamic Tensor Memory

  TensorRT improves the memory reuse by allocating memory to tensor only for the duration of its usage. It helps in reducing the memory footprints and avoiding allocation overhead for fast and efficient execution.

2.3 Path Finding Algorithm

After the localization of the robot, it is necessary for the robot to find the fastest way to approach the destination. As the map will consist of nodes, and the goal is to find a shortest path from the current position to the destination, graph search algorithms will be used.

2.3.1 Breadth First Search (BFS)

BFS is the simplest path finding algorithm, whose key idea is exploring every possible direction for each step of search until the target is found [10]. Specifically, BFS will first search the block with minimum value of the evaluation function, which is

$$f(n) = g(n) \tag{11}$$

Where $g(n)$ is the distance between the block n and the start point.

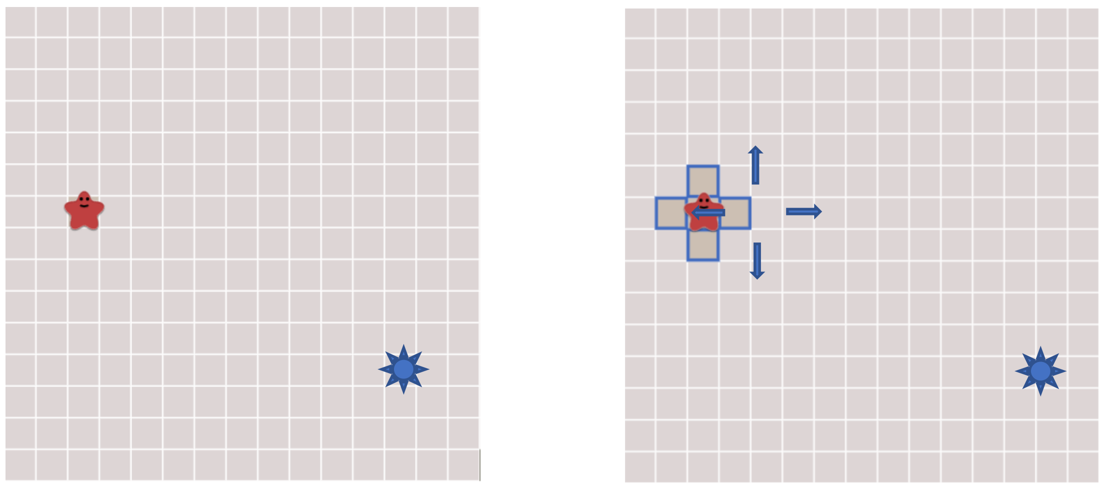


Figure 2-11.

To find a path that connects the start point (red symbol) and the target (blue symbol) shown in Figure 2-11, the BFS algorithm will set all the blocks that connect to the current block as candidates (blocks with blue boundary in Figure 2-11), and check if the final target is included until the target is reached.

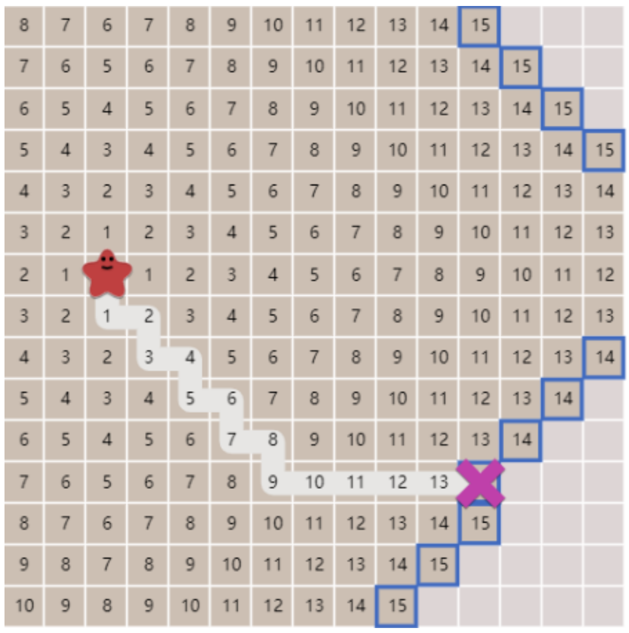


Figure 2-12.

The results shown in Figure 2-12 illustrated that the target was reached after 14 rounds of block expansion. It could be observed that most of the blocks in the map were used in the algorithm, which was probably not efficient.

2.3.2 Greedy Best First Search

BFS algorithm expands and explores all the blocks in all possible directions. However, sometimes it is much more efficient to pay more attention to the direction that the target lies. Therefore, in Greedy Best First Search, the block that is the closest to the target will be explored first. An evaluation function will be used to decide the next block to be explore, which is,

$$f(n) = h(n) \tag{12}$$

Where $h(n)$ is called heuristic function, which provides an estimation of the distance between the block n and the final target. The candidate block with the minimum value of $f(n)$ will be explored first.

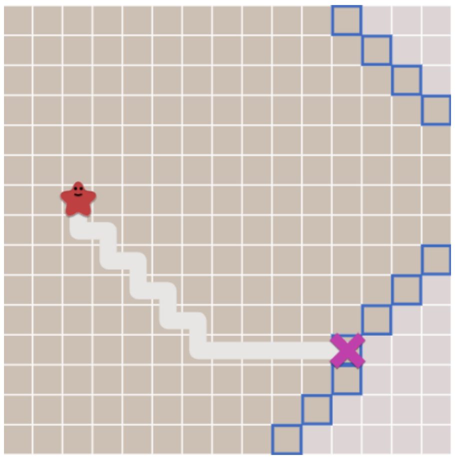


Figure 2-13

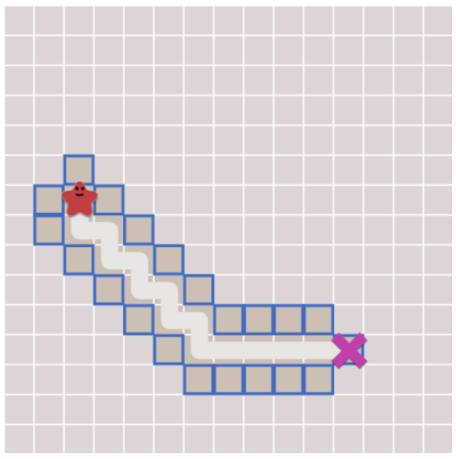


Figure 2-14

It could be observed in Figure 2-13 and 2-14 that Greedy Best First Search could avoid exploring the path that will less possibly go to the target while find the correct shortest path, which saves time and computing resources to a great extent. However, this algorithm may go astray if there are obstacles in the map.

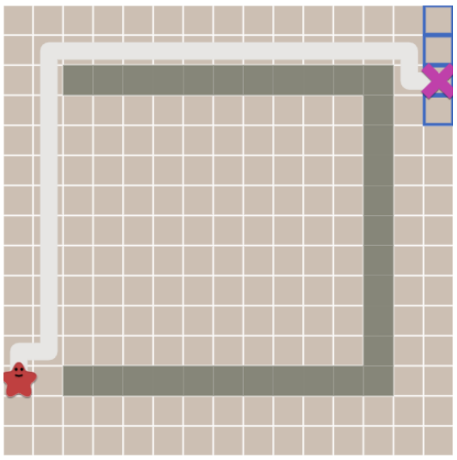


Figure 2-15

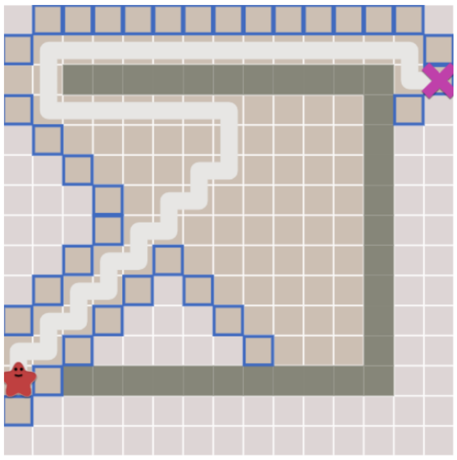


Figure 2-16

In the situation shown in Figure 2-15 and 2-16, although the Greedy Best First Search algorithm found one path to the target quickly compared to BFS algorithm, the path it found was not the shortest one.

2.3.3 A* Algorithm introduction.

A* algorithm is a compromise proposal, which uses both the estimated distance to the target and the number of steps explored so far. It decides the next block to explore by the **sum** of the distance from the start point and the distance to the target. The evaluation function of A* algorithm is

$$f(n) = g(n) + h(n) \tag{13}$$

$g(n)$ is the number of steps between the start point and the block been evaluated, which represents the current cost. And $h(n)$ is the heuristic function. The candidate block with the minimum value of $f(n)$ will be explored first.

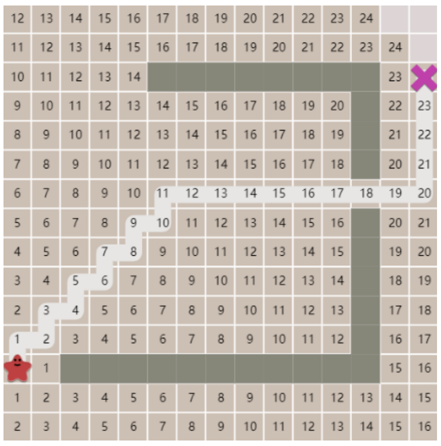


Figure 2-17

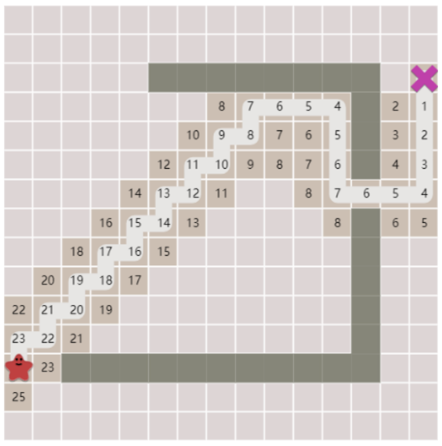


Figure 2-18

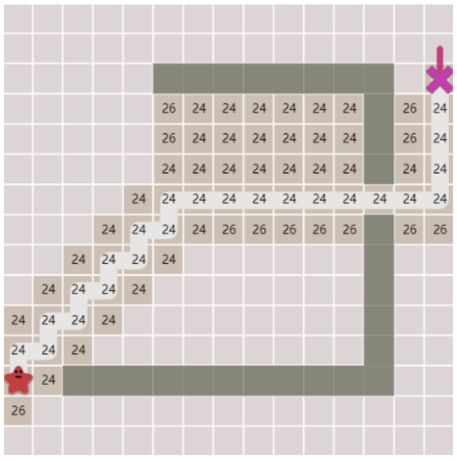


Figure 2-19.

Compared to the BFS algorithm and Greedy Best First Search algorithm, A* could find the shortest path while maintaining a relatively low computing cost, which was illustrated in Figure 2-17 to 2-19. Therefore, A* algorithm will be used in this project for global path planning.

Chapter 3 System Architecture

3.1 Overall System Architecture

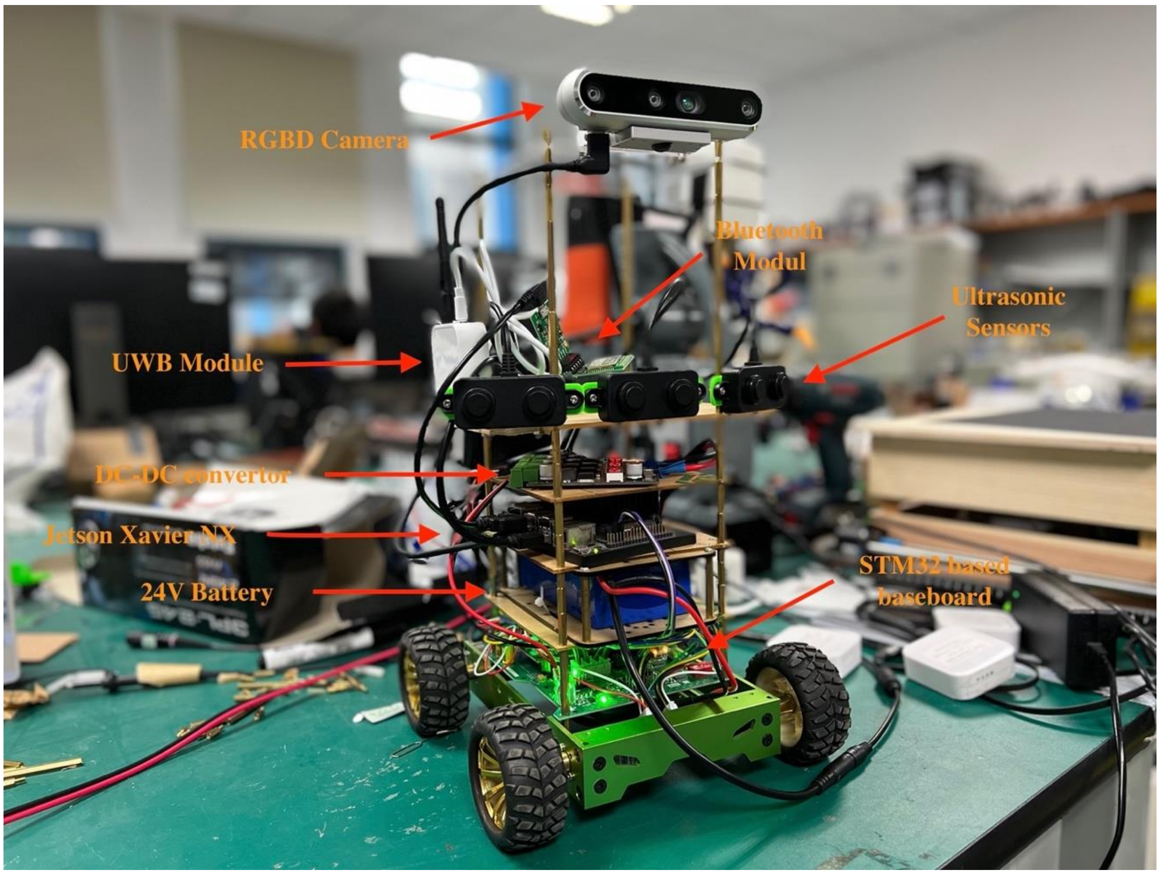


Figure 3-1. The photo of the robotic vehicle

The vehicle mainly consists of a chassis, a battery, a central controller and peripheral devices including sensors and power convertors. RGB-D camera and ultrasonic sensors for environment perception, UWB module for robot localization, Bluetooth module for robot-iPhone communication, these devices are all connected to Jetson Xavier NX board for processing and the decision making of the robot. ROS (Robot Operating System) was run on Ubuntu 18.04 on Jetson Xavier NX. The robot is powered by a 24V Li-ion battery followed by a multi-output DC-DC convertor. The detailed specs of each part of the robot are shown in

Table 2. The specification of the robotic vehicle

Items	Key Specs
RGB-D Camera	Depth Resolution and Frame Rate: Up to 1280×720 at 30 fps Depth Field of View: 87° × 58° Depth Accuracy: <2% at 4 m^2 RGB Resolution and Frame Rate: Up to 1280×720 at 30 fps RGB Field of View: 90° × 65° Ideal Range: 0.6-6m
Ultrasonic Sensors	Maximum Output Frequency: 100Hz Ideal Range: 0.2-5m Detection angle: 60 degrees
UWB Module	Maximum Data Rate: 6.8Mbps Theoretical Localization Precision: 10cm Operating Frequency: 3244-4659MHz Transmission Power: <-39dBm/MHz
Bluetooth Module	Operating Frequency: 2.4GHz Transmission Power: <4dBm Maximum Data Rate: 1Mbps
IMU	Roll/Pitch Precision: 0.1°RMS Yaw Precision: 0.5°RMS Angle Resolution: <0.01° Output Frequency: 200Hz

3.2 Hardware System

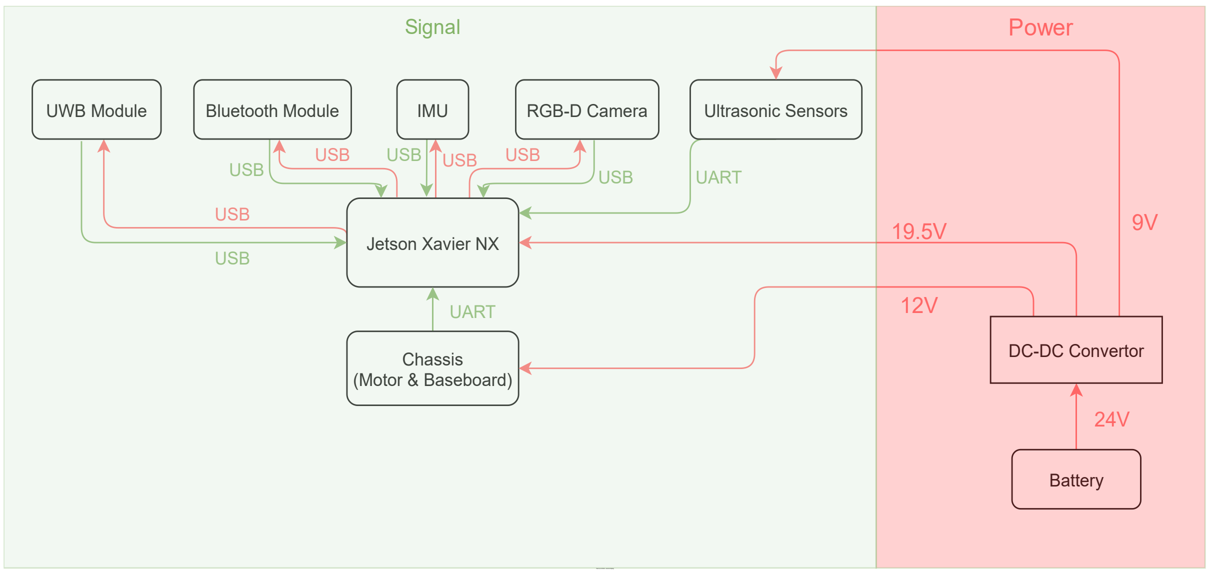


Figure 3-2. The hardware architecture of the robotic vehicle

3.3 Software System

3.3.1 ROS nodes framework

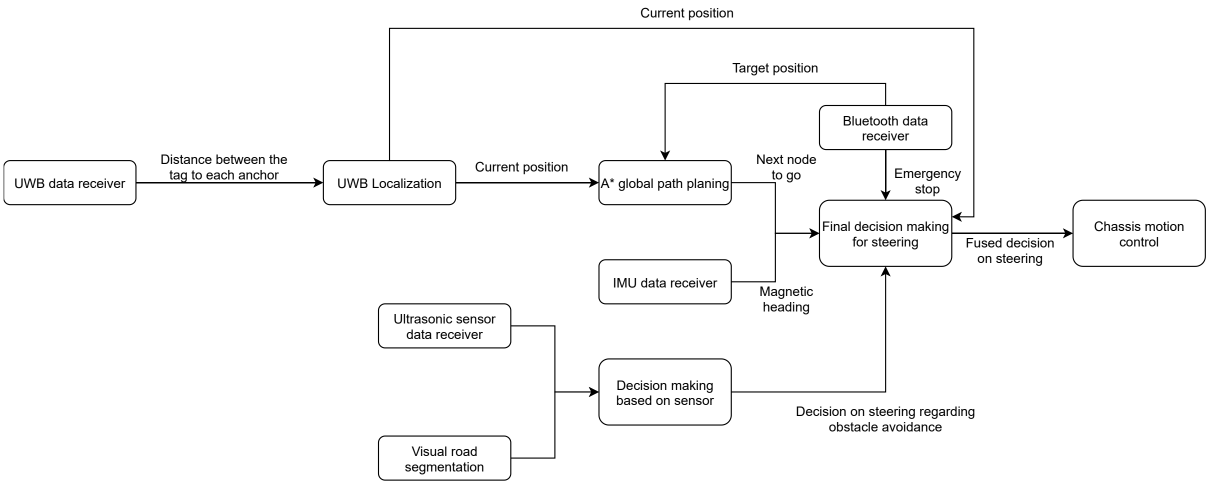


Figure 3-3. The ROS node architecture

ROS is an open-source, meta-operating system that integrates hard hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management [11].

ROS could be considered as a “social networking platform” with some integrated “mini program” for hardware connected to the host. Each hardware component can publish data on ROS and retrieve data published by other components. Integrated program like RVIZ could visualize the data published, and the inter-relations among components could be displayed by RQT tool as a graph by ROS as well, which greatly facilitate the development process.

The overall nodes framework that implemented in ROS was shown in Figure 3-3. The UWB data receiver node, the ultrasonic data receiver node, IMU data receiver node and the Bluetooth data receiver node were responsible for reading data from relative sensors and publishing them in ROS. The UWB data receiver node published the data that contained the distance between the tag and each anchor and the UWB Localization node utilized these data to calculate the tag's current location by triangulation method. The current location of the tag was used by A* global path planning node to generate a path towards the final target position, which was given by the Bluetooth data receiver. The A* global path planning node will send the next waypoint to the final decision-making node for a global path indication. Meanwhile, on another branch, the visual road segmentation node took the RGB and depth input from the camera, generated the mask of collision-free space and give an indication on steering with the ultrasonic sensors as backup in the aspect of obstacle avoidance. This local obstacle avoidance information was combined with the global path indication in the final decision-making node to generate a final steering command. The command was sent to the chassis control node, which established the communication with the microcontroller on chassis via UART.

3.3.2 TensorRT inference process

To use TensorRT to accelerate the PyTorch model, two possible ways were found. 1) Convert the PyTorch model directly into a TensorRT module by torch2trt tool, which could be utilized to perform inference with one line of code. 2) Convert the PyTorch to ONNX (Open Neural Network Exchange) model and convert the ONNX model into TensorRT inference engine. Use the TensorRT API to manually start the inference engine. It seemed that the first approach was simple. However, after testing, it was found that massive GPU memory was required to generate a TensorRT module by torch2trt tool. The Jetson Xavier NX, which was the controller used in the project, had only 8Gb unified memory for both CPU and GPU. The conversion process failed. Therefore, the second approach was finally used. The inference process was shown in the flowchart in Figure 3-4.

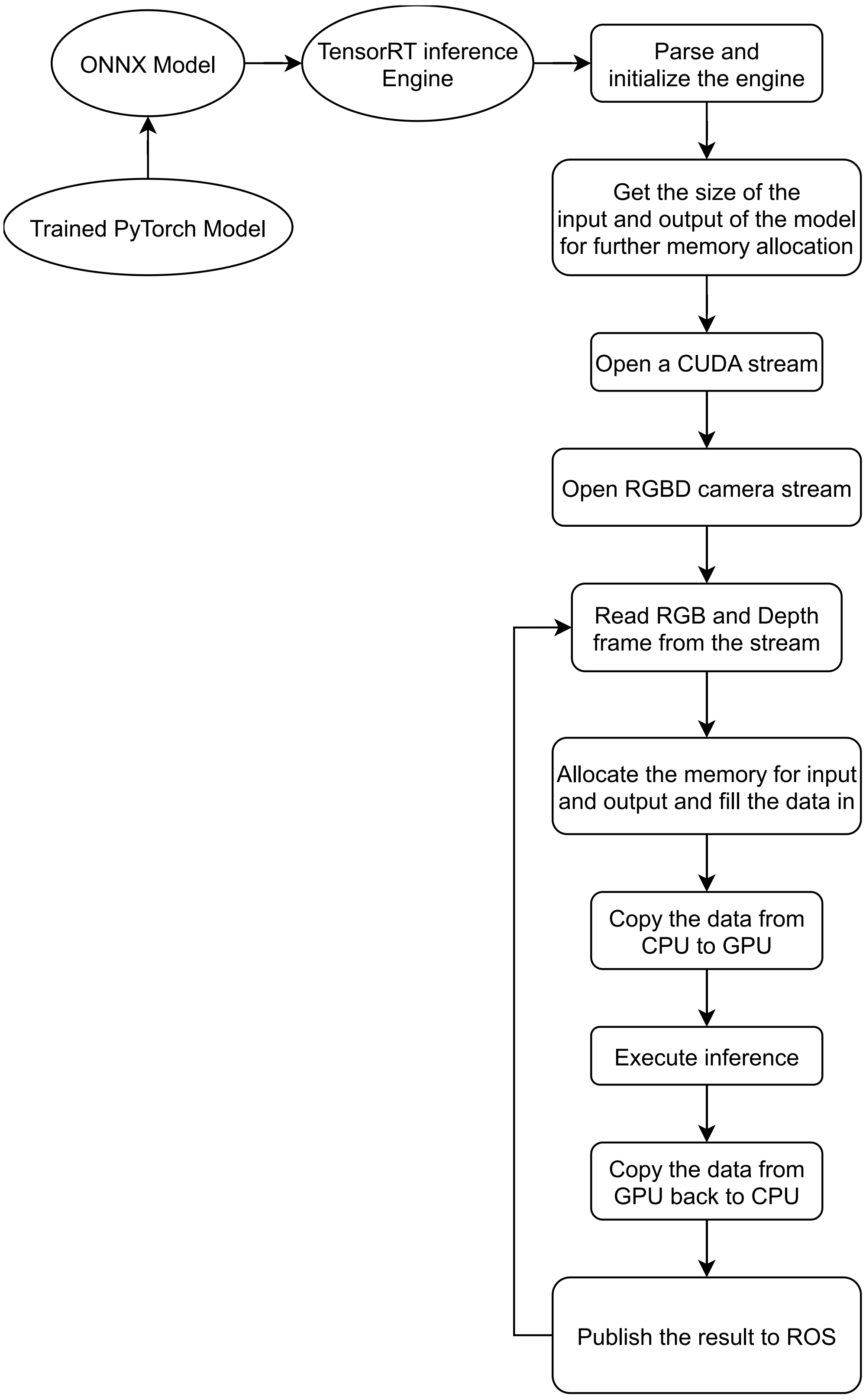

ONNX Model
TensorRT inference Engine
Parse and initialize the engine
Trained PyTorch Model
Get the size of the input and output of the model for further memory allocation
Open a CUDA stream
Open RGBD camera stream
Read RGB and Depth frame from the stream
Allocate the memory for input and output and fill the data in
Copy the data from CPU to GPU
Execute inference
Copy the data from GPU back to CPU
Publish the result to ROS

Figure 3-4. The flowchart of TensorRT acceleration

3.3.3 Navigation and Obstacle Avoidance

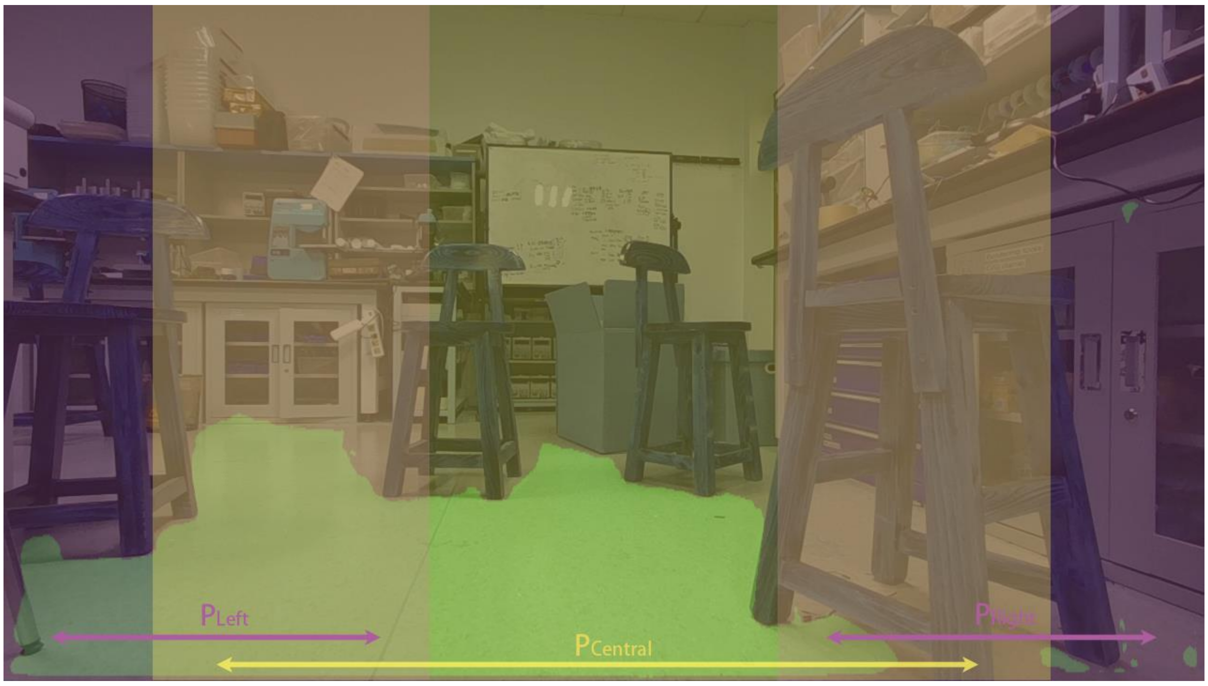


Figure 3-5.

After the machine perception results were obtained, the decision on the steering of the vehicle should be inferred based on the existing information. To achieve the smooth and safe autonomous operation, the decision making should satisfy following rules: 1) Always go towards the direction with both less obstacles and the global target; 2) When there is not enough empty space in the central front, search possible available path on two sides and temporarily ignore the global target to bypass the obstacle; 3) When there is no path available, stop in place.

The path planning was based on local path planning and global path planning. These two types of path planning will generate steering decision respectively and finally integrated to get a comprehensive result.

- Local Path Planning

The visual segmentation result was divided into three areas overlapped to each other, which located in the left, central and right part of the field of view. And the pixels that in green (without obstacles) in each area are counted as an indication of the road conditions and were denoted as P_{Left}, $P_{Central}$ and P_{Right}. In pixels counting, the pixels appeared in upper half image were ignored because no road was expected to appear there. The readings of ultrasonic sensors installed towards left side, front and right side were denoted as U_{Left}, $U_{Central}$ and U_{Right}.

The algorithm will first check whether there was enough collision-free space in the central front of the vehicle. If the area of the collision-free space was larger than the pre-defined threshold, the pixels included in $P_{Central}$ will be added up with weight that increased from left to right. In this way, the direction with more obstacles could be found so that the vehicle could turn to another side to bypass the obstacle in advance. This process was intuitive as human beings tended to go to the place that without many obstacles as well. When $P_{Central}$ was lower than the threshold, which meant that there was not enough traversable area to go, the vehicle will check the area on the two sides (P_{Left} and P_{Right}) instead to find an alternative way. If there was no way found through the camera, the ultrasonic on the two sides of the vehicle will check whether it was an empty space outside of the field of view of the camera. The vehicle will take a sudden turn to the direction with empty space if there was empty space detected by the ultrasonic sensors and will stop at the place if the ultrasonic sensors detected obstacles as well.

- Global Path Planning

The A* algorithm would continuously generate a path from the current position to the target position as a series of waypoints in the pre-defined coordinates, which was illustrated in the flowchart in Figure 3-6.

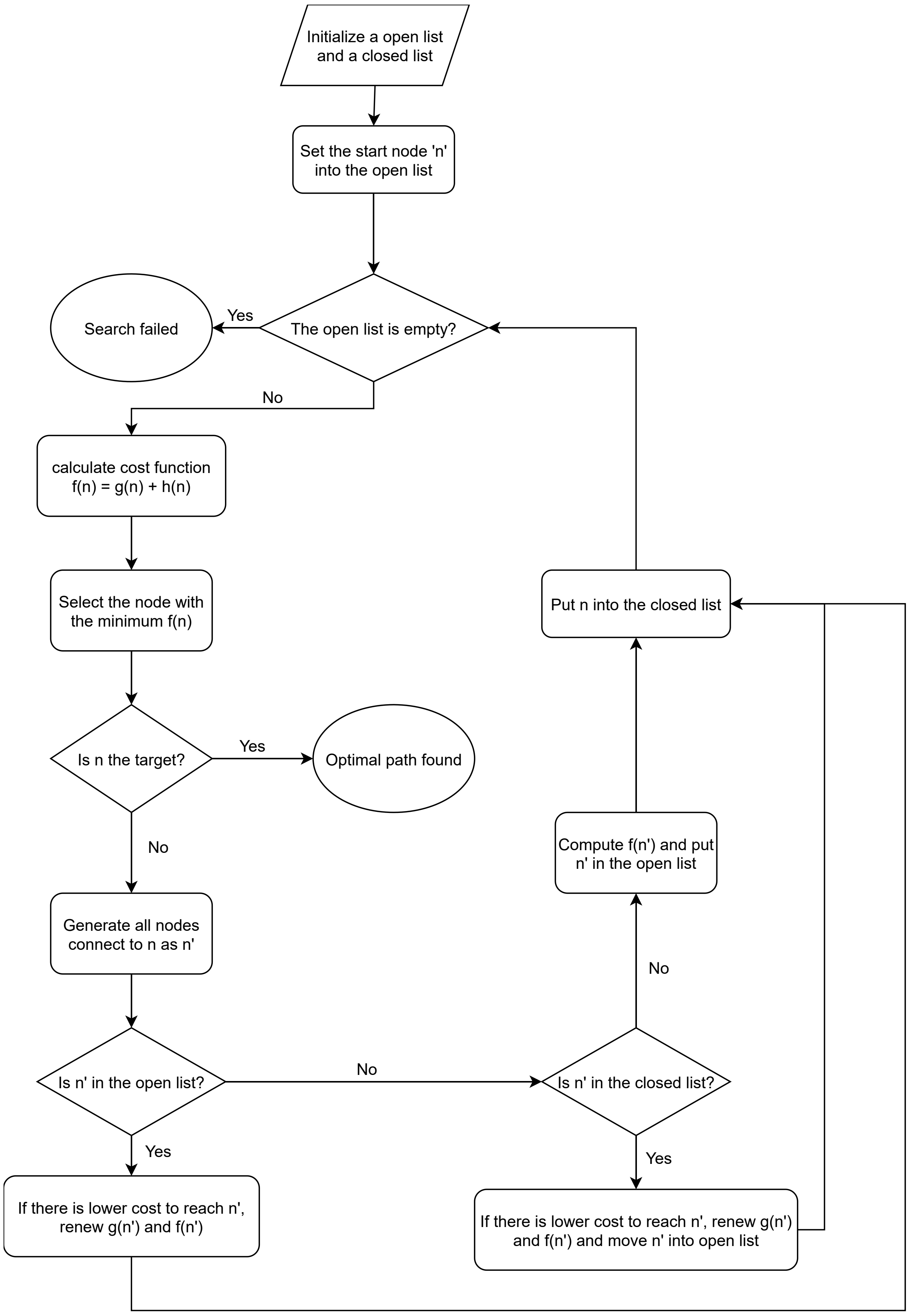

Initialize a open list and a closed list
Set the start node 'n' into the open list
Search failed
Yes
The open list is empty?
No
calculate cost function f(n) = g(n) + h(n)
Select the node with the minimum f(n)
Put n into the closed list
Is n the target?
Yes
Optimal path found
No
Compute f(n') and put n' in the open list
Generate all nodes connect to n as n'
No
Is n' in the open list?
No
Is n' in the closed list?
Yes
Yes
If there is lower cost to reach n', renew g(n') and f(n')
If there is lower cost to reach n', renew g(n') and f(n') and move n' into open list

Figure 3-6. The flowchart of the A-star algorithm

Meanwhile, the current heading could be got by the magnetometer aided IMU. And the expected heading could be computed by

$$yaw = \arctan\left(\frac{y_{target} - y_{target}}{x_{target} - x_{target}}\right) \tag{14}$$

The error between the expected heading and the current heading was used as an indication of steering for global navigation.

The flowchart of the obstacle avoidance strategies that applied to the robot was shown in Figure 3-7.

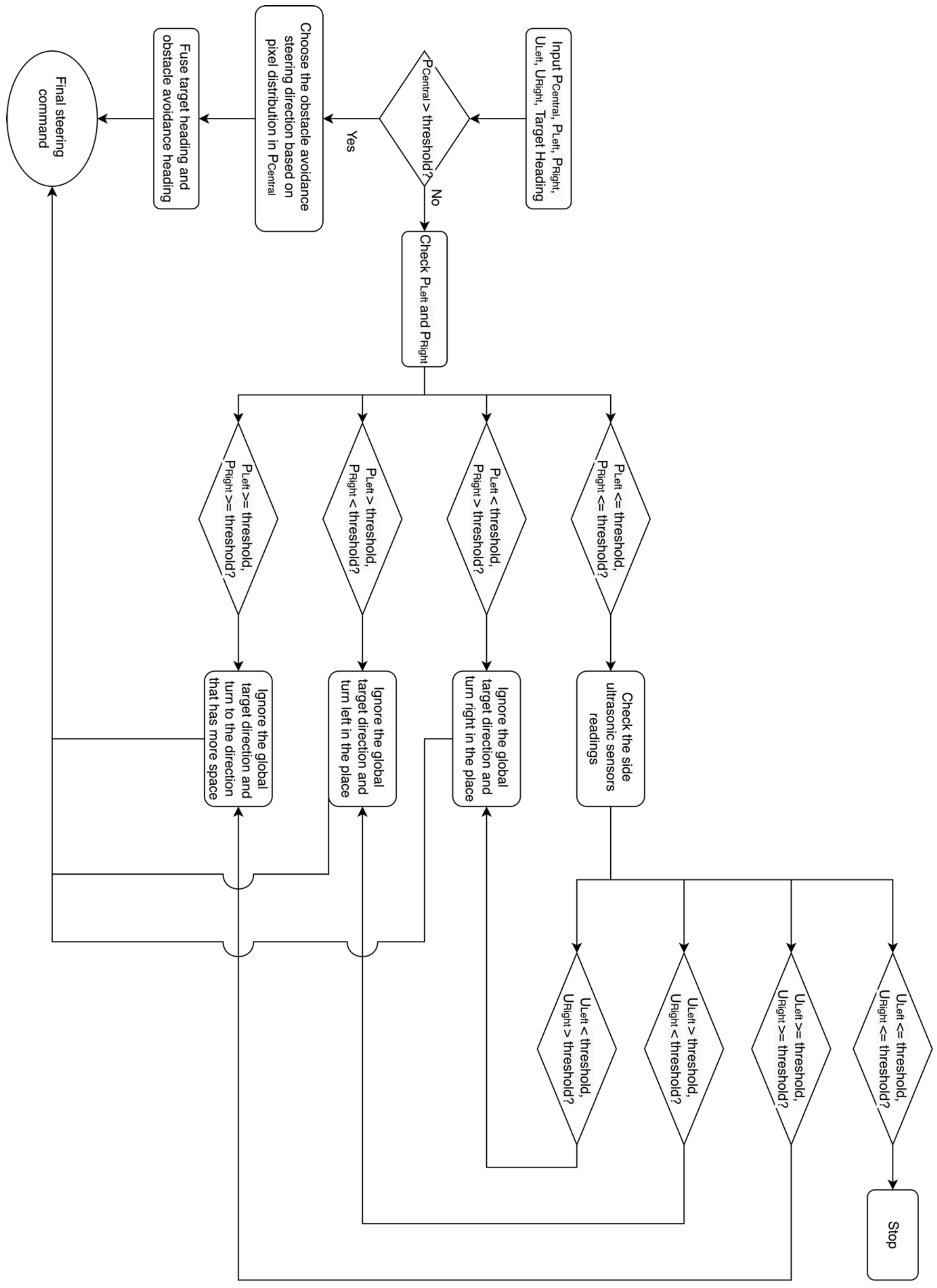

Input PCentral, PLeft, PRight, ULeft, URight, Target Heading
PCentral > threshold?
Yes
No
Choose the obstacle avoidance steering direction based on pixel distribution in PCentral
Fuse target heading and obstacle avoidance heading
Final steering command
Check PLeft and PRight
PLeft <= threshold, PRight <= threshold?
PLeft < threshold, PRight > threshold?
PLeft > threshold, PRight < threshold?
PLeft >= threshold, PRight >= threshold?
Check the side ultrasonic sensors readings
Ignore the global target direction and turn right in the place
Ignore the global target direction and turn left in the place
Ignore the global target direction and turn to the direction that has more space
ULeft <= threshold, URight <= threshold?
ULeft >= threshold, URight >= threshold?
ULeft > threshold, URight < threshold?
ULeft < threshold, URight > threshold?
Stop

Figure 3-7. The algorithm of the obstacle avoidance

After the steering decision from the global path planning and the local path planning were obtained. The final steering command should be inferred by fusing these two factors.

The idea was the less traversable area in front of the vehicle, the local path planning was more important. Therefore, the final steering command was obtained by

$$steering = global\ steering * \alpha + local\ steering * (1 - \alpha) \tag{15}$$

$$\alpha = \frac{P_{Central}}{The\ number\ of\ all\ pixels\ in\ the\ central\ area} \in \begin{bmatrix} 0 & when\ P_{Central} < threshold \\ 0.3 - 1 & when\ P_{Central} \geq threshold \end{bmatrix} \tag{16}$$

With this design, when $P_{Central} < threshold$, the steering command will be completely taken over by the local path planning because it would be very urgent to bypass the obstacles in front of the vehicle but not global navigation. When $P_{Central} \geq threshold$, the final steering will depend on both global and local path planning dynamically so that when the vehicle met obstacles, it will respond more actively and when there was no obstacle, it could go towards the target directly.

Chapter 4　Tests, Results and Discussions

4.1 UWB Localization

For localization, totally four UWB modules were used, which included **three** modules operated as Anchors and **one** module operated as a Tag. The localization accuracy test was carried out at Lobby A on the first floor of IEB, UNNC. The layout of the anchor modules and the coordinate assignment was shown in Figure 4-1. The anchor modules were installed on tripods with 2.1meters height and placed at three corners of the room. The tag module was placed at several different spots within the area enclosed by the anchors and the calculated positions were listed in Table 3.

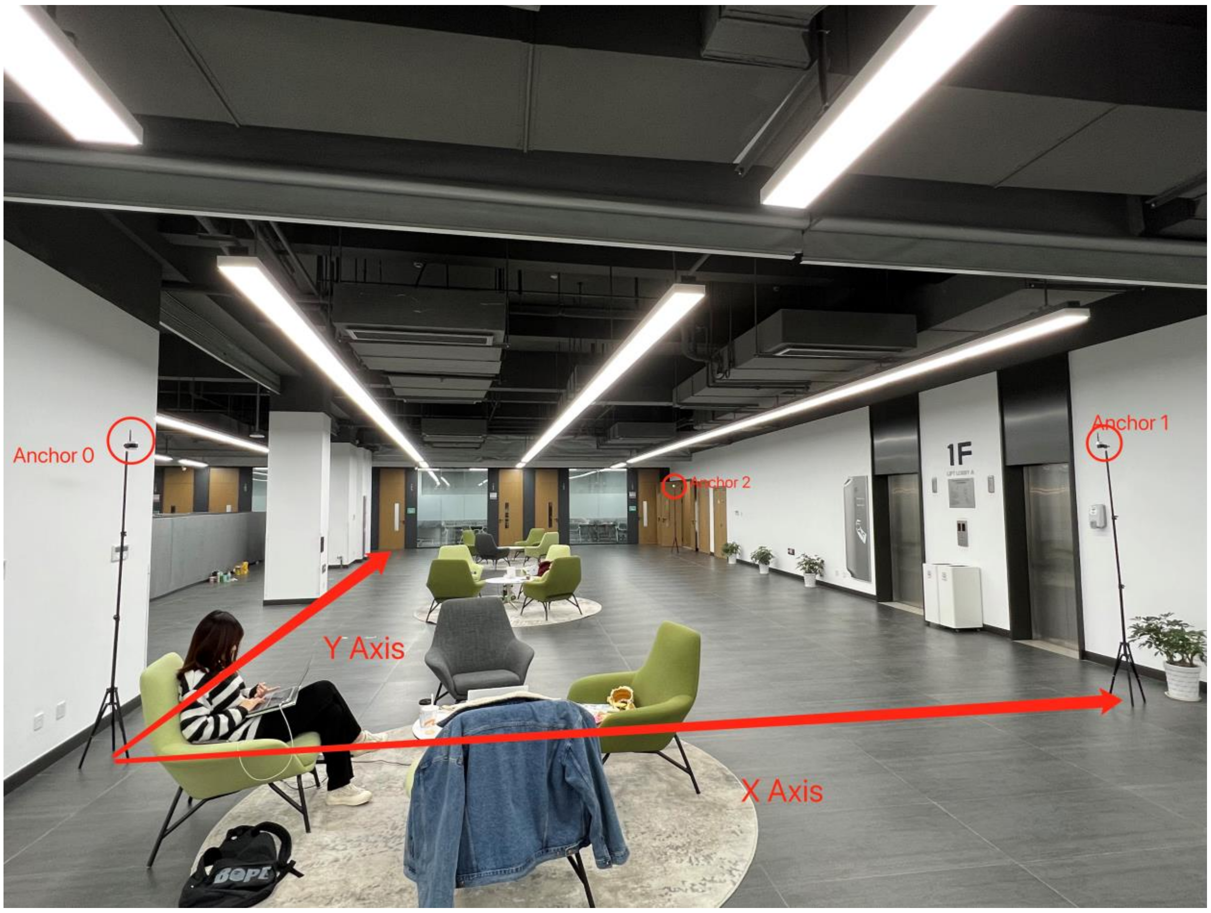


Figure 4-1. The UWB layout

Table 3. The comparison of the measured and calculated coordinates (Unit: m)

	Measured X coordinate (X_m)	Calculated X coordinate (X_c)	Measured Y coordinate (Y_m)	Calculated Y coordinate (Y_c)	Error
1	5.01	4.89	2.59	2.51	0.144
2	5.78	5.98	11.82	12.05	0.305
3	2.77	3.06	7.63	7.58	0.294

The error between the coordinates calculated by the algorithm and the measured one could be calculated by,

$$Error = \sqrt{(X_m - X_c)^2 + (Y_m - Y_c)^2} \tag{17}$$

According to the results listed in Table 3, the maximum error for localization was about 0.3 meter, which was acceptable.

4.2 Road Segmentation

4.2.1 Training Phase

The neural network was trained with a dataset of the environment in HKUST, including 1061 outdoor samples and 512 indoor samples. The dataset was divided into three sets, which are training set (20%), validation set and test set (20%).

To find the optimal training hyperparameters, several experiments were carried out in which different batch size and number of layers in ResNet were used for training. For each configuration, the model was trained for 400 epochs, which means that it was trained by all the samples in the training dataset for 400 times. The training process was super time-consuming and had a very high GPU memory demand. Therefore, two NVIDIA 2080Ti GPUs with overall 22Gb GPU memory and one NVIDIA Tesla V100 with 48Gb GPU memory were used for training. Relevant data were recorded and visualized using TensorBoard.

Figure 4-2 illustrated the comparison of the loss change on the training dataset during the training process. The loss was an evaluation of the model, and the loss function for binary classification problem used in the training was cross entropy, which was

$$loss = \frac{1}{N}\sum_{i} -(y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)) \tag{18}$$

Where N was the number of samples, y_i was the predicted label for the sample (1 for positive and 0 for negative) and p_i was the probability that the model predicted the sample as positive.

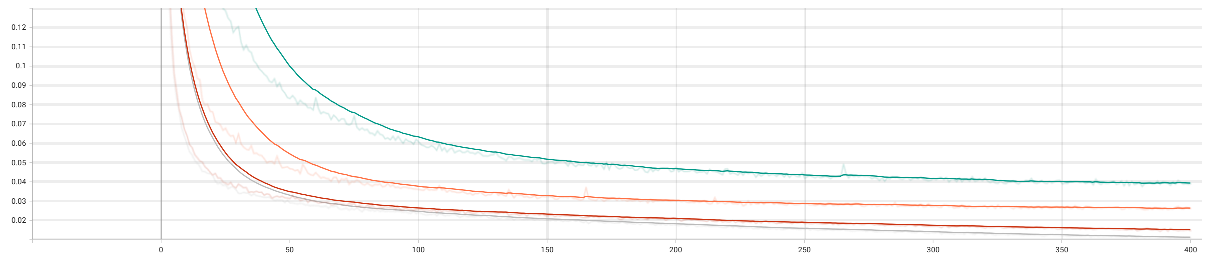


Figure 4-2. The loss changes during training (Grey: batch size = 2, ResNet 34; Red: batch size = 2, ResNet 18; Orange: batch size = 8, ResNet 18; Green: batch size = 12, ResNet 18)

According to Figure 4-2, it could be observed that larger batch size in training led to higher loss and slower convergence on the training dataset with the same layers of ResNet. With the same batch size (batch size = 2), ResNet with more layers had lower loss and higher convergence rate. As ResNet was a function block responsible for feature extraction, more layers in ResNet meant more adjustable parameters in the neural network could be used to express the relations between the input and the output. Therefore, it was intuitive that the network with ResNet 34 could achieve lower loss than the network with ResNet 18. As for the difference caused by the batch size, it was possible that for a large batch of data, loss was relevant to the mean of the whole batch, so it might be more difficult to let the loss on each sample to be small at the same time.

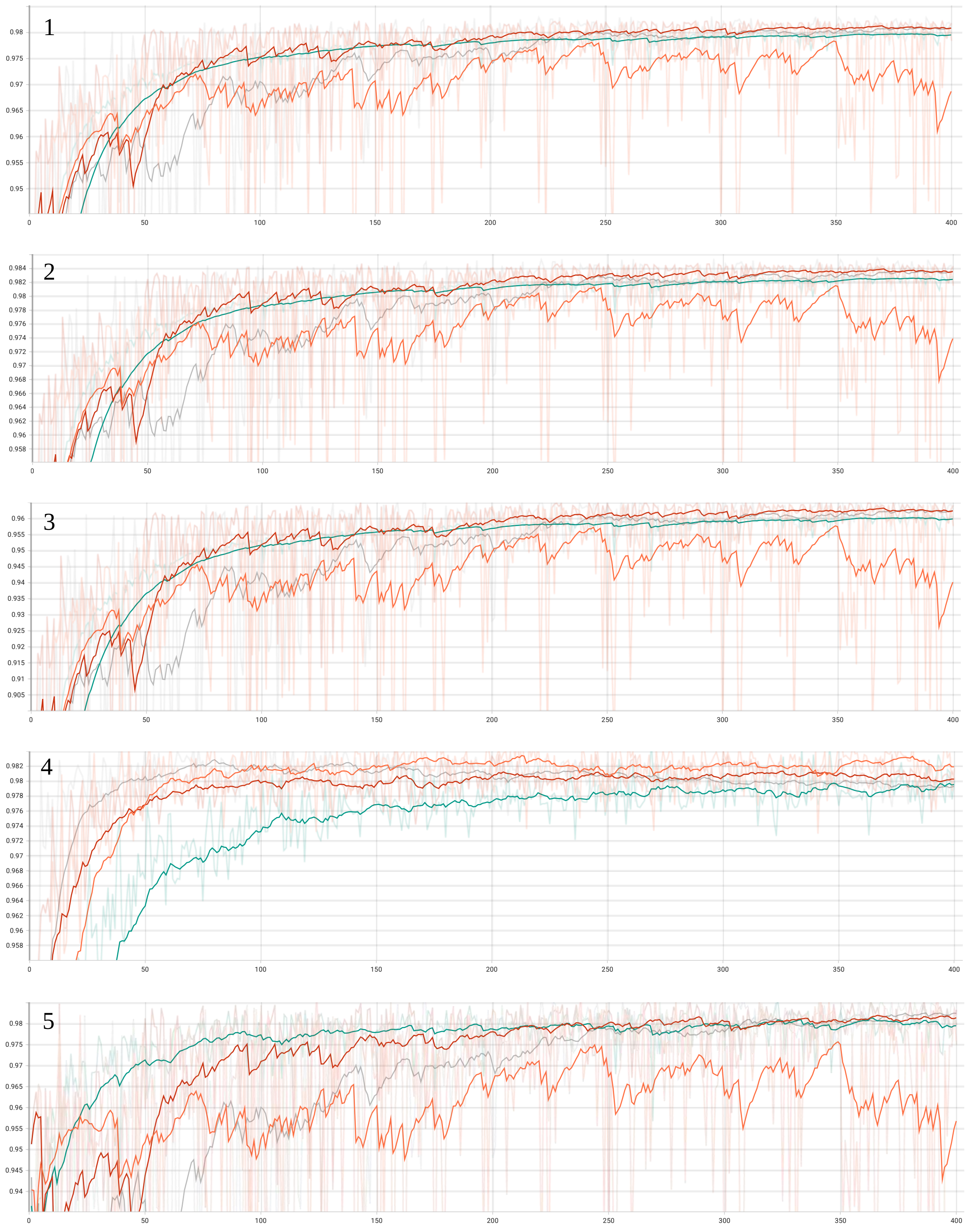


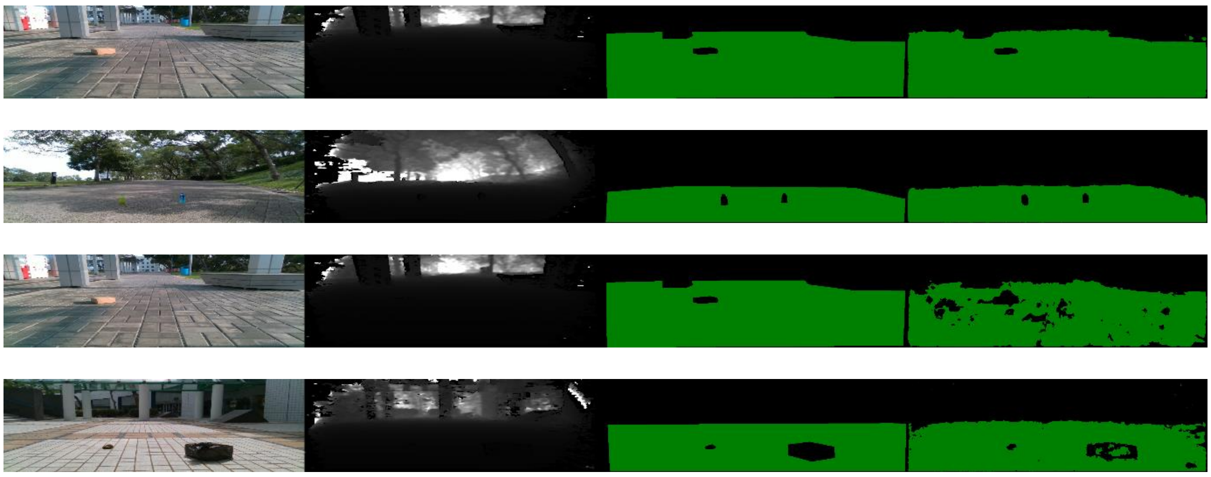
Figure 4-3 to 4-7. (1. F-score; 2. Accuracy; 3. IoU; 4. Precision; 5. Recall; Grey: batch size = 2, ResNet 34; Red: batch size = 2, ResNet 18; Orange: batch size = 8, ResNet 18; Green: batch size = 12, ResNet 18)

The changes of the five evaluation indexes introduced in Equation 6–10 on the validation dataset during the training process were illustrated in Figure 4-3 to 4-7 above. As the variation of the values were too large, the curves were applied a post-processing filter to make the trend clearer. All five indexes gradually increased after starting training and converged after about

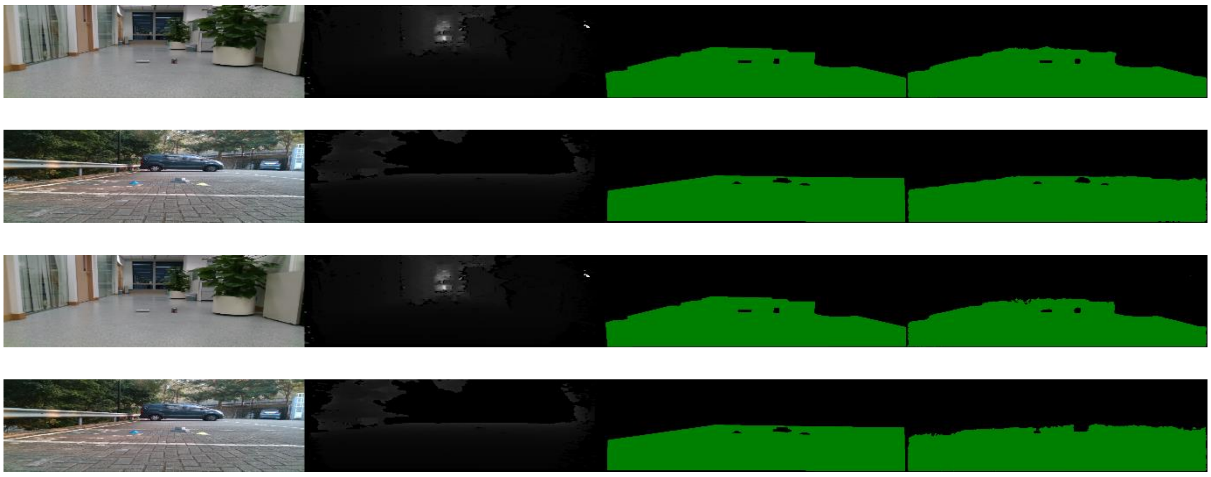
350 epochs of training for all configurations except the configuration with batch size = 8 and ResNet 18. The reason remained unknown. It was suspected that the GPU memory usage was too high which may lead to some internal problems with data transmission. In addition to the orange curve, although the rest of the curves converged at almost the same value with an error smaller than 1%, the results of small batch size were still slightly better than the results obtained by using large batch size. However, it was also possible that the large batch size needed more training steps to converge. Further investigation had to be done to confirm this assumption.

It was not obvious which training configuration was better by observing the data, however, there were obvious differences between the output of the model that was trained by different hyperparameter configurations.

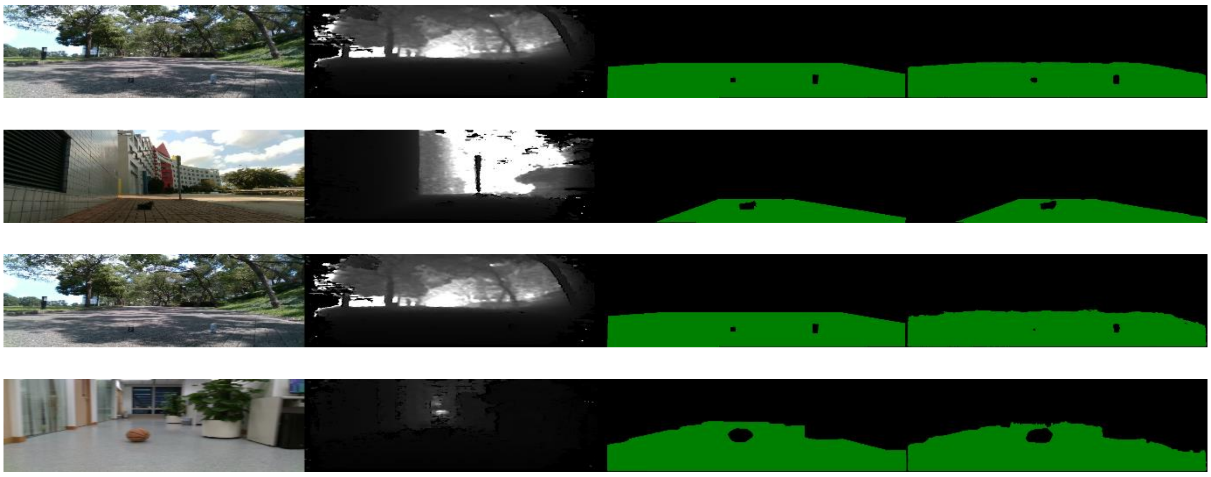
Epoch 10:



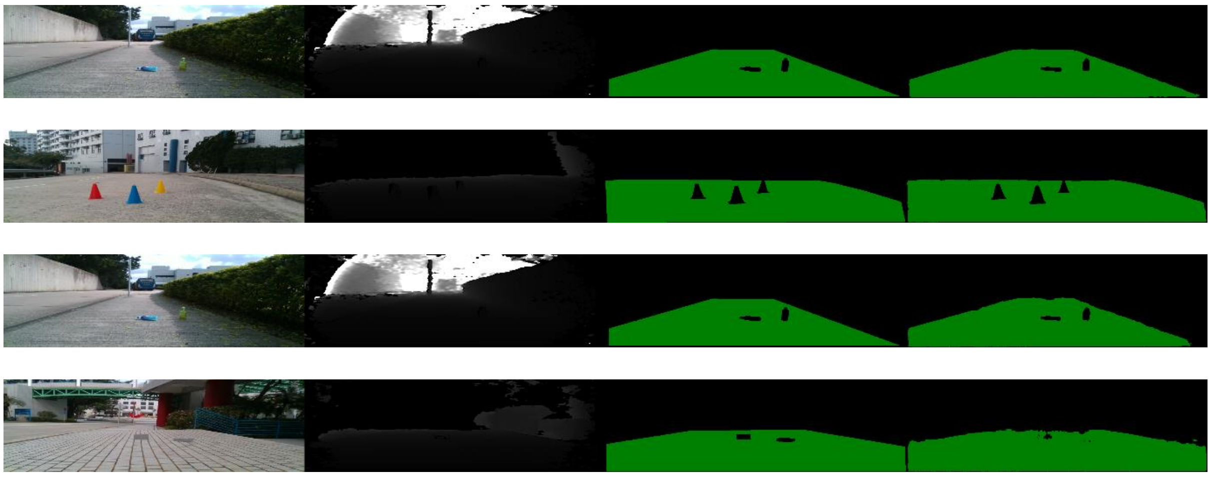
Epoch 50:



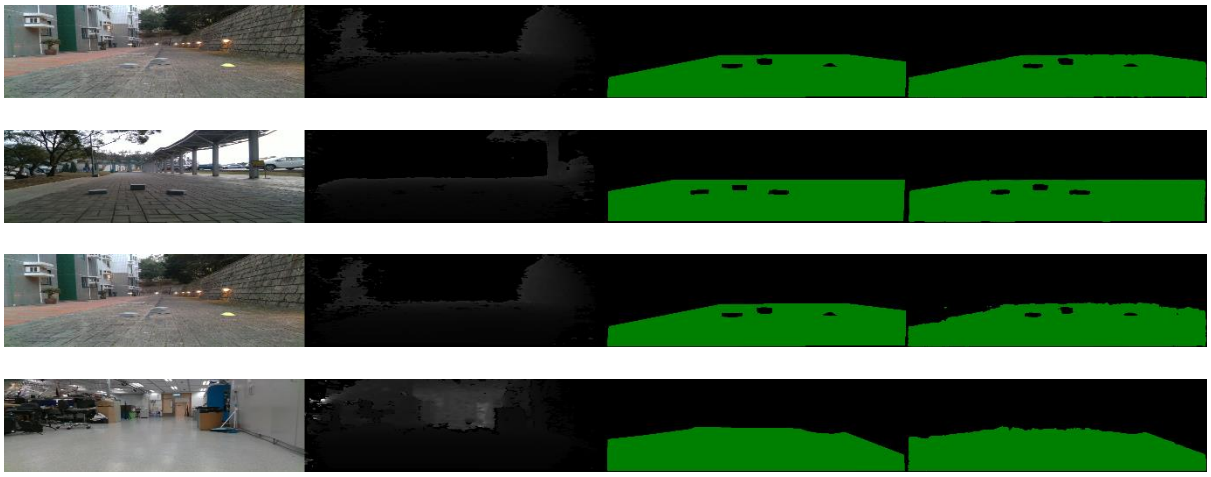
Epoch 100:



Epoch 200:



Epoch 300:



Epoch 400:

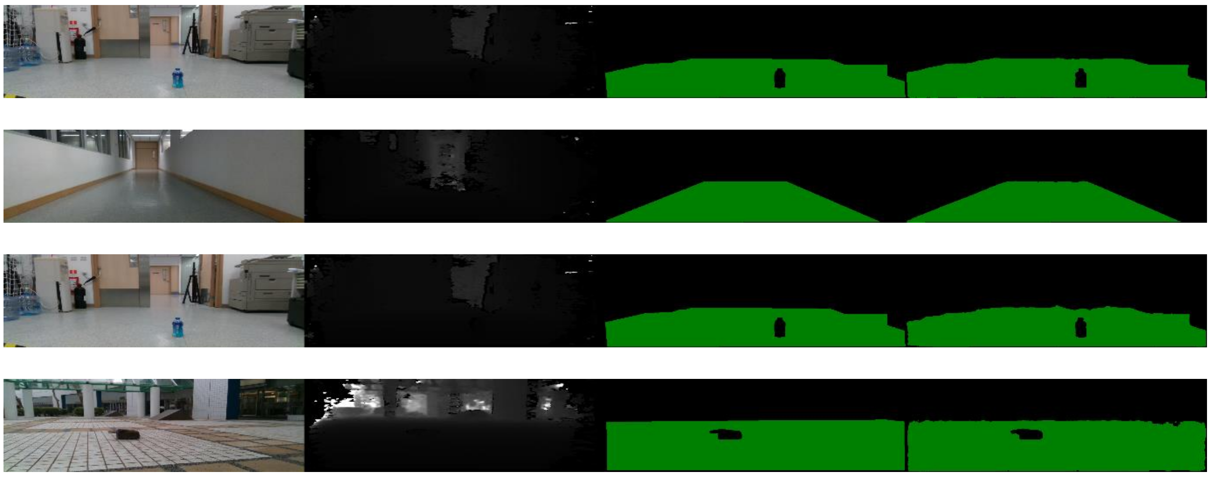


Figure 4-8. The results in the order of 1) batch size = 2, ResNet 34; 2) batch size = 2, ResNet 18; 3) batch size = 8, ResNet 18; 4) batch size = 12, ResNet 18

The test results for the model obtained after 10, 50, 100, 200, 300 and 400 epochs of training were shown in Figure 4-8. It was obvious that the model trained with smaller batch size could quickly give results with less noise. Therefore, it could be concluded that small batch size in training could accelerate the convergence speed. For this project, the model trained with batch size = 2 was finally used.

4.2.2 Deploying Phase

To obtain optimal performance during inference, the trained PyTorch model was converted to TensorRT inference engine. The performance of the original PyTorch models and the TensorRT ones will be compared in prediction accuracy on test image and inference speed.

According to Table 4, the inference speed of the same model could be accelerated by **up to 3.7 times**. Using lower model precision could achieve higher inference speed. By comparing the inference speed difference between 18-layers ResNet and 34 layers ResNet, it could be found that the inference speed was reduced by using more ResNet layers. It was interesting that the lower the model precision was used, the less frame rate was lost. For the un-accelerated model, the frame rate was about 19% lower when using more complex model with higher number of layers, while the frame rate only decreased for about 5.4% for highly optimized model, which used int8 precision. Referring to the results shown in 4.2.1, the accuracy of the model with 18-layers ResNet was only 0.09% percent lower than that with 34-layers ResNet. In this project, the real-time performance should be guaranteed at the first place, because the robot needed to respond to the change in the environment as fast as possible. Therefore, the model with 18-layers ResNet was finally used.

Table 4. The comparison of the running speed

	Original	FP32	FP16	INT8	Acceleration
18-layers ResNet	2.11	2.54	5.21	6.70	120.38%/246.92%/317.54%
34-layers ResNet	1.71	2.12	4.85	6.34	123.98%/283.63%/370.76%
Frame rate loss	18.96%	16.54%	6.91%	5.37%	

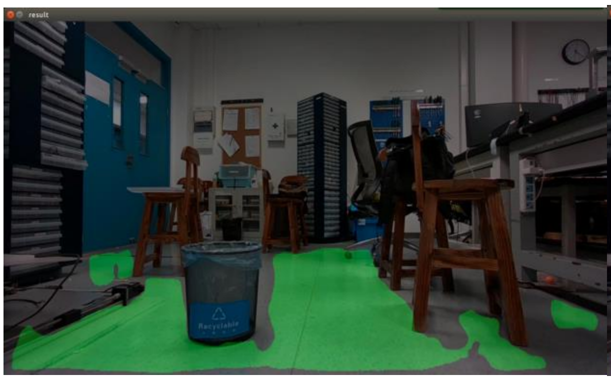


Figure 4-9. Output from the original model

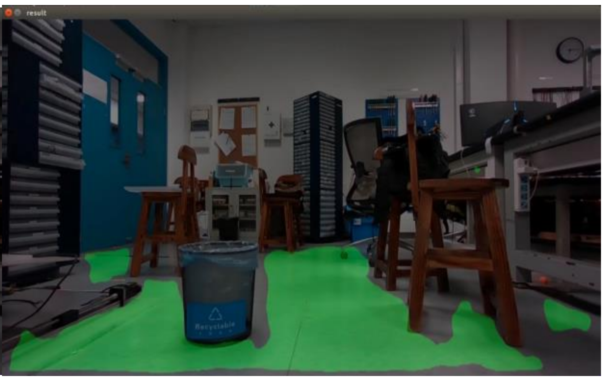


Figure 4-10. Output from the FP32 model

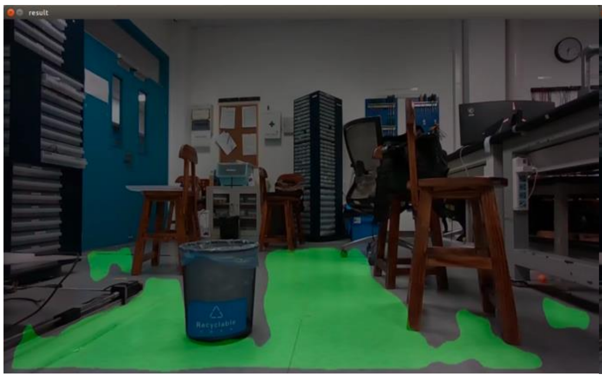


Figure 4-11. Output from the FP16 model

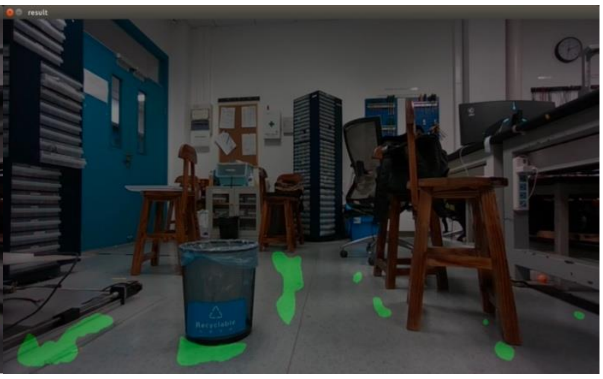


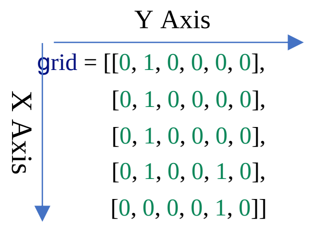
Figure 4-12. Output from the INT8 model

As for the accuracy of the TensorRT optimized model, it could be observed in Figure 4-9 to 4-11 that the output of the FP32 model and the FP16 model kept consistent with the output of the original PyTorch model. However, Figure 4-12 shown that the INT8 model apparently could not generate the correct mask for collision-free space. It seemed that the quantization process (mapping the parameters in the model from FP32 to INT8) was not correctly executed by TensorRT due to the lack of memory. However, the exact problem that led to this error needed further investigation. In this case, for this project, the FP16 model was used as it could run about 2.5 times faster than the original PyTorch while provide comparable accuracy.

4.3 Obstacle Avoidance and Global Path Planning

4.3.1 A-star path planning test

A 5×6 grid map for testing the algorithm. The positions filled with zeros were free paths while the positions with ones represented obstacles.

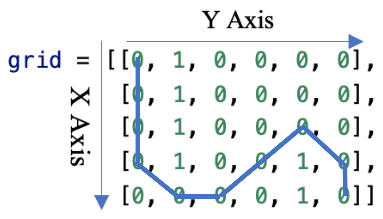


```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

Set the start point [0, 0] and the target [4, 5]. The path was generated by the algorithm as

```
[0, 0]
[1, 0]
[2, 0]
[3, 0]
[4, 1]
[4, 2]
[3, 3]
[2, 4]
[3, 5]
[4, 5]
```

The path could be visualized in the map as

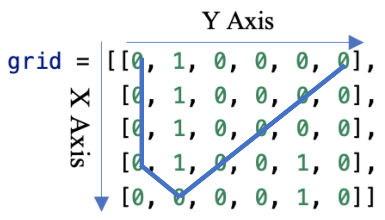


```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

Set the start point [0, 0] and the target [0, 5]. The path was generated by the algorithm as

```
[0, 0]
[1, 0]
[2, 0]
[3, 0]
[4, 1]
[3, 2]
[2, 3]
[1, 4]
[0, 5]
```

The path could be visualized in the map as



```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

The algorithm could successfully generate the correct path from the start point to the target.

4.3.2 Obstacle Avoidance

The results of the steering for obstacle avoidance were shown in Figure 4-13 to 4-16. It could be observed that the neural network could correctly generate the mask that indicated the area without obstacles. The steering indicator on the right top of the image shown that the algorithm could give a reasonable steering decision based on the visual information.

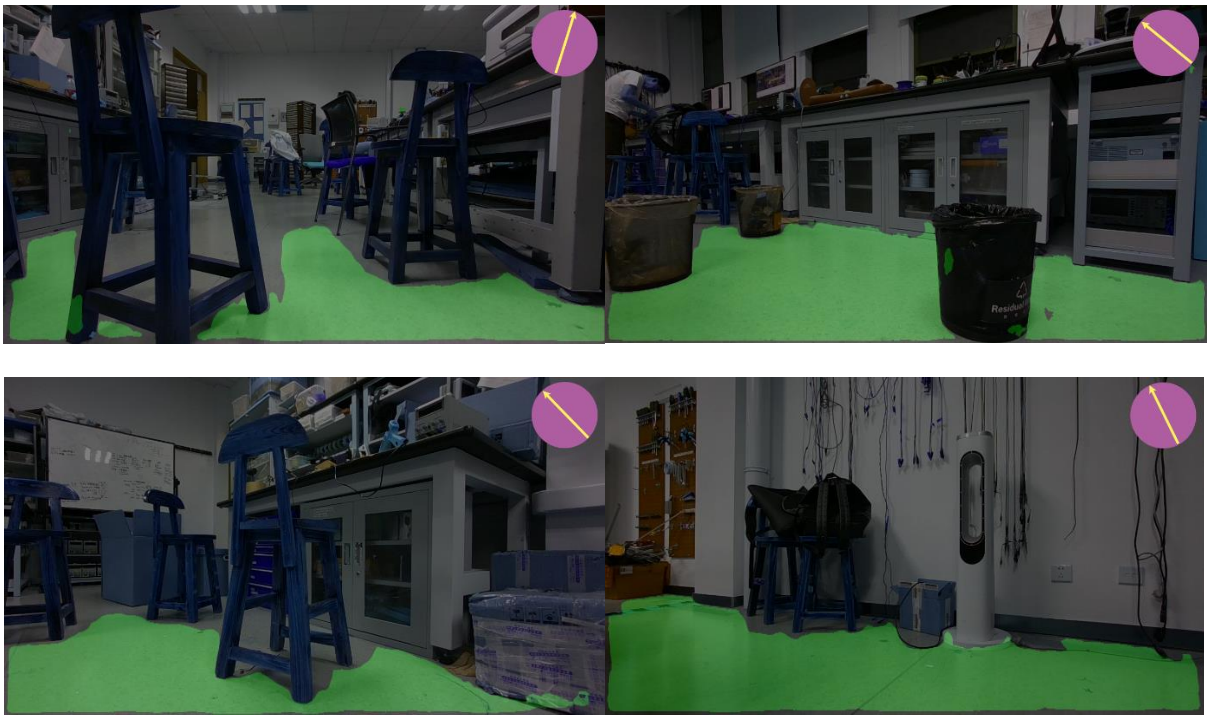


Figure 4-13 to 4-16. The direction inferred from the segmentation result

4.3.3 The Combination of the Obstacle Avoidance and Navigation

After testing the obstacle avoidance system and the global path planning system individually, the performance of the system that combined with local path planning and global path planning was tested.

During the testing, it was observed that the system could successfully navigate itself to the pre-set target in the grid map while perform obstacle avoidance. However, if the vehicle was trapped in some corners, it would stop moving. Moreover, the fusion of the local path planning and global path planning was not robust. For example, if the target was on the left side and meanwhile there were obstacles on the left side as well, the steering decision given by two system will conflict to each other. Therefore, a more robust fusion algorithm had to be developed to improve the robustness of the system.

Chapter 5 Conclusion

5.1 What has been done?

In this project, an autonomous robotic vehicle that has the ability to navigate itself to the target position without running into obstacles in the indoor environment was prototyped. The system was powered by a 24V 5Ah Li-ion battery and was controlled by Nvidia Jetson Xavier NX board running ROS. UWB modules were used for indoor localization and an RGB-D camera was mainly used for robot perception. In the aspect of obstacle avoidance, a deep neural network for detecting the collision-free space was modified, trained, and deployed onto the Jetson board with TensorRT acceleration. The robot could detect the traversable area in real-time (5.2fps) using the deep neural network with the RGB as well as depth input from the RGB-D camera and control the steering to bypass the obstacles. Three ultrasonic sensors were installed towards left, front and right of the vehicle as a backup of the visual obstacle avoidance system. For localization and navigation, A-star algorithm was used to dynamically generate a path from the current position of the robot to the assigned target in a pre-defined grid map.

5.2 What are the existing problems and what could be improved?

The obstacle avoidance control policy based on the segmentation result was simple and not robust, as it will just go towards the direction with more empty space. This strategy could only find the local optimum, as the direction with less obstacle was not necessary the best direction in the global view. Moreover, when the vehicle went into a blind alley, it was not able to turn around to find a path. Therefore, it was necessary to propose a more robust path planning algorithm to find an optimal path. It is possible to use SLAM to build a global map before navigation or use advanced method like reinforcement learning to control the robot in a more intelligent way.

Moreover, at this stage, although the vehicle could communicate with an iPhone through Bluetooth remotely, the realized function was just simple string transmission without any GUI. Therefore, the project could also be improved by building an application for vehicle status visualization and a user-friendly interface to control the vehicle remotely.

References

[1] Bleesk.com. 2021. What is Ultra-Wideband (UWB)? Here's everything you need to know | Bleesk. [online] Available at: <https://bleesk.com/uwb.html> [Accessed 8 November 2021].

[2] earian, L., 2021. ltra Wideband (W) explained (and w y it's in t e iP one 11). [online] Computerworld. Available at: <https://www.computerworld.com/article/3490037/ultra-wideband-explained-and- why-its-in-the-iphone-11.html> [Accessed 8 November 2021].

[3] Pozyx.io. 2021. Pozyx Academy | Positioning protocols explained. [online] Available at: <https://www.pozyx.io/pozyx-academy/positioning-protocols-explained> [Accessed 8 November 2021].

[4]P. Flach, Machine learning. Cambridge: Cambridge University Press, 2017, p. 3.

[5]"ML Visuals by dair.ai", Google Docs, 2021. [Online]. Available: https://docs.google.com/presentation/d/11mR1nkIR9fbHegFkcFq8z9oDQ5sjv8E3JJp 1LfLGKuk/edit#slide=id.g78327f1586_217_169. [Accessed: 06- May- 2021].

[6] Siu-Yeung Cho, Learning Machine. p. 34.

[7]Francois Chollet. and Zhang liang, *Deep Learning with Python*. Beijing: POSTS & TELECOM PRESS, 2018, pp. 97, 101, 102.

[8]R. Fan, H. Wang, P. Cai, and M. Liu, 'SNE-RoadSeg: Incorporating Surface Normal Information into Semantic Segmentation for Accurate Freespace Detection', p. 17.

[9]"NVIDIA TensorRT", NVIDIA Developer, 2022. [Online]. Available: https://developer.nvidia.com/tensorrt#performance. [Accessed: 04- May- 2022].

[10]A. Patel, "Introduction to A*," *Red Blob Games*, 01-Jan-1970. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html#astar. [Accessed: 04-May-2022].

[1] . attalo, " ntroduction to ROS," ros.org, 2018. [Online]. vailable: http://wiki.ros.org/ROS/Introduction. [Accessed: 07-Nov-2021].