CHAPTER 3
RESEARCH METHODOLOGY

Some steps that must be done in this project are :

1. Prepare for the Jetson TX2

   First of all, We flash Jetson TX2 with ubuntu as an operating system. NVIDIA has provided a package for the libraries called jetpack, in this project we used version 3.3, we only need CUDA, C compiler, TensorRT and the opencv, and after that we need to upgrade the opencv to be opencv version 3.4.6 .

2. Prepare for training the object detections

   1. We took 2200 photos whose images include single and multiple car objects images at different angles, such that YOLO can detect multi cars in one image. Examples of photos taken at different angle are shown below.

   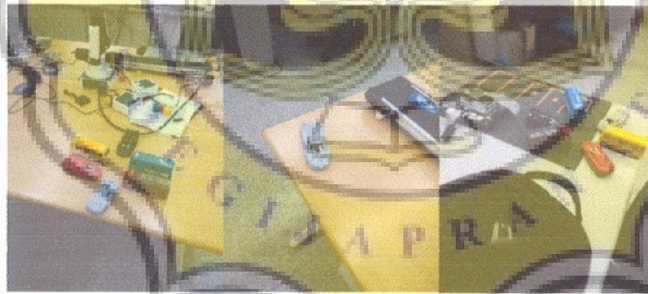

   Figure 1. Sample Photos Used for Training

   2. We convert those high resolution images to low resolution in order to make it easier for training in YOLO. Even though YOLO is trained with low resolution images, object detection is still accurate

   3. We assume that images on the left and right of Figure 2 are named as 1.jpg and 2.jpg respectively, then we label each image using Bbox labeling annotation tool to obtain the coordinate (x, y, w, h) where x and y are the

left top of coordinate of car object, and w and h are width and height of coordinate bounding box respectively and the results of the labeling are shown in Figure 3.

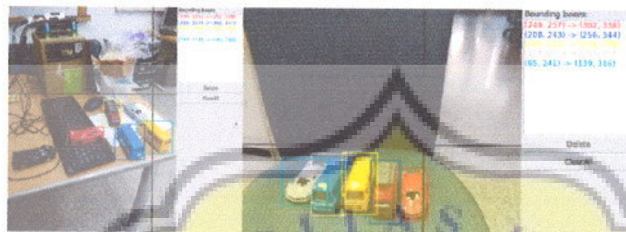

Figure 2. Labeling Objects and Specifying Coordinates with Bounding Boxes

```
1
161 373 291 467
```

```
5
310 355 371 450
257 331 316 448
213 316 279 418
157 319 232 421
121 312 198 391
```

Figure 3. Result of Labeling Objects

4. The result of Figure 3 is stored as 1.txt and 2.txt associated with its input 1.jpg and 2.jpg respectively. The number on the left represents one image with a coordinate and the number on the right represents 5 car objects with their coordinates. Figure 4 shows the normalized coordinate of x/Width, y/Height, w/Width, h/Height, where Width and Height are the width and height of the whole image. The number in the first column is the class number, i.e. 0 is the class of car and 1 is the class of human, and etc. The next four entries are all floating points in which values are from 0 and 1. The results of the normalized coordinates are saved in the same file.

```
0 0.222 0.6758 0.13333333332 0.85
```

```
0 0.7772         0.78          0.767676       0.895
0 0.127333333    0.8966666     0.459          0.233334
0 0.2323         0.13333334    0.595747       0.4445
0 0.676333222    0.456987      0.133333333    0.86757
0 0.77332        0.83656557    0.5            0.7777777775
```

Figure 4. Result of Normalized From the Coordinate.

5. split the images dataset randomly by 90% and 10% into train.txt and test.txt respectively. The car.data file is used to define the number of classes, the location of train.txt and test.txt and the output file of training result; whereas the car.names file is used to define the name of objects to be trained

6. Download YOLO pre-trained model file and modify the configuration file for GPU VRAM requirement, batch size, filters and class to meet our requirements.

7. The training is run on TX2 for about 9 hours when the average loss score close to zero and no longer decreases. Final weight file can be obtained after the training.

3. Detect the motion

Because we want to set the object detection will on work if there is an object motion of the current image, by subtracting every pixel value of the current image from the previous image, using the equation below:

$$\text{motion} = \sum_{c=0}^{2} \sum_{i=x}^{width} \sum_{j=y}^{height} \text{img}[c \cdot \text{width} \cdot \text{height} + i \cdot \text{width} + j]$$

There is a three-level nested loop. The top-level is iterated for 3 channels and the second level is iterated for horizontal traversal and third-level traverses horizontally from y to the height of the image. This loop is parallelized by pragma omp parallel for directive with a reduction clause.

```
int calculate_motion(image old_img, image cur_img, boundary subimg ){
#pragma omp parallel for reduction(+:score)
for (int c=0;c<3;c++)
 for (int i=subimg.x;i<subimg.width;i++)
   for (int j=subimg.y;j<subimg.height;j++)
score+=old_img.data[c*img.width*img.height+i*img.width+j]
- cur_img.data[c*img.width*img.height+i*img.width+j];
return(score);
}
```

Code 1: Simple Motion Detection Function Using OpenMP

4. Create the parking availability detection

   The parking slots are manually coordinated, or it can be detected with OpenCV line detection algorithm. It depends on the number of cars detected and the total number slots, for every car detected, if the bounding box of a detected car is in the area of parking slot, then it is occupied and it draws with a solid red line, else the slot is available and number of empty slot is incremented, and it draws with a dash green line as shown in Figure 5.

```
void availability(struct detected car){
   occupied = 0;
   available = 0;
   for(int i=0;i<total_car;i++){
      for(int j=0;j<slot;j++){
         if (car[i].left>slot[i].left && car[i].top>slot[i].top &&
            car[i].right< slot[i].right && car[i].bot< slot[i].bot) {
              draw_box_width(img, 900,100,1000,200,5,255,0,0);
              occupied++;
         }
         else available++;
}
```

Code 2. Parking Availability Detection Algorithm

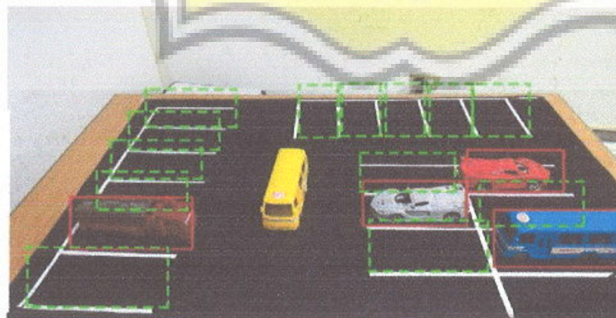

Figure 5. Result of the Parking Availability Detection