Semantic segmentation of static and dynamic obstacles

Submitted in fulfillment of the conditions for the award of the degree

B. Eng Electrical Engineering and Information Technology.

Nguyen, Trong Khiem

1235078

Supervised by Prof. Peter Nauth and M.Eng. Sudeep Sharan

Faculty of Engineering

Frankfurt University of Applied Sciences

Date 14.10.2020

Statement I hereby declare that this thesis is all my own work, except as indicated or cited in the text. Signature:

Abstract

Semantic segmentation is an important topic in the Machine Learning and Computer Vision fields. Comparing to other visual scene understanding methods like object classification and object detection, semantic segmentation is the method that presents the most information about the object since it not only produces the classification and localization of the object on the image level, but also on the pixel level, thus depicting precisely the shape and the position of the object in space (when combined with point clouds data). Therefore, semantic segmentation technique is used in many real-world applications, notably robots visual localizing, navigation and self-driving automobiles. This thesis is carried out to verify the effectiveness of semantic segmentation when used as an aid to improve the performance of indoor navigation task. The semantic segmentation model is operated as part of the Roswitha robot's operating system. In order to make the output map of semantic segmentation meaningful, as well as to improve the accuracy of the model, points cloud data extracted from Intel RealSense depth camera is adopted. This thesis also presents an efficient model which fuses the data originated from RGB stream and depth stream to improve the speed and accuracy when compared to other algorithm. A literature overview is also included in thesis to discuss about recent RGB-D image segmentation as well as how Deep Learning techniques can be used to tackle the semantic segmentation task. Due to the lack of local hardware, this thesis proposes a pipeline to train and deploy in real-time the semantic segmentation algorithm on cloud services. As part of the thesis, a new dataset is collected and labelled to fit the model on the settings of the robotics labs at the Frankfurt University of Applied Sciences. Also, a baseline model, which was trained on additional massive synthetic indoor environment dataset of more than 500.000 images, can be used to shorten the training time and increase the accuracy of the model when trained on a different dataset in the future. The result of the thesis has verified the effectiveness of combining semantic segmentation and indoor robot navigation, which opens up many promising future development rooms for the project

Acknowledgements

In the process of writing this thesis I have received a great deal of support and assistance. I would first like to thank my supervisors, Prof. Peter Nauth and M.Eng. Sudeep Sharan, whose expertise was invaluable in the formulating of the research topic and methodology in particular. I am also grateful to have received enormous support from M.Sc. Julian Umansky on the hardware aspects.

Contents

Table of figures

1. Introduction

In recent years, robotics has emerged as a fast-developing field in both industrial and research contexts. At the same time, due to advancements in hardware accelerators such as Graphical Processing Units (GPU) and Tensor Processing Units (TPU), the Computer Vision (CV), Machine Learning (ML), Artificial Intelligence (AI) and notably Deep Learning (DL) fields have achieved many remarkable breakthroughs. Recognizing the potential, researchers recently have tried to corporate components of Deep Learning and Computer Vision field in order to enhance the abilities of robots, namely the ability the see, feel and understand the real world like a human, through its sensors, actuators and cameras. Self-driving automobiles companies such as Tesla and NVIDIA are some notable examples of the industry fields trying to make their machines more intelligent by powering them with AI to make them fully discernable about their surrounding objects, as well as to navigate and localize themselves in new environments.

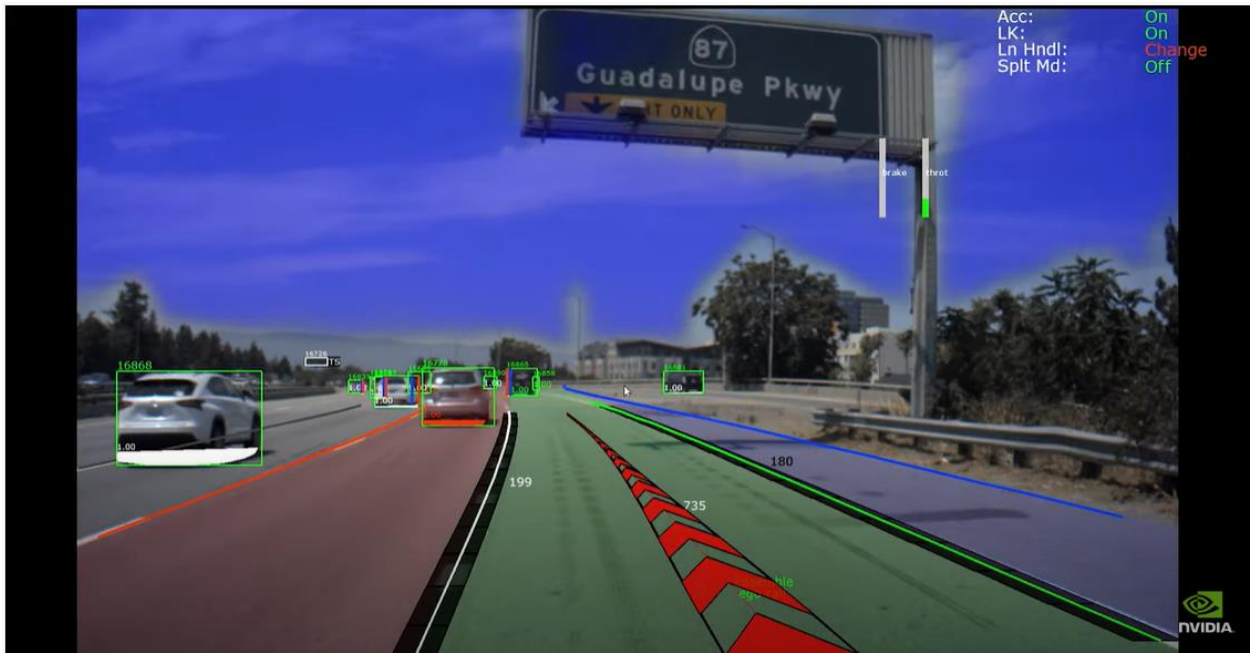


Figure 1: Deep Learning application in Self-Driving car [1]

1.1. Motivation

Indoor navigation is one of the field that has lots of rooms for improvements. Unlike outdoor navigation task, indoor navigation requires the robot to fully and precisely understand the complex surrounding environment with many different types of obstacles and navigating paths. A conventional approach is to use Laser scanner or Ultrasonic sensor in order to detect obstacles around the robots, from which a suitable navigation path can be calculated to avoid such objects. However, it is well known that these devices have a serious limitation that it cannot detect all types of obstacles (e.g. laser scanner can only detect 4 legs of a table but not the whole surface of the table), as well as difficult navigating paths (e.g. laser scanner cannot detect that there is staircase in front of it). Not only such devices are limited with the scanning ability, they are also unable tell the classes of the objects. This will be a huge drawback for complex tasks that require flexibility such as path planning when encountering different types of obstacles, which is common in outdoor navigation (e.g. a car will stop when detect a human passing by but continue running and moving into a different lane when detecting a car in front of it). To tackle such issue, rich visual and space information retrieved from RGB-D camera can be used to make the robots fully aware of the indoor spaces and obstacles. From these data, the robots can react differently when encountering static obstacles versus dynamic obstacles. Semantic segmentation is the perfect solution to serve this task, since it can:

- Detect exactly the shape of the objects
- Detect exactly the classes of the pixels in the image.
- Since each pixel also contains the coordinate of a point in space (due to the usage of Depth camera), the robots can build a costmap on top of the provided dense point clouds.

1.2. Aims and Objectives

In order to aid the robots in terms of discerning the objects surrounding it, the following criteria has to be set for the developed Deep Learning indoor image semantic segmentation model:

- **Accuracy**: The model has to be able detect correctly the classes with accuracy metrics higher than 70% on the test dataset (this is an acceptable goal considering the lack of available dataset).
- **Latency**: The model has to be able to run in real-time or at least the same speed of the path planning algorithm so that the coordinates returned by the segmentation model match with the current position of the robots with regard to surrounding environment. A low processing speed will make the data returned by the semantic segmentation model lag behind the data stream from Lidar, making the fusion of both data source to create the local costmap become inaccurate.
- **Complexity**: The model has to be small enough to fit into the VRAM of the GPU/TPU during the training as well as the inference stage. Also, the model should not be too large in number of parameters to avoid the overfitting phenomenon, given that the available training data is quite small.
- **Robustness**: The model should be robust to different lightning condition as well as different obstacles settings and motions. Due to the nature of semantic segmentation, unlike object detection, the model should also be able to detect parts of the occluded objects (e.g. human body parts) without having to see the fully visible object.

1.3. Thesis structure

In this thesis, a literature review regarding recent breakthroughs in RGB-D semantic semantic segmentation is discussed in Chapter 2. Then, a theoretical background on the general working mechanism of Deep Learning methods and how Deep Learning is applied in semantic segmentation field are briefed on Chapter 3. Afterwards, a new semantic segmentation model architecture is proposed to fully leverage on the data extracted from RGB-D camera, as well as an experiment is carried out to validate the efficiency of our model over other models on Chapter 4. In Chapter 5, the training and deployment pipeline as well as the data collection process in explained in details. In Chapter 6, a thorough evaluation scheme is executed to verify the result of the semantic segmentation model on the indoor environment settings as well as how the model can help the robot navigate and plan the path better when encountering static and dynamic obstacles. In Chapter 7, the future development plan is proposed to improve the whole system.

2. Related Works

Merging visual data stream and depth data stream to improve the accuracy of the RGB-D image semantic segmentation algorithm is not a new approach in the research field. Also since depth image are invariant to lightning condition, making use of depth image will help the model detect objects in extreme and variant lightning environment.

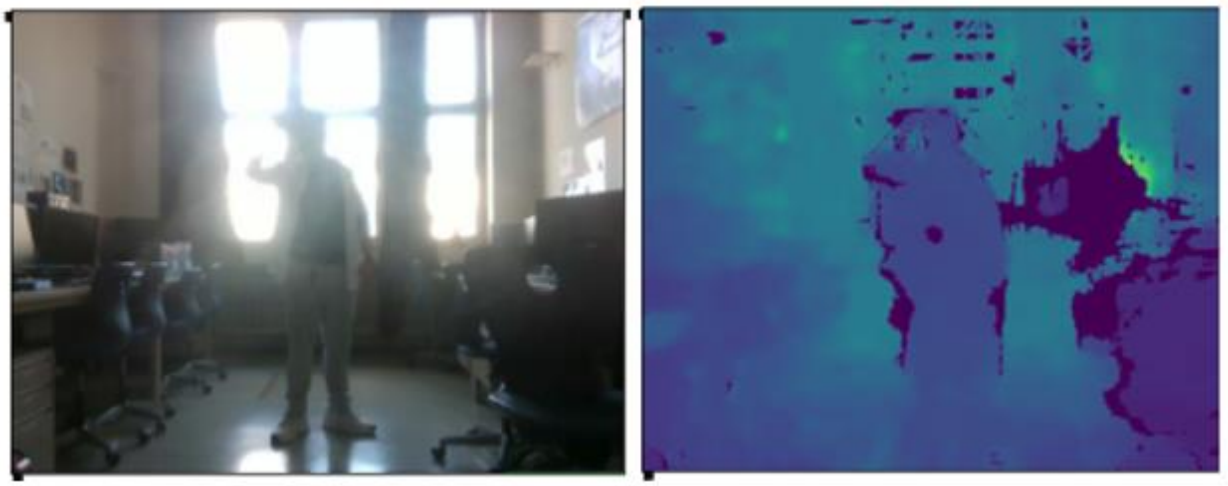


Figure 2: Example of RGB image and its corresponding depth image

In overall, there are 2 main researching branches for this task:

- Making use of basic features of the image such as contour, color homogeneity … to divide images into smaller clusters of superpixels regions, from which regional higher level features can be extracted via handcrafted filtering operations before being fed into a traditional machine learning classifier such as Support Vector Machine to classify the class of each regions. Afterwards, a further post-processing algorithm such as Conditional Random Field (CRF) is run over the image to refine the output of the classification stage:

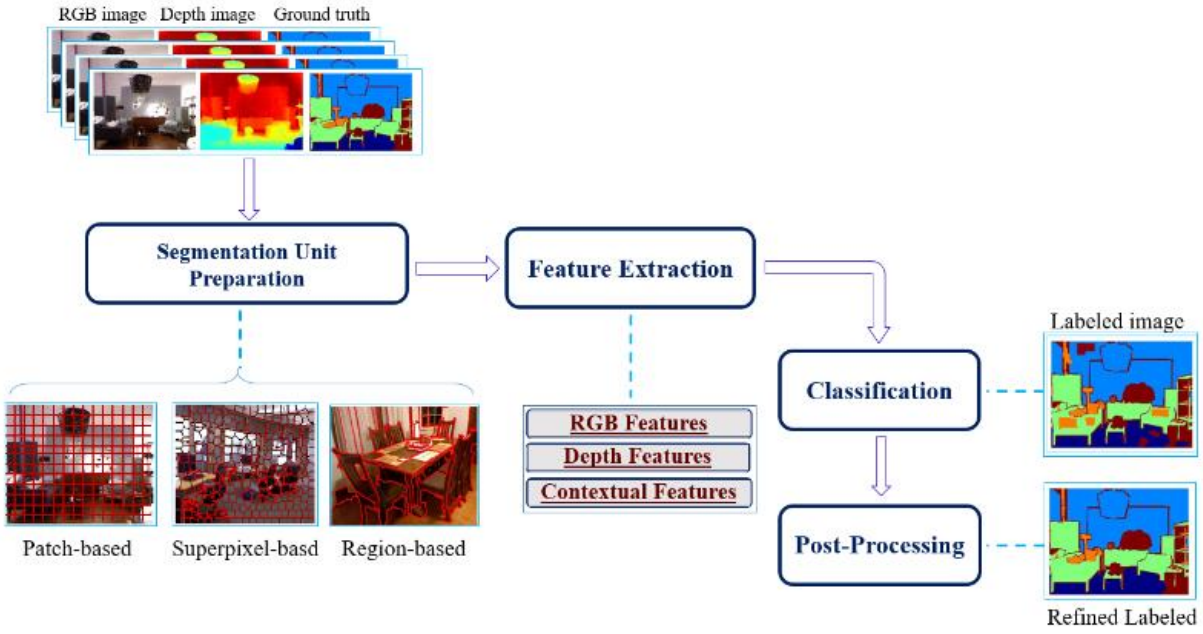
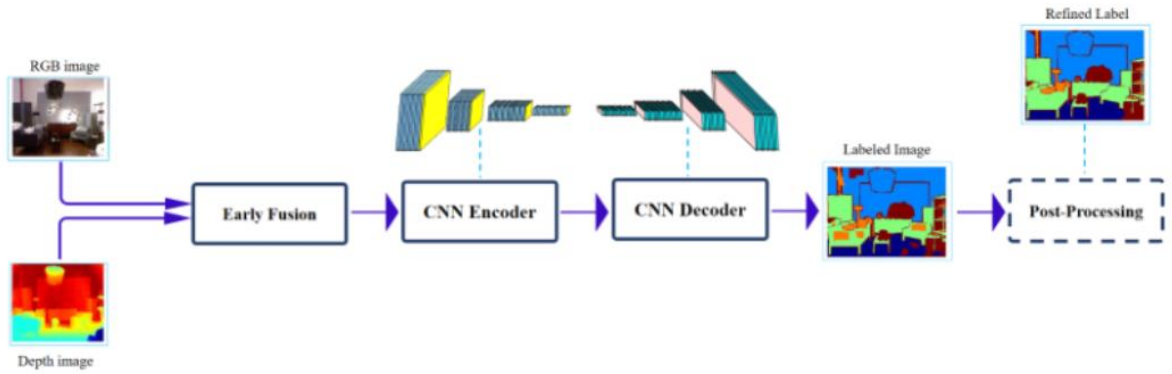


Figure 3: Traditional RGB-D image segmentation [2]

- Making use of new Deep Learning techniques to produce the segmentation map directly without having to iterate through all the segmenting, feature extracting, classification and refining steps in the former method. Such Deep Learning methods normally utilize the powerful learning capability of Convolutional Neural Network. In addition, the data from RGB camera and Depth camera are usually extracted in different streams (encoder branches) before being fused with each other later:



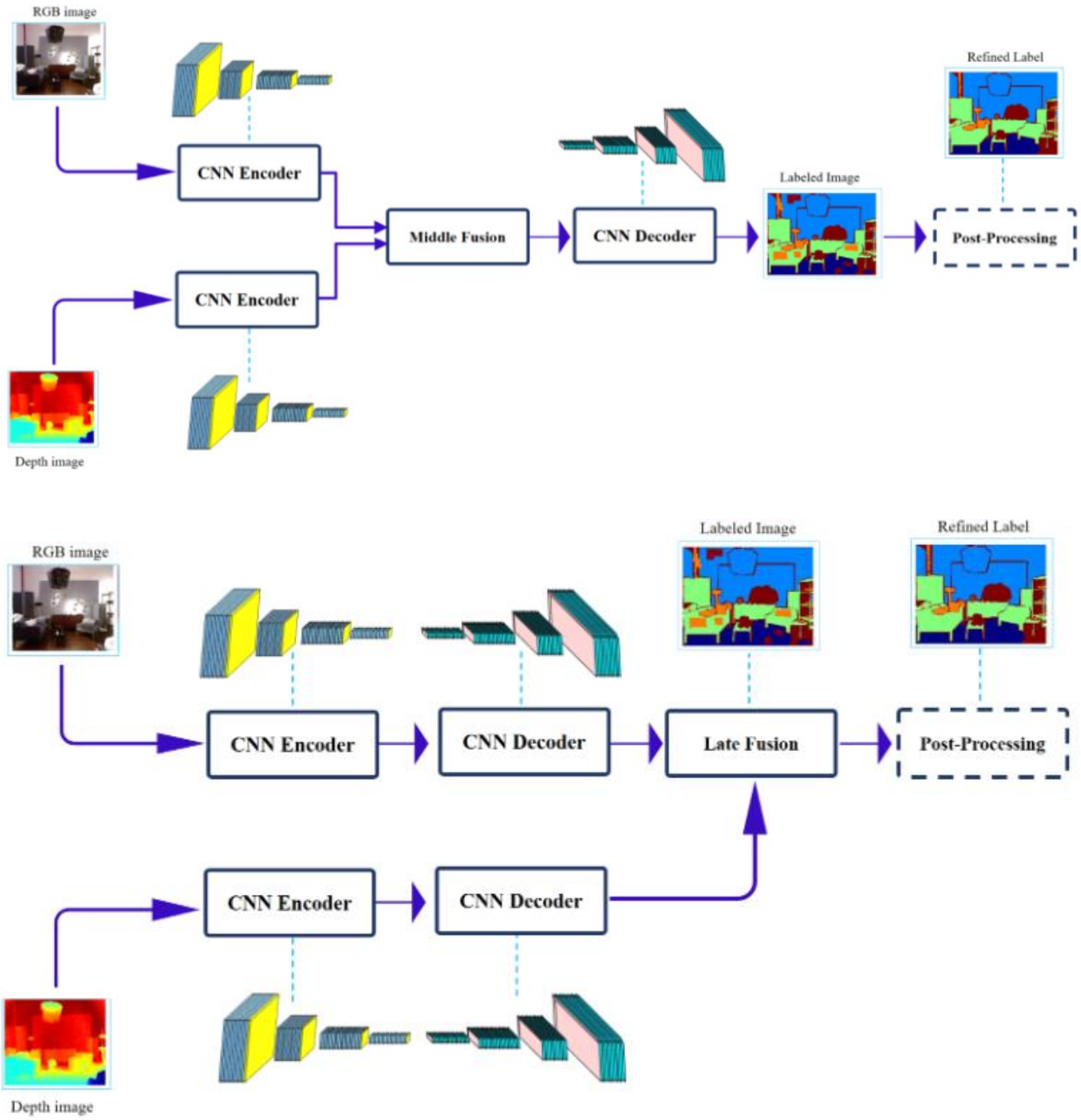


Figure 4: Deep Learning model with RGB and Depth data fusion during early stage, middle stage and later stage respectively [2]

In general, the former methods are more complicated to reliably implement in real-world or industrial scenarios due to the involvement of multiple separate stages, as well as the fact the filtering operations during the feauture extraction stage are mostly handcrafted, thus leading to

the liability of not being able to capture all the data presented in the image and not being robust to different environment scenarios. Also, the classifier in such methods are also very naïve and simple. Last but not least, by dividing the image into multiple different regions, the algorithms cannot learn the interconnected features between different adjacent regions (e.g. a table is normally lies above a ground or a human head is normally connected to the human body). On the contrary, Deep Learning methods execute most of the aforementioned stages in one-shot fashion, leading to higher flexibility and feasibility to be implemented in reality. In addition, Convolution Neural Network tends to outperform traditional Computer Vision methods due to its ability of generalizing of different types of images and capturing low-level and high-level (features from faraway regions on the images) precisely and automatically. Therefore, Deep Learning methods are becoming the main researching directions in the field nowadays. Some notable works that representing this trend are FuseNet [3], MMAF-Net [4] or ACNet [5].

However, Deep Learning approaches still contain some drawbacks that should be addressed:

- Most Deep Learning model for RGB-D segmentation has huge and complicated architecture with more than 100 million parameters. This makes running such model in real-time, in production settings and on mobile or embedded device virtually impossible [4].
- Due to large size, these models are also prone to being overfitted when trained on small datasets, which would be proved later in this report.
- Also due to large size, it takes more time for these model to train and update their parameters as well as requires GPU with high RAM to fit the whole model. Therefore, such model usually has to resort to the Transfer Learning technique, which will be discussed in section 5. In general, Transfer Learning is a technique that requires the model to be trained on other huge datasets before it can be trained on the desired dataset.
- Some DL models have tried to leverage the depth data into the model, but their approaches are not optimal.

In order to solve such issues, we will present a new and more efficient architecture, in which:

- The depth data and RGB data will be incorporated wisely.

- The model backbone architecture is extremely light-weight, easy-to-train and easy-to-deploy. The model is capable of running in real-time on a medium GPU.
- The input to our model is simply depth map and requires no conversion to HHA encoding like many other models.
- Even though our model has a small number of parameters, and can be considered extremely small compared to other RGB-D segmentation mode, it can achieve an acceptable accuracy, even outperforms many bigger Deep Learning model.
- Due to its small size, our model can easily be fitted on small datasets, which would be suitable for small-scale real-world usage where collecting huge dataset up to hundred thousands of images are not feasible.

3. Theoretical Background

This chapter demonstrates the theoretical background of Deep Learning for Computer Vision in general, as well as Semantic Segmentation algorithm.

3.1. Deep Learning in Computer Vision

Deep learning is named after its ability to learn deep and complex representation of data via combinations of many linear and non-linear filtering operations and mapping. The following illustration demonstrates the diversity of features that a deep learning network can extract from the original data.

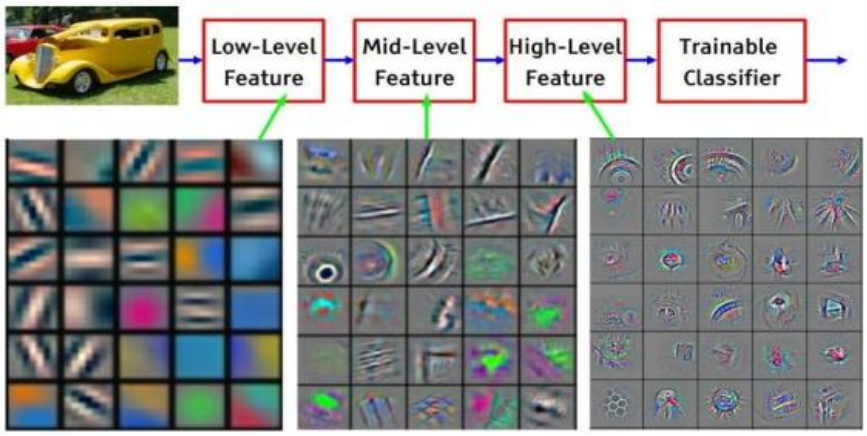


Figure 5:Extracted features visualization

Small details such as edges and colors of the images tend to be captured in early stage of the network, while more complex and broader features are learnt in later stages. In general, there are 4 components that must be included when constructing and training a Deep Learning model for Image processing:

- The convolution layers and non-linear mapping function that allows the model to learn complex

representations. These layers are usually called as the encoders of the model, since they encode the most important information before passing them to the neural network/decoder to produce the predictions. Such models that make use of convolution layers and neural networks are usually called as Convolutional Neural Network (CNN).

- An objective/loss function to let the model know how well it is learning, as well as to guide the model to learn in such a way that minimize this function.
- A label or so-called ground truth (in case of supervised learning), so that the model can compare its output versus the true class of the data. Without the label, the loss function cannot be calculated.
- The gradient descent method and its variation to optimize the objective/loss function via the automated update of the convolution kernels, which helps the overall network extract the best features from the presented images.

The core component of a CNN is the convolution layer. The convolution layer of the network consists of a group of kernels with different parameters inside. This kernel will slide across the input image, compute the weighted sum to produce the new feature map. This feature map is the new feature (it could be edges, gradients …) that the model has been able to infer from the original data. The mathematical formula to describe the convolution is as follow:

$$O(u,v) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) I(u + (m - \frac{M-1}{2}), v + (n - \frac{N-1}{2}))$$

where:

O: *the output feature map*

f: *the learnable weights of each filter*

I: *the original input image*

Since convolution is a linear operation, while our data source is very complex and cannot be learnt correctly with mere linear operation, non-linear operation has to be introduced into the model. There comes the non-linear mapping functions like Rectified Linear Unit (ReLU) or Signmoid function:

$$ReLU(x) = x^+ = \max(0,x)$$

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

Also, one of the notorious problem when training deep learning model is the limitation of the hardware memory. This is normally caused due to the image resolution being too large. Therefore, researchers have came up with new idea to reduce the image resolution while maintaininig the most important features of the images. One of the solution is to insering the max-pooling layer multiple times in between the network to gradually reduce the resolution of the feature maps:

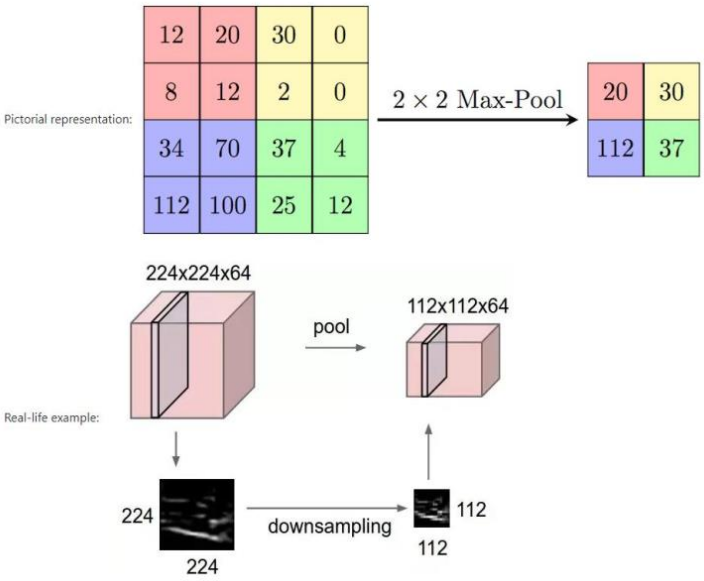


Figure 6: Max pooling operation

By consecutively applying the max-pooling and convolution layers in the CNN architecture, the model is capable of seeing a broader region of the image. Imagine when convoluting a 3x3 kernel on top of a 10x10 max-pooled input, the model is effectively seeing a 20x20 input since every 1 max-pooled pixel represent 4 original pixels.

Finally, a typical CNN architecture looks like this:

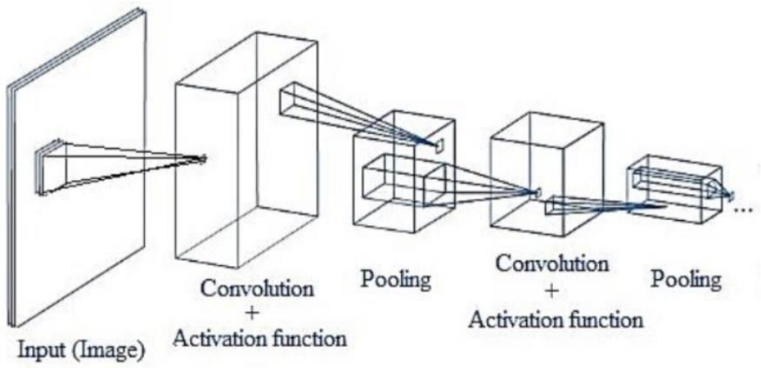


Figure 7: General architecture of CNN

In can be seen that as the CNN grows deeper, the depth of each block increases so that more features are learnt while the resolution of the feature maps decrease so that faraway features can be captured as well as due to limitation of GPU's memory.

After outputting the final feature maps, which represent the output classification probability for each class of the model, a loss function is applied to calculate the correctness between the prediction and the label. One of the popular loss function is *Multi-class categorical cross-entropy loss:*

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K}[y_{ij}\log(p_{ij})]$$

where:

n: *number of pixels*

K: *number of classes*

y: *class label*

p: *predicted probability of class j*

Normally, the class label will be encoded as 'one-hot' vector:

Class 0 -> [1,0,0]

Class 1 -> [0,1,0]

Class 2 -> [0,0,1]

This *Multi-class cross entropy* loss function will penalize the model heavily if it outputs a low confidence/probability for the correct class. The lower of the loss function, the better the model is performing.

In order to minimize the loss function, the parameters/weights of the model have to be updated accordingly. One way to do this is to calculate the derivative of the variables with respect to the loss function, from which we can update the parameters in the reverse direction of the derivative. This method is called Gradient Descent:

$$x_{t+1} = x_t - \eta f'(x)$$

$$where:$$

$$x_{t+1}: Variables' \ value \ at \ iteration \ t+1$$

$$x_t: Variables' \ value \ at \ iteration \ t$$

$$f'(x): Derivative \ of \ loss \ function \ f \ with \ respect \ to \ variable \ x$$

$$\eta: learning \ rate$$

Finally, in order to calculate the gradients of the loss function with respect to each weights of the kernel, chain rule theorem is applied to back propagate the gradient from the later layers to earlier layers:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Normally, the process of gradient calculation and parameter updating will be automatically calculated by the programming framework.

3.2. Semantic segmentation

Most of the semantic segmentation deep learning architecture share the following common points:

- The models are divided into 2 stages:
    - An *encoder* stage to encode the details of the original images as well as decrease the resolution of the images gradually to reduce computation cost and learn more complex features.
    - A *decoder* stage, which gradually restore the resolution of an image, as well as combine the features from the encoder stage to produce the prediction map. This stage is usually built with deconvolution layer such as *transposed convolution, bi-linear interpolation* or *unpooling operation*.

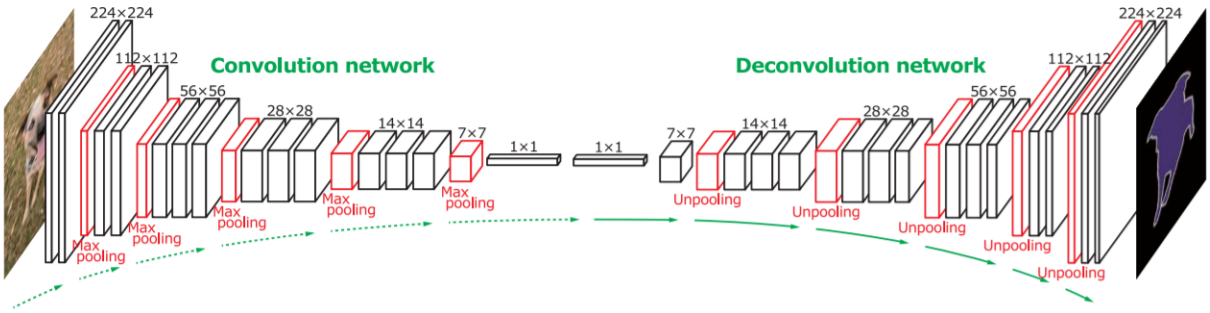


Figure 8: Semantic Segmentation model architecture

- Before generating the final probability map, a *Softmax* function is applied to the last convolution layers in the decoder to convert the output float values into probability ranging between 0 and 1.

$$f_i(\vec{a}) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

where index *i* indicated the considered class, and indexes *k* are all the classes of the dataset; a_i, a_k denotes the output value of the convolution layers (raw scores); $f_i(a)$ is the probability score.

After this operation, the output of the model would be of shape *(width,height,number_of_classes).* The class of each pixel will then be assigned to the one that has the highest probability.

- Some models forward the features learnt in the encoder directly to the decoder via shortcut connections (so-called *skip-connections*), besides sequentially via the stacking of convolution layers like in figure 9. These skip connections preserve the encoded information better as well as allow the model to grow deeper to learn more complex feature [6].

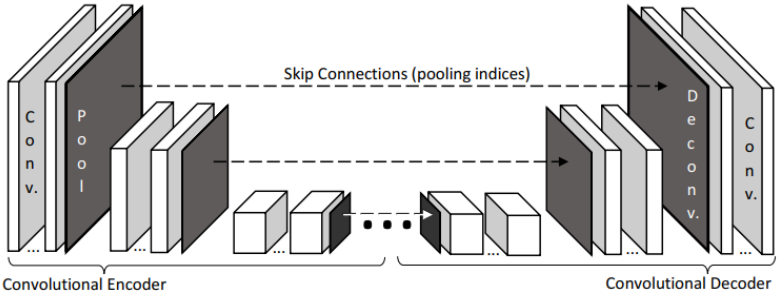


Figure 9: Illustration of skip connections

4. Explanation of RGB-D image segmentation model

Our backbone model that we use is Fully Connected Harmonic DenseNet [7] (FC-HarDNet), which is an upgraded version of the Fully Connected DenseNet model [8]. The original source code from the FC-HarDNet author can be found at https://github.com/PingoLH/FCHarDNet/blob/master/ptsemseg/models/hardnet.py. Since the original source code is written in Pytorch framework, which is used mainly in research but not in production, we rewrote the code back in Tensorflow framework to make it easier to deploy on TPU for training and optimized on GPU for inference. The reasons for us to choose FC-HarDNet are following:

- In terms of efficiency, FC-HarDNet ranks number 1 on the benchmarking outdoor dataset Cityscape with IoU score up to 75% and speed up to 75 FPS.
- Since we have our FRAUAS dataset is quite small (only ~400 images), we have to find a model containing small number of parameters to avoid the overfitting phenomenon. FC-HarDNet is a perfect option for us, as it contains a really small number of parameters, but outperforms many bigger and complex model on the Camvid outdoor benchmarking dataset [7].
- One of a rule of thumbs in Deep Learning is that if ones want their models to be more precise, they have to increase their models' number of parameters by making them grow deeper (more layers). Some good DL model on the world have up to 100 million parameters and approximately 400 layers. The deeper the model grows, the more complex and broader features the model can generalize on. However, the growth in depth is going to lead to the vanishing gradient phenomenon. Vanishing gradients happens after a couple of training iterations, when the gradient starts to become smaller and closer to zeros. As stated in the chain rule theorem, the gradient of the earlier layer is the multiplication of later layers' gradients. If all the later layers' gradient is small, then this ultimately leads to the gradient in the earlier layer becomes almost 0, meaning that the earlier layers no longer being updated:

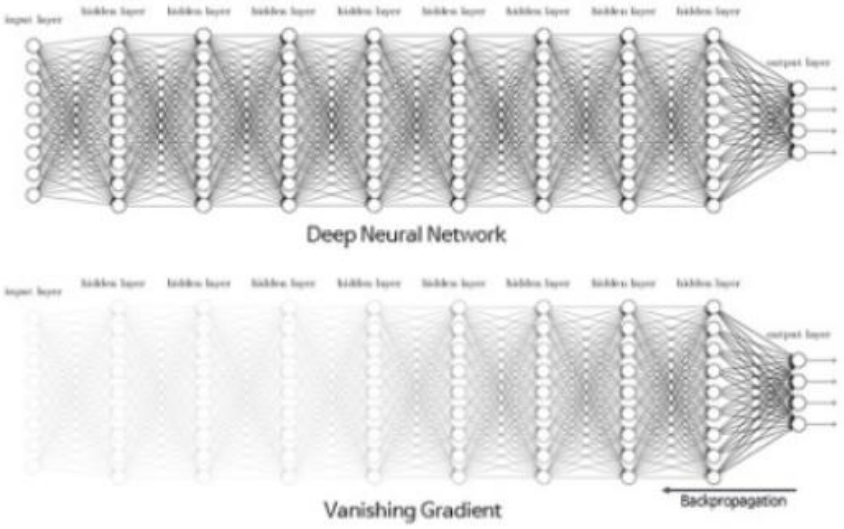
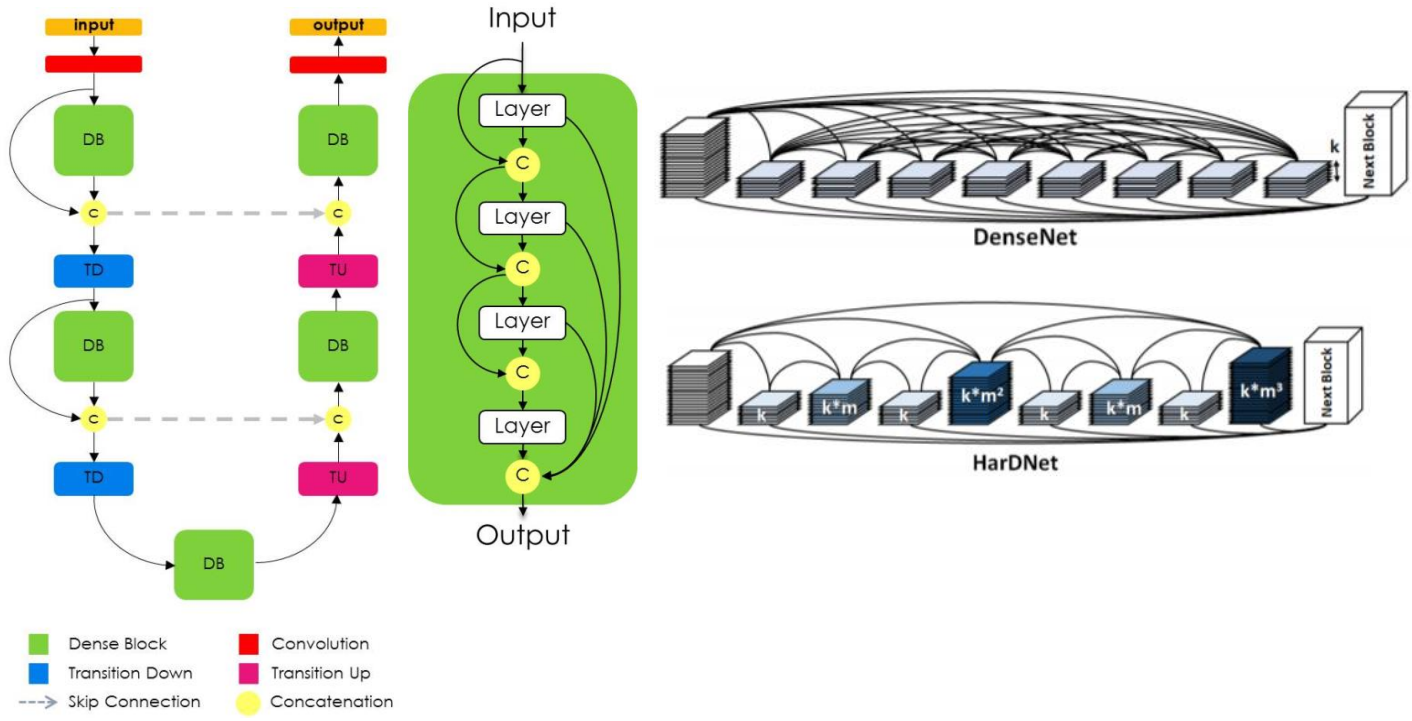


Figure 10: Illustration of Vanishing Gradients phenomenon

- FC-HarDNet contains 76 layers, which is a relatively deep model. To tackle the issue of vanishing gradient while maintaining the depth, FC-HarDNet employs the skip connection wisely (as explained in section 3.2) so that not only the model can grow deeper, but also it allows the model run at a higher speed than its predecessor FC-DenseNet as well as prevents the overfitting phenomenon. Skip connection is concatenation or elementwise-addition of layers, allowing for gradients of later layer to flow directly to earlier layers without getting attenuated.
- With the reordering of the skip connections resembling the harmonic waves, the authors of FC-HarDNet proved that such organization will lead to faster I/O operation on the DRAM, resulting in faster processing speed up to 36%. The author also injected a parameter k to control the depth of each convolution layer in the HarDBlock. Apart from the change of skip connection architecture in dense block, the rest of the FC-HarDNet still follows the original U-Shape of FC-DenseNet, with just some little modifications, which are shown and explained in the source code at ***~/segmentation/scripts/model.py***:



Layer
Batch Normalization
ReLU
3×3 Convolution
Dropout $p = 0.2$

Transition Down (TD)
Batch Normalization
ReLU
1×1 Convolution
Dropout $p = 0.2$
2×2 Max Pooling

Transition Up (TU)
3×3 Transposed Convolution $stride = 2$

Figure 11: Original FC-DenseNet architecture and comparison between DenseBlock and HarDBlock (Top left: Overall architecture of FC-DenseNet; Top middle: DenseBlock; Top right: Comparison between DenseBlock and HarDBlock; Bottom row: Structure of other building blocks of FC-DenseNet)

- In order to fuse the data from RGB camera and Depth camera, we used the architecture from MMAF-Net [4], in which, there are 2 separate encoders to extract data from 2 camera, and we fuse them before each downsampling stage to create a main branch. The main branch is then used to upsample the feature map back to the original:

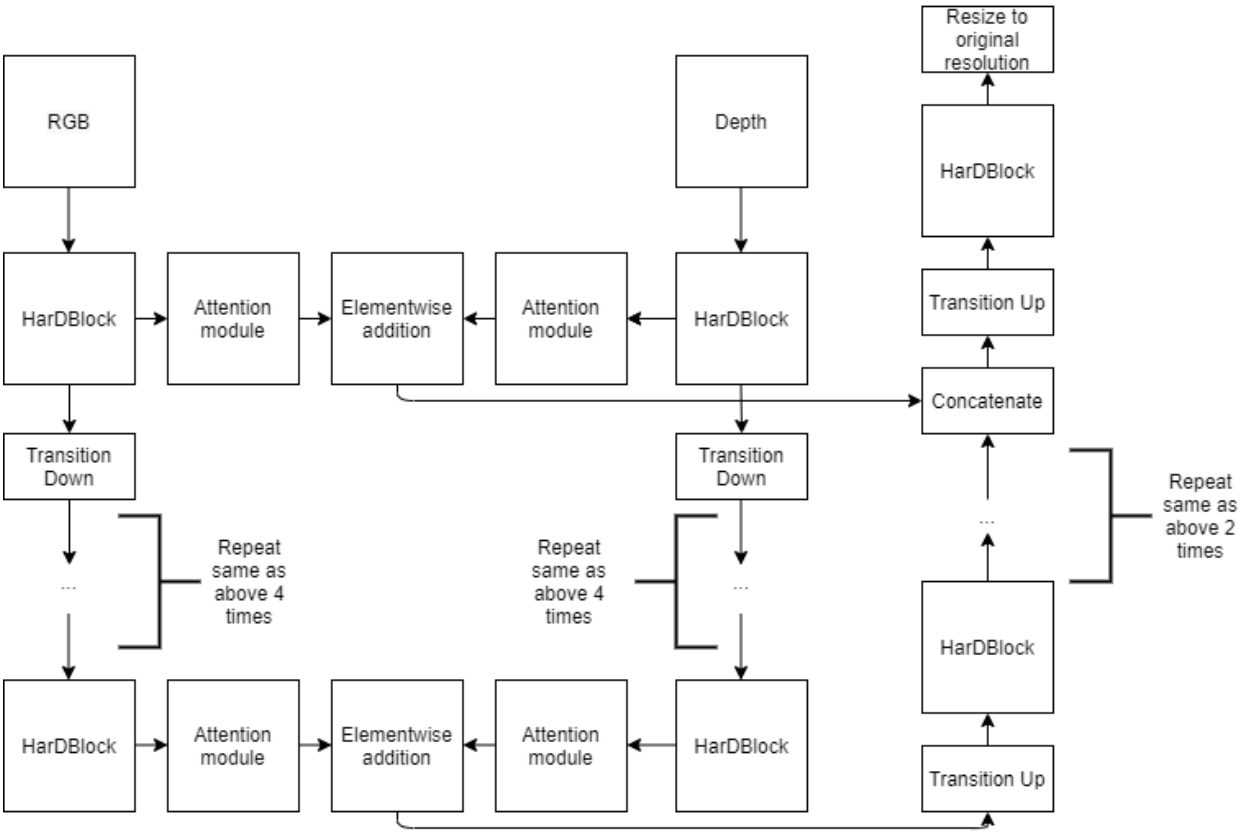


Figure 12: Architecture of RGB-D semantic segmentation

However, our model used a slightly different attention module from the MMAF-Net. Our attention module is taken from the Google Brain's Image Classification model called EfficientNet [9]. EfficientNet is among the best State-of-the-art Deep Learning model that make use of the Attention mechanism (or so-called Squeeze – Excitation Network [10]). Attention module helps the model know which features are useful and which features are not for the learning process, by multiplying an addition weight to each feature map. The weight will be 0 if the feature map is useless, and be positive or negative depending on the usefulness of the feature map:

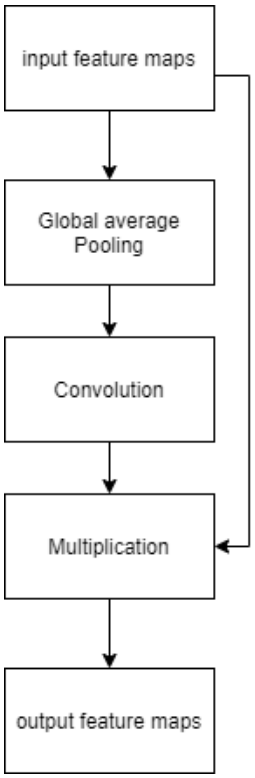


Figure 13: Architecture of Attention Module

In order to verify the advantage of our proposed model over other Deep Learning RGB-D segmentation models, we have tested our model on the renowned benchmarking indoor dataset NYU Depth V2 [11], which contains 1449 labelled images of indoor scenes with 41 classes. Here is the result:

Model	***No. of params (millions)***	***Pretrained with ImageNet dataset***	***Use HHA encoding***	***Global Accuracy***	***Mean Accuracy***	***Mean IoU***
DeepLab [12]	>21	No	No	50	23.9	15.9
HAA-CNN VGG16 [12]	>42	No	Yes	59.1	30.8	21.9
D- CNN +HHA VGG16 [12]	>42	No	Yes	61.4	35.6	26.2

D-CNN VGG16 [12]	>21	No	No	60.3	39.3	27.8
FCN - 32s [13]	>138	Yes	No	61.5	42.4	30.5
Facebook AlexNet [14]	>63	Yes	No	62.9	41.3	30.8
FuseNet [3] [15]	>30	Yes	No	66.0	43.3	32.7
FCH – 32s + HHA [13]	>138	Yes	Yes	64.3	44.9	32.8
FCN -16s + HHA [13]	>70	Yes	Yes	65.4	46.1	34.0
Facebook VGG16 [14]	>138	Yes	No	65.6	45.1	34.1
Ours	6.5	No	No	65.1	46.1	34.1
DeepLab-L + HHA [16]	>37	Yes	Yes	68.4	49.0	37.6
HAA-CNN VGG16 [12]	>42	Yes	Yes	Not found	51.1	40.4
CFN+HHA VGG16 [17]	>60	Yes	Yes	Not found	Not found	41.7
D-CNN+HHA [12]	>42	Yes	Yes	Not found	56.3	43.9
MMAF-Net-152 [4]	122.3	Yes	No	72.2	59.2	44.8
3M2RNet [18]	225.4	Yes	No	76.0	63.0	48.0

Pretrained denotes that the model has been trained in advanced on the 1-million-image 1000-class dataset ImageNet, before it is trained on the NUYV2 dataset. Model's name with ***HHA*** denotes that the depth data has been pre-preprocessed and converted to HHA encoding [19] before being fed to the model.

Due to our lack of time and resources, we cannot fully examine the exact number of parameters of other models. We can give only the approximation of such numbers, based on the model's encoder architecture that the authors discussed in their papers. The total of number of parameters of such models could be much higher in reality since we have not taken into account of their decoder architecture.

It can be seen that our model is the smallest one among all the evaluated models, and smaller than ~3 times the next smallest model in the list, and ~40 times smaller than the largest model in the list. Despite the small in size, our model can beat many other larger models without having to be pretrained on any other dataset. Considering only models that are trained from scratch, then our model is the best among them, in terms of both size and accuracy. Not to mention that we also outperformed many models that use the HHA encoding input rather than the raw depth map like us. HHA encoding is a method to convert raw depth as horizontal disparity, height above ground and the angle the pixel's local surface normal makes with the inferred gravity direction. Even though the HHA conversion is complicated, most models have reported an increase in performance accuracy when use with HHA input as can be seen in the table. However, in this test, our model just used raw depth, therefore, there is still lots of room for our model to increase its accuracy if used with HHA encoding. All in all, this test has verified the efficiency of our models over many other State-of-the-art Deep Learning models out there, making ours more suitable for real-world production settings and latency-prioritized applications.

5. Implementation details

This section will discuss about the data used to train the model as well as the data collection and processing stages. Furthermore, details on how to setup and train the model using the TPU and GPU of Google Colaboratory service is going to be presented. Lastly, instructions on how to deploy the model in real-world scenario and integrate with the Robot operating system (ROS) will also be discussed.

5.1. Data

Before feeding our depth data in the model, we first converted the depth image into point cloud. The reason is by having point cloud data, the model can infer more features from the environment. For example, walls tend to have same relative horizontal distance to the camera, while ceilling tends to have higher height than the camera. Meanwhile, if we use only the original depth image, only Euclidian distance from the point to the camera or the distance of the projection of the point onto the center axis of the camera can be extracted. In order to convert to point clouds, we used the following imaging model:

$$x = (x_d - cx_d) * \frac{depth(x_d, y_d)}{fx_d}$$

$$y = (x_d - cy_d) * \frac{depth(x_d, y_d)}{fx_d}$$

$$z = depth(x_d, y_d)$$

$where$:

x, y, z: $real\ world\ coordinates$

(x_d, y_d): $pixel\ coordinates$

cx_d, cy_d: $pixel\ coordinate\ of\ the\ principal\ point\ (center\ of\ projection)$

fx_d, fy_d: $focal\ length\ of\ the\ image, as\ a\ multipl\ of\ pixel\ width\ and\ height$

$$depth(x_d, y_d): real\ world\ depth\ at\ the\ pixel\ coordinate\ (x_d, y_d)\ in\ milimeter$$

To prove that our point-cloud data processing approach is superior than just feeding in the raw depth image, here is the difference of our model's performance when trained on 2 different types of input:

	Raw depth	Point clouds
Mean IoU score	0.31	0.40

Please be noted that since JPG image file extension only allows for positive integers, while the point clouds are real and float numbers, our pipeline also scale the point clouds frames back into integers ranging in [0,255] to save them as image and save up the disk memory. This also offers an advantage that the model will generalize on the relative position of points in space, rather by learning by heart the exact coordinates of them.

Our data came from 2 main sources:

- An external synthetic indoor RGB-D dataset with segmentation label: Unlike outdoor semantic segmentation, there are not many large authentic indoor segmentation datasets. There are only approximately 5 indoor datasets, and the largest one contains only more than 5000 images with many mislabeled images, which is far fewer than other type of DL problem like object detection with up to 100.000 images. To solve this problem, we decided to switch to synthetic datasets, which has been proved to may also work in Computer Vision task [20]. Synthetic dataset's labels are also much more precise than human-labelled dataset:

Figure 14: Mislabelled sample from SUN RGB-D dataset (the floor is mislabelled)

The dataset that we used is SceneNet RGB-D [21] with up to 5 million images of 14 classes (Background, bed, books, ceiling, chair, floor, furniture, objects, picture, picture, sofa, table, TV, wall, window) at resolution 240x320. The dataset can be downloaded from this link https://robotvault.bitbucket.io/scenenet-rgbd.html. However, we also have stored a preprocessed SceneNet dataset in the local PC in the lab under the folder ***~/segmentation/data/scenenet***, in which we only select only 1 in every 5 images in the original dataset to eliminate similar images and to save disk space. If you want to re-run the whole preprocessing steps with SceneNet dataset, the code can be found under the folder ***~/segmentation/pySceneNetRGBD/parse_scenenet.py.*** on the local workstation in the robotics lab.

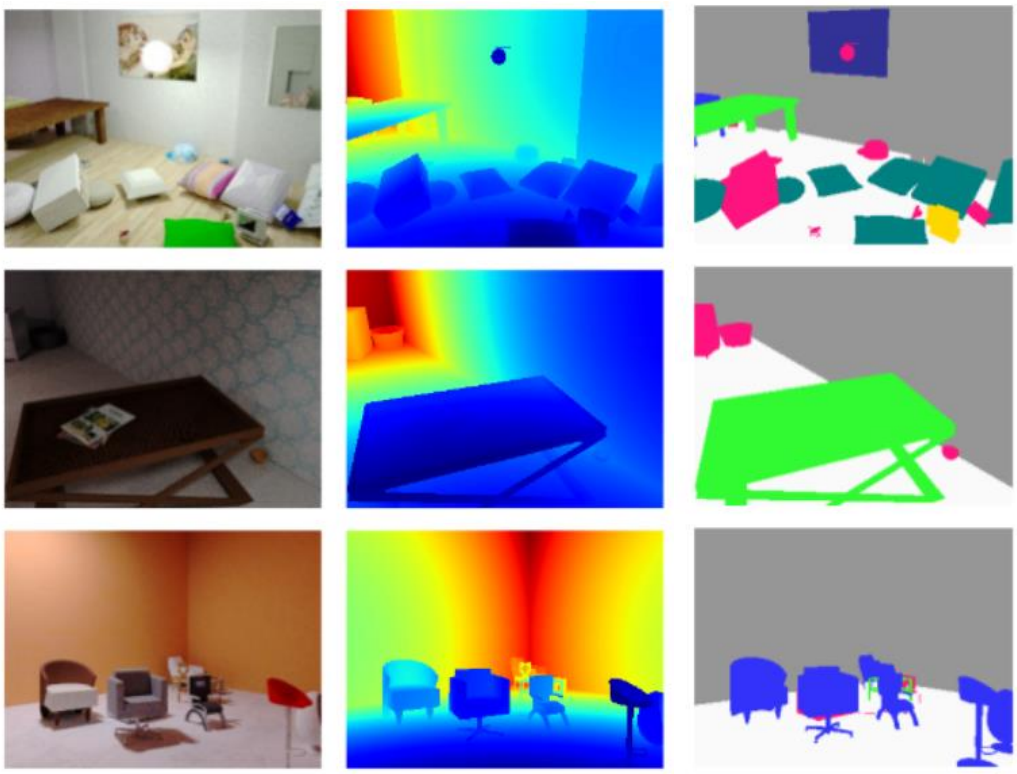


Figure 15: Examples synthetic images from SceneNet dataset (left to right: RGB, depth and label images)

- A custom dataset at FRAUAS labs: The SceneNet does not include some classes that we are interested in, such as human, backpack, robots, and obstacles in general. Meanwhile, there are many irrelevant classes such as bed, toilet or ceiling. Not to mention that the environment at FRAUAS labs are much different than the scenario presented in the SceneNet dataset, which is more about household scenarios. Therefore, in order to make the model works well with our scenario, we also have to collect more images around the robotics lab. The images are collected by a D435 Intel Realsense attached to the Roswitha. The images can be captured at resolution 640x480 at 90 FPS, and point clouds converting speed is 30 FPS.

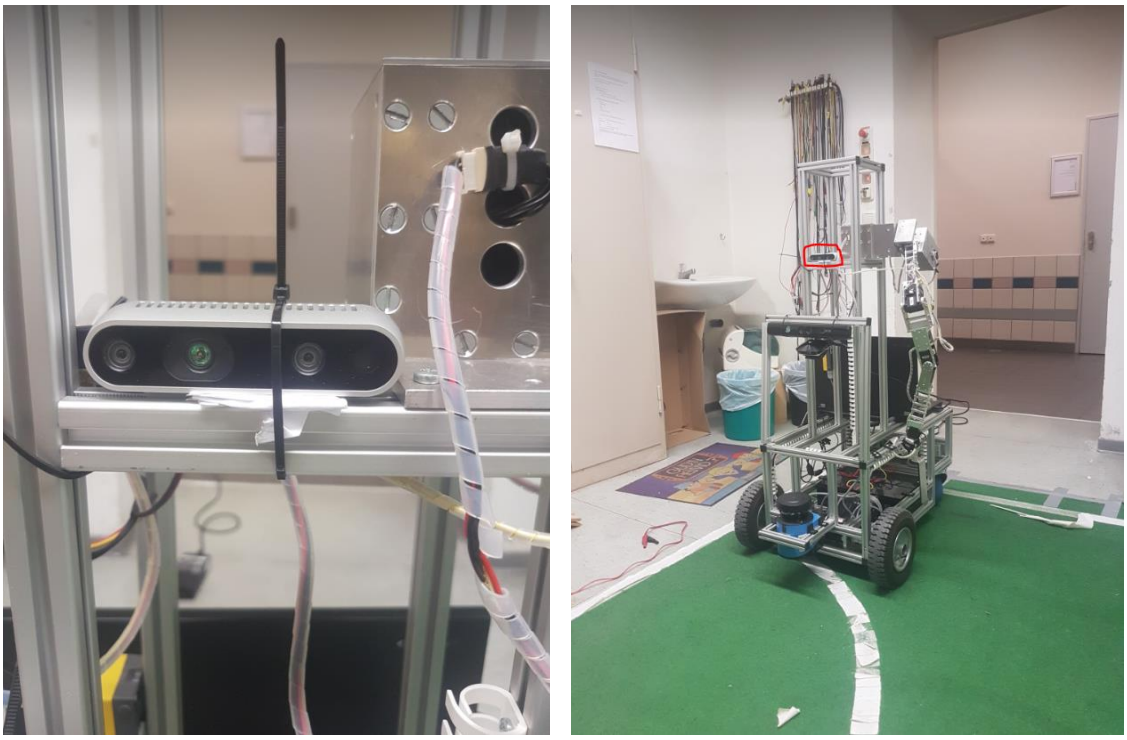


Figure 16: Intel RealSense camera attached on Roswitha robot

We moved the robot around the corridor and in the 2 robotics lab to collect the images. In order to connect to the RealSense camera on the laptop, the Intel RealSense SDK package has to be installed on the laptop. The detailed installation instruction can be found here: https://github.com/IntelRealSense/librealsense/blob/master/doc/distribution_linux.md for pre-built packages or here: https://github.com/IntelRealSense/librealsense/blob/master/doc/installation.md for building from source. After installing, plugging in the camera to the USB 3.0 port and type ***realsense-viewer*** in a new terminal. After that, a GUI of the Intel RealSense will pop up, and toggle the switch of RGB stream and Depth stream to visualize the data from the camera

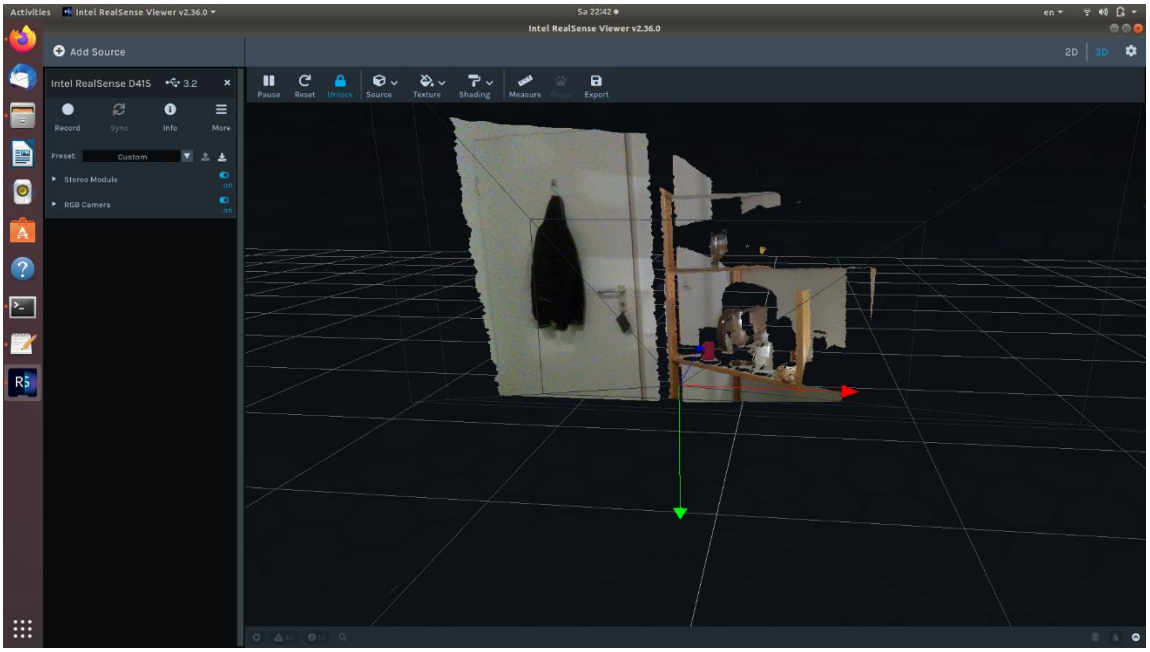


Figure 17: Intel RealSense GUI

Also since our code is written in Python 3, it is necessary too to install a Python binding, which is called *Pyrealsense 2*, to the original C++ Librealsense package. To install it on Linux, simply open a new terminal and type ***pip3 install pyrealsense2***. Verifying the installation by typing ***python3 -> import pyrealsense2*** in the terminal.

However, we have already installed all the package in the laptop. In order to run the whole data processing pipeline, please run the file at ***~/segmentation/scripts/data_collect.py***. By running the script, the RGB frame will be saved into ***~/segmentation/data/custom/rgb*** folder, and the x,y,z point clouds frame will be saved into ***~/segmentation/data/custom/x***, ***~/segmentation/data/custom/y*** and ***~/segmentation/data/custom/z*** respectively. This scripts also help removing the outliers in the depth of the image, since the depth is usually distorted when steered at the window or glass surface, by detecting whether there is high skewness in the depth values:

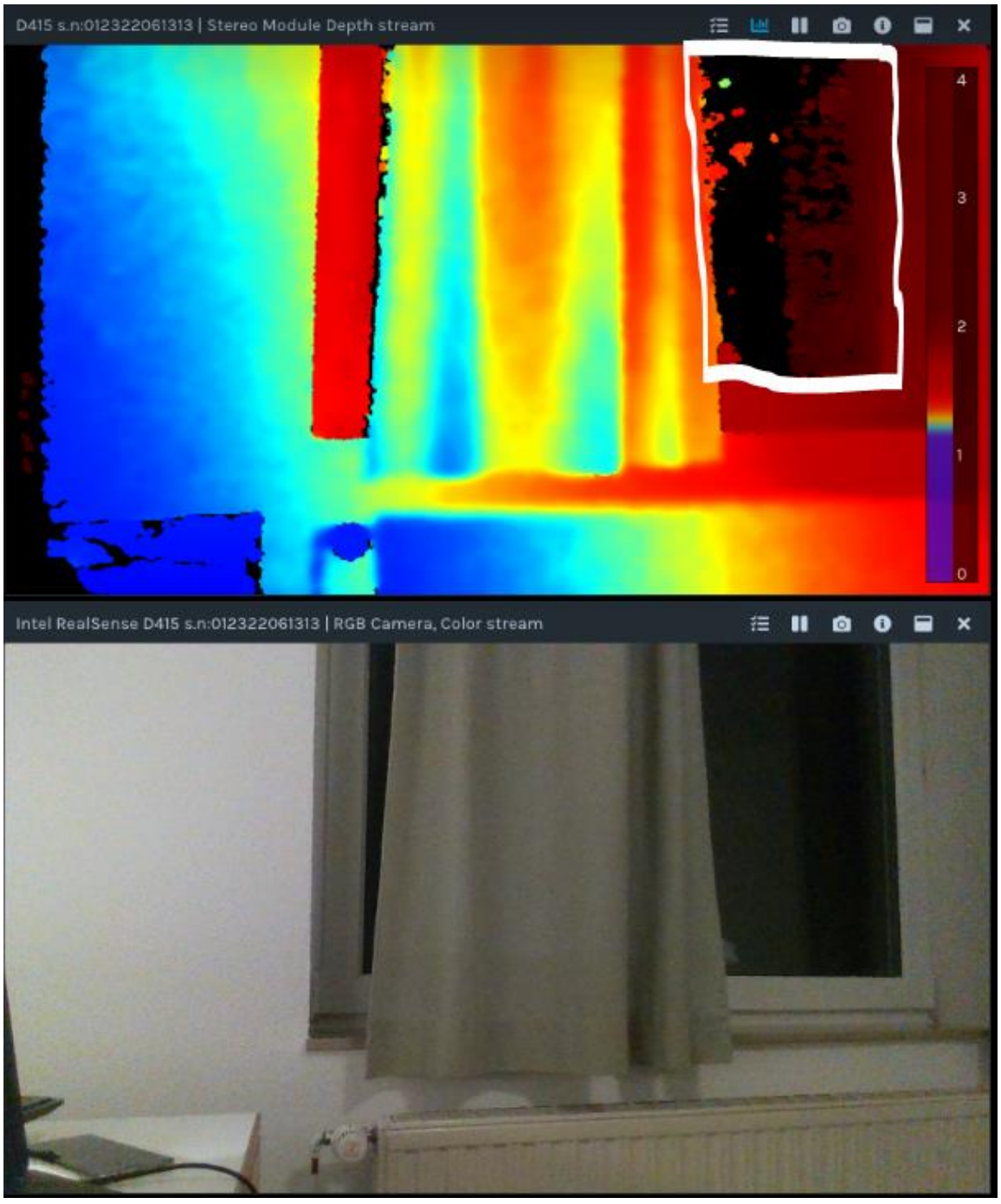


Figure 18: Distortion of window's depth (region inside white rectangular)

Our FRAUAS dataset is divided into 2 subdatasets:

- 1st dataset with ~1400 images of 3 classes: Obstacles, free space and human. We constructed this dataset to first test the performance of the model on real-world

data. Also, we use this dataset as a backup plan in case our experiment with the 2nd dataset (which is more difficult for the model to learn and takes longer time to annotate due to the increase number of classes) fails. This dataset takes more than 2 minutes to label each image manually, resulting in around ~47 hours to label this whole dataset.

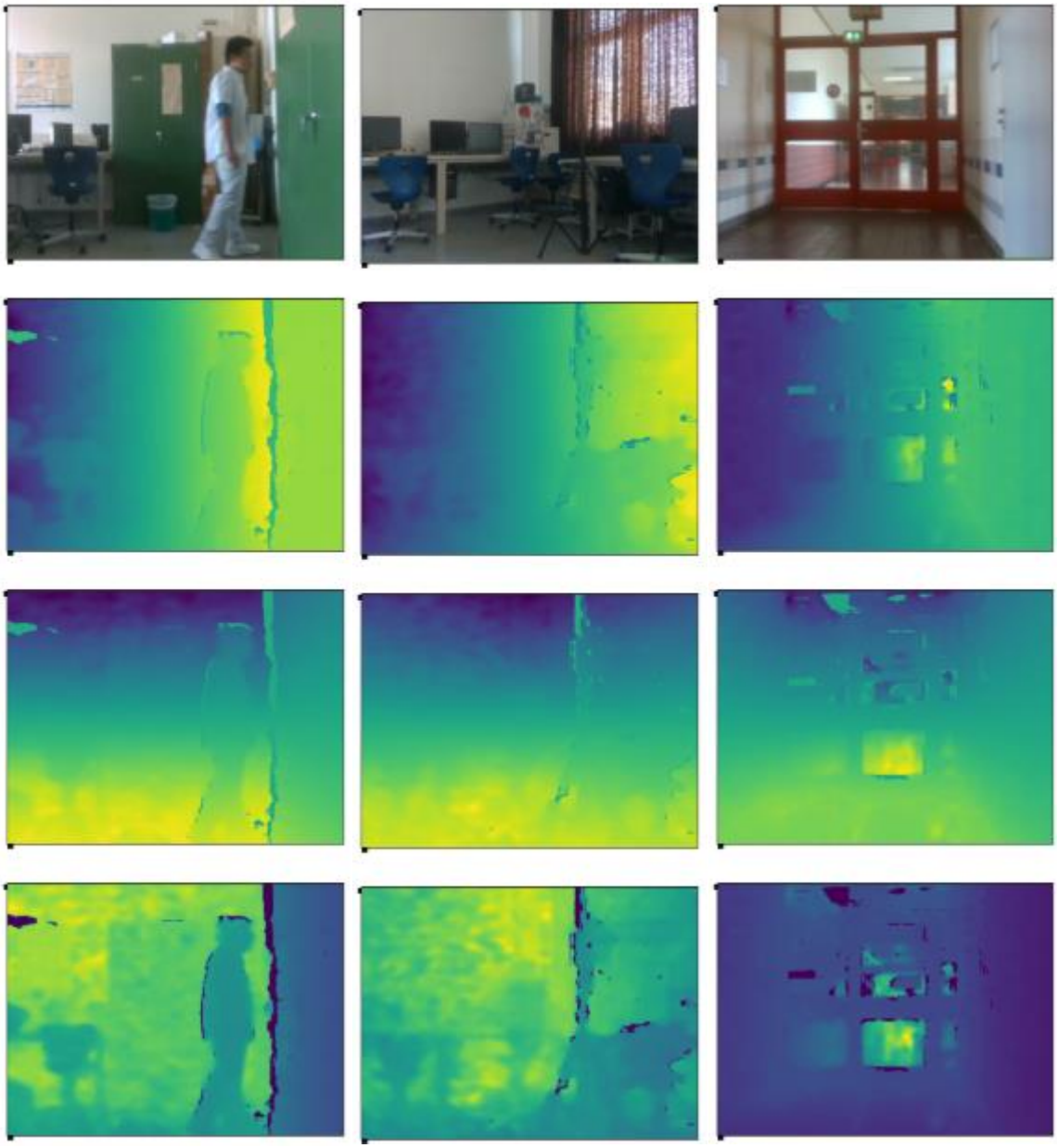


Figure 19: 3-class FRAUAS dataset (from top to bottom: RGB, x, y, z coordinates and label)

- 2nd dataset with ~390 images of 7 classes: Free space, human, chair, table, robot, backpack and other obstacles. This dataset takes ~10 minutes to label each image, resulting in ~65 hours to fully label the 2nd dataset:

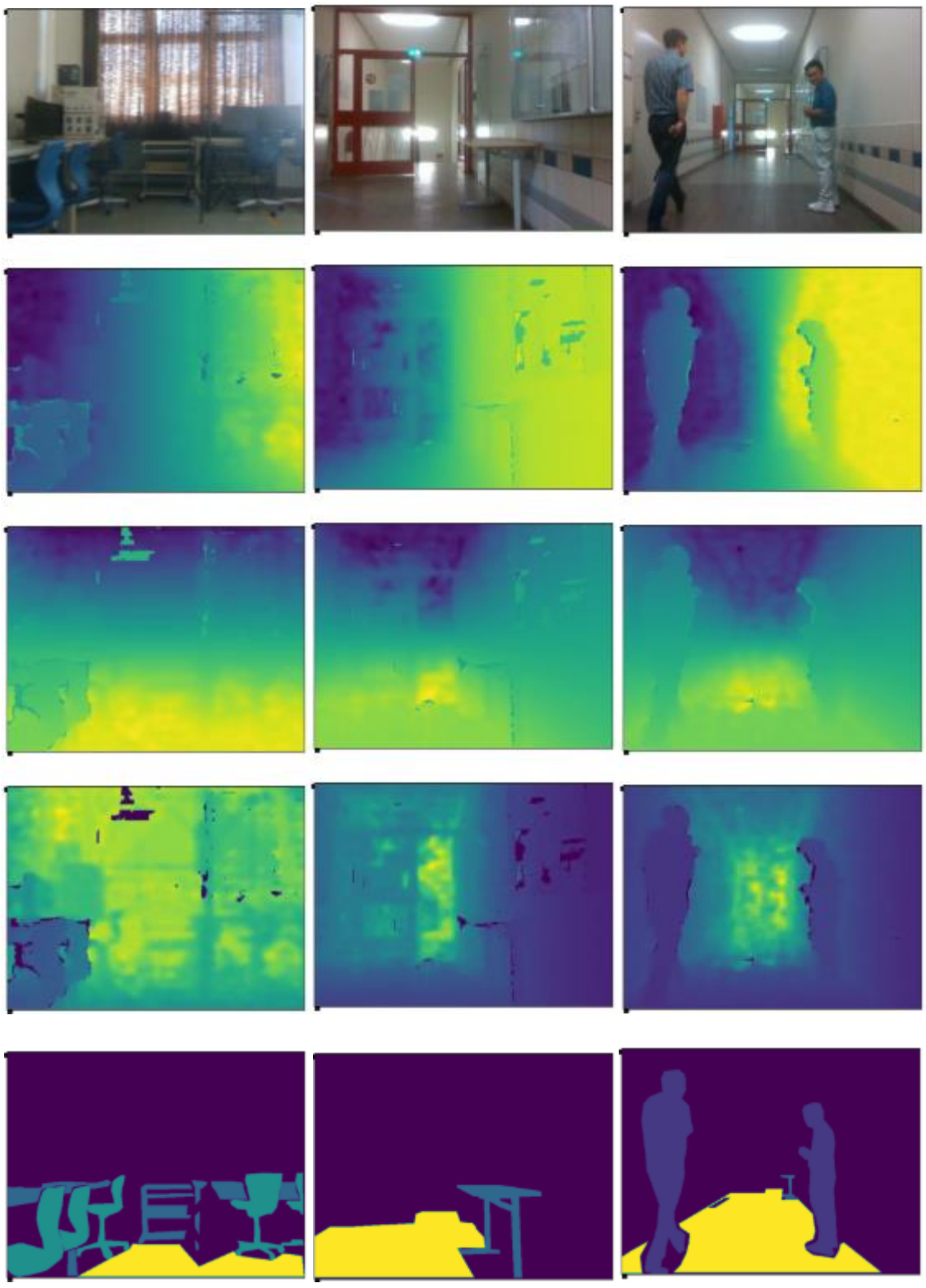


Figure 20: 7-class FRAUAS dataset (from top to bottom: RGB, x, y, z coordinates and label)

The software used to label the FRAUAS dataset is Labelme. Labelme is an open-source polygonal annotation software, which can be installed from here (we have installed it on the laptop): *https://github.com/wkentaro/labelme*. To label your own image, first launch it from the terminal by typing ***labelme*** on the terminal, after which the GUI will pop up. Click on ***open image/open folder*** to load the image, the click ***create polygon*** and start annotating the vertex of the of the polygon bounding the object. Afterwards, assigning the class that the polygon belongs to. After saving the annotation, a JSON (Javascript Object Notation) file will be generated. You should save your own JSON files in the folder ***~/segmentation/data/custom/json.*** If you do not want to use our label, but create you brand new label sets, then please run the **~/*segmentation*/scripts/custom_json_to_label.py** script to generate the PNG label in greyscale from the JSON files (saving label images in JPEG format will cause inconsistencies around the edges due to the compression mechanism) around boundaries. Please modify the class mappings to map each class to certain integers in the scripts as your choice. The ***~/segmentation/scripts/*** also contains scripts to convert FRAUAS dataset label JSON files to PNG files.

Figure 21: Labelme GUI

However, since we are using Tensorflow platform, which is built by Google, to train and deploy our models, we have to convert the .jpg extension files into *.tfrecords* format suggested by Tensorflow. TFRecords is a file format in which data is converted to binary, which takes less space on disk, less time to copy and can be read efficiently from disk. Also, by storing files as TFRecords, the whole data does not need to be loaded into the RAM all at once, which will be very helpful for large dataset of 100GB like SceneNet. It is also well-known that the processing speed of GPU is much higher than CPU, therefore, in normal cases, the GPU will have to wait for the CPU to feed in the images before they can start training, which is a waste of idling time. With TFRecords, ones will be able to minimize such overhead by parallelizing and prefetching the data preprocessing pipeline, which we be discussed in more detail in training section later. To convert the our FRAUAS dataset image files to TFRecords, please run the script at ***~/segmentation/scripts/3classes_frauas_to_tfrecords.py*** for the 3-class dataset and ***~/segmentation/scripts/7classes_frauas_to_tfrecords.py*** for the 7-class FRAUAS dataset***.*** Also, the script to convert the SceneNet dataset to TFRecords is available at ***~/segmentation/scripts/scenenet_to_tfrecords.py*.** We partitioned the SceneNet dataset into 17 smaller TFRecords files to conveniently transfer and store them, as well as the fact that *TFRecords* work better for large number of small files rather than small number of large files.

To convert your own dataset to TFRecords, then please run the file ***~/segmentation/scipts/custom_to_tfrecords.py.*** The generated TFRecords will be stored at ***~/segmentation/data/custom/tfrecords.***

Please be noted that for the FRAUAS dataset are stored in 5 folders, each corresponds to different environment settings that we organized in 5 different days. All the JSON and the image files of the SceneNet and FRAUAS are stored at ***~/segmentation/data/*** folder.

5.2. Training

Our model training process is divided into 3 stages:

- Train the model with SceneNet dataset to give good initial weights to the model.
- After training the model with SceneNet, we trained it with the 1st FRAUAS dataset (3 classes only). This technique in Deep Learning is called Transfer Learning. Transfer Learning is used to depict the process of pre-training the model on a large dataset, and then use the pretrained weights of classifier's part of a model (or some last layers of the model) to continue learning on another quite similar task. For example, a DL model trained on 1 million images to classify Dog and Cat can be used later for training a classification model of Cat and Horse with only 1000 training images. This approach has been adopted in many DL image classification models, which have been trained in lengthy time by mega-tech companies like Google on large dataset (e.g. ImageNet [22]) of millions images on mass-scale computing clusters. These models then can be later used for training on other classification task to save training time, resources and increase the generalization capabilities of the algorithm. Since SceneNet is a quite big dataset, we believed that adopted the same strategy will save the training time and enhance the flexibility and extensibility of the model when transfer it on different domains in the future (e.g. segmenting images in outdoor environment rather than in indoor environment).

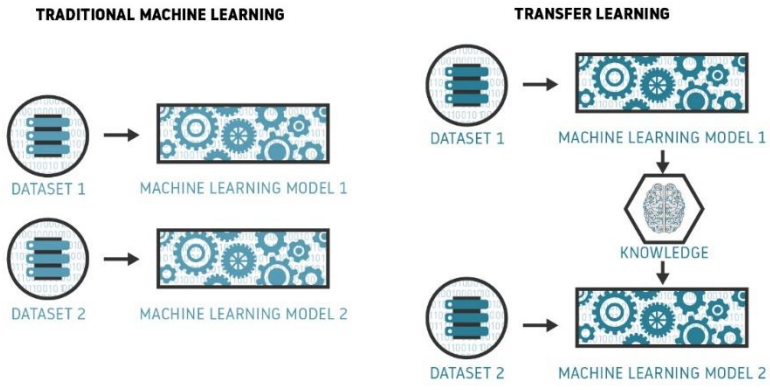


Figure 22: Illustration of Transfer Learning technique

Normally, when training for other domain tasks, only the weights of some of the last layers will be trained, while earlier layer's weight will be frozen. This is because early layers have been trained very well on large dataset to extract the best features from any images, while later layers are more specific to a certain task (e.g. edges features will be necessary for any classification task and has been extracted correctly in early layers, however complex features like body parts will be important for human classification rather than vehicle classifications, and those features needed to be learnt in later layers). In our scenario, we will freeze the encoder branches of the model, and only train on the decoder, since the decoder is the part of the model that is responsible for inferring from the data extracted by the encoder. As the training progress, once we saw no improvement in the validation accuracy, we unfroze the encoder branches and fully train the model with small learning rate (10^{-4}) to squeeze some more accuracy gain from the model but at the same time to ensure that the weights would not drift to much from the original numbers. A same training approach can be found here at the official documentation of Tensorflow (https://www.tensorflow.org/tutorials/images/transfer_learning):

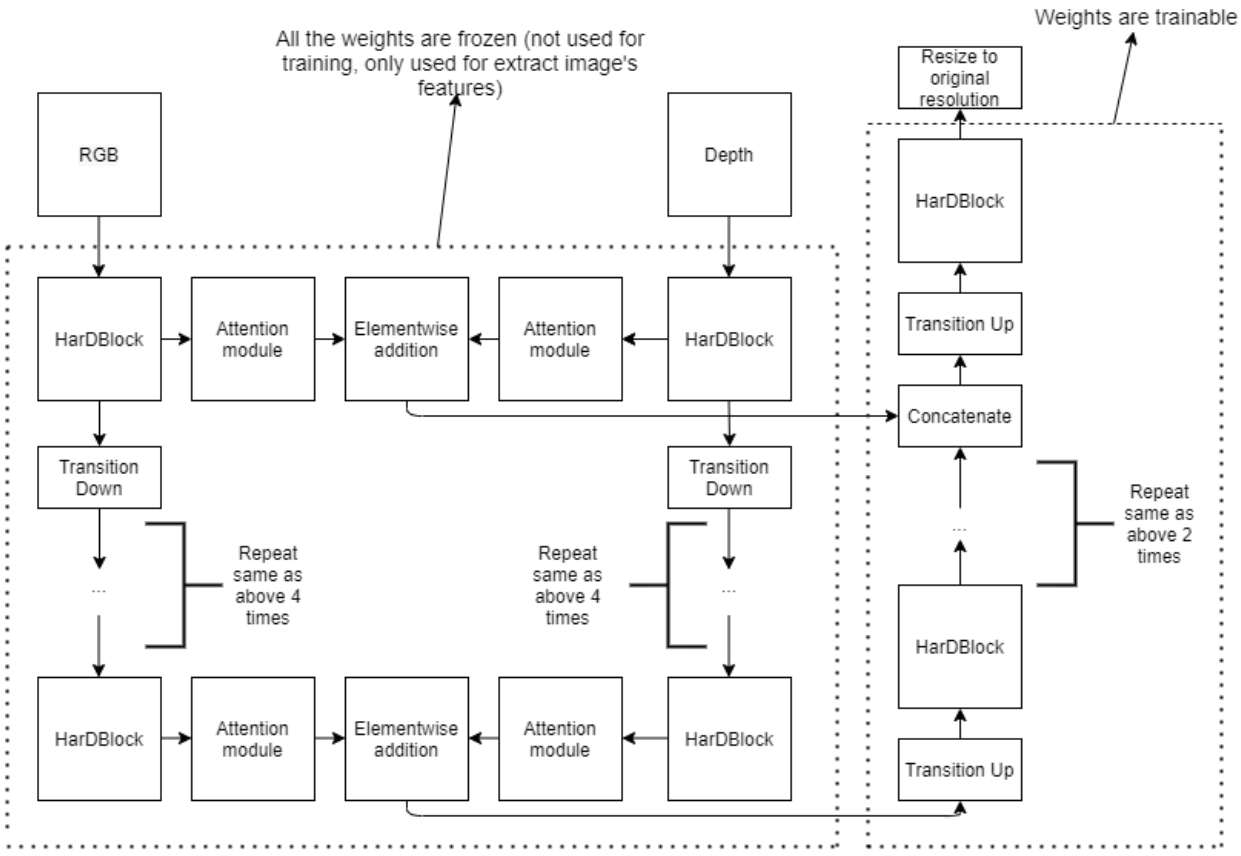


Figure 23: Transfer Learning in RGB-D segmentation

- After training the model with the 1st FRAUAS dataset, we continue training it with the 2nd FRAUAS dataset (with 7 classes).

To verify the advantages of the transfer learning approach versus learning from scratch, we have conducted an experiment between our model training on the 7-class FRAUAS dataset with 2 different set of weights: Completely random weights (train from scratch) and pretrained weights of SceneNet dataset and 3-class FRAUAS dataset. In our pretrained version, we only continue the training process of the decoder, while freezing the encoder. Here is the result:

	Trained from scratch	Pretrained
Mean IoU score	0.67	0.70
Training iterations until convergence	10000	8000

%RAM required when train with batch size = 28	100%	78%

As can be seen, our model when trained from scratch achieved a lower accuracy, took longer to converge and occupied more RAM in compared to when trained by the transfer learning policy (since we only train half the model in the pretrained settings).

The first training stage took place on a TPU (Tensor Processing Unit) offered on Google Colaboratory services. Google Coloboratory is free cloud-based service, on which users can run their Python codes on GPU or TPU-powered Linux virtual machine. The virtual machine is also run on Docker image with pre-installed Machine Learning library like Tensorflow, Pytorch or Numpy, saving setup time and dependencies conflict between different software. An introduction on how to use Google Colab can be found at https://colab.research.google.com/notebooks/intro.ipynb#. Google Colab can only runs codes written in IPynb files. Ipynb is a Python notebook file extension (IPython), which is an interactive command shell for interactive computing in mainly Python programming language as well as other languages. In other words, an IPython notebook file is in fact a collection of many different Python scripts, which are divided in code cells, in which all scripts share a same memory and can access other scripts' functions and variables as long as they are compiled before they are called. The result of running each scripts (or code cells) will be displayed without being deleted right afterwards underneath the code cells for later inspection. By working with IPython notebook, we found it more convenient to debug the code (as we can print out the result or behavior of each section of the code), divide the code into different sections without having to save them into many different Python scripts, and since IPython is an interactive shell, it is easier to experiment and modify the codes as well as to present the visualization to other people. Besides using Google Colab, IPynb files can also be opened with Jupyter Notebook, which can be installed from here: https://jupyter.org/install.

Another advantage of Google Colab is that they can be launched directly from the web browser without having to setup and install an additional software and can be mounted directly to the Google Drive to perform read and write operations.

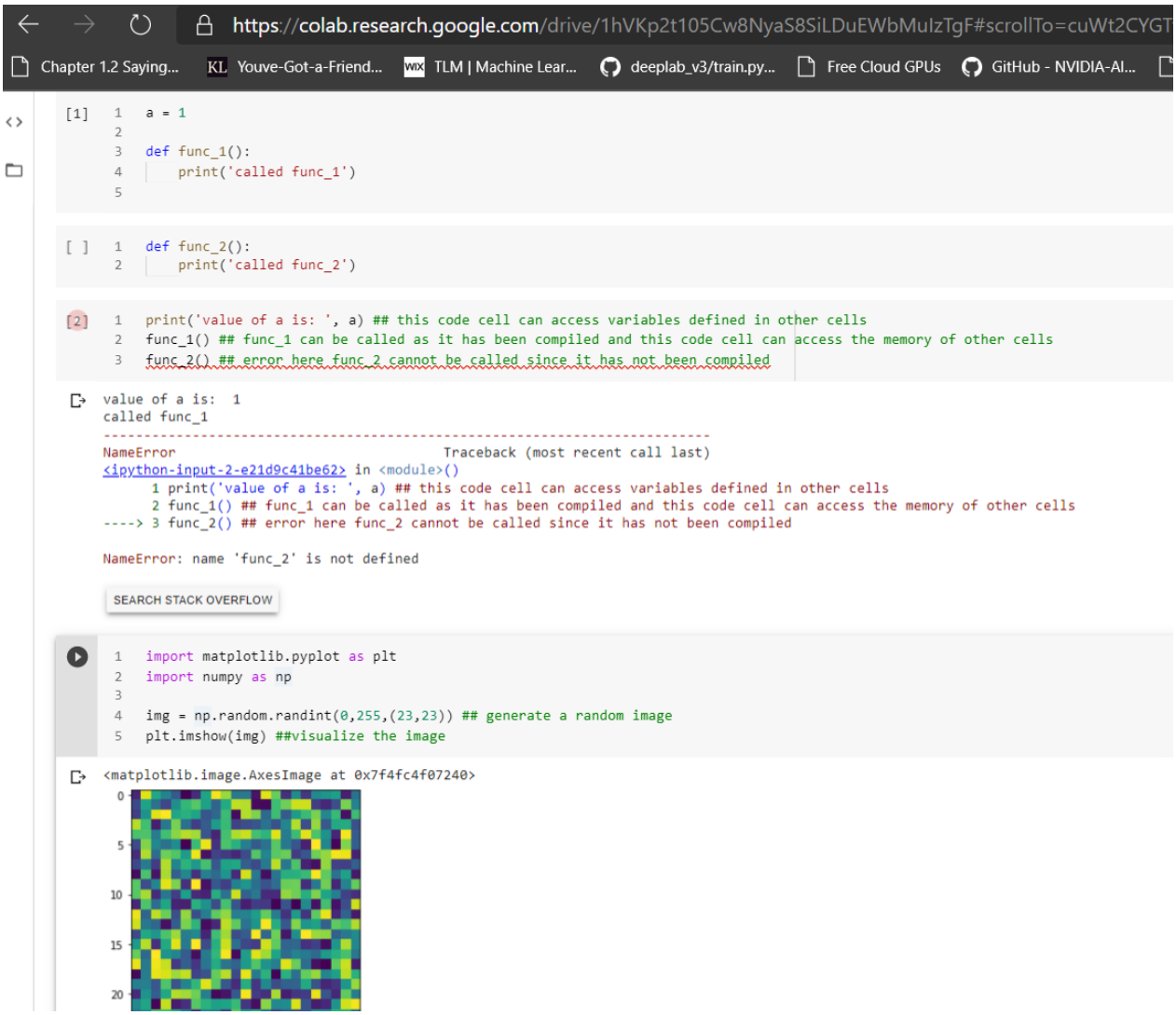


Figure 24: Demo of Google Colab on web browser

The number on the top left corner of each code cells refer to the order that they have been compiled (e.g. 1 means this is the first compiled cell; 2 means this is the next compiled cell …). It can be seen the second cell has not been compiled (no top left corner number is displayed), thus leading to the third cell not recognizing the function *func_2().* It can also be seen that the result of each cell can be displayed separately, as well as each cell can be run separately without any

order. All of these features lead to easier organization of code, as each functionality of the code can be separated in a different cell and run independently. The programmer also easily knows where the bug is and can conveniently test out their solution within the cell. Also, while exiting from a .py file will automatically release all the memory occupied when running the file and destroy all the objects, stopping a cell in notebook file will not release any memory unless the user specifically reset the whole notebook. This helps saving time loading the model or compiling every parts of the code, since they are still stored in the memory.

One of the disadvantages of Google Colab is, the free version of it only allows for 30 hours of usage per week, and no more than 12 consecutive hours of usage. The CPU has only 2 cores with disk size of 64 GB and RAM of 12GB. Regarding the GPU, Google Colab will randomly assign users a K80 GPU with 12GB VRAM or a P100 GPU with 16GB VRAM. However, if users upgrade their Google Colaboratory to the Pro tier, they will get longer runtime (up to 24 consecutive hours of runtime), more CPU's RAM of 25GB and the strongest Deep Learning GPU in the moment - NVIDIA V100 of 16GB VRAM with TensorCore technology (which halves the training and inference time when converting the model's weight to 16 bit floats), for 10$ per month.

Tensor Processing Unit (TPU) is an AI accelerator application-specific integrated circuit (ASIC) developed by Google specifically for neural network machine learning, particularly using Google's own TensorFlow software [23]. TPU offers more RAM (64 GB versus 12 GB of a standard K80 GPU) and more speed (100x speed up in compared to GPU). However, to use TPU, users need a Google Cloud services to upload their data onto their own Google Cloud Storage Bucket, since TPU does not allow reading operation from local disk but only from cloud service. Google Cloud is offering 300$ credits for first-time user, which would be enough for short-time usage or experimenting purpose. However, if user intends on lengthy, large-scale usage, it is suggested that the cloud account's budget should be constantly monitored, otherwise unwanted excessive fee will be charged.

We used TPU for the SceneNet dataset since it is very large in size, therefore require long training time and high throughput (many simultaneous trained images per training iteration – also known as batch size). With TPU, the training took approximately 1 days until the model's performance

stops improving. Meanwhile, for the training the FRAUAS dataset, we only make use of GPU since the dataset is quite small and we have to use small batch size (TPU batch size has to be large, which is suggested to be larger than 32, and is divisible by 8 since TPU has 8 cores and each core must be assigned a same number of images). Training the FRAUAS dataset takes approximately 3-4 hours on the V100 GPU.

The training pipeline, which is similar for all 3 datasets, can be illustrated as follow:

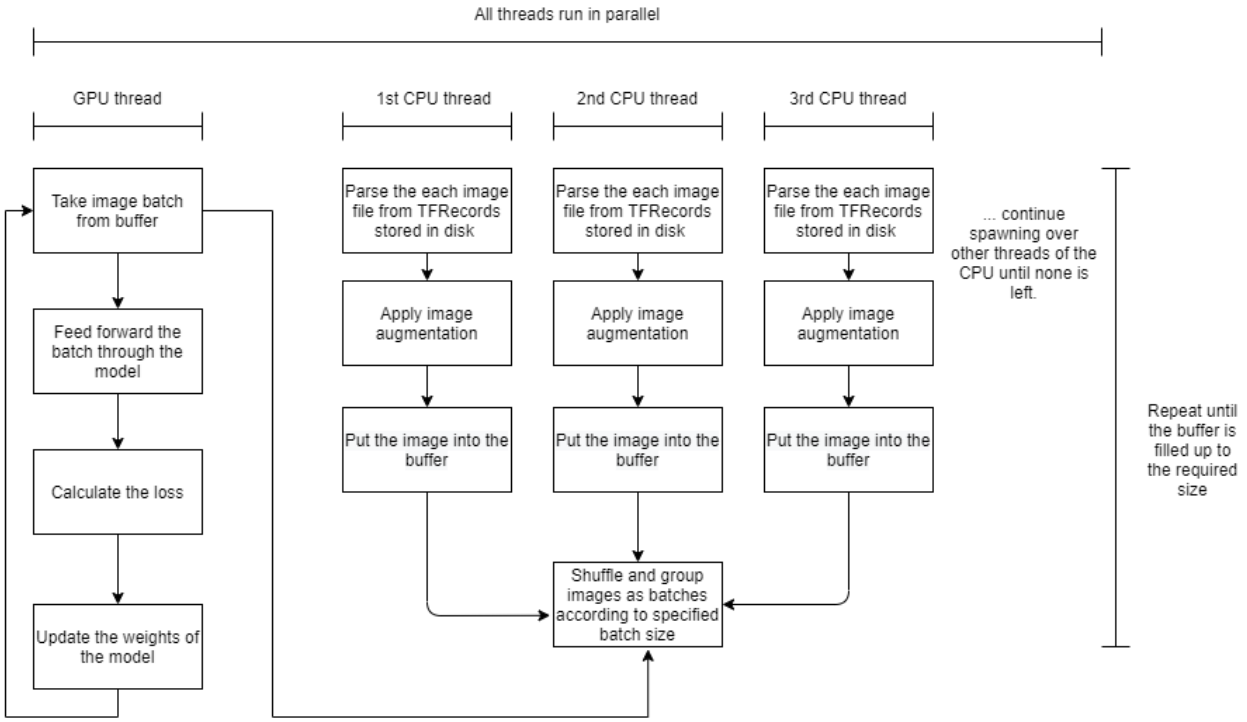


Figure 25: Diagram of training pipeline

There are 2 well-known problems when training small to medium DL model, that are:

- In reality, the GPU has to wait most of the time until the CPU finishes preparing the batch to feed in the GPU. This is due to the training time on GPU is usually much smaller than the time required by a CPU to perform data I/O operations and complex preprocessing operations.
- Ones might argue that all the data I/O and preprocessing steps should be done entirely on the GPU. However, this is usually not the case since we have to save VRAM for GPU to store model's weights and gradients information to update the weights, hence doing both

things at once will easily cause memory overflow for the GPU. It is a rule of thumb that all the data-related steps should be done in CPU, as GPU is more suitable for parallel simple humongous-scale matrix multiplication operations required by a DL model ‘s millions of parameters with its high memory bandwidth, while complex and I/O operations (transferring data from disk to RAM for example) is more suitable for CPU. A GPU is more bandwidth optimized, and CPU is more latency optimized (considering an analogy of a CPU as a Ferrari, which can transport small number of person in short time, and GPU as a truck, which can carry more people but require longer time):

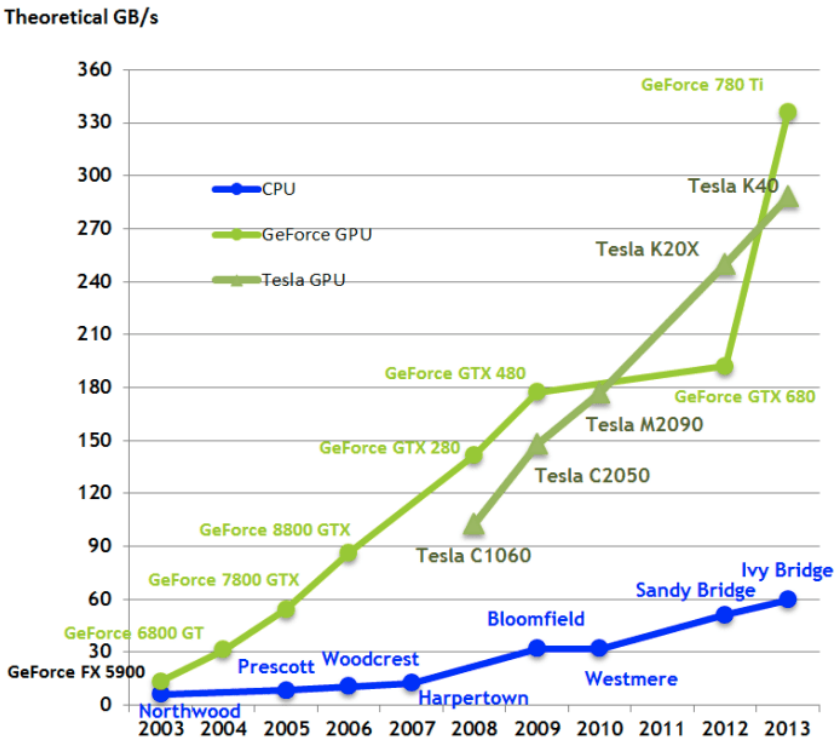


Figure 26: Illustration of GPU's bandwidth versus CPU's bandwidth

As introduced in section 5.1, TFRecords file format allows for data pipeline optimization to tackle such data parsing and processing problem via the prefetching and parallelized mapping mechanism:

- Prefetching overlaps the preprocessing and model execution of a training step. While the model is executing training step s, the input pipeline is reading the data for step s+1.

Doing so reduces the step time to the maximum (as opposed to the sum) of the training and the time it takes to extract the data. In particular, the transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of the time they are requested. The number of elements to prefetch should be equal to (or possibly greater than) the number of batches consumed by a single training step [24].

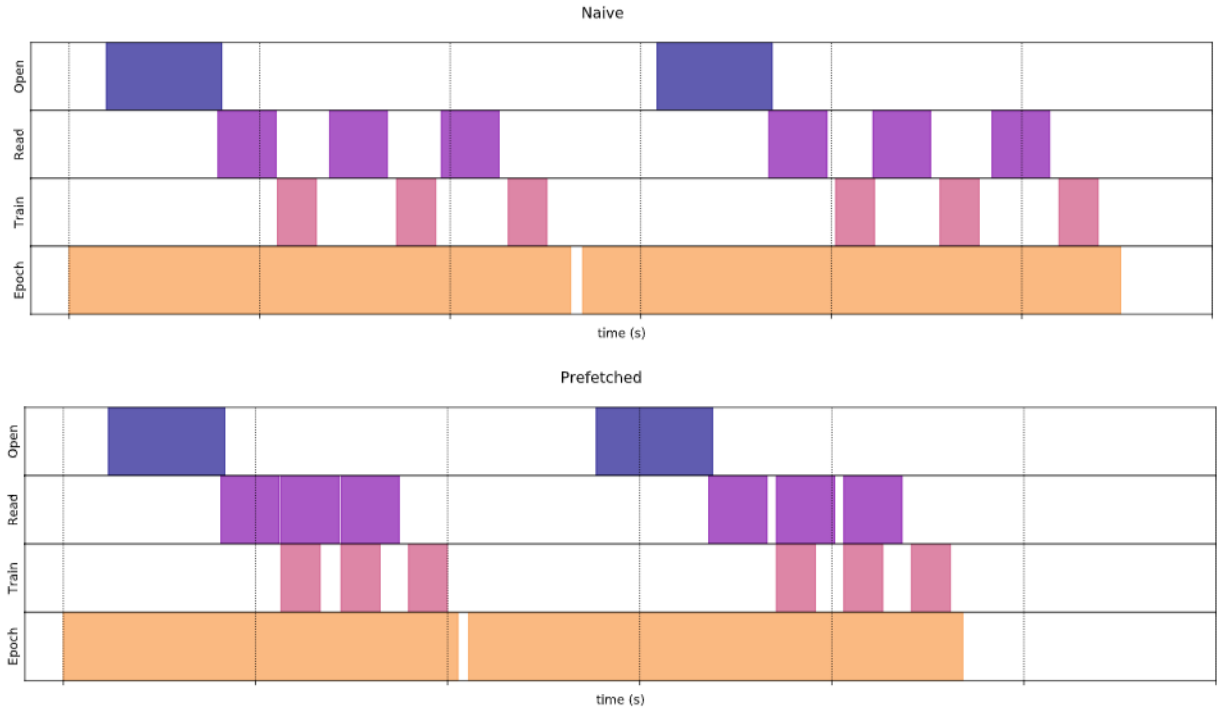


Figure 27: Timing diagram comparison between naive pipeline and prefetching pipeline

Normally, after opening and reading operations are performed on the TFRecords file, further mapping function may still be needed to perform on the data (e.g. random data augmentation, image normalization …). In order to save time, TFRecords also allow for parallelizing such mapping operations over all cores of the CPU in an easy fashion:

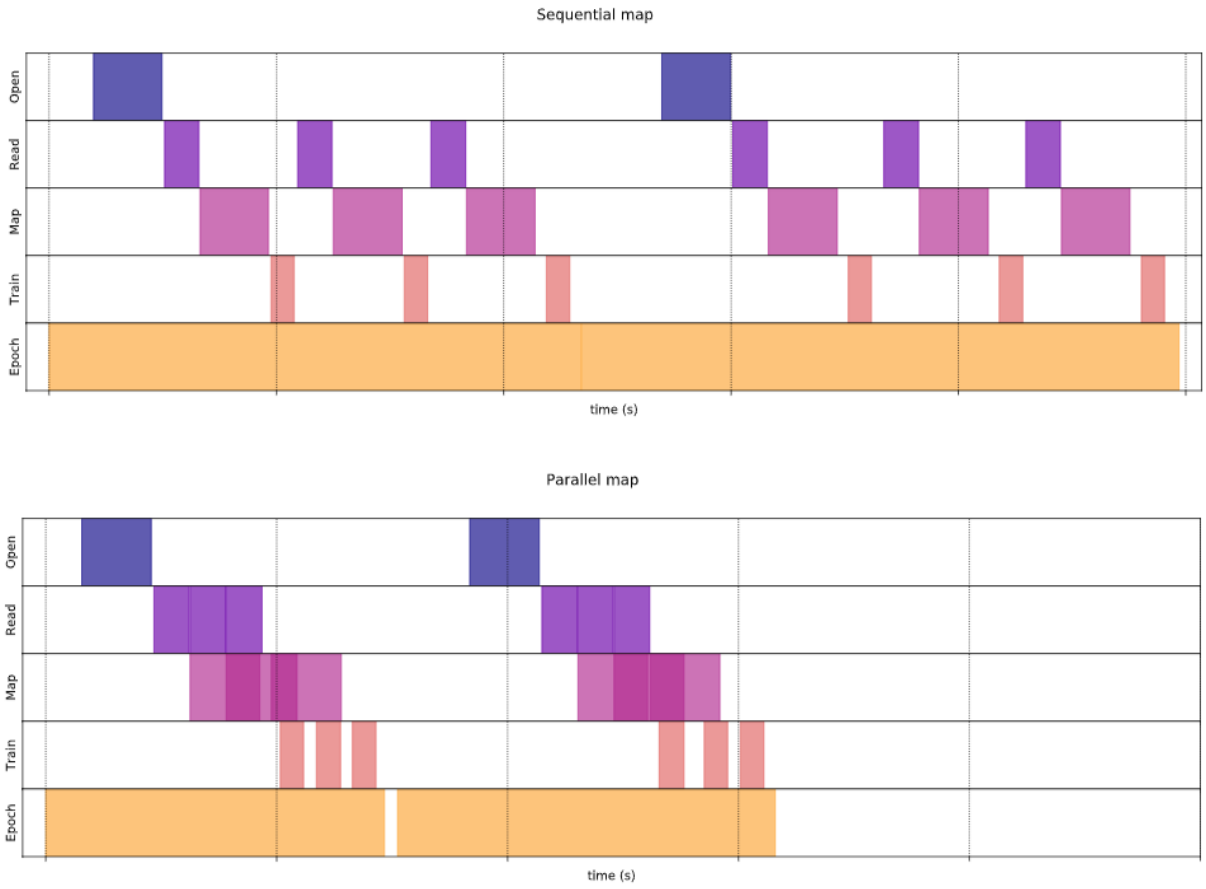


Figure 28: Timing diagram between sequential mapping and parallel mapping

In our training pipeline, there is a data augmentation steps in order to render more samples for our dataset, which is a small dataset:

- Randomizing the brightness, contrast, saturation, blurriness and hue channel (only for ScenetNet dataset) of the image.
- Random cropping (then resizing back to 480x640 resolution), flipping, shuffling and adding random Gaussian noises to the point clouds.

Also, as a measure to prevent overfitting due to small size of FRAUAS dataset, we also applied the following techniques:

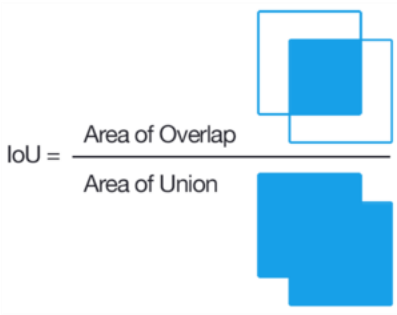
- Add L2 regularization to the weights. L2 regularization prevents the any weights from the getting too large (otherwise this will cause the loss function value surge significantly), thus

prevents the model from relying on a certain feature to heavily. Now, with w_k represent the weight k^{th} of our model, the loss function, which previously just contains the categorical cross-entropy becomes:

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K}[y_{ij}\log(p_{ij})] + \frac{1}{2}\cdot 10^{-4}\sum_{k=0}^{P}\|w_k\|_2^2$$

We multiply the L2 loss with 10^{-4} to make sure that the L2 term will not be too influential over the our main term in the loss, which is the categorical cross-entropy part.

- Add another term to reflect the accuracy metrics of the model onto the loss function, called Dice Loss (IoU loss). IoU (Intersection over Union) is a metric to measure how close is the segmentation mask compared to the label. IoU is a more trustworthy metrics over conventional accuracy, which tends to be in favor of majority class rather than minority class, to measure the performance of the model. IoU of a certain class can be calculated as follow:



$$IoU(c) = \frac{\sum_i (o_i == c \wedge y_i == c)}{\sum_i (o_i == c \vee y_i == c)},$$

where:

c: *the considered class*

o: *predictions*

y: *label*

i: *pixel indices*

In order to reflect such metrics in the loss function, the following modification is made to the loss function [25]:

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K}[y_{ij}\log(p_{ij}) + 1 - \sum_{i=1}^{n}\sum_{j=1}^{K}\frac{y_{ij}\cdot p_{ij}}{y_{ij}+p_{ij}-y_{ij}\cdot p_{ij}+10^{-7}}] + \frac{1}{2}\sum_{k=0}^{P}\|W_k\|_2^2$$

10^{-7} is added to the denominator of the IoU term to avoid dividing by 0.

Also, the initial learning rate is controlled via this policy:

$$lr = \max(\frac{0.002}{0.2 * epoch + 1}, 0.0001)$$

Our weights updating policy is *RMSProp*, which is a variant from traditional Gradient Descent algorithm, in which each variable will have its own adaptive learning rate and this learning rate will also take into account its past gradients to smooth out the update process and to speed up the convergence speed:

$$For\ each\ Parameter\ w^j$$

$$v_t = p * v_{t-1} + (1-p) * {g_t}^2$$

$$\Delta(w_t) = -\frac{\mu}{\sqrt{(v_t+\epsilon)}} * g_t$$

$$w_{t+1} = w_t + \Delta w_t$$

$$where:$$

$$p: Momentum$$

$$\mu: Initial\ learning\ rate$$

$$v_t: Exponential\ Acerage\ of\ squares\ of\ gradients$$

$$g_t: Gradient\ at\ time\ t\ along\ w^j$$

We also scaled the image and point clouds back to [0,1] range to make the training more stable.

The trained model's weights are located at ***~/segmentation/weights/weights_scenenet.h5***, ***~/segmentation/weights/weights_frauas_3classes.h5*** and ***~/segmentation/weights/weights_frauas_7classes.h5.***

The training code for the 3 training stages can be found at ***~/segmentation/scripts/train_scenenet.ipynb,*** ***~/segmentation/scripts/train_frauas_3classes.ipynb*** and ***~/segmentation/scripts/train_frauas_7classes.ipynb***. If you want to reproduce the results, before running these notebooks on Google Colab, please setup your Google Account as follow:

- Create a Google Cloud Storage Bucket in region US with the name ***segmentation_scenenet*****,** then upload all the TFRecords of SceneNet onto that bucket. In case that the ***segmentation_scenenet*** has been selected by other users, then please choose a different name, but then modification is needed to the ***train_scenenet.ipynb*** notebook file. Remember that when creating Google Cloud account, the Google Account must be the same as the one Google Colab and Google Drive use. Otherwise, conflicted access permission may arise later.
- Upload the notebooks files located at the ***~/segmentation/scripts/*** folder, the TFRecords of FRAUAS dataset to the folder ***Drive/segmentation/*** on your Google Drive as follows:

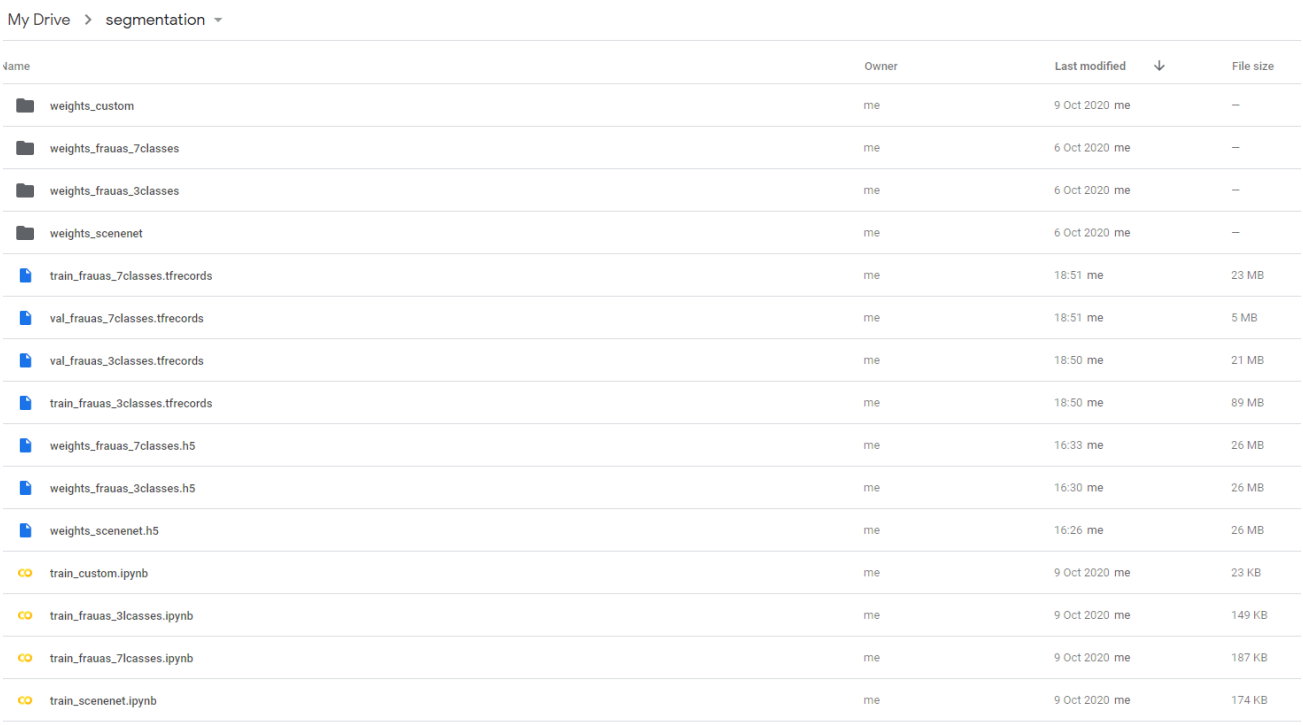


Figure 29: Drive folder structure

- In your Google Drive, click on the file ***train_scenenet.ipynb*** and select open with Google Colab. A new browser tab will pop up. Click on ***Edit -> Notebooks setting -> Hardware Accelerators -> TPU***. Then please run all the code cells from top to bottom by clicking on each cell and pressing ***Ctrl + Enter*** to run each cell. Please be noted that on the first cell, Google will require the users to click on 2 links to verify that the users give the permission for Google Colab to read and write to the Google Drive and the Google Cloud Bucket. As the training progress, users will see the progress bar update the training loss and accuracy after every training iteration. After a certain number training iterations (or so called 1 epoch), a validation is carried out on the test set to see the real performance of the model. Only weights of model performing better than previous epochs are stored in the folder ***Drive/segmentation/weights_scenenet/*** with the following format:
  trainloss:4f_IoUScore:.4f. If users want to stop the training process as there is no improvement of the IoU score, please click on the square of top left corner of the current running cell. If users want to the resume the training process, please set the ***continue_training*** to True, and specify the ***starting_epoch*** and the ***checkpoint_path***. At the end of each notebook, there are also some code cells to visualize the prediction produced by the model.
- The above process also applies for ***train_frauas_3classes.ipynb*** and ***train_frauas_7classes.ipynb.*** Remember to change hardware accelerators to GPU. Users are also free to change whether to freeze the encoder or any layers of the model in the script. The weights will be saved respectively to
  /Drive/segmentation/weights_frauas_3classes/ and
  /Drive/segmentation/weights_frauas_7classes/.
- Since Google Colab will randomly shut down the training process if it cannot detect any mouse/keyboard action from the users, we had to use some Javascripts tricks to prevent the notebook from becoming idle. After opening the notebook file, please press ***Ctrl + Shift + I***, from which a Javascript code panel will display on the right hand side. Then, paste the following code in the box:

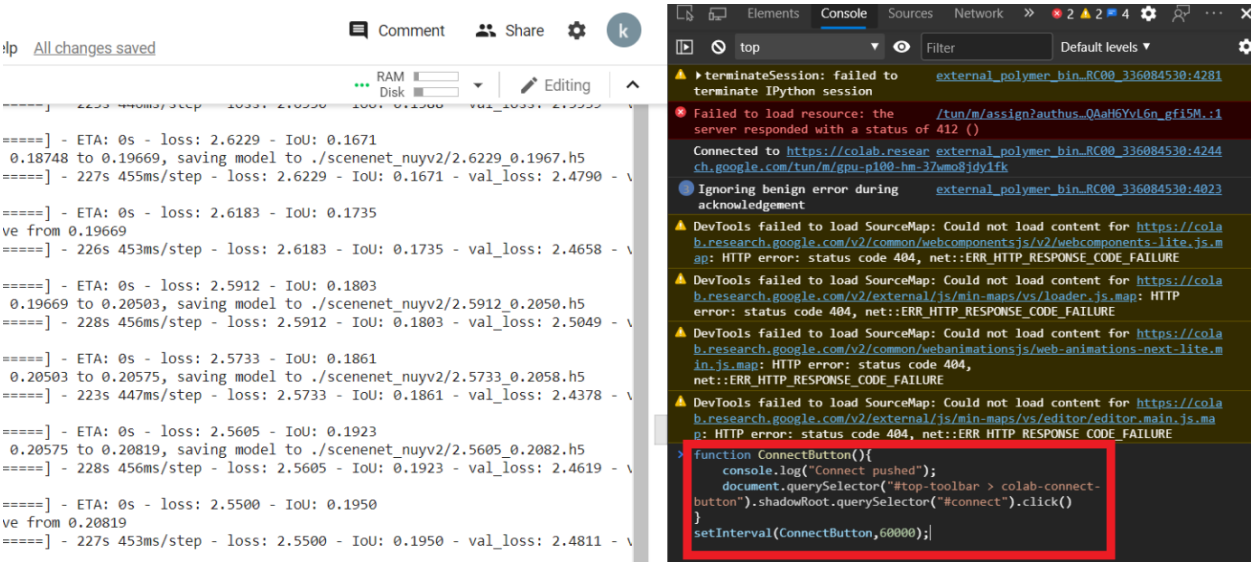


Figure 30: Javascript trick to prevent the notebook from shutting down

5.3. Deployment and Integration with ROS

Since the Google Colab machines are located in US, which would cause significant delay in uploading and downloading time if we were to host the whole inference pipeline and the model on Google Colab. Therefore, we had to shift our attention to machine run on Google Cloud. However, our attempts to get a GPU-powered on Google Cloud were somehow constantly rejected by Google Team. Consequently, our last resort was to switch to Amazon Cloud Service (AWS). Amazon finally granted us a NVIDIA Tesla T4 GPU, which is among the most efficient GPUs for production as well as among the rare GPU that is capable of saving processing time when running 16-bit float models, which we discussed later as a method to optimize the model's processing time. We first uploaded our model on to our Ubuntu OS, Deep Learning Image Virtual Machine (VM) hosted on AWS. Then, we run a server on the VM that listens for image frames sent from our client hosted at FRAUAS on our laptop attached on the Roswitha via TCP socket; produces semantic segmentation outputs from these received frames; and returns these predictions back to the client so that they can be processed and published to ROS. The communication via TCP sockets between the server and the client is handle by the ImageZMQ library. However, in order for TCP communication to work, the client and the server must know

each other public IP addresses. While the server's IP can easily be known since Amazon provided each VM a public IP's address, the client's public IP address cannot be determined since it is alias to the router's public IP address, we had to make use of ngrok services. Ngrok is a cross-platform application that enables developers to expose a local development server to the Internet with minimal effort. Ngrok helped us expose our local TCP port to the outside world via a URL, from which our server can listen and send data to:

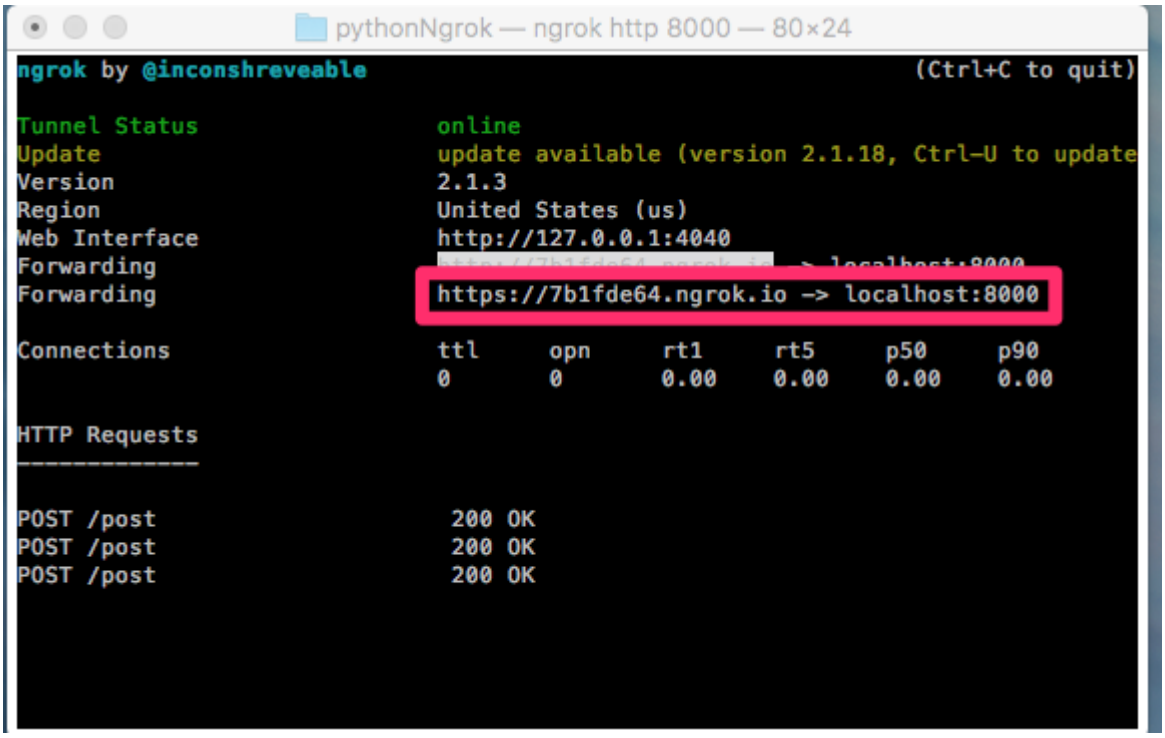


Figure 31: Ngrok terminal

In order to speed up the whole deployment process on the client side, as well as to take advantage of multiple CPU cores on the laptop, we used the ***multiprocessing*** library of Python to spawn many parallel processes across CPU:

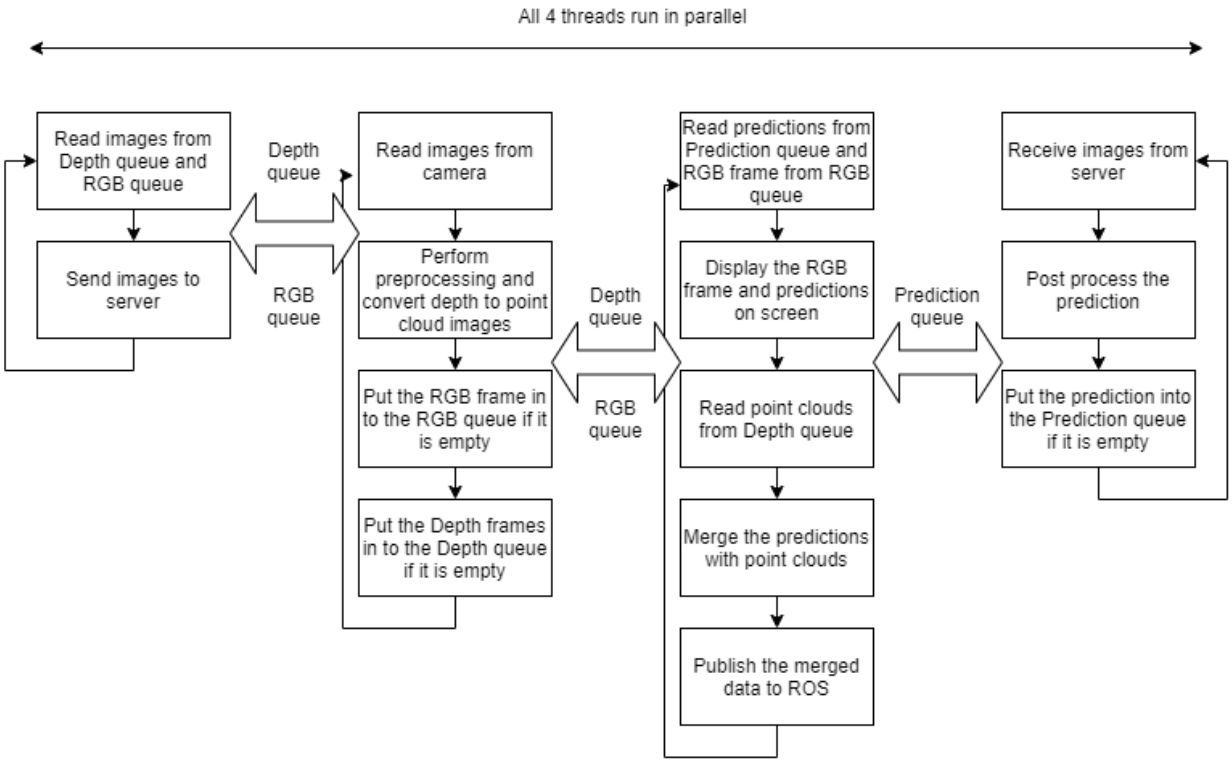


Figure 32: Client pipeline

These parallel threads can connect with each other via Queue objects. Unlike other programming language, the parallel running threads in Python do not share a common memory stack, thus, rather than saving the data to disk and then loading them back, we chose using Queue mechanism to reduce latency in communication and data transfer between 2 threads.

By parallelizing the whole pipeline, we have seen a 5x times boost up in speed in compared to sequential approach. On the other hand, we did not use multithreads on the server side since the server's pipeline is simple and the number of CPU of core available on the VM is small.

Also, as part of the segmentation map post-processing, we decided to retain only pixels that have the class probability higher than 50%, and pixels that do not satisfy this are marked as class *uncertain*. The reason behind this through experiments, we have realized that Depth camera tends to output the wrong point clouds coordinate of edges. At the same time, edges of objects are also regions the model predicts with low confidence. Therefore, by filtering out this regions, we could simultaneously effectively remove the data artifacts from the camera.

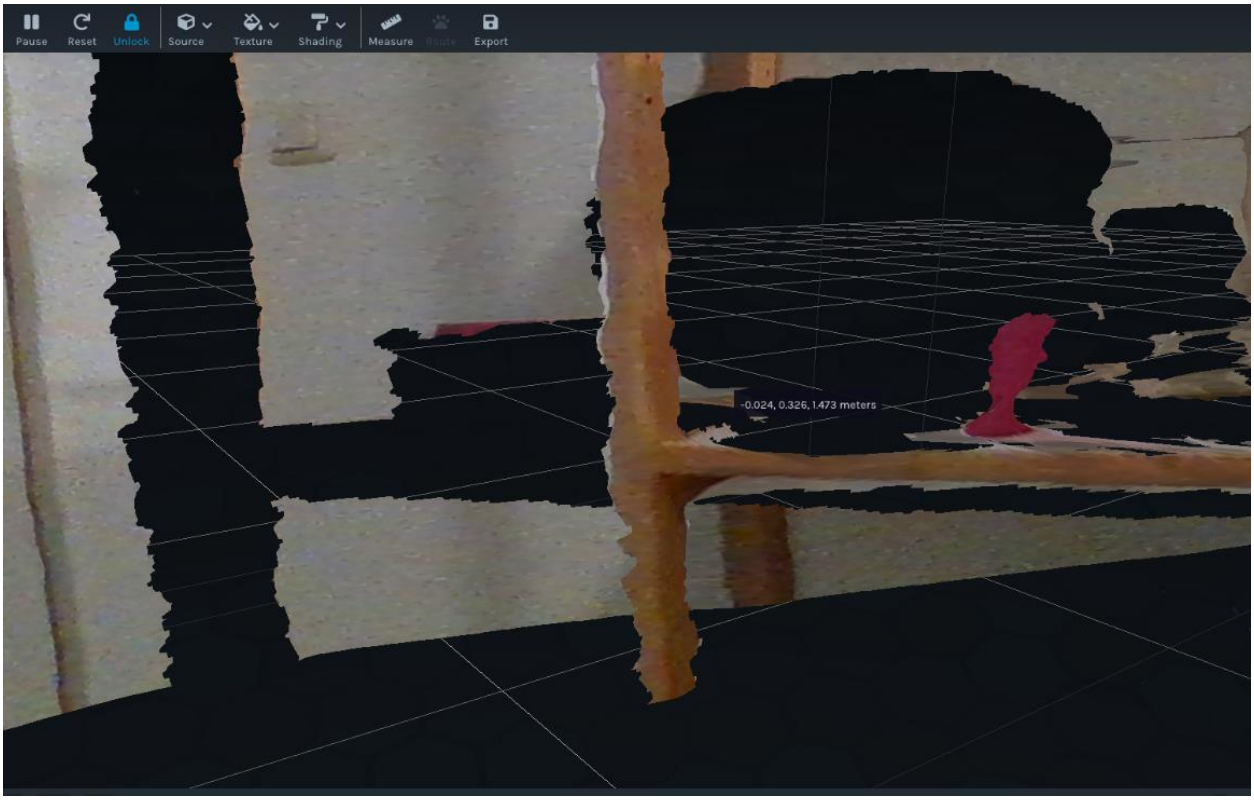


Figure 33: Depth artifacts around object's edges

Regarding the point clouds published to ROS, we only choose 10% of points that are within 1 meter from the floor and in between the range of 50cm and 4 meter in front of the camera. This is because RealSense camera works best in this distance range, and points outside this range are no longer accurate. Also, taking only 10% of satisfying points will speed up the publishing operation to ROS. Even though the number of points have been reduced, the number of retained points is still very large (~20000 points versus ~1000 points of Lidar), and the ROS's localization has no problem creating the cost map from those points.

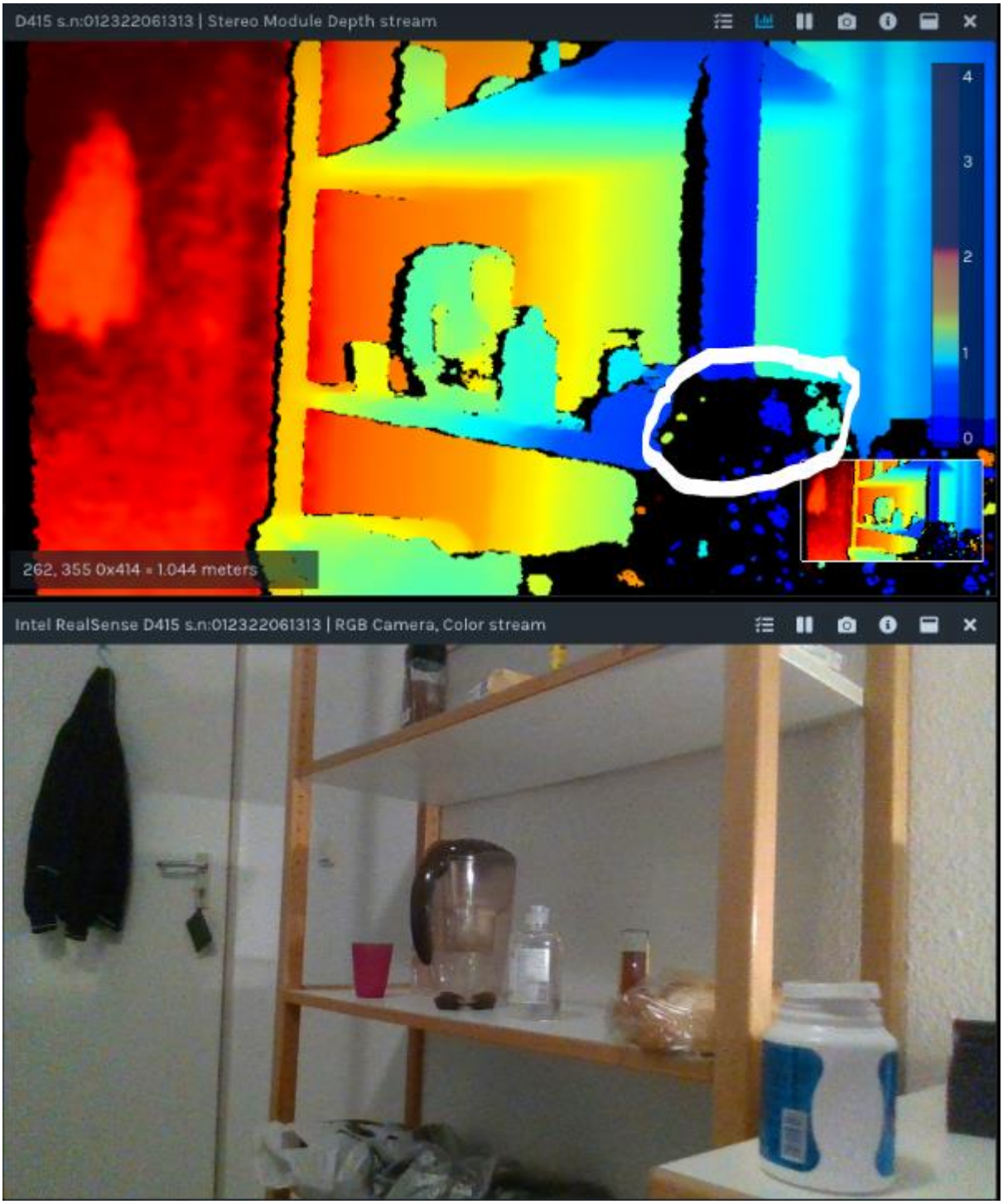


Figure 34: Depth artifacts of objects that are too close to the camera

Each point sent to ROS has the following structure: *(x_coordinate, z_coordinate, class number).* We omit *y_coordinate* part of the points clouds, since they are not necessary for creating the costmap as well as to reduce the publishing time. The class number will be represented as follow:

0 – Static obstacles, 1 – Free space, 2 – Human, 3 – Chair. Human and Chair are considered as dynamic obstacles in our test scenarios, while static obstacles are general obstacles, table, robot and backpack.

In addition, the point clouds image frame resolution is downsized 4 times on the client size before being upsized on the server back to the original resolution to reduce latency and due to the fact that point clouds images do not contain as much information as RGB frames so some loss of resolution is tolerable.

However, besides the optimization in coding aspects, we also performed post-training optimization on the model via NVIDIA TensorRT Software Development Kit, whose documentation can be found here https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html. NVIDIA TensorRT is a high-performance inference optimizer and runtime that can be used to perform inference in lower precision (16-bit float and 8-bit integer) on GPUs. TensorRT takes a trained network, which consists of a network definition or graph and a set of trained parameters, and produces a highly optimized runtime engine which performs inference for that network. TensorRT applies graph optimizations, layer fusion, among other optimizations, while also finding the fastest implementation of that model leveraging a diverse collection of highly optimized kernels. Since Tensor RT is also integrated in Tensorflow (TF-TRT), we took advantage of its ability to optimize the model graph and the model's weights to 16-bit (as well as the special Tensor Core technology of NVIDIA Tesla T4 GPU that allows for specialized matrix math to accelerate on Deep Learning in 16-bit mode), to speed up our model by approximately 300% (from 35 FPS to around 100 FPS on NVIDIA T4 GPU). Please be noted that TensorRT requires the optimization to be performed on the machine that the inference job is going to be performed (e.g. model optimized on V100 GPU and CUDA 10.0 will not work on T4 GPU or V100 GPU but with CUDA 11.0). Also, in order to achieve speedup for 16-bit model, it must be run on GPU that possesses Tensor Core technology, otherwise users will not see any speed improvement between a 32-bit model and a 16-bit model. All the codes regarding the server, client and the optimization can be found at ***~/segmentation/scripts/server.***py, ***~/segmentation/scripts/client.py*** and ***~/segmentation /scipts/optimization.py*** respectively.

Another point that is worth noting is that a further speedup (up to 200 FPS) can be achieved when running the optimization process in the NVIDIA Machine Learning Docker with pre-built TensorRT. We assumed such impressive speed up is due to the fact that the NVIDIA Docker (Docker is a container of pre-installed software) has been carefully installed and set up by NVIDIA team to get the most out of the software (e.g. Tensorflow only run fastest with the right combination of CUDA, CudNN and many other software dependencies). Despite the speedup in processing time, we still opted for running it without using Docker, since the loading of model optimized with TensorRT on Docker takes really long. Also since the TEB local planner package's processing speed is merely 10 HZ, running the model at 200 HZ instead of 100 HZ does not make any significant difference.

In order the replicate the whole deployment and ROS integration pipeline, please set up the AWS machine as follows:

- Create an account on Amazon Web Service, then go to Services and choose EC2, then choose Limits, choose option Running Instances of the filters. This step must be carried out in order to request for a CPU quota from AWS. Without the CPU quota, the user cannot hire a GPU machine from AWS. For T4 GPU, a quota of 4 CPU must be satisfied.
- Choose Running On-Demand All G Instances (NVIDIA T4 GPU are of G-family instance on AWS), then choose Request limit increase on the top right corner. A new tab will be displayed, then scroll down and choose Region to be Frankfurt, and New limit value to be 4 (or more), and declare the purpose of usage is educational.
- After 1 day, the Amazon team will respond to the quota increase request from the users. If it is granted, then please to proceed to create the GPU-powered Virtual Machine. Get back to the tab EC2, click Launch Instances, then search for Deep Learning Ubuntu 18.04 Version 34.0 and choose it. Then in the next tab, please filter the machine type to GPU, then choose the *g4dn.4xlarge* machine, which is the Tesla T4. Then go to the tab Configure Security Group, choose Add Rule, then choose Custom TCP rule, Port Range is 5555 (since the ImageZMQ will be listening on port 5555), Source is 0.0.0.0/0:

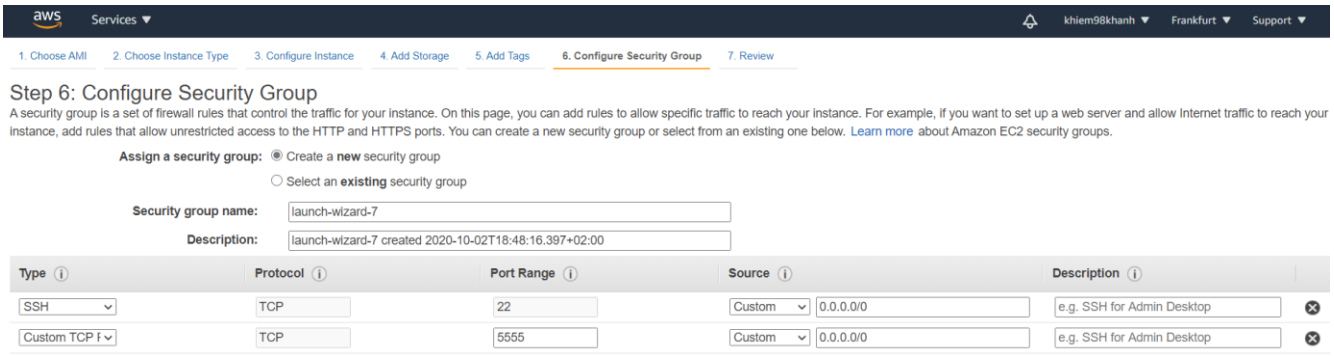


Figure 35: Security Setup for EC2 Machine

- Click Review and Launch. Now, a pop-up will display information regarding the SSH key .pem file to access to the virtual machine. Create one and download the ***<your .pem filename>.*** key to the folder ***~/segmentation/,*** make sure that no on have the access to the .pem key other than you. After that, fire up the terminal and go the folder storing the .pem key, and type ***chmod 0400 <your .pem filename>.***
- Now go back to the EC2 window, click on the tab Instances. If the instance is not running, then click on the instance name, click Actions -> Instance state -> Start Instance. Wait until the Instance State becomes the Running state, then click on the tab Details to see the Public IPv4 DNS. Please be noted that sometime, Amazon will sometimes inform that due to lack of GPU capacity within the region, users cannot launch their GPU instance. Therefore, please wait for a couple of minutes before launching the instance again.

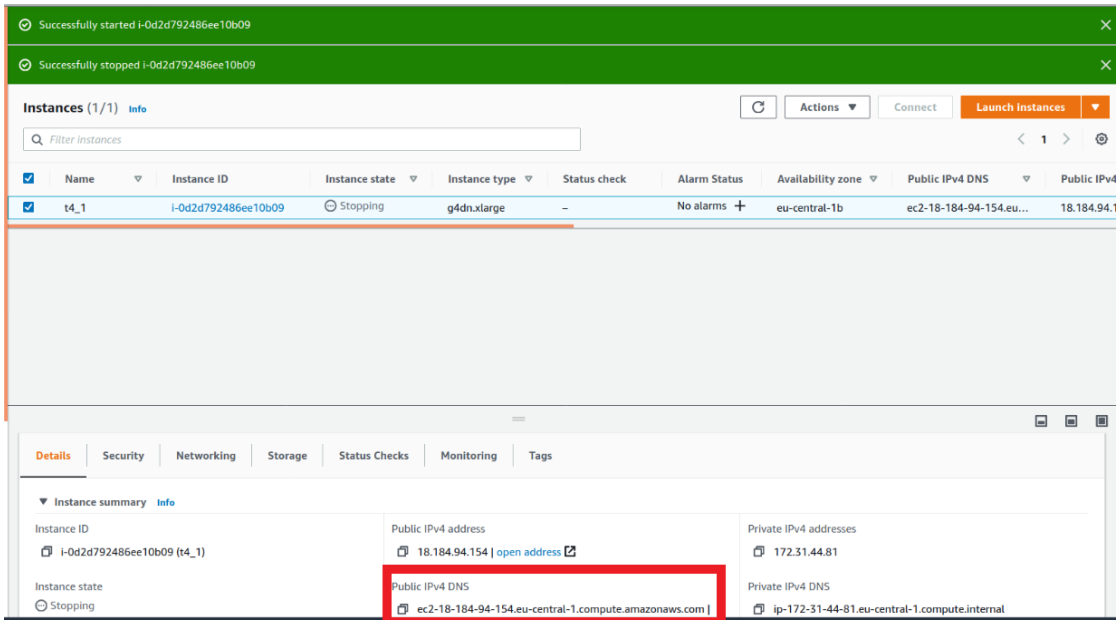


Figure 36: Public IPv4 DNS of AWS EC2 instance

- Now open a terminal, go to the ***~/segmentation/*** directory, then type ***ssh -i <your .pem filename> ubuntu@<your instance DNS>***. Remember to replace the .pem key name and the Public IPv4 DNS of the VM to the command's bracket.
- Further reference and instruction can be found here: https://aws.amazon.com/getting-started/hands-on/get-started-dlami/

Now that you already SSH into the virtual machine, please do the following steps to optimize the model on the VM (you may do this step only once):

- Open up a new terminal to copy the model checkpoint to the VM via the following command: ***scp -i <your .pem filename> ~/segmentation/weights/frauas_7classes.h5 ubuntu@<your instance DNS>:<~/segmentation/***
- Copy the optimization code to the VM via the command: ***scp -i <your .pem filename> > ~/segmentation/scripts/optimizer.py ubuntu@<your instance DNS>:<~/segmentation/***
- SSH into the VM
- Activate the Tensorflow environment via the command: ***source activate tensorflow2_latest_p37***, which for our case would be Tensorflow version 2.3

- Run the optimization script via the command: ***python3 ~/segmentation/optimization.py.*** The generated optimized 16-bit model will be generated at ***optimized_model_frauas_7classes*** folder.

Now that the optimized model has been generated, please do the following steps in order to activate the whole server-client-ROS pipeline. Make sure that ngrok has been installed on the local client side (we have installed all the package on the laptop), as well as a ngrok account has been created and the authentication token has been verified (more information can be found at: https://ngrok.com/download).

- Open a new terminal, then type ***ngrok tcp 5556 -region=eu*** in order to expose our client side's TCP port 5556. The terminal now will display the URL representing our *localhost:5556*, which should be taken note for later steps. This terminal should not be closed in order to maintain the ngrok connection.
- Copy the server code to the VM via the command: ***scp -i <your .pem filename> > ~/segmentation/scripts/server.py ubuntu@<your instance DNS>:<~/segmentation/***.
- SSH into the VM and run the server by typing: ***python3 server.py -client_url <ngrok URL>.*** After that, there will be some warnings from Tensorflow regarding the model loading, but please ignore it. Wait until the line "Server is ready" is displayed before proceeding with next steps.
- Open a new terminal and type ***roscore*** to start ROS.
- Open a new terminal and type ***python3 client.py -server_dns <server DNS>.*** Now that the connection between server and client has been established, the terminal will start printing the time taken each by thread. Also, the RGB frame and the segmentation map will also be displayed. The point clouds data stream now has been subscribed to ROS via the node **/segmentation/data**. Users can see the points cloud data publishing speed via the command ***rostopic hz /segmentation/data***.
- If somehow, there is a technical failure that leads to the disconnection between the client and the server, then please stop the server and restart it. Also, on the client side, open up a terminal and type **ps auxww | grep 'python3' | aws 'print {$2} | xargs kill -9** to kill all

Python Threads and Queue objects, then start the client again. The above kill command is a temporary solution for us to solve the problem of the threads and queue persisting to exist even after the Python program has exited. We still do not know the reason behind this bug, even after we have set the threads to be in daemon mode (daemon processes are processes that will automatically die when the main program exits).

- Remember to stop the VM once the task is finished, otherwise additional fee will be charged.

6. Results and Discussion

Our model trained on the SceneNet dataset achieve the following result:

Global and mean metrics		Per-class IoU													
Global Accuracy	Mean IoU	Background	Bed	Books	Ceiling	Chair	Floor	Furniture	Objects	Picture	Sofa	Table	TV	Wall	Window
0.73	0.39	0.92	0.36	0.00	0.69	0.34	0.66	0.22	0.4	0.4	0.05	0.33	0.25	0.67	0.20

Figure 37: Segmentation samples on validation set of SceneNet dataset

(top row: predictions, middle row: ground truth, last row: RGB image)

Our model pretrained with SceneNet dataset, and fine-tuned with the 3-class FRAUAS dataset achieve the follow results:

Global and Mean metrics		Per-class IoU		
Global Accuracy	Mean IoU	Obstacles	Free space	Human
0.982	0.909	0.9801	0.8503	0.8966

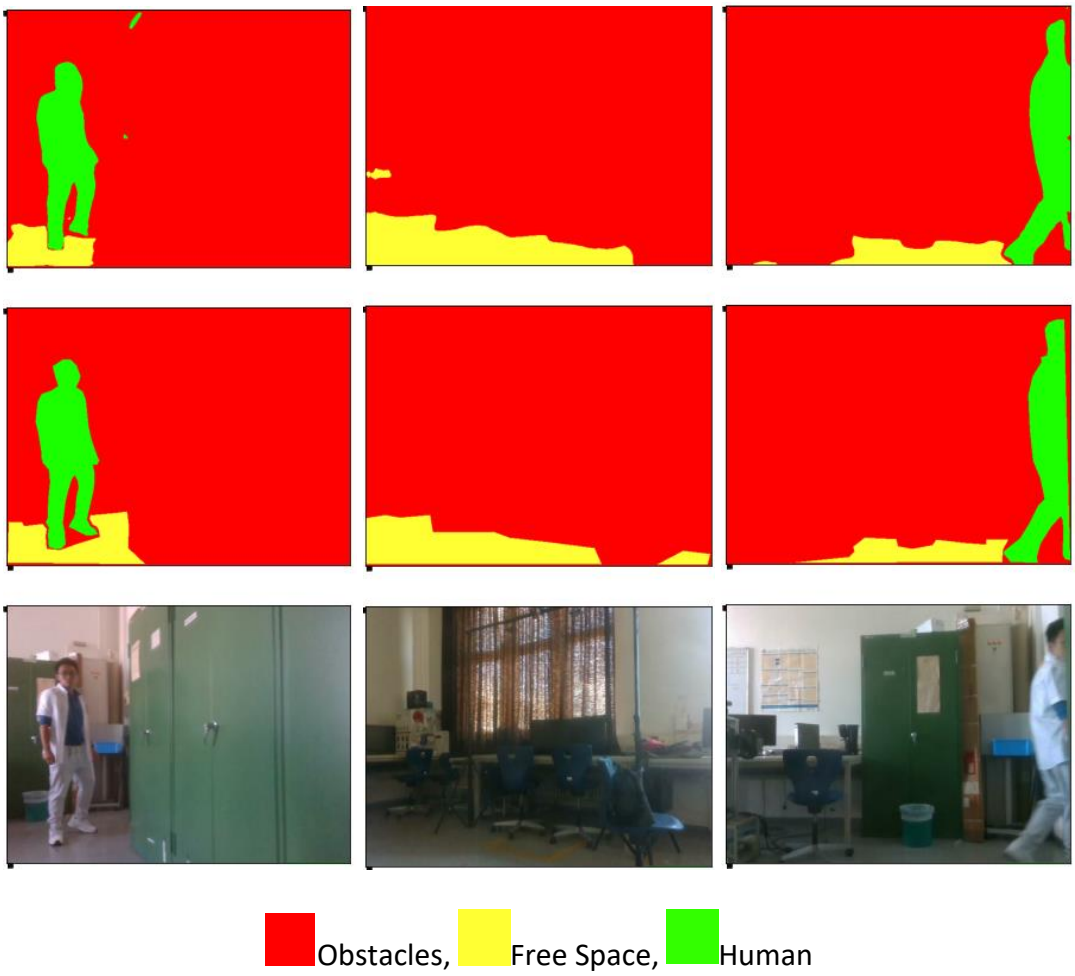
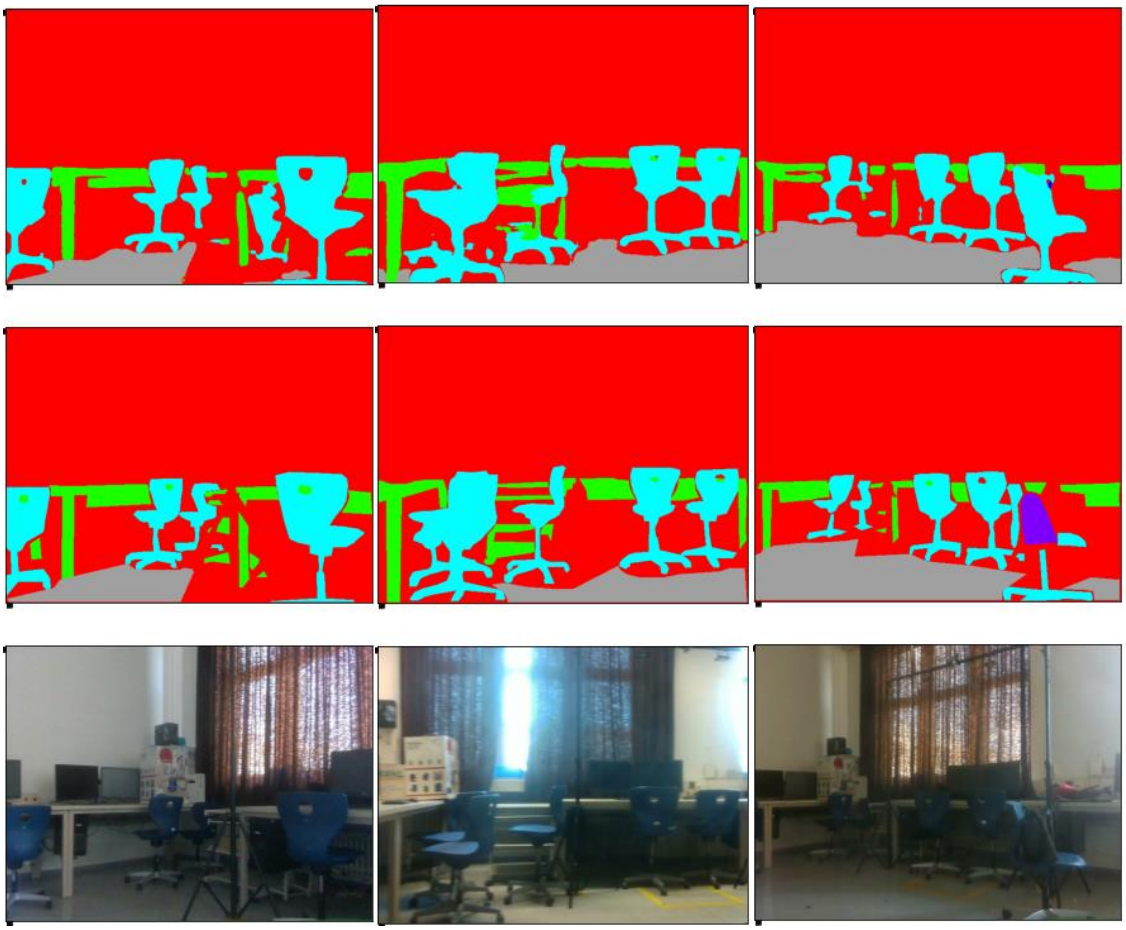


Figure 38: Segmentation samples from 3-class FRAUAS dataset

(top row: predictions, middle row: ground truth, last row: RGB image)

Our model pretrained with SceneNet and 3-class FRAUAS datasets, fine-tuned with the 7-class dataset achieve the following results:

Global and Mean metrics		Per-class IoU						
Global Accuracy	Mean IoU	Other Obstacles	Human	Table	Chair	Robot	Backpack	Free space
0.9564	0.7203	0.9506	0.8524	0.6628	0.7632	0.7085	0.2129	0.8914



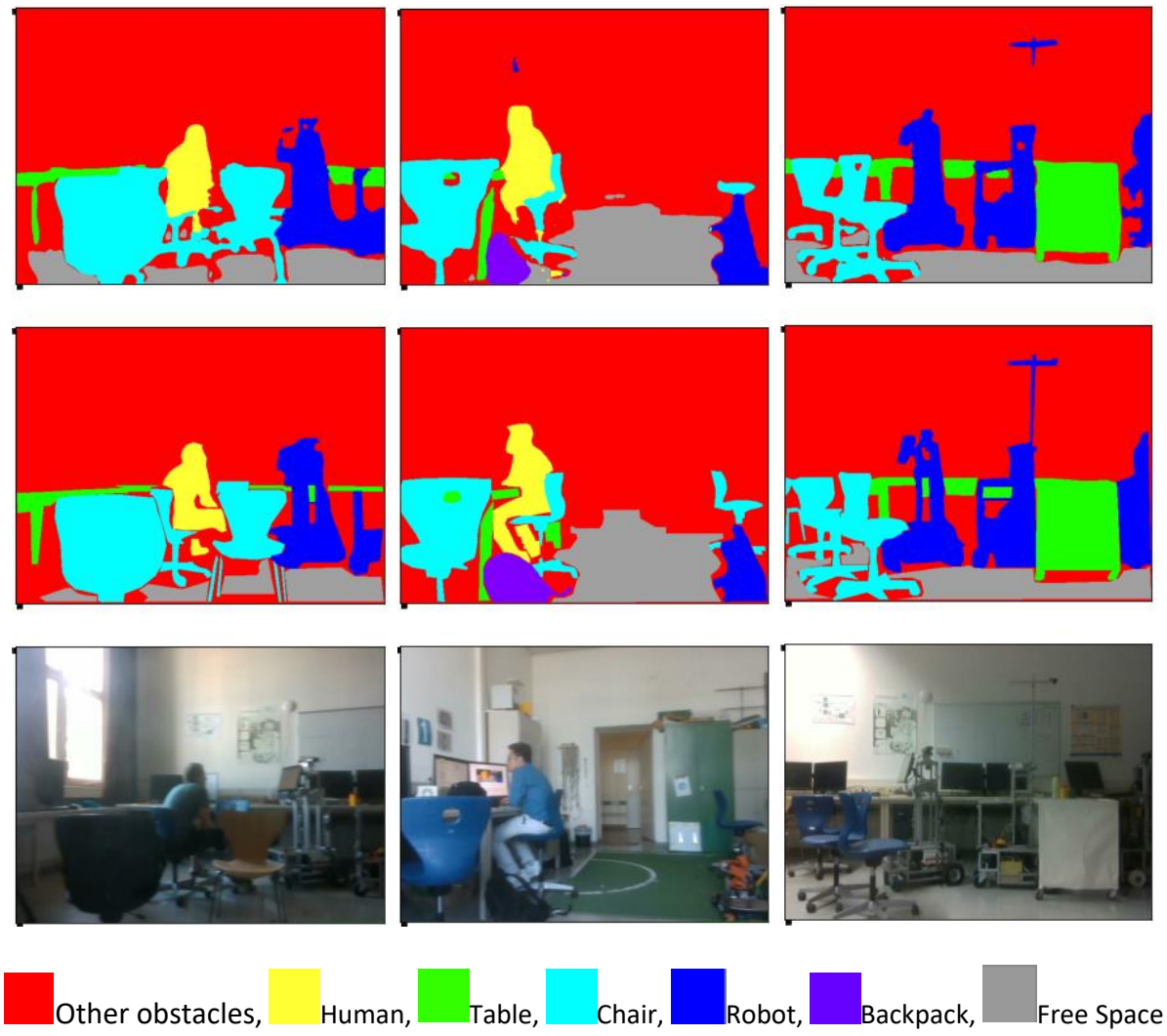


Figure 39: Segmentation samples from 7-class FRAUAS dataset

(top row: predictions, middle row: ground truth, last row: RGB image)

Our model segmentation output, when used to create the robot's local costmap, result in impressive detection of difficult-to-detect objects like table (The dark blobs of Lidar are in fact inflated since Lidar can only detect the 4 legs on a table). In some testing scenario, using our model merely is enough for the robots to navigate in indoor environment without the use of Lidar.

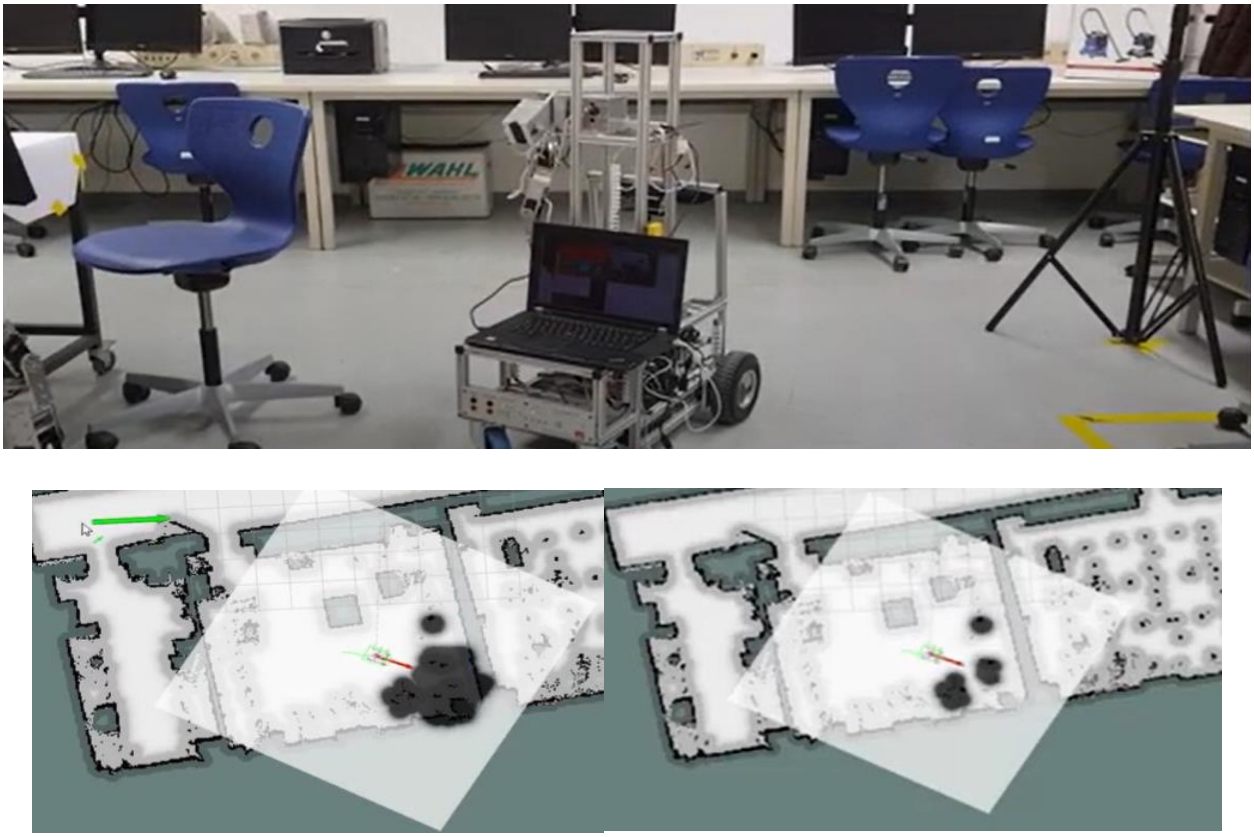


Figure 40: Costmap visualization for indoor testing scenario (bottom left: Costmap with Lidar and Segmentation, bottom right: Costmap with only Lidar)

We also filmed the real-world operation of the Roswitha robot running with navigation package TEB_local_planner, integrated with RGB-D semantic segmentation at: https://www.youtube.com/watch?v=pJ9bY4G6Ja0&fbclid=IwAR2yecYJXDhCOM1bsMiVRjhstZVjYOiec3a5Ldl7UZTCdJCSIimqmKjdTvQ&ab_channel=QuynhNguyenKhanh.

Our model's performance in extreme lightning condition is also very robust. For example, in the image below, due to direct sunlight, the color of the human's face cannot be differentiated from the color curtains, however, by relying on the information provided by depth camera, the human and the obstacles are detected correctly:

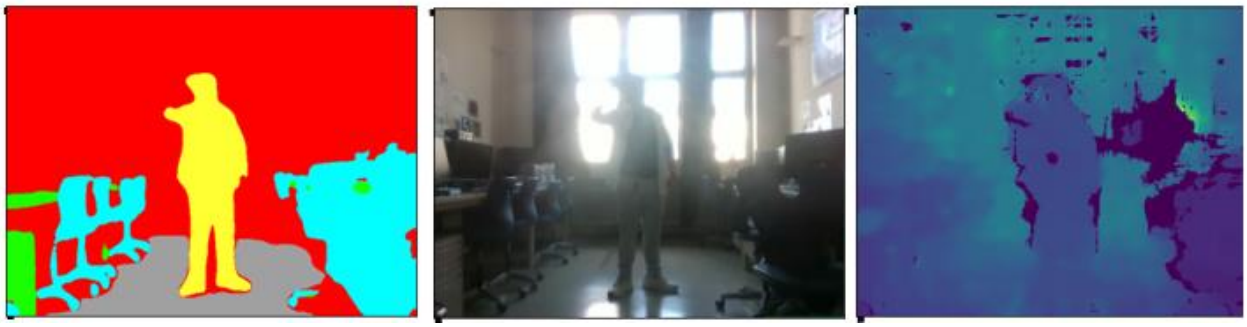


Figure 41: Semantic segmentation result in extreme lightning conditions (left: predictions, middle: RGB image, right: Depth image)

In order to visualize the effect of fusing the data from the Depth encoder and RGB encoder branches, we also presented here how Depth features has been incorporated to enhance RGB learnt features:

Figure 42: Visualization of features learnt from the model (left: fused RGB and Depth features, middle; RGB features, right: features learnt from Depth)

Notice the features located inside the red circle in the above figure. As can be seen in the middle figure, the region of human and the behind curtains are learnt similarly in the RGB encoder, however, in the right figure, the depth encoder noticed the difference between these 2 objects. Hence, by fusing the data between the 2 encoders, the Depth branch has corrected the mistakenly learnt features from the RGB branch (left figure).

However, there are also certain drawbacks in our approach:

- First of all, our model cannot perform well if the image is extremely blurred. This usually happens when the robot makes a turn. In order to cope with such limitation, further processing needed to be done on the ROS side, for example, to reduce the confidence on

the point cloud sent by the model when the robot is turning fast. Another method to overcome such limitation is to exploit the time component of the video stream. The model may rely more on information on previous frames to infer on the current blurred frames. Some researchers have tried to incorporate optical flow [26] or Long Short Term Memory (LSTM) network into the CNN model [27] to capture the relationship between adjacent frames. However, due to the limitation of time and the complexity of such methods, we only used the temporary solution on the ROS side.

- Secondly, there is a bottleneck when uploading images to the server and downloading the segmentation maps from the server. This is due to the limitation of the Internet bandwidth, as well as due to the nature of the problem of not having a local GPU machine set up at the lab. We tried some testing with NVIDIA Jetson Nano, which is an embedded device equipped with a GPU. However, the Jetson Nano is the weakest board in the NVIDIA Jetson family. Jetson Nano only have a shared 4GB RAM (GPU and CPU have to share RAM with each other), and a low-end GPU. With such low computing ability of the Nano, we failed to optimize our model on it as the memory keeps getting overflown, while running an un-optimized model on the Nano is as slow as running it on a laptop. Therefore, if the deployment pipeline's speed is required to be higher than 15 FPS, then a local GPU or an Internet connection whose uploading speed is higher than 200Mbit/s is required.
- Thirdly, our model currently does not react well to extremely close objects to camera. This is due to the depth coming from close-range objects being extremely noisy, and our training dataset not containing many close-range objects samples.

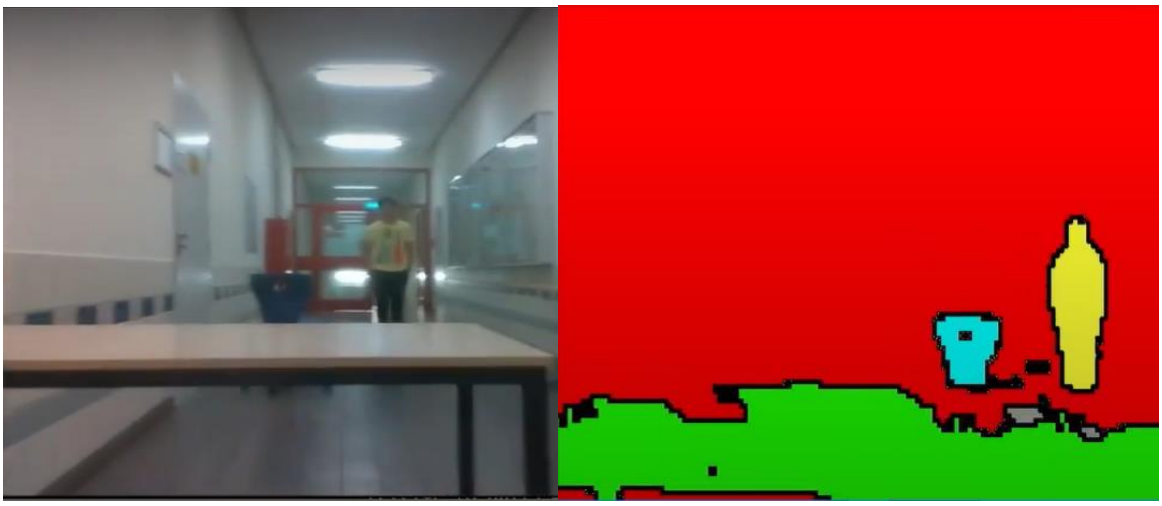


Figure 43: Segmentation result for extremely close-range objects

7. Future development

One of a possible direction to further develop this project is to convert the original image segmentation problem into a video segmentation problem. Video segmentation problem will help the model become more robust to blur image and fast-moving scenes and objects as it can incorporate the time-series features of the data into the learning process. Also, learning images as a video will enhance the stability of the prediction, since 2 adjacent frames tend to not differ very much, thus reduce the outliers in the predictions of the model.

Another possible direction to increase robustness of the model would be enriching the real-world indoor dataset used to train and validate the model. Since our collected and labeled dataset is relatively small, collecting more data would definitely make the model robust to different environments and the model's performance evaluation result will be more trustworthy.

8. Conclusion

To conclude, this thesis has incorporated the scene understanding capability of a RGB-D semantic segmentation Deep Learning model to enhance the robustness of an indoor local planning algorithm. Different indoor navigation scenarios and difficult obstacles that conventional Lidar cannot detect has been tested to verify the functionality of the whole system. Also, a new architecture of RGB-D semantic segmentation has been presented in order to fully exploit the information from the RGB and Depth visual data as well as to satisfy the requirement of real-time processing speed and to compensate for the lack of indoor training dataset and local hardware resources. Due to the lack of local hardware, 2 pipelines have also been carefully explained in this report, one for training on the Google Colaboratory service, and one for inference stage on Amazon Web Service. Last but not least, a semantic dataset around FRAUAS robotics lab has been collected and labelled, from which future works can make use of. Three pretrained models with SceneNet and custom FRAUAS datasets are also available for further development in the future.

Bibliography

[1] *Ride in NVIDIA's Self-Driving Car.* [Film]. USA: Youtube, 2019.

[2] F. Fooladgar and S. Kasaei, "A survey on indoor RGB-D semantic segmentation: from hand-crafted features to deep convolutional neural networks," Multimedia Tools and Applications, May 2019.

[3] C. Hazirbas, L . Ma, C. Domokos and D. Cremers, "FuseNet: Incorporating Depth into Semantic Segmentation via Fusion-Based CNN Architecture," in *Asian Conference on Computer Vision*, 2016.

[4] F. Fooladgar and S. Kasaei, "Multi-Modal Attention-based Fusion Model for Semantic Segmentation of RGB-Depth Images," Dec 2019.

[5] X. Hu, K. Yang, L. Fei and K. Wang, "ACNet: Attention Based Network to Exploit Complementary Features for RGBD Semantic Segmentation," IEEE International Conference on Image Processing, Taipei, Sep 2019.

[6] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016.

[7] P. Chao, C-Y. Kao, Y-S. Ruan, C-H. Huang and Y-L. Lin, "HarDNet: A Low Memory Traffic Network," in *International Conference on Computer Vision*, 2019.

[8] S. Jégou, M. Drozdzal, D. Vazquez, A. Romero and Y. Bengio, "The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017.

[9] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *ICML*, 2019.

[10] J. Hu, L. Shen, S. Albanie, G. Sun and E. Wu, "Squeeze-and-Excitation Networks," in *Conference on Computer Vision and Pattern Recognition*, 2017.

[11] N. Silberman, P. Kohli, D. Hoiem and R. Fergus, "Indoor Segmentation and Support Inference from RGBD Images," in *European Conference on Computer Vision*, 2012.

[12] U. Neumann and W. Wang , "Depth-aware CNN for RGB-D Segmentation," in *European Conference on Computer Vision*, 2018.

[13] J. Long, E. Shelhamer and T. Darrel, "Fully Convolutional Networks for Semantic Segmentation," in *Conference on Computer Vision and Pattern Recognition*, 2015.

[14] D. Eigen and R. Fergus, "Predicting Depth, Surface Normals and Semantic Labels," in *International Conference on Computer Vision*, 2015.

[15] Cremers, C. Hazirbas and L. Ma and C. Domokos and D., "Github," Technische Universitat Munich, [Online]. Available: https://github.com/tum-vision/fusenet. [Accessed 9 October 2020].

[16] B. Kang, Y. Lee, and T. Q. Nguyen, "Depth Adaptive Deep Neural Network," *IEEE Transactions on Multimedia,* vol. 20, no. 9, pp. 2478 - 2490, 2018.

[17] D. Lin, G. Chen D. Cohen-Or, P-A. Heng and Hui Huang, "Cascaded Feature Network for Semantic Segmentation of RGB-D Images," in *International Conference on Computer Vision*, 2017.

[18] F. Fooladgar and S. Kasaei, "3M2RNet: Multi-Modal Multi-Resolution Refinement Network for Semantic Segmentation," in *Advances in Computer Vision*, 2019.

[19] S. Gupta, R. Girschick, P. Arbelaez and J. Malik, "Learning Rich Features from RGB-D Images for," in *European Conference on Computer Vision*, 2014.

[20] S. I. Nikolenko, "Synthetic Data for Deep Learning," 2019.

[21] A. Handa, V. Patraucean, V. Badrinarayanan, S. Stent and R. Cipolla, "SceneNet: Understanding Real World Indoor Scenes With Synthetic Data," 2015.

[22] J. Deng, W. Dong, R. Socher, L-J. Li, K. Li and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[23] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Tensor_Processing_Unit. [Accessed September 2020].

[24] "Tensorflow," Google , [Online]. Available: https://www.tensorflow.org/guide/data_performance#caching. [Accessed 30 September 2020].

[25] A. Galloway, "Angusg," [Online]. Available: https://angusg.com/blog/2016/optimizing-iou-semantic-segmentation/. [Accessed 1 October 2020].

[26] M. Ding, Z. Wang, B. Zhou, J. Shi, Z. Lu and P. Luo, "Every Frame Counts: Joint Learning of Video Segmentation and Optical Flow," in *Association for the Advancement of Artificial Intelligence*, 2020.

[27] A. Pfeuffer, K. Schulz and K. Dietmayer, "Semantic Segmentation of Video Sequences with Convolutional LSTMs," *IEEE Intelligent Vehicles Symposium,* 2019.

[28] I. Ulku and E. Akagunduz, "A survey on deep learning - based architectures for semantic segmentation on 2D images," 2020.