POLITECNICO DI TORINO

Master Degree in
Mechatronic Engineering

Master Degree Thesis

Semantic Scene Segmentation for Indoor Robot Navigation

Supervisor
Prof. Marcello Chiaberge

Candidate
Daniele Cotrufo

2021-2022

A mia madre Mercedes

Summary

Scene Segmentation is an important component for robots which are required to navigate in an indoor environment. Obstacle avoidance is the task of detecting and avoiding obstacles and represents a hot topic for autonomous robots. To obtain a collision free motion, a robust module for obstacle detection is needed. The objective of this thesis is to make a robot able to navigate autonomously, relying only on visual perception, performing real-time segmentation of the indoor scene. In accordance with the state of the art, the proposed method is based on a Deep Learning model for Semantic Scene Segmentation. A Pyramid Scene Parsing (PSP) Net with a ResNet-34 as a backbone is chosen as a model to train. At first, the backbone has been pre-trained on ImageNet dataset, then, maintaining these weights fixed, the PSP Net is trained on the labeled dataset for semantic segmentation. In order to detect the viable part of the scene with high robustness, binary segmentation is chosen, so pixels are labeled as floor (1) or not floor (0). A 91% Intersection over Unit (IoU) score is achieved on the test set with this approach. Once the image is correctly segmented, a post-processing is applied, in order to obtain a "pixel-goal" for navigation purposes. Navigation is performed through a proportional controller which links the steering angle with the coordinates of the pixel-goal. The linear velocity is handled by the navigation algorithm too. A ROS2 net with a segmentation and a navigation node is built. The model and the ROS2 net are then deployed on a Jetson AGX Xavier platform and the pipeline is tested on a Turtlebot3 robot. The coefficient of the proportional control is tuned directly with real world tries and multiple tests are performed to analyze the performance, mainly from a qualitative point of view. The best results are achieved in corridor scenarios, with the robot able to avoid obstacles along its path, while staying far enough from the walls. In situation with multiple objects with more complex shapes, such as people and chairs in an office, performances are worse, but the robot still often exploit obstacle avoidance correctly.

Acknowledgements

This thesis would not be the same without the generosity and contribution of a large number of people. I am very grateful to my supervisor at polytechnic university of Turin, Professor Marcello Chiaberge, I appreciate very much his suggestions and the time he spent to support me, helping to achieve this important milestone. Special thanks to my tutor Simone and all the guys of PIC4SeR who often gave me guidance regarding many practical issues. I am also very grateful to all my friends and colleagues of the PIC4SeR team who were always supportive. Finally, last but not least, I would like to express my greatest gratitude to my family and also my girlfriend for their continuous support.

Contents

List of Tables 6

List of Figures 7

I BACKGROUND 9

1 MACHINE LEARNING AND DEEP LEARNING 10
1.1 Machine Learning . . . 10
1.2 Neural Networks . . . 11
1.2.1 Perceptron and Multi-Layer Perceptron (MLP) . . . 11
1.2.2 ReLU Function . . . 13
1.2.3 Sigmoid Function . . . 13
1.2.4 Softmax Function . . . 14
1.3 Deep Learning . . . 14
1.4 Convolution Neural Networks . . . 15
1.4.1 CNN Architecture . . . 15
1.4.2 Transfer Learning . . . 17

2 ARCHITECTURES FOR IMAGE UNDERSTANDING 18
2.1 Architectures for Image Classification . . . 18
2.1.1 LeNet . . . 18
2.1.2 AlexNet . . . 19
2.1.3 VGG Net . . . 19
2.1.4 ResNet . . . 20
2.2 Architectures for Segmentation . . . 24
2.2.1 Fully Convolutional Networks . . . 26
2.2.2 Convolutional Models with Graphical Models . . . 26
2.2.3 Models with Encoder-Decoder Structures . . . 27
2.2.4 Pyramid Network-Based Models . . . 29
2.2.5 Other Models . . . 30
2.3 Metrics for Classification . . . 33
2.3.1 Accuracy . . . 33
2.3.2 Other Metrics . . . 33
2.4 Metrics for Image Segmenation . . . 34
2.4.1 Pixel Accuracy . . . 34
2.4.2 Intersection over Union (IoU) . . . 35
2.4.3 F-Score . . . 36
2.5 Loss Functions . . . 36

2.6 Datasets 37

II WORK DESCRIPTION 40

3 PRELIMINARY STEPS 41
3.1 Hardware Presentation 41
3.1.1 Jetson AGX Xavier 41
3.1.2 TurtleBot3 44
3.1.3 Workstation 44
3.2 TensorFlow 45
3.3 State of the Art 45

4 SCENE SEGMENTATION 53
4.1 NVIDIA SDK Isaac 53
4.1.1 Data Collection 54
4.1.2 Training 55
4.1.3 Results and Conclusions 56
4.2 Working Approach 57
4.3 Dataset 58
4.4 Models 58
4.5 Training 62
4.6 Results 66
4.6.1 UNet 66
4.6.2 LinkNet 68
4.6.3 FPN 68
4.6.4 PSPNet 69

5 NAVIGATION 72
5.1 Post-Processing 73
5.1.1 Middle-Point 74
5.1.2 Waypoints 76
5.2 ROS Integration 78
5.2.1 Scene Segmentation Node 78
5.2.2 Navigation Node 79
5.3 Implementation 80
5.4 Final Test 82

6 CONCLUSION 85

List of Tables

2.1 LinkNet performance with Res18 backbone compared with other models . 29
2.2 Results of PSPNet in terms of performances with different versions of ResNet 31
3.1 Training parameters. 47
3.2 Metrics results on test set. 48
3.3 Memory and time performance. 48
3.4 Number of acquisitions on the base of light conditions 49
3.5 Test resutls. 50
3.6 Hyperparameters configuration. 51
3.7 Test results. 52
4.1 Hyperparameters values of the training process 56
4.2 U-Net training results. 67
4.3 LinkNet training results. 68
4.4 FPN training results. 69
4.5 PSPNet training results.. 70
5.1 Corridor test recap 84

List of Figures

1.1 The Machine Learning approach. . . . 11
1.2 Rectified Linear Unit Function (ReLU) graph. . . . 13
1.3 Sigmoid Function graph. . . . 13
1.4 Softmax Function graph. . . . 14
1.5 Convolution Neural Network architecture. . . . 16
2.1 AlexNet architecture. . . . 20
2.2 Architecture of a VGG16. . . . 21
2.3 ResNet residual block. . . . 21
2.4 ResNet architecture. . . . 23
2.5 Preditcion time of different ResNet architectures on two big datasets. . . . 23
2.6 Training and test accuracy of different ResNet architectures on two big datasets. . . . 24
2.7 Semantic and instance segmentation example. . . . 25
2.8 Segmentation models timeline. . . . 26
2.9 U-Net architecture. . . . 28
2.10 LinkNet architecture. . . . 29
2.11 PSP-Net architecture. . . . 30
2.12 Intersection over Union (IoU) - Graphical representation. . . . 35
3.1 NVIDIA Jetson AGX Xavier shape. . . . 42
3.2 NVIDIA Jetson AGX Xavier architecture. . . . 42
3.3 NVIDIA Jetson AGX Xavier Module Interface. . . . 43
3.4 NVIDIA Jetson AGX Xavier Specifications. . . . 43
3.5 Qualitative examples of predicted images. . . . 47
3.6 Metrics vs memory consumption models comparison. . . . 49
3.7 Example of inference. . . . 51
4.1 Warehouse top view. . . . 55
4.2 Example of acquired image and label. . . . 55
4.3 Isaac inference result. . . . 56
4.4 Dataset inspection. . . . 59
4.5 Learning curve example. . . . 66
4.6 Image prediction examples. . . . 66
4.7 U-Net and PSPNet inference comparison 70
4.8 U-Net and PSPNet Learning Curve comparison 71
5.1 Middle path estimation. . . . 72
5.2 Example of bad detection when an obstacle is placed in the middle of the path. . . . 73
5.3 Segmented Images. . . . 73
5.4 Image1 with meanrow function applied. . . . 75
5.5 Image2 with meanrow function applied. . . . 76

5.6 Image3 with meanrow function applied. 76
5.7 Waypoints calculation. 78
5.8 Robot front view. 81
5.9 Robot lateral view. 81
5.10 Robot during navigation. 82
5.11 Test schemes. 83
5.12 AMR test structure. 84

Part I

BACKGROUND

Chapter 1

MACHINE LEARNING AND DEEP LEARNING

In the last decades the importance of having technologies able to solve problems and tasks increasingly complex is growing in the Artificial Intelligence field. Machine Learning is a technique developed to reach the objective of solving problems which are typical of the human mind. Robotics, defined as the engineering field which studies and develops systems and methods with the purpose of making robots reproducing automatically these typical human tasks, highly draws from Machine Learning and Deep Learning.

1.1 Machine Learning

Machine Learning (ML) is the research field and science of programming computers so they can learn from data. Two famous definitions are:

- "Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed." [Arthur Samuel]
- "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [Tom Mitchell]

A system with these features often gives results with high accuracy, that's why Machine Learning is rapidly becoming a topic of high interest for the Artificial Intelligence field, Gèron [2019]. The power of a ML approach is the possibility to get results without writing rules but make the machine able to get the rules from data, that's why in fields where building the rules is very difficult, such as speech recognition, images segmentation and object classification, is an essential technology.[Fig. 1.1]

In general, there are three types of Machine Learning: Supervised Learning, Unsupervised Learning, Reinforcement Learning.

In supervised learning, a dataset is given, and it's known what the correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, the objective is to predict results within a continuous output, so trying to map input variables to some continuous function. In a classification problem, the goal is to predict results in a discrete output. In other words, trying to map input variables into discrete categories.

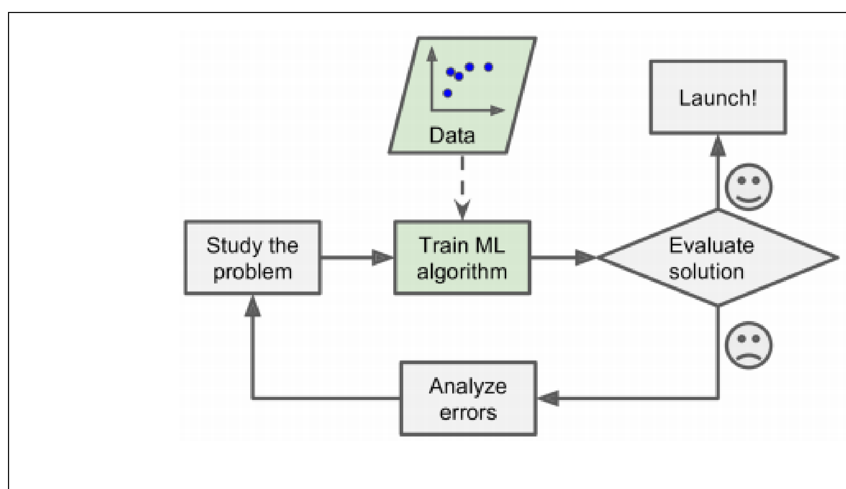

Figure 1.1. The Machine Learning approach.

Unsupervised learning allows us to approach problems with little or no idea what results should look like. We can derive structure from data where it is not necessarily known the effect of the variables. This structure can be derived by clustering the data based on relationships among the variables in the data. With unsupervised learning, there is no feedback based on the prediction results. Reinforcement learning is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment to maximize some notion of cumulative reward.

1.2 Neural Networks

There are various mathematical models used in Machine Learning problems, such as Naïve Bayes Classifier, Support vector Machine, decision trees and multiple types of regressors. The choice of the proper model to use is always based on the problems that need to be solved. A very powerful and complex branch of Machine Learning models is represented by Neural Networks. Artificial Neural Networks are mathematical models that are inspired by the biological structure of the human brain. These models are composed of multiple layers, with each layer connected with the previous one and the next one. Output data of a layer is used as input by the following one. Nodes (or neurons) are the basic part of each layer.

1.2.1 Perceptron and Multi-Layer Perceptron (MLP)

The simplest model of Artificial Neural Network is represented by the Rosenblatt perceptron: it is a classification model which receives as input a certain number d of elements x_1, $x_2 \dots x_d$ to give a binary output 0 if $\sum(w_i * x_i)$ is below the threshold, 1 otherwise. In the Rosenblatt model we have only one input layer with a number of neurons equal to d and an output layer.

The following steps define the learning algorithm of the model:

- At the start (t=0) $w_i(0)$ and $b_i(0)$ are defined for i=1...d
- An input vector X with its corresponding output Y is given
- The output is calculated with the former formula

- The weights are updated, based on the difference between the real output (y) and the calculated one (a), according to this formula: $w_i(t+1) = w_i(t) + \eta * (y - a(t)) * x_i(t)$. η is a tuning positive parameter, between 0 and 1.

The steps are iterated until enough elements are classified correctly or when the maximum number of cycles is reached. According to Rosenblatt, the model can classify correctly all the elements only if they are linearly separable.

It's clear that the former model has some big limitations, that's why a more complex one was created: the Multi-Layer Perceptron (MLP). Basically, there is the same structure of the Rosenblatt model but with a higher number of layers. Each node of the input layer is connected to every neuron of the following layer: fully connected layers. Moreover, to handle non-linear problems, each node has a non-linear activation function to calculate its output. For each couple of layers, the formula is the following:

$$a_{lj} = \sigma(\sum(w_{ljk} * a_{l-1k} + b_{lj}))$$

Where:

- w_{ljk} is the weight for the link between the k-th node of the l-1-th layer and the k node of the l-th layer
- b_{lj} is the j-th bias element of the l-th layer
- a_{lj} is the j-th element of the l-th node
- σ is the activation function

The MLP gives better results for problems where data are not linearly separable but are equivalent to the single-layer structure in case of data which can be separated by a hyperplane. The learning algorithm of the Multi-Layer Perceptron has the same steps of the former model. There are some input vectors with the corresponding correct output that are given to the model. At each iteration, the weights of the nodes are updated with the aim of minimizing the error that the net can commit with its activity. For the training of an Artificial Neural Network, the different methods previously analyzed can be used: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. The limits of the MLP are related to images as input data.

To better understand this, let's analyze the structure of a MLP for a classic problem of image classification on the MNIST dataset. The MNIST dataset is composed of 28x28 grayscale images (pixel values between 0 and 256) of numbers from 0 to 9. Images must be given as arrays to the net, so 784 neurons (784 is defined as the number of features) for the input layer are needed, then there is a hidden layer and 10 neurons for the output one. The problems emerge when there are images with bigger size: for a 128x128 RGB image dataset 49.125 features are present, so the number of neurons (and so the complexity of the structure) becomes much bigger as the size of the images grows. Moreover, this kind of Neural Network is not invariant to rotations, translation, and distortions of the images. To handle these issues, networks with different structures are considered, with a different approach for the feature extraction part: these are known as *Convolutional Neural Network* which will be analyzed in the next chapter.

The choice of the activation function can be crucial in terms of convergence of the net; the main function we can find are:

- Step Function
- Sigmoid Functions (Logistic Function and Hyperbolic Tangent)
- Rectified Linear Unit (ReLU) Function

Since 2017, Agarap [Feb. 2019], ReLU is the most popular activation function and the only one that will be found in the models present in this thesis.

1.2.2 ReLU Function

The Rectified Linear Unit Function (ReLU) is defined as following:

$$\sigma(z) = max\{0, z\}$$

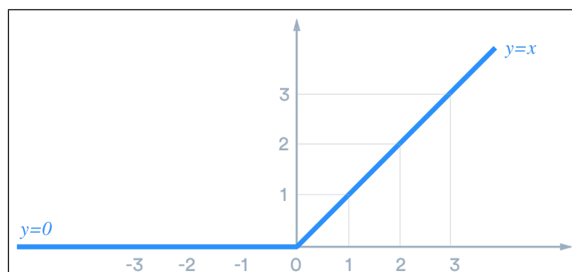

Figure 1.2. Rectified Linear Unit Function (ReLU) graph.

The function gives positive output values for positive input values, whereas 0 for negative inputs. The first reason ReLU has so wide use today is because, differently from other activation functions, it has the derivative which does not converge too fast to 0. Avoiding this is, in general, very good, considering that the training of a neural network is based on a gradient descent algorithm: if there are multiplications for values close to zero, deep layers have slow learning. Moreover, both this function and its derivative have a simple structure, which helps in case of training of a very deep network, limiting power consumption.

1.2.3 Sigmoid Function

The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

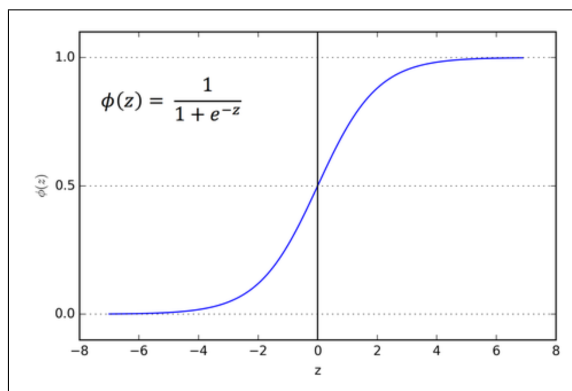

Figure 1.3. Sigmoid Function graph.

This function can assume values between 0 and 1. As z increases, sigma goes closer to 1. The biggest use of sigmoid function in Artificial Neural Network occurs in the output layer Gèron [2019], in case of 2 classes classification problems. When the output of the sigmoid tends to 1, output is classified as part of a class, otherwise if it is closer to 0, the label output is the other class.

1.2.4 Softmax Function

The softmax function is defined as following:

$$\sigma(z) = \frac{e^{z*j}}{\sum e^{z*k}}$$

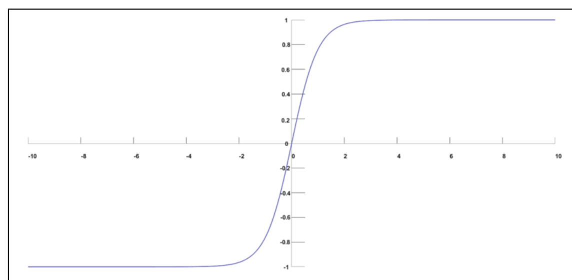

Figure 1.4. Softmax Function graph.

When there are problems with 2 or more classes, the sigmoid function is not sufficient anymore. In these cases, the activation function of the output net layer is chosen as the softmax. Considering a number d of output classes, what this function allows is to have d probabilistic values, with each representing the probability of being in the j-th of the d classes.

1.3 Deep Learning

When a Neural Network has two or more hierarchical hidden layers, it is defined as a *Deep Neural Network (DNN)* model. More precisely, Deep Learning is defined as a class of algorithms having these features:

- they use various levels of cascaded non-linear units to perform feature extraction and transformation tasks. Each subsequent level uses the previous level's output as its input. Algorithms can be both supervised and unsupervised and applications include pattern analysis (unsupervised learning) and classification (supervised learning)
- they are based on unsupervised learning of multiple hierarchical levels of data characteristics (and representations). Higher-level characteristics are derived from lower-level characteristics to create a hierarchical representation.
- they are part of the larger class of data representation learning algorithms within Machine Learning.
- they learn multiple levels of representation that correspond to different levels of abstraction; these levels form a hierarchy of concepts.

Usually, a DNN has a number of layers between 7 and 50, Nets with 100 Sayed [2018] or more layers have often better performances, with lower efficiency as a drawback. Layers are not the only parameter to consider when we talk about complexity of a network: we must consider the number of neurons and connections (nets are not necessarily fully connected, so this is not directly related to the number of nodes) too. The wide usage of DNN is related to the following factors:

- presence of BigData: there is a wide presence of labeled data
- GPU Development: by means of technologies like CUDA for NVIDIA GPUs, time of training has been strongly reduced

- vanishing Gradient: this problem was solved by using ReLU function instead of Sigmoid, as previously described

Models of DNN to approach regression and classification problems are categorized as follows:

- Supervised Learning:
  - Convolution Neural Networks (CNN or ConvNet)
  - Fully Connected Nets (FC DNN): MLP nets with 2 or more hidden layers
  - Hierarchical Temporal Memory (HTM)
- Unsupervised Learning:
  - Stacked Auto-Encoders
  - Restricted Boltzmann Machines (RBM)
  - Deep Belief Network (DBN)
- Recurrent models:
  - Recurrent Neural Network (RNN)
  - Long Short-Term Memory (LSTM)

1.4 Convolution Neural Networks

Convolution Neural Networks are a class of DNN models mainly developed for image processing. They differ from MLP because of two main reasons:

- local processing: neurons are locally linked with the ones of the former level and not with all of them (NOT a fully connected structure)
- shared weights: weights are shared in the same level, so different nodes can process different parts of the same input vector, reducing the total number of weights.

1.4.1 CNN Architecture

A CNN has a hierarchical structure of the layers:

- input layer, directly connected to the image pixel
- feature extraction layers, where the local processing and the shared weights are. Here there is a repetition of three kinds of layers: convolutional layer, polling layer, ReLU layer.
- fully connected layers (or dense layers), which work as a MLP classifier.
- output layer with softmax (or sigmoid) as activation function

- **Convolutional Layer:** the first layer is the convolutional one in which a digital filter is applied to the image through a convolution. A digital filter means the application of a 2D mask (matrix) of scrolled weights on the different positions of the input, for each position a scalar product is generated as output (weighted sum) between the mask and the covered portion by the input. An important parameter for this operation is the stride, which modifies the amount of movement over the image in terms of pixels (and so the mask overlap). For example, for a 4x4 mask, if the stride is 4 there is no overlap.

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)\tau'$$

  What the filter really does is a 3D convolution, because an image is represented as a tensor (with 3 channels for an RGB and 1 for a greyscale). A hidden convolution

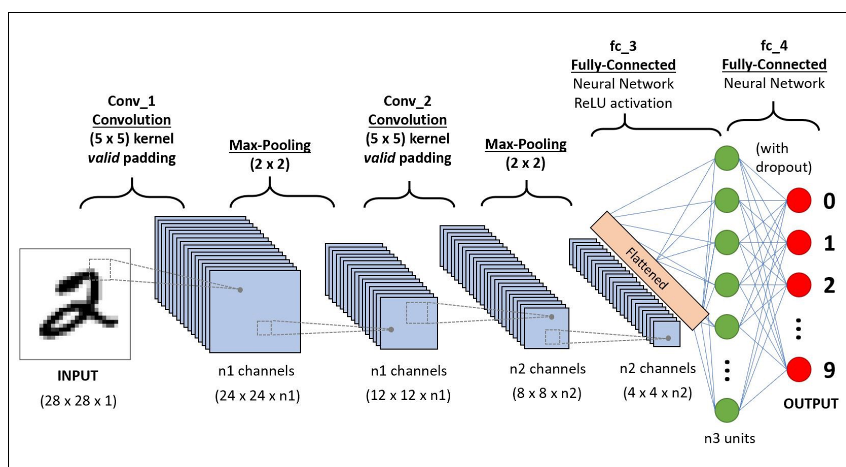

Figure 1.5. Convolution Neural Network architecture.

layer is divided into different feature maps where each of them looks for the same feature in a different portion of the image. Besides the number of filters (F) and the stride (S), there is another important parameter of this layer, called zero-padding (P), which allows control of the output size, giving the possibility of adding an edge of zeros to the input tensor. This avoids the loss of information when information passes to another layer. The output size is defined by this formula:

$$O = \frac{(I - F + 2P)}{S} + 1$$

ReLU function is then applied to the feature map, setting all the negative values to 0.

- **Pooling Layer:** the pooling layer is necessary to merge the results of the former level in order to obtain feature maps with lower size. The downsampling is crucial to achieve invariance to transformations, while maintaining the significant information extracted with convolutions. The pooling layer is also necessary in order to deal with two main problems introduced by the convolutional one. The first is the high complexity of the output: analyzing every little part of the input with multiple overlaps led the output of the convolution layer to have much bigger size than the input one. Without downsampling, the size would become bigger for each convolution layer of the net, making the final output untreatable. The other problem is related to overfitting: if the dataset is big, the convolutional level of the net adapts too much to the training set, with poor inference performances. Pooling can be done by taking the highest input of the filtered window (Max pooling) or going for the mean value of the window (Average pooling).
- **Fully Connected layers:** the ending part of the net is an MLP. The input of the dense part of the net is a 3-dimension tensor that needs to be flatted, obtaining a vector. The final layer has usually a softmax (could be a sigmoid in case of binary classification), so the output is a vector with size equal to the number of classes of the problem, where each node is the probability of the initial input to be in that specific class.

1.4.2 Transfer Learning

Training complex CNNs on large datasets can take days or weeks, even if run on GPU (at least only for the training phase). In addition, the training of a CNN on a new problem requires a large, labeled training set Bryan S. [1998]. For these two reasons, training is often not done starting from zero, but what is called transfer learning is carried out through two techniques:

- Fine Tuning: a net already trained on a different problem is used. The output layer is changed according to the number of classes of the current problems. Starting weights are set as the final of the trained net, except for the ones between the last two layers which are initialized randomly. Some training iterations are done on the new dataset to optimize the weights for the current problem.
- Feature Reuse: features related to intermediate layers of a trained net are taken to train another classifier.

Chapter 2

ARCHITECTURES FOR IMAGE UNDERSTANDING

Image understanding is the process of interpreting the various regions and objects exploited by low-level operation (described in the former chapter), in order to deal with images related problems, such as segmentation, images classification, regression, object recognition and segmentation, etc Bryan S. [1998]. Image understanding is strongly related to computer vision Jacome et al. [May 2019], the batch of processes with the aim of reproducing human sight. Convolutional Neural Networks have the capacity to extract features from images, without the need of ad-hoc algorithms, that's why they are so important for image understanding Jacome et al. [May 2019]. In the following paragraphs the main tasks related to image understanding will be described with an analysis of the main CNNs architectures related, with particular attention to semantic segmentation models.

2.1 Architectures for Image Classification

Image classification is the task of assigning a single label to an image input data. CNN are widely used for image classification. In fact, one of the main applications is the diagnosis of cancer using histopathological images. There are several classification architectures based on convolutional networks, which bring a revolutionary approach to the field of image recognition, from the simplest AlexNet Simonyan and Zisserman [2012], to more complex ones such as VGG Krizhevsky et al. [2015] and ResNet He et al. [2015]. In the following paragraphs, some of the models used for image classification are presented. The focus is on models which will be recurrent in the thesis.

2.1.1 LeNet

The LeNet architecture LeCun and al. [1995] can be considered as the first architecture of CNN-based Deep Learning. CNNs were initially limited to handwritten numbers recognition (MNIST dataset). These early techniques did not work very well with other types of images. Architecture is the following:

First, after the input, there is a convolutional layer with 6 filters of 5x5 size.

Second layer is a subsampling/pooling one with each cell in each feature map that is connected to 2x2 neighborhoods in the corresponding feature map in the former layer.

Third layer is convolutional with 16 filters 5x5. The input of the first six feature maps is each continuous subset of the three feature maps in the second layer, the input of the next six feature maps comes from the input of the four continuous subsets, and the input of the next three feature maps comes from the four discontinuous subsets. Finally, the input for the last feature map comes from all feature graphs of the second layer.

Fourth layer is equal to the second one.

Then there is the fifth layer, a convolutional one with 120 kernels 5x5. Each cell is connected to the 5*5 neighborhood on all 16 feature graphs of the fourth layer.

Last layer before the output one is fully connected with 84 features map as output.

The choice of the parameters of each layer are set with the idea to work on the MNIST dataset (28x28 grayscale images), masks of size 5x5 avoid the input tensor to go out from the boundary of the convolutional layer LeCun and al. [1998].

2.1.2 AlexNet

The LeNet's successor, called AlexNet Simonyan and Zisserman [2012], is considered the first architecture of deep CNN able to guarantee good results both on classification activities and image recognition. This architecture has improved the learning abilities of CNNs, making them deeper by adding more hidden layers, and applying different optimization strategies of the parameters Khan and al. [2019].

At the beginning of the 2000s, the evolution of hardware systems limited the deep CNN learning skills. To overcome those limitations, AlexNet was trained in parallel on two NVIDIA GTX GPUs of the highest level. The depth has been extended from 5 (LeNet) to 8 levels allowing to improve generalization at different levels of image resolution. However, the large number of layers introduced an overfitting problem. For this reason, superimposed subsampling and local response normalization was applied. Another resolution to this problem was introduced based on the work of Hinton Srivastava et al. [2014], which made it possible by creating an algorithm (called dropout) which consists in skipping some drives on the network randomly during the training process, in order to force the learning of more robust features. Furthermore, the ReLU was applied as an activation function, to improve velocity of convergence, avoiding the problem of vanishing gradient. Other changes introduced by AlexNet concerns the use of large filters (11x11 and 5x5) at the initial levels of the net. Thanks to all this, the effectiveness of learning with the use of AlexNet was proven, thus leading to a significant evolution in the field of CNN, initiating a new series of research for new types of architectures Khan and al. [2019].

2.1.3 VGG Net

VGG, is a CNN architecture, which is based on a simple and efficient algorithm, realized with 19 levels of depth, compared to the 8 of AlexNet. VGG starts from the observation that smaller filters give higher net performances. Based on these results, VGG replaced the 11x11 and 5x5 filters with a set of 3x3 filters, demonstrating in an experimental way that the simultaneous use of small filters simulates the use of larger filters. The use of these very small filters made it possible to reduce the computation complexity and learn a feature maps linear combination, reducing the number of parameters. To achieve this, 1x1 convolutions were added too. Subsequently, to get more performance of the neural network, a max pooling algorithm was used after the layer convolution, with the use of padding to hold spatial resolution, avoiding the problem of the loss of features.

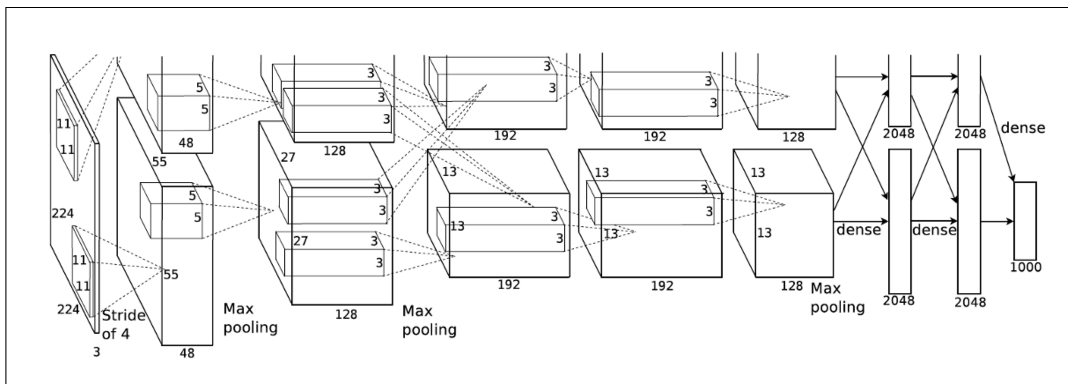

Figure 2.1. AlexNet architecture.

The biggest limit of this architecture is based on the use of 138 million parameters that make it heavy from a computational point of view, requiring high power to implement Khan and al. [2019].

The original architecture of the net was proposed by Karen Simonyan in 2014. The input image is passed through the stack of 3x3 (or 1x1) filters previously described. The convolution stride is set to 1 pixel; the spatial padding of convolutional layer input is set in order to preserve resolution after convolution, so the padding is 1 pixel for 3×3 convolutional layers. Pooling is carried out by five max-pooling layers, which follow some (not all) of the convolutional layers. Max-pooling is performed over a 2×2 pixel, with stride equals to 2. The convolutional part is then followed by three fully connected layers: first two have 4096 channels and the last has total channels equals to 1000 multiplied by the number of classes. Last layer of the net is the one with the softmax function. All the hidden layers have ReLU as activation function. The original architecture of the VGG has 13 convolutional and 3 dense layers, so it's also called VGG16. A variance is represented by the VGG19 with a higher number of levels in the convolutional part Simonyan et al. [2014].

2.1.4 ResNet

Residual Neural Network (ResNet) was introduced by Shaoqing Ren, Kaiming He, Jian Sun, and Xiangyu Zhang in 2015. It is considered an extension of the deep CNNs because the concept of residual learning was introduced, changing the entire architecture of convolutional neural networks. This technology allows to elaborate a new method useful for the construction of Deep Learning systems based on neural networks. Although this architecture has a greater depth than the previous ones, it is related to a lower computational complexity. It has been shown Simonyan et al. [2014] that a ResNet with 50/101/152 (ResNet50, ResNet101, ResNet152) layers presents higher accuracy on the classification of the images compared to the 34 layers of the normal network, achieving a performance improvement of approximately 28% on COCO dataset Lin et al. [2014]. A crucial role in the network is interpreted by the residual blocks, which are the revolutionary elements of this architecture.

As one can notice, a direct connection which skips some layers in between is present (this structure can vary according to the specific model). This is what is called skip connection and represents the core of this block. If the skip connection is ignored the input (let's call it x), passes through the layers, and is multiplied by their weights and the

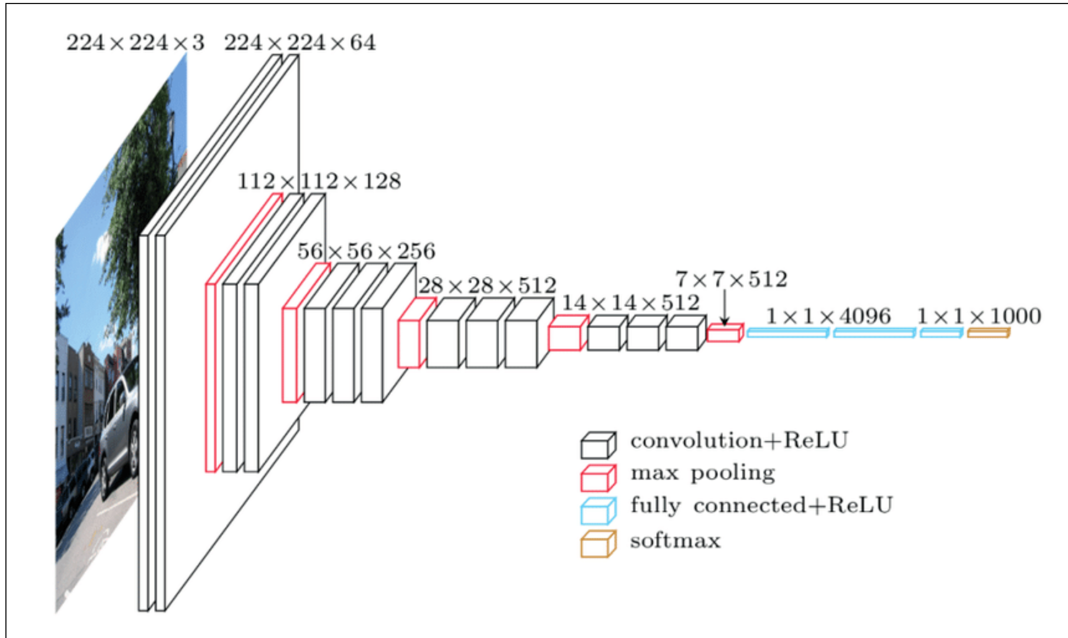

Figure 2.2. Architecture of a VGG16.

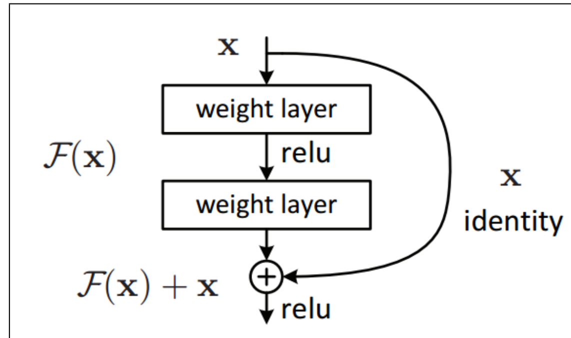

Figure 2.3. ResNet residual block.

bias term is added. Then, this term goes as input of the activation function f(), generating the output H(x), such as:

$$H(x) = f(w * x + b)$$

or

$$H(x) = f(x)$$

With the presence of the skip connection, input x is added to f(x), so:

$$H(x) = f(x) + x$$

At this point, there is no guarantee that f(x) and x are of the same dimensions, because of the possible presence of convolutional and pooling layers in f(). There are two main approaches to deal with this:

- the skip connection is modified through zero padding to increase the dimensions

- a 1x1 convolutional layer is added (projection method), so the output changes to be:

$$H(x) = f(x) + w_1.x$$

with w_1 as the added parameter

As said before, the presence of skip connections in ResNet avoids the problem of vanishing by allowing this alternate shortcut path for the gradient to flow through. Moreover, these connections are very useful because they allow the model to learn the identity functions, ensuring that the higher layer will perform at least as good as the lower layer, and not worse. Here is a deeper explanation.

There is a shallow network and a deep network that map an input ‘x’ to output ’y’ by means of the function H(x). The objective is that the deep network must perform at least as good as the shallow network and not degrade the performance. One way of achieving it is if the additional layers in a deep network learn the identity function and thus their output equal inputs which do not allow them to degrade the performance even with extra layers. To learn an identity function, f(x) must be equal to x which is grader to attain whereas in case of ResNet, which has output:

$$H(x) = f(x) + x$$

$$f(x) = 0$$

$$H(x) = x$$

What is needed is to make f(x)=0, and x will be taken as output which is also the input. In the best-case scenario, additional layers of the deep neural network can better approximate the mapping of ‘x’ to output ‘y’ than it’s the less deep counterpart and reduce the error by a significant margin. And thus, we expect ResNet to perform equally or better than the plain deep neural networks. The usage of ResNet has significantly increased the performance of neural networks with more layers and here is the plot of error % when comparing it with neural networks with plain layers. It’s clear that the difference is big in the networks with 34 layers where ResNet-34 has much lower error % as compared to plain-34. Furthermore, it’s possible to see that the error % for plain-18 and ResNet-18 is almost the same (https://www.mygreatlearning.com/resnet).

The most common architectures of the ResNet are:

- ResNet18
- ResNet34
- ResNet50
- ResNet50V2
- ResNet101
- ResNet101V2
- ResNet152
- ResNet152V2

The number following the name is simply the total number of layers. In ResNet architectures there are always n-1 convolutional layers and only a fc1000 dense one, preceded by an average pooling function. In V2 variance there is a batch normalization before each weight layer and it is a nomenclature mostly used for coding frameworks useful for DL, such as Tensorflow and Keras.

Here is a comparison, in terms of accuracy and prediction time between versions of Resnet with 18, 52, 101 and 152 layers. The study is named “Evaluating the Performance of ResNet Model Based on Image Recognition”, by Riaz Khan, Emella Opoku Abogye,

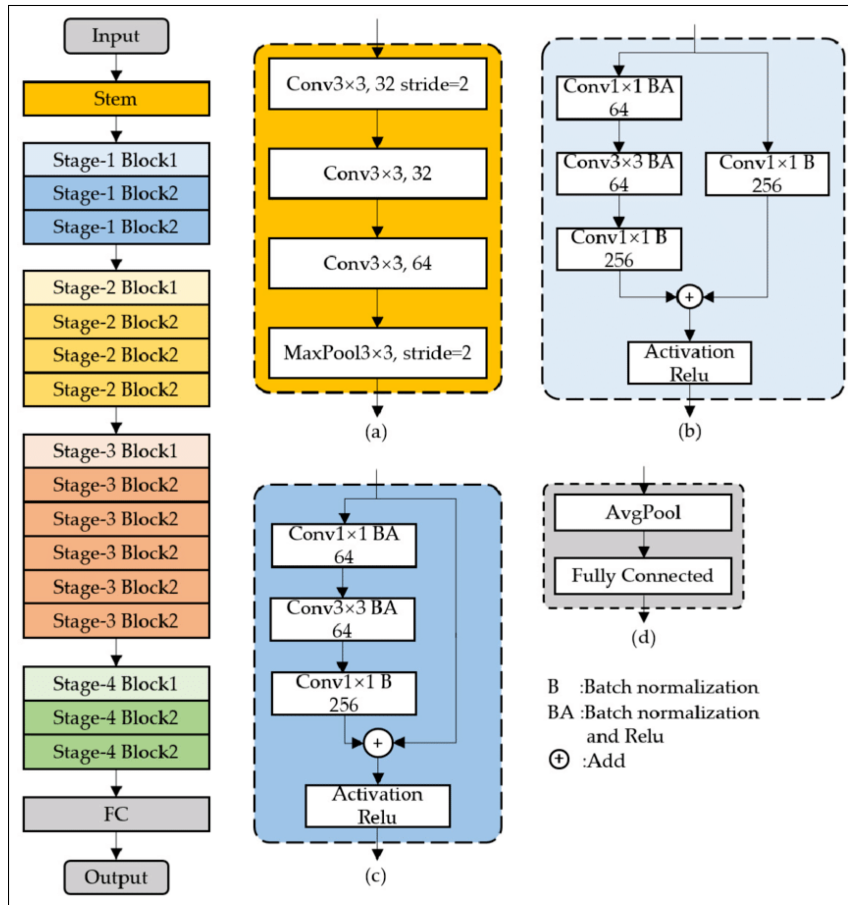

Figure 2.4. ResNet architecture.

Rajesh Kumar, November 2018. Two datasets of medical images for cancer diagnosis are used. The first, Malware Dataset, is a Microsoft dataset for 9 classes classification challenge and contains 21741 images. In these tests, 10868 elements of the dataset are used for training, the rest for testing. Test dataset is completed with 3000 more images from different sources. Cancer Dataset is made by medical images for diagnosis of Chengdu Sichuan Cancer Hospital, taken in a period of around 10 years. No specific indication is given on hardware, but the OS (Ubuntu) and the RAM (8 GB) Riaz Khan [Nov. 2015].

Figure 2.5. Preditcion time of different ResNet architectures on two big datasets.

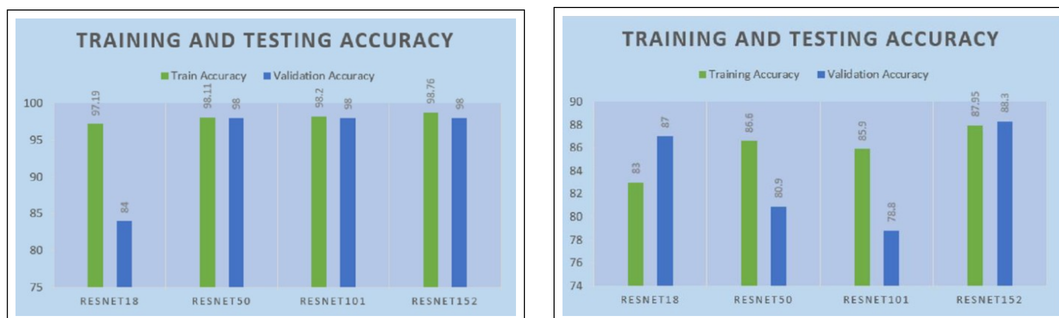

Figure 2.6. Training and test accuracy of different ResNet architectures on two big datasets.

2.2 Architectures for Segmentation

Image segmentation is a key part for computer vision systems and consists in partitioning an image into different regions representing different objects. This process is based on extraction of sets of pixels, from the domain image, of one or more regions. Segmentation plays a particularly important role in various applications including face recognition, image searching, medical imaging, video surveillance and augmented reality. Numerous segmentation algorithms have been developed, from the most basic which are based on histograms, to algorithms more advanced. We speak of semantic segmentation when the model is also capable of establishing the class for each of the regions identified. In recent years, the networks of Deep Learning have produced a new generation of semantic image segmentation models with a noticeable performance improvement, reaching very high accuracy values. The result of a segmentation technique must respect some properties:

- segmentation must be complete, so all the pixels of the image must belong to at least one region of the partition.
- the set of pixels of a region of the image must be connected.
- all regions of an image must be separated from each other.
- it must satisfy the criterion of features homogeneity deriving from components spectral defined based on intensity, color, or texture.

It's possible to distinguish two main types of segmentation, based on the objective:

- Semantic Segmentation, which consist in giving a label to every image pixel of the correct class (e.g. floor, window, wall, ceiling, table, chair for an indoor scenario)
- Instance Segmentation, that is a sort of merging of semantic segmentation and object detection. The objective is always labeling the pixels correctly, according to the classes of the task, but it's also necessary to distinguish the different objects of the same class.

It's possible to identify a third type of segmentation, named Panoptic Segmentation which is a further step and consists in combining semantic and instance segmentation. Both a label and an instance ID are given to each pixel of the image: if two pixels have the same class and the same ID, they are part of the same object.

Another kind of classification for image segmentation is done based on the segmentation technique.

- Region-based Segmentation:
  - Threshold Segmentation
  - Regional Growth Segmentation

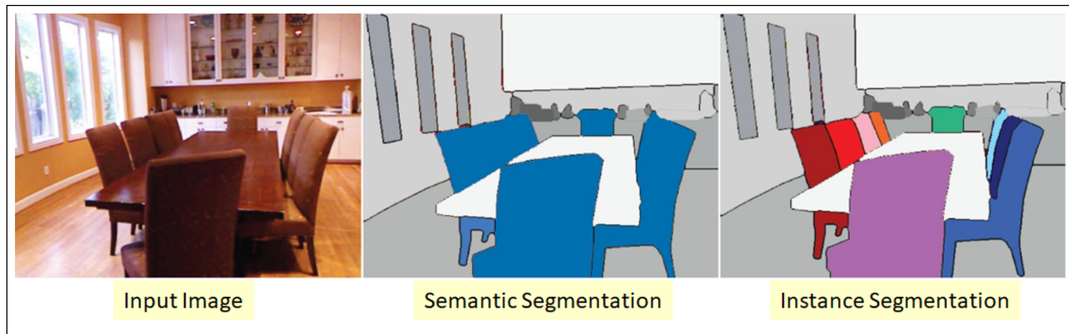

Figure 2.7. Semantic and instance segmentation example.

- Edge Detection Segmentation
- Segmentation based on clustering
- Segmentation based on DL models

Threshold segmentation is the simplest approach for segmentation. It takes into account only the greyscale information and labels the pixels based on the gray value of the target. This method can be global, subdividing the image into two big regions (one for the targets and one for the background) or local, subdividing the input in different regions with different threshold values. This region-based technique is very fast but works properly only if there is high greyscale contrast between target and background; it is also very sensitive to noise and ignores the spatial information Yuheng and Hao [2017].

The regional growth method, instead, is based on the idea to put pixels with similar properties on the same region. First a seed pixel is selected and then the similar pixels are merged around the seed one into the region where it is located. This method is able to provide good boundary information and segmentation results, because it usually separates regions with the same characteristics. The disadvantages are related to the high computational cost and (like the threshold method) the sensitivity to noise and greyscale unevenness.

Edge Detection Segmentation is based on searching grey edges, which separates two adjacent regions. With the usage of mathematical differential operators it is possible to analyze the discontinuities of these boundary elements. Parallel edge detection is done using a spatial domain differential operator to perform image segmentation by convoluting its template and image Yuheng and Hao [2017]. The so-called parallel edge detection is usually used as an image preprocessing method. First order most used differential operators are Prewitt operator, Sobel operator and Robert operator. For the second order there is the Laplacian operator, Kirsch operator and Wallis operator.

Image Segmentation can be seen as grouping pixels with the same characteristics. Clustering is an unsupervised machine learning technique which is based on grouping elements on the base of analogue characteristics and at the same time maximizing the difference (in terms of features) with elements of other clusters. One of the most common clustering algorithms is the K-means, which is specifically based on the distance among points and can be used well in an image segmentation scenario. A clustering algorithm, in particular the K-means, is usually fast, due to its simplicity, highly efficient and scalable for large datasets Yuheng and Hao [2017]. The biggest drawback is related to the difficulty to estimate or select K (the number of clusters). Moreover, it is important to remind that K-means is a distance-based partition method, so it can work only for convex suitable

datasets. Finally, there is another possible disadvantage: for each iteration of the algorithm the set of data is traversed completely, so sometimes it is possible to have scenarios with very expensive algorithm time.

There are several DL-based architectures for image segmentation. First well-performing models were born around 2013 and 2014 and several have been developed until nowadays. As for the case of image classification, some models are presented in the next paragraphs.

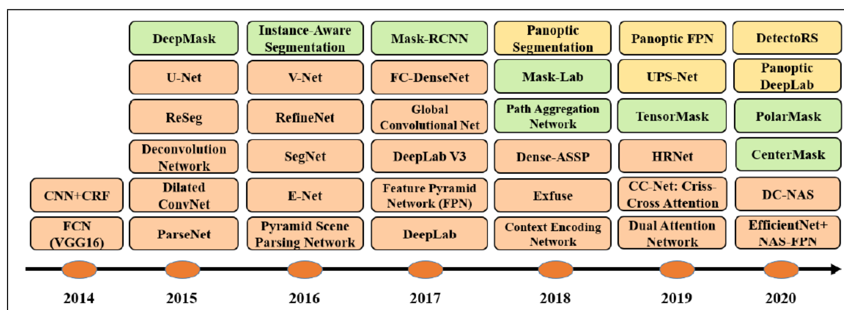

Figure 2.8. Segmentation models timeline.

2.2.1 Fully Convolutional Networks

Fully convolutional networks (FCN) represent the first DL-based approach to image segmentation. They are composed only by convolutional layers, obtained starting from existing CNNs such as VGG16, and replacing the fully connected part with other convolutional layers. These networks represent an important step for image segmentation, because they were the first to demonstrate that CNNs can be trained for segmentation purposes. Long et al. were the first in proposing a work of this kind but, even if their model worked properly and became the state-of-the-art, there were some limitations related to the high rime of inference and the difficulty in transferring the information in 3D. Moreover, FCN did not consider the context information in an efficient way. A solution for this was found by Liu et al. who proposed a network, called ParseNet, very similar to the FCN but with some changes in the last convolutional layers with a module specifically built.

2.2.2 Convolutional Models with Graphical Models

In order to solve the limitation of FCN of ignoring some useful semantic context information, a further step was done with convolutional models which incorporate probabilistic graphical models. A relevant semantic segmentation algorithm proposed by Chen et al. starts from the observation that the response of the final layers of CNNs are not sufficiently localized for accurate object segmentation: it is possible to better recognize segment boundaries replacing the last convolutional layer of the FCN with fully connected conditional random fields (CRF) layer. A similar work was proposed by Schwing and Urtasun with a network made by a combination of convolutional layers and fully connected CRFs for semantic segmentation, reaching good results on the PASCAL VOC 2012 dataset. A slightly different approach was related to Liu at al. work who proposed a usage of MRFs, but instead of going for an iterative optimization, there is a CNN model named Parsing Network Chaurasia and Culurciello [2017].

2.2.3 Models with Encoder-Decoder Structures

Models with this structure have a wide use in the image segmentation field. The general structure consists of two parts, an encoder which uses convolutional layers and is the part with the feature extraction task, and a decoder, with the objective of building the segmentation mask.

First work of this kind was proposed by Noh et al. and uses the convolutional layers of a VGG16 as encoder, reaching a 72.5% accuracy Chaurasia and Culurciello [2017] on the PASCAL VOC 2012.

Another important model is the SegNet, which still uses the VGG16 as encoder, but it has a convolutional decoder, able to do up sampling of the input feature map: this is done using the indices of the corresponding pooling part in the encoder. Compared to its predecessors, SegNet has the big advantage of having a decoder part with no need for learning to up-sample, reducing a lot the number of trainable parameters.

Another model with similar elements has been recently developed, the HRNet which maintains the high resolution during the encoding process.

Several new works use this net as backbone. Several models with encoder-decoder structure were born to address tasks related to medical and biomedical imaging: U-net and V-net are an important example in this sense. U-net is analyzed in a deeper way below; V-net was proposed for the segmentation of 3D medical images, by Milletari et al. and, as well as the U-Net, a FCN-based model, related to the task of prostate volume segmentation. In Milletari work, a particular objective function based on the Dice coefficient is proposed, in order to deal with the high difference of the number of voxels in foreground and background. Some variants of these nets have been also proposed during years, usually to achieve good enough segmentation results on medical images of a specific organ. An example in this sense is the Progressive Dense V-Net (PDV-Net).

U-Net

U-Net was proposed for the first time by Ronneberg et al. in 2015 with the purpose of segmenting medical images obtained with a microscope. The CNN based on the U-Net architecture Minaee et al. [2020] consists of a series of codes and contractions that serve to extract the context of the image, followed by a sequence of decoding and symmetrical expansion. The latter processes are useful to derive the classification of each single pixel. Every step that concerns the contraction receives an image as input and is described by 4 blocks with chains, each of which has two convolutional layers, a batch layer normalization and a ReLU activation function, and finally a last block of max pooling. For each block, the number of filters that are applied will be the double of the previous one, i.e., it starts from 32 feature maps extracted from the first block, up to 256 features of the fourth block. During this phase we notice that information about features is incremented, while information that concern spatial information diminishes through the sampling operation.

The second part of the architecture, that is the one that concerns the expansion, it is also made up of 4 blocks and each of these allows to concatenate the over-sampled image with the one that corresponds to the contraction path and then applies the two convolutional units, the ones identical to the previous one.

In the end, the features that come out of the last level will be of the same size as the input image but with a higher number of channels. So, for the U-Net there are two main parts: the contracting path and the symmetric expanding path. The former is an FCN-like that uses 3x3 convolutions to extract features, whereas the last one decreases the number of feature maps while increasing the size of the tensor. It's important to

remark that the feature maps of the down-sampling part are copied to the up-sampling part, with the objective to avoid loss of information.

Finally, the softmax classification layer, with 1x1 convolution processes which defines the segmentation map, assigns the label to each pixel of the image, based on the category to which it belongs.

Different variants of the U-Net have been developed during these years for different tasks such as 3D images and road segmentation.

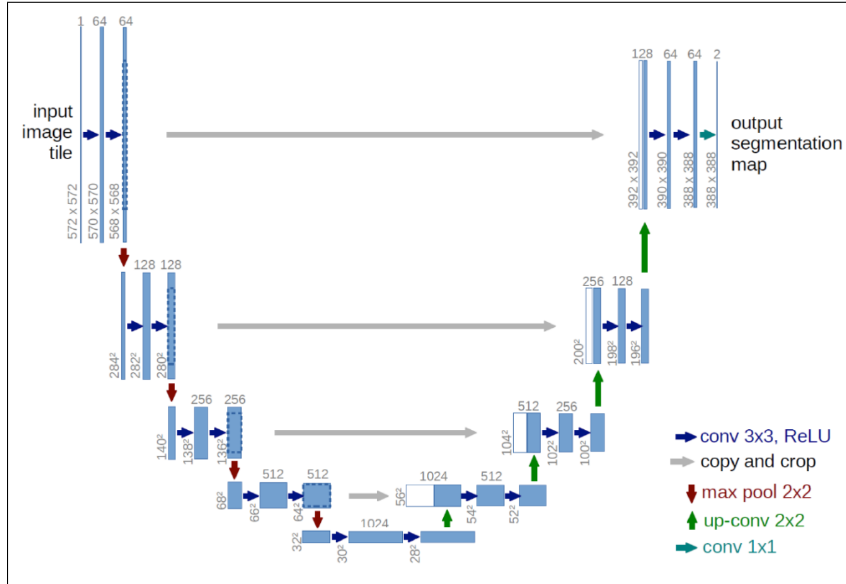

Figure 2.9. U-Net architecture.

LinkNet

The novelty introduced by LinkNet in 2017 was the way it links each encoder with decoder. This idea started from the observation that during the down-sampling process some information is lost and impossible to recover only by the simple up-sampling process of the decoder. Unlike the previous models, where the output of the encoder goes directly to the decoder as input, in LinkNet each encoder layer is bypassed to the output of its corresponding decoder layer, in order to recover potentially lost spatial information, which results useful for the decoder task. Because of the communication at each layer between encoder and decoder, the net results to have fewer trainable parameters in the down-sampling part, resulting more efficient if compared with the predecessors. For what concerns the architecture, it is shown in figure 2.10. Between each convolutional layer, batch normalization is performed, then a ReLU function is applied. The first block of the encoder applies a 7x7 kernel size convolution with stride equals to 2. It also performs a max pooling in an area of 3x3 with stride 2, while the rest of the encoder is made by residual blocks. The original work of Link-Net used a Res18 as backbone, instead of the more used Res101 and VGG16, to have a lighter model overall Chaurasia and Culurciello [2017].

In table 2.1 the LinkNet performance with Res18 backbone is shown, compared with some models previously analyzed.

Figure 2.10. LinkNet architecture.

Model	Class IoU	Class iIoU
SegNet*	56.1	34.2
ENet*	58.3	34.4
Dilation10	68.7	-
Deep-Lab CRF (VGG16)	65.9	-
Deep-Lab CRF (ResNet101)	71.4	42.6
LinkNet without bypass	72.6	51.4
LinkNet	**76.4**	**58.6**

Table 2.1. LinkNet performance with Res18 backbone compared with other models.

2.2.4 Pyramid Network-Based Models

Pyramid network-based models are a set of networks which in some sense incorporate multi-scale analysis for image processing. Lin et al. developed the Feature Pyramid Network (FPN), initially to perform object detection, but with time the models have been used for image segmentation too. The idea is to mix features with high and low resolutions, so there is an ascending path, a descending path, and some lateral connections. Then the merged feature maps enter in 3x3 convolutional layers, to give an output for each stage. The object detection is finally performed by each stage of the descending part. To obtain segmentation instead of object detection, some MLPs layers must be added at the end.

Pyramid Scene Parsing (PSP) Network, developed in 2016 by Zhou et al. is a very important and widely used multi-scale model. Its structure is described and analyzed below. PSP net is very important to address scene parsing based tasks. The previous approach was mainly related to FCNs that have the problem of the absence of a strategy to understand complex scenes.

PSP-Net

Pyramid Scene Parsing Network for semantic segmentation was developed with the objective to obtain a good-performing DL model for the scene parsing task. With scene parsing is intended a computer vision topic that consists in understanding the scene. Achieving this in a good and complete way can be useful for applications such as robot sensing and

autonomous navigation.

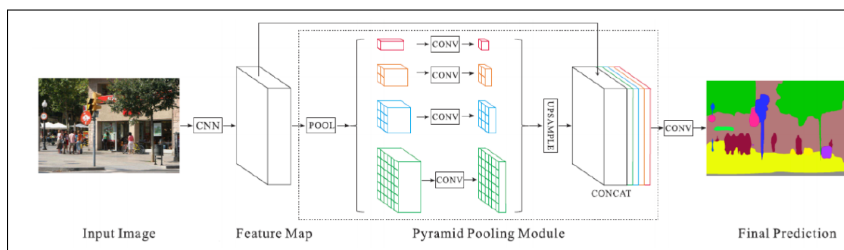

Figure 2.11. PSP-Net architecture.

What really characterizes this model is the pyramid pooling module, which constitutes the core of the network. This module works as a global contextual prior and basically merges features under four different pyramid scales Zhao et al. [2017].

The first level is the roughest one and is a global pooling, which gives a single unit as output.

Second level, starting from the feature maps, operates a division in some sub-regions and then applies pooling. The pooling operation on features is applied using 4 different scales corresponding to 4 different pyramidal levels: 1x1, 2×2, 3×3 and 6×6. To reduce the size of feature maps a convolution operation is applied using a convolutional layer of 1×1 in order to sub-sample it. The features thus obtained are finally chained to the initial feature maps in order to obtain contextual information global. Subsequently, these results are subject to a new processing by a new convolutional layer in order to generate the predictions regarding pixels. The network in question therefore allows us to examine all the features for each sub-region in different positions to better understand the image and have excellent results regarding semantic segmentation.

The whole structure can be seen in figure 2.11. So, given an input image, first it passes through a CNN, which represents the backbone of the PSPN, to extract the feature map. In the original work ResNet models with a different number of convolutional layers were used. Once the feature map, which will be of a reduced size, compared to the input image, is obtained, it enters the pyramid pooling module, which works as descripted. The 4 levels work on the whole image, half of it and on some portions. Their fusion constitutes the global prior. In the final part of this module, the global prior is concatenated with the feature map at the end of the CNN module. The final part is made of some convolution layers that build the segmentation map. In table 2.2, results of PSPN in terms of performances with different versions of ResNet are shown. Statistics are directly taken from the original PSP paper, as specified previously. In 2016, PSP was the best net in ImageNet scene parsing competition and was 1st classified both on PASCAL VOC 2012 benchmark and urban scene Cityscapes data benchmark.

2.2.5 Other Models

R-CNN Models

Other sets of models for image segmentation have been developed in these years, such as the regional-convolutional neural network (R-CNN) and its variants. These networks are mainly developed for object detection, but at the same time produce high quality segmentation for each instance.

Method	Mean IoU(%)	Pixel Acc.(%)
PSPNet(50)	41.68	80.04
PSPNet(101)	41.96	80.64
PSPNet(152)	42.62	80.80
PSPNet(269)	**43.81**	**80.88**
PSPNet(50)+MS	42.78	80.76
PSPNet(101)+MS	43.29	81.39
PSPNet(152)+MS	43.51	81.38
PSPNet(269)+MS	**44.94**	**81.69**

Table 2.2. Results of PSPN in terms of performances with different versions of ResNet.

Fast R-CNN, Faster R-CNN, Masked R-CNN, Path Aggregation Network (PANet) are the best examples of this kind of model. The core they have is represented by a region proposal network (RPN), that gives as output the so-called region of interest (RoI) and a layer that applies polling to this output (RoIPooling) to finalize the object detection. With some changes that we can notice in some extensions, instance segmentation is performed.

Other models are part of this family, like the one developed by Diu et al. which is characterized by a very complex architecture based on 3 parallel networks with different tasks: instance differentiation, masks estimation and object categorization. Feature maps are continuously shared among all the three branches. Moreover, Hu et al. developed a model which can be trained with datasets where just a little part has corresponding labels. This is possible due to an innovative weight transfer function.

MaskLab by Chen et al is a relevant model developed for instance segmentation too, based on Faster R-CNN. The three bins output produced (box detection, semantic segmentation, and direction prediction) can be given as input of the Faster RCNN detector to achieve object detection and instance segmentation.

RNN Models

Recurrent Neural Networks (RNN) based models represent an alternative to CNNs for computer vision tasks. RNNs can be useful in the sense of handling the dependencies among the pixels of an image: in this way segmentation maps could be predicted in a better way. An important example of this set of models is ReSeg by Visin et al., developed for semantic segmentation, which have a structure with four different RNNs that yield to important global information. To perform semantic segmentation, convolutional layers are taken from a VGG net and then some layers that work as a decoder are added to the architecture.

Another model that achieves good performance in terms of both semantic segmentation and image classification is the one proposed by Byeon et al., called long-short-term-memory (LSTM) network. A generalization of this model, named Graph Long Short-Term memory network was developed afterwards. Images are not yet divided into pixels (like in LSTM), each superpixel is taken and considered as an entity with semantic importance. Then, with an adaptive approach, a graph for the image is built, where the relations among the superpixels, from a spatial point of view, are used as edges. With the goal of achieving both semantic segmentation and 3D mapping of the scene, Data Associated Recurrent Neural Networks (DA-RNNs) was developed.

Dilated Convolutional Models

These sets of models have as distinctive traits the dilation rate as a new parameter for convolutional layers. The dilation operation is useful in the sense that a certain filter can have a bigger size, while maintaining the same number of parameters. If a 2x2 kernel is applied in normal conditions, it has a receptive field of size 2x2 and 4 parameters. With dilation operation, supposing a rating of 2, the receptive field is increased to 3x3, but the parameters are still 4. This operation has a wide usage in networks for segmentation, and specifically for real time tasks.

DeepLab, proposed by Chen et al. (with the two versions DeepLab1 and DeepLab2) is a milestone in this sense, bringing three big novelties: the presence of dilation operation in convolutional layers; a spatial pyramid pooling module that allows to use features at different sampling rate; improved localization of objects boundaries. With a Res-Net 101 as a backbone, this model can reach almost 80% mIoU score on the 2012 PASCAL VOC challenge and more than 70% on the Cityscapes challenge Chaurasia and Culurciello [2017]. Better results are obtained with the successors of these models: DeepLabv3 and DeepLavv3+ with the addition of batch normalization and 1x1 convolution layer at the level of the pyramid pooling module.

CNN Models with Active Contour Models

This family of models is based on the idea of merging fully convolutional networks and Active Contour Models (ACM). This operation is mainly done by introducing an ad-hoc loss function that is based on active contour models principles. The basic model of this set is the one developed by Chen et al. that starts from a normal FCN and adds a supervised loss layer that acts during the training of the FCN part.

The main works in this sense are the ones by Le et al. where the ACM is used only as a post-processor; the Deep Active Lesion Segmentation (DALS), developed for semantic segmentation of medical images; Deep Structured Active Contours, known as DSAC by Marcos et al; Deep Active Ray Network (DarNet), by Cheng et al.

Models Based on Attention Mechanisms

In the history of computer vision, attention mechanisms represent a recurrent topic. Models based on this are developed to address semantic segmentation. An attention mechanism-based model is the one developed by Chen et al. that learns how to weight multi-scale features at the location of each pixel. A model for semantic segmentation is taken and trained with both multi-scale images and the attention model. The attention mechanism surpasses both average and max pooling mechanisms and allows the model to appraise the relevance of the features at different positions.

Another relevant work is the Reverse Attention Network (RAN) that is based on a totally different idea of training with respect to all the examples named until now. When the model is trained, the objective is to extract the features that don't match with any of the classes of interest. This is possible thanks to an architecture that is characterized by three different branches, which allow to perform both the direct and the reverse learning at the same time.

Other relevant works are represented by a Pyramid Attention Network by Li et al. that puts together spatial pyramids and attention mechanisms to achieve semantic segmentation, Expectation-Maximization Attention Network (EMANet), OC Network, Criss-Cross Attention Network (CCNet), Discriminative Feature Network (DFN).

2.3 Metrics for Classification

2.3.1 Accuracy

To evaluate the performance of a given classification model, different and varied metrics exist to analyze the results. These functions require in input the real ground truth labels and those deriving from model inference. They return a numeric value indicating the quality of the results. Before the analysis of them in detail, it's necessary to define:

- True Positive (TP): the set consisting of the samples labeled and classified in the same way.
- False Positive (FP): a set consisting of the samples classified by the system as belonging to one class, but really belonging to another.
- False Negative (FN): made up of samples classified by the system as not belonging to a specific class which are part of it.

Starting from this definition, it is possible to define the confusion matrix. For a classification problem with d classes, the confusion matrix M is a matrix dxd where the element M_{ij} corresponds to the number of samples belonging to the class i but it classified by the network as j. The elements on the diagonal are the elements correctly classified. The simplest confusion matrix is the one of a binary classification problem. Assuming you have two labels, -1 and 1, the confusion matrix is a 2x2, where M_{11} corresponds to the true-positive. The M_{12} element corresponds to a false negative (FN). Similarly, M_{21} is a false positive and finally M_{22} a true negative (TN). For a multi-class classification problem, the matrix becomes larger, where the correctly classified samples (true positives) appear on the diagonal, while FP and FN are present in the rest of the matrix. Thus, we can conclude that the confusion matrix allows for understanding the performance of the model used, in case the classes involved are not excessive.

In cases of classification where the number of classes is too large, it becomes very difficult to extrapolate useful information for a first evaluation. In an image classification problem, the term "accuracy" refers to the number of images correctly classified, over the total number of images used in the test. It's a usual practice, during the training, doing an evaluation on a part of the training dataset (validation dataset), by means of the accuracy metric.

2.3.2 Other Metrics

Accuracy can't be always the only metric parameter to consider, especially in situations of unbalanced datasets. Other significant metrics for image classification are the precision, the recall and the F1 score (or harmonic mean).

Precision and recall are two indices that allow a simplified reading of the confusion matrix allowing to measure the goodness of results. We define precision as the ratio of the number of samples that have been correctly classified and the total of classified elements, including FP.

Recall is defined as the ratio between the total number of correctly classified elements and the total number of elements that the system should recover if this worked perfectly.

These two metrics are very useful for understanding the reliability of a system, but it can often be convenient to consider only one value. For this reason, the F1 Score value is calculated, that is the harmonic average of the two values previously calculated.

$$Precision = \frac{M_{ii}}{\sum_j M_{ji}}$$

$$Recall = \frac{M_{ii}}{\sum_j M_{ij}}$$

$$F1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

F1 can assume values between 0 and 1, where 1 indicates a perfect result, while 0 indicates an unreliable result. In the case of multi-class problems, this value is equal to the average of the F1 values of each class. In relation to the semantic segmentation, this parameter indicates the proximity between the traced contour and segmented contour. In particular, for each class, the average F1 is the average of the F1 of the class over all images. For each image, the value of F1 is given by the mean of the F1 of all the image classes for the single image. For the entire dataset, F1 is the average of F1 of all classes in all images.

2.4 Metrics for Image Segmenation

Evaluation metrics for segmentation are useful to understand how good, from a quantitative point of view, the image segmentation is. The importance of metrics is not only limited to the evaluation of a model already trained, but they are crucial during the training process: having knowledge of how some metrics change (both on the training and validation dataset) at each step of the training gives information about the quality of the training. For image segmentation there are always present, for a certain image, the predicted mask and the actual mask, also named ground truth. The most used metrics for image segmentation are described in this paragraph.

2.4.1 Pixel Accuracy

Accuracy (or precision) is the simplest function to measure the quality of a classifier and corresponds to the fraction of correctly classified samples. Accuracy is the simplest metric to consider. It's a parameter expressed as the percentage of pixels correctly classified over the total number of pixels.

In a semantic segmentation problem for each class, accuracy is the ratio between the number of correctly classified pixels and the total number of the single class, as specified from the ground truth. For each image, the accuracy value is the average accuracy of all classes for the single image. For the entire dataset, the accuracy value is the average accuracy of all classes in all images. The overall accuracy, on the other hand, is the fraction of correctly classified pixels, regardless of the class. Accuracy can be useful to evaluate how well a model performs segmentation, but it is not always significant. To make an example, consider a binary segmentation where there is a dataset of images of trees with apples and the semantic segmentation has the objective of labeling pixels related to apples as part of class 1 and the rest as part of class zero. Consider a dataset of images of 128x128, so the total number of pixels is 16 384. Number of pixels related to apples, for each image are, on average, 64. Now, imagine having a model that labels every pixel as non-apple, independently from the input image. It is clear that this hypothetical model is totally useless, but if we consider only the accuracy, on the dataset, on average there are 16 384 − 64 pixels correctly classified, so 99.6% accuracy is achieved. Accuracy can be a relevant parameter in cases where numbers of pixels related to each class are similar, but as the example above shows, there are cases of very bad performing models that achieve high levels of accuracy.

$$FP_i = \frac{\sum_i M_{ii}}{\sum_i \sum_j M_{ji}}$$

2.4.2 Intersection over Union (IoU)

The Intersection over Union, known also as Jaccard index, and the Jaccard similarity coefficient (firstly proposed by Paul Jaccard), is a metric used for comparing the similarity and diversity of two sets. The Jaccard coefficient measures similarity between finite sets and is defined as the size of the intersection divided by the size of the union of the sets.

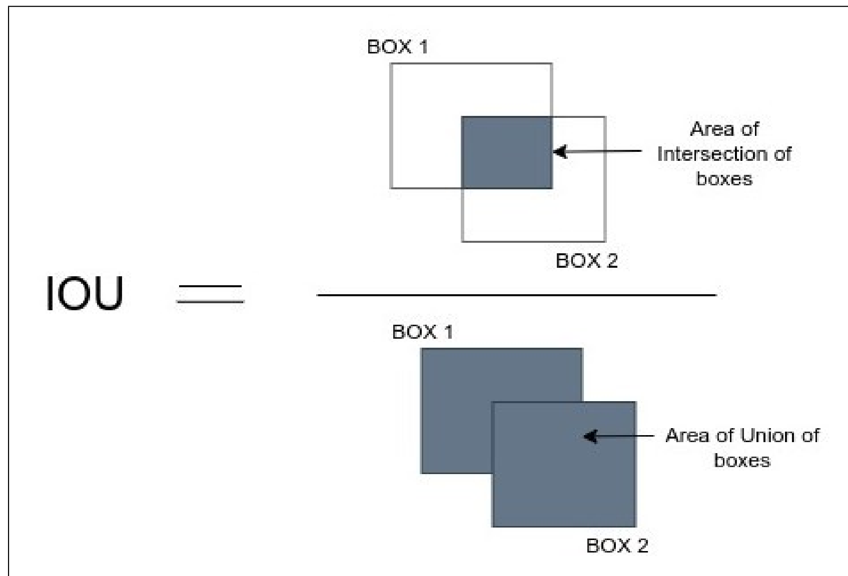

Figure 2.12. Intersection over Union (IoU) - Graphical representation.

The formula is:

$$J(A, B) = \frac{A \cap B}{A \cup B}$$

From a confusion point of view, it's possible to define the IoU as follows. For each class, the IoU is calculated as the ratio between the pixels that are correctly classified (true positive) divided by the sum of the total number of pixels that have been labeled with that class or that in the ground truth they belong to that same class. The numerator therefore corresponds to the intersection between the class pixels according to the predictions and according to the ground truth. The denominator, on the other hand, is the union of class pixels for the ground truth and for predictions. For each image, the average IoU value (mIoU) corresponds to the average of the values of all classes for the image considered. For the whole dataset, mIoU corresponds to the average of the values of all classes for all images. If the dataset is unbalanced, the average IoU value can be calculated based on the number of total pixels for each image. Unlike accuracy, here there are no problems related to how many pixels are present for each class, so this metric value best describes the quality of the resulting segmentation.

$$IoU = \frac{TP}{(TP + FP + FN)}$$

2.4.3 F-Score

In paragraph 2.3.2, F1-score is described as the harmonic mean of recall and precision. If a weighted parameter is added to that formula, the more general (and weighted) F-Score or Dice Coefficient is obtained. Considering beta as the weighting parameter, F-score can be expressed as:

$$F_\beta(precision, recall) = (1+\beta^2)\frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

From a confusion point of view:

$$L(tp, fp, fn) = \frac{(1+\beta^2)\cdot tp}{(1+\beta^2)\cdot fp + \beta^2 \cdot fn + fp}$$

With

- tp: true positives
- fp: false positives
- fn: false negatives

2.5 Loss Functions

The choice of the objective function for a deep learning application can be crucial and it is typically based both on the specific model and type of problem. In this section, some relevant losses for training image segmentation models are presented.

Jaccard Loss

$$L(A, B) = 1 - \frac{A \cap B}{A \cup B}$$

As one can notice by the expression above, it is a loss function which directly derives from the jaccard loss metrics, discussed in paragraph 2.4.2. Normally, when Jaccard loss is calculated all classes are considered, but there are some situations, i.e., an unbalanced dataset, where it's better to include only a few classes in the calculation. Jaccard loss can refer directly to the whole batch or can be calculated for each image of a batch and then averaged. Finally, a smooth parameter is introduced sometimes, to avoid division by zero.

Dice Loss

$$L(precision, recall) = 1 - (1+\beta^2)\frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

As well as the case of Jaccard Loss, Dice Loss is directly obtained by the metric with the same name. From a type I and type II errors point of view, it is written as:

$$L(tp, fp, fn) = \frac{(1+\beta^2)\cdot tp}{(1+\beta^2)\cdot fp + \beta^2 \cdot fn + fp}$$

Where tp, fp and fn stay for true positive, false positive and false negative respectively. β is an index for choosing the balance between recall and precision. As in the previous case, it's possible to include all the classes or only a fraction; loss can be calculated on the complete batch or for each single image and then averaged; a smooth parameter can be introduced to handle division by zero.

Binary Cross Entropy

$$L(gt, pr) = -gt \cdot \log(pr) - (1 - gt) \cdot \log(1 - pr)$$

In a two classes problem of classification, binary cross entropy compares each of the predicted probabilities to actual class output which can be either 0 or 1. Then, the score that penalizes the probabilities based on the distance from the expected value is calculated. Gt and pr are the ground truth and the prediction respectively.

Categorical Cross Entropy

$$L(gt, pr) = -gt \cdot \log(pr)$$

In a similar way to the previous loss, categorical cross entropy can be defined in this way. In this case it's possible to consider only a few classes instead of all.

Binary Focal Loss

$$L(gt, pr) = -gt\alpha(1 - pr)^{\gamma} - (1 - gt)\alpha pr^{\gamma} \log(1 - pr)$$

Parameter α carries out the same function of the weight in the F-score metric. γ can be considered as a focusing parameter that modulates the factor (1 - pr).

Categorical Focal Loss

$$L(gt, pr) = -gt \cdot \alpha \cdot (1 - pr)^{\gamma} \cdot \log(pr)$$

An objective function similar to the binary focal loss, but for multiclass scenarios. Here it's possible to consider only a part of the total number of classes.

2.6 Datasets

In this section, the main datasets for image segmentation are presented. Even if the core of thesis work is semantic scene segmentation, some datasets mainly used in image classification or in cases of non-semantic segmentation are analyzed. This is important because some modules of the models used during the work are trained or pretrained on datasets not specifically related to semantic segmentation tasks.

MNIST and Fashion MNIST

Modified National Institute of Standards and Technology database (MNIST) is a dataset of handwritten numbers from 0 to 9. 60 000 constitutes the training part and 10 000 the validation one. Each image is a greyscale 28x28 pixels and the dataset was born on the base of the old NIST database. MNIST is a very common dataset and is appropriate for a 10 classes classification problem. Nowadays, the main usage of this dataset is as a first test to understand if a model works properly. Good performance on the MNIST is necessary for a model to be good, but not sufficient to guarantee a good network behavior for more complex scenarios. Fashion MNIST has the same concept and characteristics of the former dataset, but instead of 10 digits, there are 10 different types of wearing articles. Images are directly taken from Zalando catalog. The introduction of the FMNIST was necessary because from a certain point of the story, models started to be too good for MNIST, always achieving accuracy higher than 99%. FMNIST is a little bit more challenging but remains simple enough to represent a good first step test for a new model.

ImageNet

ImageNet dataset is composed of more than 14 million of images, divided in more than 20 000 categories, thought for object recognition. This database represents a key element for computer vision and CNN training. Each year, as anticipated in paragraph 2.2.5, there is a competition, named ImageNet Large Scale Visual Recognition Challenge (ILSVRC), with the objective to test which model offers the best performance. This dataset was fundamental also for the GPU training of CNNs: the first time a model was trained using this hardware was in 2012, during the yearly ILVRSC edition. Even if ImageNet is a dataset for object recognition, in a lot of models used for segmentation, there is a CNN backbone trained on the ImageNet, making convergence faster for the main training of the segmentation model.

Pascal Visual Object Classes (VOC)

A very famous and versatile dataset for computer vision tasks is the VOC, which contains images for classification, segmentation, detection, recognition, and person layout. Nowadays, almost each model is tested on this dataset. For what concerns segmentation, 21 classes are present: vehicles, household, animals, airplane, bicycle, boat, bus, car, motorbike, train, bottle, chair, dining table, potted plant, sofa, TV/monitor, bird, cat, cow, dog, horse, sheep, and person, plus the background class. 1464 images are present for training and 1449 for validation. An extension of this dataset is represented by PASCAL Context, which increases to 400 the number of classes. It is more specific for segmentation tasks than the more general purpose VOC.

Microsoft Common Objects in Context (MS COCO)

MS COCO represents another big dataset used both for object detection and segmentation problems. There are 328 000 images which contain more than 2.5 million of object instances (labeled) subdivided into 91 different types. Images with objects present are taken in common situations. For the MS COCO detection challenge, there are more than 200 000 images divided into training, validation, and test set, with a total number of classes equal to 80.

Cityscapes

Cityscapes is a dataset with focus on semantic segmentation of urban streets, by means of short stereo videos recorded in the streets of over 50 cities with a high quality of the pixels. In total there are more than 5 000 frames from videos and 20 000 static images, all with the respective label associated. The classes are 30, subdivided into 8 macro classes.

ADE20K

All images belonging to this dataset have been annotated with objects. In turn, a good part of the objects was annotated with the parts that compose them. For each object, there is further information, such as the fact that objects are occluded or truncated. The images of the validation sets are fully annotated, while training set images have not annotated them exhaustively. Information regarding this dataset is constantly updated. In total there are 20 210 images for the training set, 2 000 for validation and 3 000 for the test set.

SUN-3D and SUN RGB-D

SUN-3D is a dataset containing a large quantity of RGB-D videos. For each frame there is the segmented ground truth and the depth information. In total there are more than 500 short videos with more than 300 different spaces and buildings. The SUN RGB-D contains more than 10 000 RGB-D images and is widely used for scene understanding tasks, providing a benchmark in this sense.

Part II

WORK DESCRIPTION

Chapter 3

PRELIMINARY STEPS

The aim of the practical part of the thesis is to perform a robust semantic scene segmentation with the usage of a suitable deep learning model. This segmentation must be the perception information used as the base on which the navigation algorithm for an autonomous indoor robot must be developed. The result must be a robot able to move in an indoor environment performing obstacle avoidance, by using camera information only.

The provided hardware consists of a Jetson AGX Xavier and a Turtlebot3 for the deployment, and a workstation for the deep learning part development. TensorFlow is assigned as the framework to use for the DL part. No other constraint is present. The first period of the work consists in getting information about ML, DL, scene segmentation and significant paper about indoor robot navigation based on semantic segmentation of the scene. All the concepts that are reported in part I of this thesis, such as basics of machine learning, networks for image classification and segmentation, are a resume of what has been learned during the initial period of the thesis: the aim is to have a strong base from a theory point of view to handle the actual problem. Once the theory and the state of the art are clear, it is necessary to learn the suitable tools that allow us to implement a deep learning model for segmentation.

As previously said, the framework is TensorFlow, so the basics are studied and some simple models are implemented in order to take confidence with data preparation, learning, validation and testing. First, a dense neural network is implemented to perform classification on the Fashion MNIST dataset, then by adding some convolutional layers, a model able to recognize if an input image represents a horse or a human is developed. The last step consists in experimenting with segmentation, in particular a Unet with a VGG16 encoder is hardcoded, trained, validated and tested to perform segmentation of animal images. In the following paragraphs of this chapter the description of the main hardware components, TensorFlow and a brief presentation of the state of the art are reported.

3.1 Hardware Presentation

In this section there is a description of the three most important hardware elements: Jetson AGX Xavier, Turtlebot3 and a workstation.

3.1.1 Jetson AGX Xavier

NVIDIA Jetson AGX Xavier is an embedded system-on-module (SoM), whose main characteristic is the presence of an integrated Volta GPU with Tensor Cores and Deep Learning

Accelerators that allow good performance in deep learning and computer vision applications. The limited dimension (100x87x16 mm) and power consumption (<30W) make this module suitable for autonomous robots which requires powerful hardware for accelerated AI applications. Here are the specifications and the performance (in terms of inference) of different CNNs elinux.org [2022].

Figure 3.1. NVIDIA Jetson AGX Xavier shape.

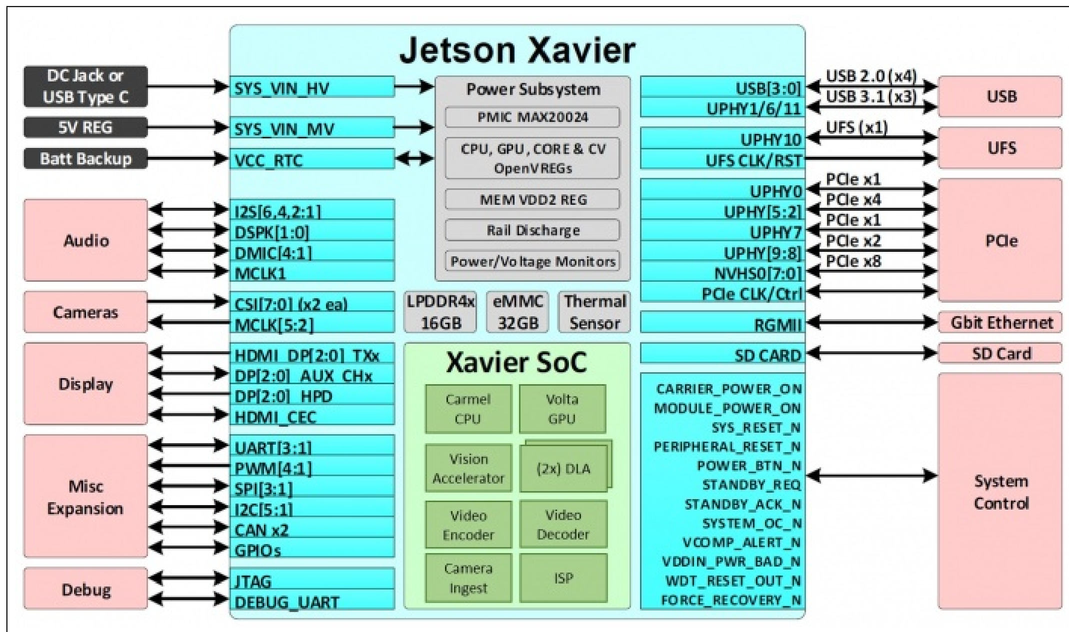

Figure 3.2. NVIDIA Jetson AGX Xavier architecture.

The software support part is based on the NVIDIA Jetpack, which main components are:

- L4T
  - Linux kernel 4.9
  - Reference filesystem based on Ubuntu 18.04 aarch64

DEVELOPER KIT I/OS

Developer Kit I/Os	Jetson AGX Xavier Module Interface
PCIe X16	x8 PCIe Gen4/x8 SLVS-EC
RJ45	Gigabit Ethernet
USB-C	2x USB 3.1, DP (Optional), PD (Optional) Close-System Debug and Flashing Support on 1 Port
Camera Connector	(16x) CSI-2 Lanes
M.2 Key M	NVMe
M.2 Key E	PCIe x1 + USB 2.0 + UART (for Wi-Fi/LTE) / I^2S / PCM
40-Pin Header	UART + SPI + CAN + I^2C + I^2S + DMIC + GPIOs
HD Audio Header	High-Definition Audio
eSATAp + USB3.0 Type A	SATA Through PCIe x1 Bridge (PD + Data for 2.5-inch SATA) + USB 3.0
HDMI Type A	HDMI 2.0
uSD/UFS Card Socket	SD/UFS

Figure 3.3. NVIDIA Jetson AGX Xavier Module Interface.

The Tech Specs

JETSON AGX XAVIER

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x | 137GB/s
Storage	32GB eMMC 5.1
DL Accelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Encoder/Decoder	(2x) 4Kp60 | HEVC/(2x) 4Kp60 | 12-Bit Support
Size	105 mm x 105 mm x 65 mm
Deployment	Module (Jetson AGX Xavier)

Figure 3.4. NVIDIA Jetson AGX Xavier Specifications.

- CUDA Toolkit
- cuDNN
- TensorRT
- Multimedia API
- GStreamer
- V4L2 media controller support
- VisionWorks
- OpenCV
- OpenGL / OpenGL ES / EGL
- Vulkan
- NVIDIA Nsight Systems
- NVIDIA Nsight Graphics
- NVIDIA Nsight Compute

For the specific case of the thesis, the 4.5 version of Jetpack is used, because, when the

work started, it was the latest jetpack version. Moreover, it is necessary to specify that in the work was specifically used the Jetson AGX Xavier Development Kit which bundles together all the pre-assembled parts for developers to get started creating applications and includes:

- Open source reference carrier board (105x105mm)
- Integrated active thermal solution
- Jetson AGX Xavier compute module
- USB 3.0 Type-A to Type-C cable (for flashing)
- USB 2.0 OTG adapter
- 19V power supply
- AC power cable

3.1.2 TurtleBot3

TurtleBot is a ROS standard robot. It is limited in terms of dimensions, manageable, programmable with its own API, ROS-based mobile platform with use in several fields such as prototyping and research, but for educational purposes too. TurtleBot3 has the advantage to be very limited in terms of price and dimensions but without lowering its functionality and quality, while at the same time giving the possibility to be expanded. The TurtleBot3 platform offers high possibilities of customization from different points of views depending on how the mechanical parts are reconstructed and optional parts such as the computer and sensors are used. Moreover, TurtleBot3 is developed to be suitable for robust embedded systems, by means of little and cheap SBC, 360-degree distance sensor and additive manufacturing technology.

Two different versions are present for the Turtlebot3: TurtleBot3 Burger, more compact and smaller, and TurtleBot3 Waffle Pi, larger in terms of dimensions. The last is the one provided as hardware for the thesis work. Here it's possible to notice the specifications, the design and the main components for a TurtleBot3 Waffle Pi.

3.1.3 Workstation

Workstation is the hardware used for the code development behind the final application. It is mainly used for the training of the deep learning model used to achieve a robust semantic segmentation. What is important for a workstation in a project like this is performing fast-enough training and this can be achieved by means of an NVIDIA GPU with high CUDA capability. In this case there is an RTX 2080, which is not the ultimate generation of NVIDIA GPUs, but good enough to train everything without big problems. In the following table the workstation specifications:

Manufacturer	INTEL
Hard Disk	512GB SSD NVME + 1TB HDD SATA
Processor Speed	3GHZ to 4.7GHZ
Operating System	WINDOWS10 PROFESSIONAL
RAM	32GB DDR4
CPU	I7-9700
Grade	A

3.2 TensorFlow

TensorFlow is an open source software python library for machine learning and deep learning, which provides tested and optimized modules, useful in the realization of algorithms for different types of perceptual and language comprehension tasks. It is a second generation of API, used by about fifty teams active both in scientific research fields and in production fields; it is the foundation of dozens of Google commercial products such as voice recognition, Gmail, Google Photos, and Search. These teams previously used DistBelief, the first generation of API. TensorFlow was developed by the Google Brain team and released on November 9, 2015, under the terms of the Apache 2.0 open source license. TensorFlow is compatible with major 64-bit operating systems (Windows, Linux and Mac OS X) and Android. Although at the beginning the official documentation spoke of limited hardware compatibility, the library can work on numerous types of CPUs and even on GPUs, thanks to the support of languages such as CUDA or OpenCl. In addition, Google has designed and built an ASIC processor specifically dedicated to this language, called TPU (Tensor Processing Unit), with a computing capacity of 180 teraflops, in the second version.

3.3 State of the Art

In chapter 2 the most used models to perform segmentation have been analyzed, with the description of the structure, the segmentation results achieved on the most used datasets and the specific learning processes. Scene segmentation is a specific kind of semantic or panoptic segmentation that has, as aim, the analysis, from a perception point of view, of a particular environment or a part of it. As previously specified, in this thesis work the objective is performing real-time semantic scene segmentation to achieve indoor robot navigation. As also seen in chapter 2, segmentation is a general technique, often obtained via deep learning, that can be applied in a lot of different fields, so the examination of the state of the art can result useless if done from a general point of view and is more powerful if bounded to the interested scope.

The applications of segmentation in navigation problems are mostly related to outdoor scenarios, such as streets for autonomous car driving; this is the reason why there are no public semantic segmentation datasets to perform indoor robot navigation. By contrast, some works that address this problem have been published in the last year, defining the state of the art in indoor scene semantic segmentation for autonomous navigation.

One of the most relevant works in this sense is proposed by Yao Yeboah, Cai Yanguang. Wei Wu and Zeyad Farisi with “Semantic Scene Segmentation for Indoor Robot Navigation via Deep Learning”. The idea presented in this paper is to start from deep convolutional neural network-based semantic scene segmentation to make an autonomous robot, with limited embedded hardware, to navigate autonomously in a domestic environment, putting the main focus on corridor scenarios.

The main part of the paper is about performing good and robust semantic segmentation on top of which a navigation algorithm can be developed. To reach this goal, models with encoder-decoder structure are chosen by the authors, in particular a SegNet, analyzed in paragraph 2.2.3 and a Fully Convolutional Network, considered good choice to detect the navigable part of the scene for the robot through semantic segmentation and let the robot moving with a collision-free motion, while deploying the software in a limited embedded hardware.

For the training of the two networks the approach is based on transfer learning, this is

done because, as said previously, public scene semantic segmentation dataset for indoor scenarios are just a few and not suitable for segmentation that is done to achieve navigation, so where the most important class is represented by the usable path Yao Yeboah and Farisi [2019]. A solution can be collecting images and doing manual labeling but this is a very time-consuming solution, especially if the aim is to build a balanced and large-enough set of images. To handle these problems, as said, the solution is to go for a two phases transfer learning: pretraining and fine-tuning.

In the former step, what is done is to train the backbone of the model (in this case a VGG16) on the ImageNet dataset: as seen in paragraph 2.6, this is not a set suitable for segmentation, being built for image classification, but this allows the model to learn low-level features, for example boundaries or object shapes, that results useful in the majority of the tasks related to computer vision. At the end of this phase what is obtained is a model that is not able to perform semantic scene segmentation, but with the weights of the encoder that are near to the suitable values to achieve good enough segmentation.

For the pretraining phase, in order to significantly reduce the training time, the first 13 layers of the VGG16 are taken already trained on the ImageNet dataset from a work with a different objective. Then, the fine-tuning step is entered: after the models are pretrained until there is no more improvement in terms of loss and accuracy, the training for segmentation starts. With the use of the camera mounted on the robot, a set of videos are recorded while the robot moves with manual driving, then these videos are split in order to obtain about 2000 single images. Each image is labeled, assigning at each pixel the floor class or the obstacle class. During the frames collection the attention is put in taking videos with textures, light conditions, obstacles that varies, to obtain a dataset with variance high enough to get a robust segmentation. No rotation of the images is considered since the z coordinate of the robot remain always the same during the motion.

Collecting the fine-tuning dataset in the same environment where the robot will move through real-time segmentation is a solution that makes the network able to do good predictions, but there is no guarantee that the segmentation is good in different indoor scenarios than the one where it has been tested. In the training process, stochastic gradient descent is used, with batch size equals to 8 with batch shuffling at every step. This is a common choice that regards all the 4 models trained.

The hardware used for the training process results in a pair of GTX 1080Ti by NVIDIA and Intel i7 processor (generation is not specified) with CUDAv8 and cuDNNv5 as libraries for parallel computation. When the fine-tuning is terminated, a series of post-processing operations are made by means of a series of morphological operations that make the edge improved, connecting disjoint pixels that result from the raw prediction. In total there are four different models trained, the first is a SegNet, while the other three are FCNs with little changes in the hyperparameters. In table 3.1 they are shown in detail. In figure 3.5 qualitative examples of predicted images are reported. Each row represents a model and each pair of columns reports the image overlapped by the mask and the inference. In the second column, black represents the true positive and purple the false positive, while the last column reports the true positive in purple.

Tables 3.2 and 3.3 show in detail the results of the model on the test set, for pixel accuracy, Jaccard score and BF score, with all data referring to the mean values. Observing the reported score, one can notice that there is no difference between the models in terms of pixel accuracy and IoU score, reaching values of more than 98% and 96% respectively. For the BF Score there is a little difference, with the SegNet model reaching only 84% and the FCNs obtaining 92%, 90% and 84%. Although some differences are present, they are very small and all networks perform segmentation with very good results from a quantitative point of view.

Models	Learning Rate	Momentum	Epochs
Model-1	1e-3	0.9	100
Model-2-v1	1e-3	0.9	50
Model-2-v2	1e-3	0.9	50
Model-2-v3	1e-3	1.0	100

Table 3.1. Training parameters.

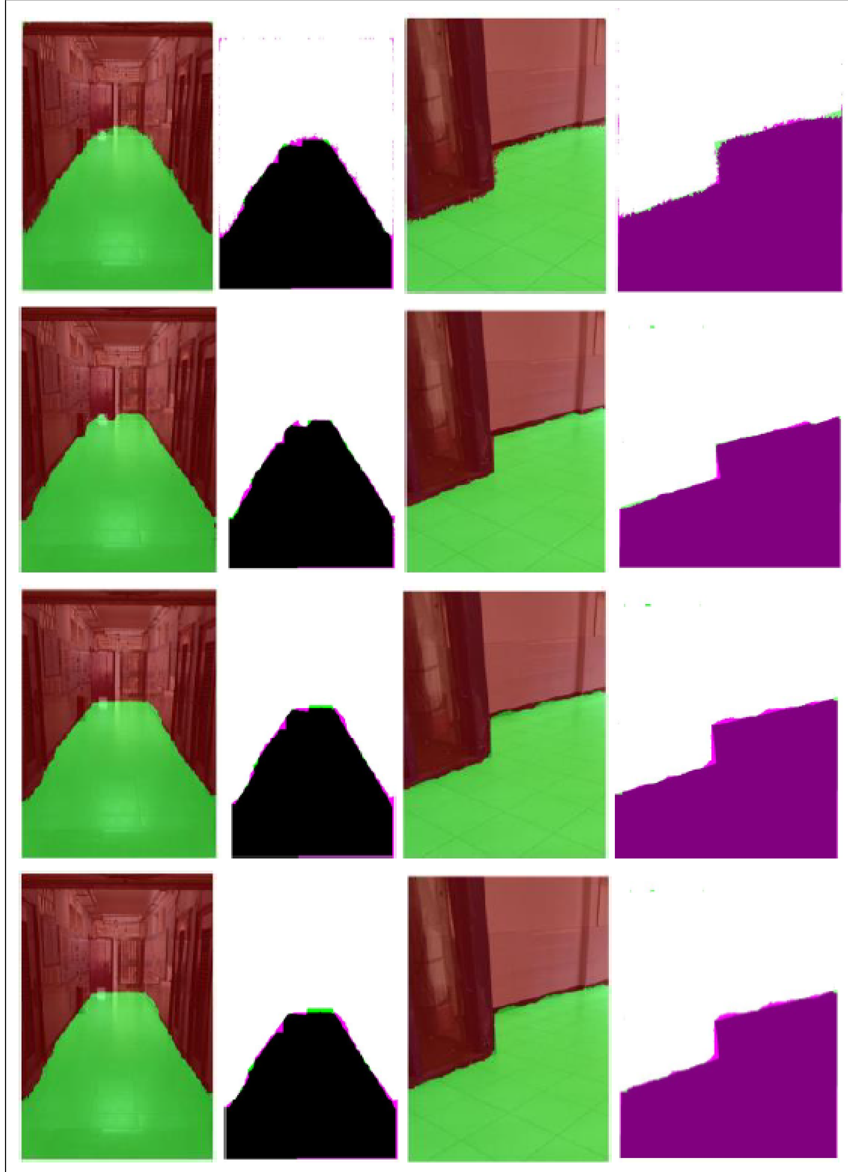

Figure 3.5. Qualitative examples of predicted images.

The biggest difference, as is shown in the table is on the base of the memory occupation: SegNet performs better in this sense, with a 111 MB weight compared to the 501 of the FCNs, being a much better candidate in condition of limited hardware. As a consequence,

the load time of the model is smaller in the SegNet, 0.8s compared to 3.1/3.2s of the FCNs, while for the inference time the difference is only 0.05s: 0.18s for the former model and 0.23s for the heavier ones.

Models	Accuracy	IoU	BFScore
Model-1	0.9835	0.9669	0.8444
Model-2-v1	0.9905	0.9802	0.9293
Model-2-v2	0.9882	0.9756	0.9036
Model-2-v3	0.9840	0.9664	0.8477

Table 3.2. Metrics results on test set.

Models	Memory (MB)	Load Time (s)	Inference Time (s)
Model-1	111.1	0.8	0.18
Model-2-v1	501.2	3.2	0.23
Model-2-v2	501.2	3.1	0.23
Model-2-v3	501.3	3.1	0.23

Table 3.3. Memory and time performance.

Another relevant work is “Corridor segmentation for automatic robot navigation in indoor environment using edge devices” by Surbhi Gupta, Sangeeta R, Ravi Shankar Mishra, Gaurav Singal, Tapas Badal and Deepak Ga.rg that proposes semantic segmentation with focus on corridor scenarios with a region-based method that uses a series of mathematical operations such as gray conversion, detection of the edges and some morphological operations, so segmentation is achieved without using any deep learning model. After the pipeline, pixels are classified as part of the corridor class or part of the obstacle class.

The pipeline is composed of the following steps: data acquisition, frame extraction, pre-processing, edge detection, line extraction (both horizontal and vertical), slope extraction and actual corridor segmentation. The embedded hardware for the autonomous robot is a 64-bit Raspberry Pi 3B+. First, by means of an RGB camera directly connected to the raspberry (camera is the only sensor of the robot), videos are acquired while the robot moves at speed of 0.6 m/s, then the recordings are split in order to obtain RGB images with 1080x1920 resolution. Videos are taken with varying conditions in terms of illumination and reflection, as table 3.4 shows.

The total number of frames can be different in recordings with different environment conditions. Once the frame extraction is completed, the pre-processing phase is entered with each image that is converted into grayscale, using the BT.601 formulation Helland [2011]. With the frame in the greyscale and with the Gaussian blurring applied, the edge detection step is carried out and represents the crucial phase, because of the importance of edges in corridor segmentation Gupta et al. [2020]. A convolution filter with a specific kernel is applied on the image in order to operate a discrete approximation of the pixel values and then a suppression of the pixels that are not part of the boundaries is carried out.

The extraction of both horizontal and vertical lines is the next passage of the pipeline and is made through a series of morphological operations: dilation, erosion and closing. The use of dilation is made with the objective of adding pixels at the boundaries of the objects in the image, by contrast erosion operation has the effect to remove pixels at the

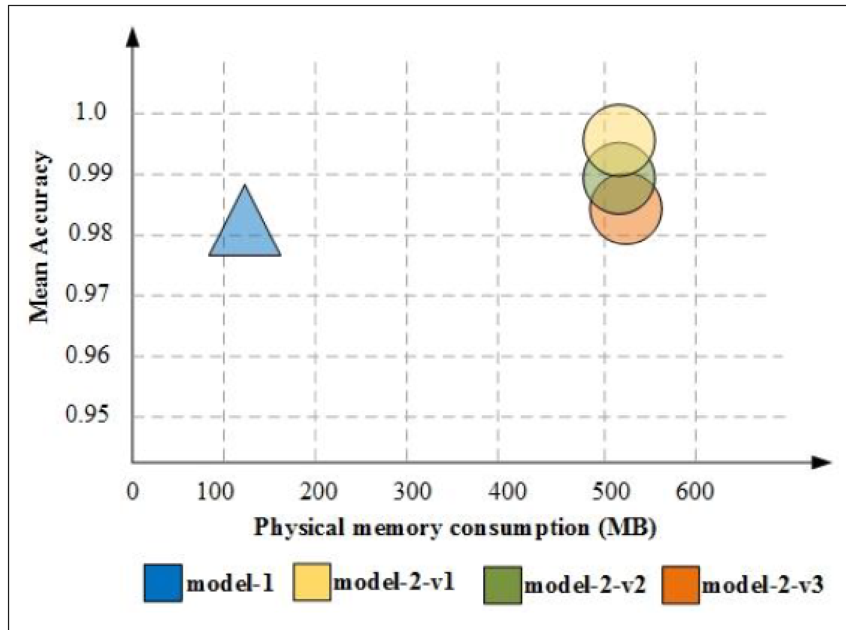

Figure 3.6. Metrics vs memory consumption models comparison.

Video Environment	Total number of videos	Total number of Frames
High Illumination	9	2127
Low Illumination	8	1561
Reflection	12	3032
Standard	11	2139
Total	40	8859

Table 3.4. Number of acquisitions on the base of light conditions

object boundaries. In this way, both horizontal and vertical lines are obtained, then on these bases the slope extraction phase is entered and allows to obtain two lines with the appropriate slope that define the lateral limit of the corridor region. These lines with the best candidate horizontal one and the bottom horizontal limit of the image, defines the definitive segmented corridor area, that is the actual space that the robot can use to navigate. In table 3.5 all the results in terms of TP, FP, FN, precision, recall, f1 score and accuracy are reported changing the reflection and illumination environment conditions. As one can notice, the accuracy values are in the 50-71% range, resulting significantly worse than the ones of the former study, where more than 98% is achieved. This clearly shows that an approach not based on deep learning is not a good choice, according to the metrics. On the other hand, it is necessary to consider that in this study the hardware used is only a Raspberry Pi, so there is a very big constraint in this sense and a DL model would not work in such a limited hardware.

Daniel Teso-Fz-Betoño, Ekaitz Zulueta, Ander Sánchez-Chica, Unai Fernandez-Gamiz and Aitor Saenz-Aguirre in 2020 proposed a work, called "Semantic Segmentation to Develop an Indoor Navigation System for an Autonomous Mobile Robot" with the aim of training a suitable deep learning model for semantic scene segmentation in order to make an AMR able to recognize the viable path and on this base developing a navigation

Video Type	TP	FP	FN	Precision	Recall	F1 score	Accuracy
High Illumination	1328	138	661	0.91	0.69	0.77	62%
Low Illumination	781	405	375	0.66	0.68	0.67	50%
Reflection	2155	247	630	0.90	0.77	0.83	71%
Standard	1328	312	499	0.81	0.73	0.77	62%
Total	5592	1102	2165				

Table 3.5. Test resutls.

algorithm to achieve obstacle avoidance. The only sensor used for the mobile robot is an RGB camera.

In paragraph 2.1.4 the Residual Neural Network for image classification is described and it represents the starting point for the choice of the model in this work. The selected ResNet-18 is used as the backbone of an encoder-decoder structured model that is built on the base of the DeepLabV3+ structure, in order to make the model suitable to perform semantic scene segmentation. The learning process is developed in three main phases: the acquisition of the data, the preparation of the data and the actual training. For the acquisition what is made is a collection of images in the environment where the robot will move. At the end of the acquisition phase, there are 391 images ready to be labeled. The idea is to build an ad-hoc dataset since the availability of public dataset specific for indoor segmentation for navigation is poor. The drawback of this is that, since the robot will move in the same environment where the images of the training dataset are taken, there is no information about the robustness of the obtained segmentation: it is impossible to know if it would perform well even in a different indoor environment. The collected images are then part of the manual labeling process. In this specific case, the semantic scene segmentation discriminates two classes (binary segmentation): the pixels labeled in blue are the viable path, while the ones in orange are part of every object that is not floor, such as walls, chairs, people, ceiling and any other obstacle. What is done in a nutshell is manually building a mask for each collected photo. The complete dataset is then organized putting the 80% of the elements in the training part, 19% are reserved for the validation dataset and the rest is for testing. For the images selected for the learning process, improvements are introduced such as data augmentation with reflection and translation function along both the horizontal and vertical directions. Authors decided to avoid the presence of flipped or rotated images, because the robot actually moves only on the X and Y plane, maintaining the Z coordinate constant. At this point all the elements are ready to start the training of the neural network so the hyperparameters are set. In this study they are not optimized Teso-Fz-Betoño et al. [2020a]. 100 is the maximum number of epochs in the training, but some callbacks functions are developed to perform early stopping if the loss on the validation set does not improve for a certain number of epochs (validation patience). A batch of 10 images is used for validation at each step of the learning process, to calculate the validation loss and accuracy. The metrics are evaluated once every 50 epochs. Learning rate is set initially to 0.003 in order to avoid a too slow process, but at the same time getting a good sub-optimal solution (or maybe the optimal). Learning rate is not necessarily always the same during the whole training: with a callback if the loss does not decrease for a certain number of epochs (learn drop period), the rate is reduced for a certain quantity (learn drop rate factor). The L2Regularization is present. In the learning, the gradient threshold method is active and consists in controlling a learnable parameter: if it results to be bigger than a set threshold (gradient threshold), the gradient

is scaled according to a function, such as the L2 norm. The dataset is shuffled at each epoch. In table 3.6 all the details and values about the used hyperparameters are reported.

The Training Parameters	Values
Learning Rate Drop Factor	0.1
Learning Drop Period	5
L2 Regularization	0.005
Gradient Threshold Method	L2norm
Gradient Threshold	Infinite
Validation Frequency	50
Validation Patience	4
Shuffle	Every epoch
Learn Rate Schedule Setting Method	Piecewise
Learning Drop Rate Factor	0.01
Learn Drop Period	5
Initial Learn Rate	0.003
Max Epochs	100
Mini Batch Size	10
Momentum	0.9
Solver	sgdm

Table 3.6. Hyperparameters configuration.

After the training, the network is ready to perform inference to help the navigation algorithm. It is developed on the base of the predicted mask and consists in a sort of path calculation on the part of the image recognized as navigable. The analysis and calculation are done at the pixel level.

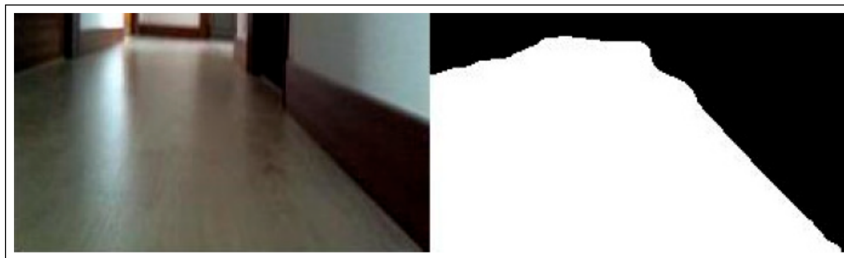

Figure 3.7. Example of inference.

The path obtained is then used to create the navigation commands for the AMR. In this study, there is no embedded hardware, but a laptop is physically mounted on top of the movable part of the robot, a big difference with the other works where there are some limitations in this sense. With the described method, that is image acquisition, image inference, path calculation, command extraction and publication, the system needs between 1.05s and 1.24s to produce a navigation command using CPU. The hardware of the laptop is a GTX 660 Ti as GPU and 4GB of RAM DDR5. No details about CPU are given. CUDA libraries are used for inference acceleration. With GPU the network can produce a segmented image in less than 0.1 seconds, but with path calculation and definition of the navigation command, the robot needs 1s to actually react.

The study finishes with the implementation and the comparison of performance using

similar models: DeepLabV3+ with ResNet18 and ResNet50m SegNet and U-Net with VGG19 as backbone. Metrics for all the models are collected and reported in table 3.7. From the data, it's clear that the first two models behave well in every situation, while SegNet has good results for what concerns accuracy and Jaccard metric, but the BF score is only 68-76%, compared with the 90% of the other models. U-Net, as one can notice, has significantly lower values in every metric compared to every other model used in this work.

Name	ResNet-18	ResNet-50	Segnet(vgg19)	Unet
Floor Accuracy	0.9859	0.9934	0.9768	0.7383
Floor Intersection over Union	0.9608	0.9775	0.9318	0.6322
Floor MeanBF Score	0.8516	0.8943	0.6856	0.2916
Wall Accuracy	0.9836	0.9898	0.9697	0.8945
Wall Intersection over Union	0.9750	0.9857	0.9558	0.7682
Wall MeanBF Score	0.8786	0.9169	0.7636	0.4744

Table 3.7. Test results.

Chapter 4

SCENE SEGMENTATION

As described in chapter 3, the approach to this work consists in discomposing the whole problem in two main modules: the first has the objective to perform a semantic scene segmentation with high score in terms of metrics and robustness. Then, on this base the navigation module will be implemented with the final objective to generate the motion command for the robot. In this chapter, the focus is on the analysis of the segmentation module, such as the model choice, dataset construction, training process and test results both from a quantitative and qualitative point of view.

4.1 NVIDIA SDK Isaac

In the presentation of the state of the art of indoor navigation based on semantic scene segmentation, a recurrent issue is about the choice of the dataset. From a more detailed perspective, public and widely used dataset for indoor segmentation are present, for example the ADE20K and the SUNRGB-D, but they are datasets with a large number of classes that goes from 13 up to 150. To perform a scene segmentation that can result in useful indoor navigation tasks, it is crucial that the floor class is labeled as best as possible, because it is on those pixels that the navigation command is calculated. The problem with the named datasets is about a not large enough presence of floor elements and this results in segmentation of a high number of elements that are completely useless for navigation (i.e. a window or a door), while the segmentation of the pixels of a hypothetical navigable arises not sufficiently good.

To address navigation tasks is more convenient a dataset with binary masks that give a high attention to the floor class. Public datasets with this structure are just a few and usually related to a specific work and that's why in all the studied papers the dataset is built autonomously with manual labeling. In this thesis, the same kind of problem is present and the first try for a suitable solution needs the usage of the NVIDIA SDK Isaac.

Isaac is a platform for robotics powered by the Isaac software development kit (SDK) and offers a big number of high performant algorithms working on GPU (GEM) for manipulation tasks, navigation and computer vision. With the usage of the related engine it is possible to develop applications with Isaac (C or python API) and do the deployment on NVIDIA hardware platforms such as Jetson Xavier or Jetson NANO. Isaac already offers some basic applications with the purpose of both presenting some examples of what it's possible to do and giving pre-developed starting points to the developer to make the construction of advanced applications easier. It is also possible to do development and tests using the Isaac Sim, a virtual environment specific for robotics.

The “Free Space Segmentation” Isaac GEM is specifically implemented for semantic segmentation of outdoor and indoor scenes. The aim of the GEM is to segment images and videos into classes of interest such as navigable space and obstacles. The provided input is an image or a frame of a video, while the output is represented by the mask of the segmented image. The GEM proposes an easy way to train a model for the segmentation of the free space in a simulated environment and then uses it to carry out inference in a real-world scenario.

4.1.1 Data Collection

For what concerns the dataset problem for indoor segmentation, Isaac proposes two different possibilities: simulated data and autonomous data collection.

The first solution consists in building a virtual scenario (“Free Space Segmentation” uses Unity3D) that simulates a real indoor environment in which the synthetic images can be collected to build the dataset. In virtual environment, it is possible to achieve domain randomization in various ways, such as changing the level of intensity and the colors of the lights (light randomization), using different kind of materials (material randomization), applying different textures on the materials (material randomization), changing color to the textures (color randomization), changing the reflection and refraction properties of the materials.

The second solution is related to a pre-developed application of Isaac that directly allows autonomous collection of data. What is necessary is simply a Jetson Platform to deploy the app and a Carter robot (a specific mobile robot powered by NVIDIA). For the motion of the robot it’s possible to do both manual drive navigation and autonomous navigation giving it a set of waypoints to reach, while images are collected. Each time an image is taken, the mask is automatically produced by a set of built-in algorithms based on the split of the image into a batch of superpixels and clustering.

Considering the hardware available for the thesis, this last solution can’t be used for the definition of the dataset, simply because there is not a Carter bot. A solution could be to try to do an exact replication of the Carter using the TurtleBot, but this is a time-consuming process with low possibilities to work, so the solution that involves the synthetic environment is chosen. A warehouse scenario is built in the Unity3D environment, based on a similar pre-existing scenario: objects like boxes, shelves, trash bins, windows are created and inserted, floor and walls with different textures and materials are present and light conditions change in time. It’s important to remember that in the Unity3D environment, there is a function that automatically segments the scenario, so for each image collected the correct mask is available. In the synthetic warehouse a camera spawns in a random position with 5Hz frequency, and each time collects an image to add to the dataset. Photos are taken at totally random points, with different angles and heights. It is possible to set the maximum inclination that the camera can have: 15 degrees is considered a reasonable value, because it allows to add a little more variability in the dataset, maintaining a realistic situation where the robot camera is still at zero degrees and fixed height. A similar idea led to the choice of 0.20m as the maximum change of the z coordinate during data collection. Random objects (boxes, bins, shelves) spawn and disappear with the same camera frequency to increase the variability of the scene. In other words, it’s like having a Carter robot with the autonomous data collection application of Isaac, but in a virtual environment. The application parameters that can be changed are the following:

- Number of collected images: 60 000

- Camera frequency: 5Hz
- Object spawn rate: 5Hz
- Max camera rotation angle: 15deg
- Max z translation value: 0.20m

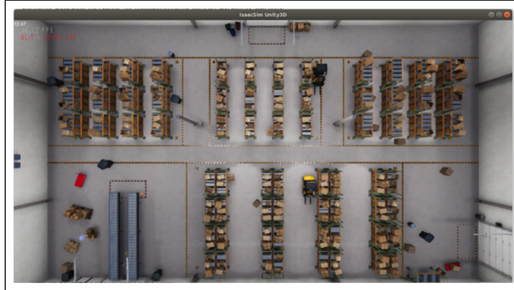

Figure 4.1. Warehouse top view.

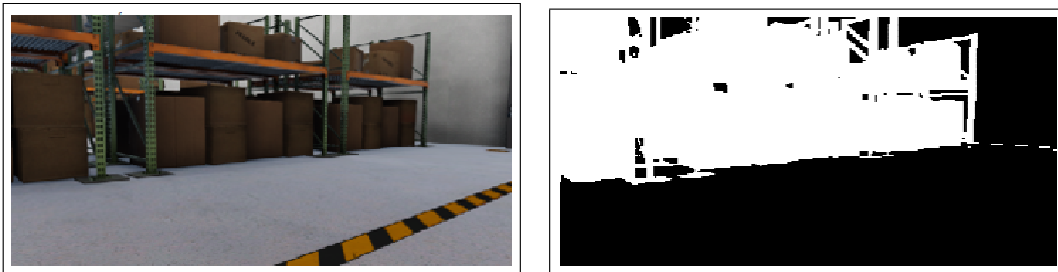

Figure 4.2. Example of acquired image and label.

4.1.2 Training

"Free space segmentation" Isaac GEM already offers a series of nodes that are suitable to perform training while the data collection app is running. TensorFlow is the framework on which all deep learning tasks present in Isaac work. TensorRT is then used for model optimization and inference. The pre-existence network in the Isaac app is a vanilla U-Net (encoder-decoder structure) with a VGG13 backbone. For what concerns the set of the trainig, it's important to remember that no callbacks can be implemented and the choice of the hyperparameters can't be done directly on the TensorFlow script, but it's necessary a specific node. The size of the dataset is of 60 000 images with 10% used for validation, and the rest for the learning process. A batch size of 128 is set, with 0.003 as the value of the learning rate: this (first) choice is inspired by Teso-Fz-Betoño et al. [2020a]. The number of epochs (fixed) is 100, but the final model is taken as the one with the lower loss value during the process. Binary cross entropy is the metric, since the aim is to perform two classes segmentation. The complete pipeline (collection + training) is run on a GTX 1060 3GB, AMD FX-8350 with 4.2Ghz clock, 16 GB RAM DDR4 workstation and takes about 33h to complete. Because of the pre-built settings, there are no validation and training metrics during the training process, so it's necessary to export the weights of the model in native TensorFlow environment and do the calculation to get a quantitative evaluation. What the Isaac application allows us to do is image and video inference, for

a qualitative evaluation of the trained model in doing segmentation. In total the training process is repeated three times with the hyperparameters values reported in table 4.1.

Images	Epochs	Learning Rate	Batch Size	Loss	Shuffle	Metric
60 000	100	0.003	128	Binary crossentropy	Yes	Accuracy
40 000	50	0.001	64	Binary crossentropy	Yes	Accuracy
60 000	100	0.01	128	Binary crossentropy	Yes	Accuracy

Table 4.1. Hyperparameters values of the training process.

4.1.3 Results and Conclusions

As one can notice in figure 4.3 just from a qualitative point of view the results of segmentation are clearly not good enough to be the base for a navigation algorithm. The test image put in the network is just a corridor with no obstacles, but the central part of the floor is full of pixels labeled as obstacles (FN). The image is a test example of the first of the three training but other cases result in similar situations.

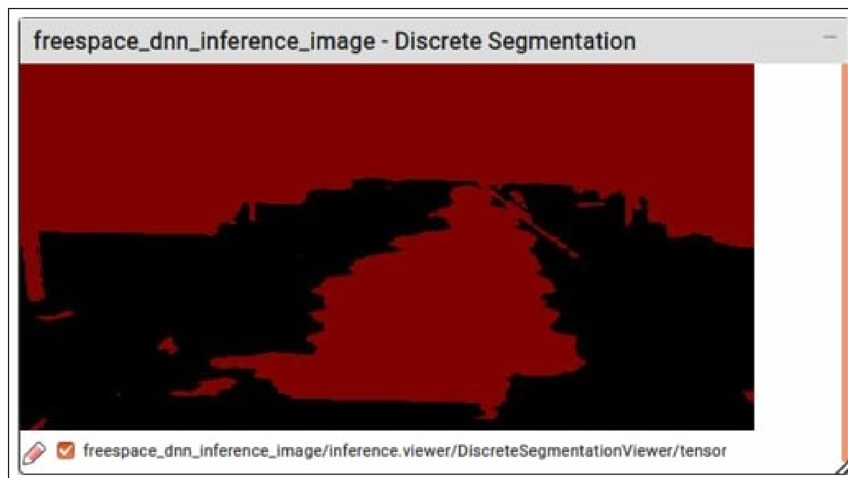

Figure 4.3. Isaac inference result.

Viewing these qualitative results, it is considered not necessary to export the models and go for a more significant metric evaluation to judge the resulting model not good enough for the wanted application. To try to improve the results, two solutions are considered: trial and error approach for hyperparameter tuning and a different model than the VGG13 U-Net proposed by Isaac.

The first one was actually already carried out in the three different training previously presented, where the values of learning rate, batch size and datapoints change. It's necessary to consider that for the complete pipeline described (data acquisition in Unity 3D synthetic environment + actual network training), a time between 25 and 36 hours is spent, depending on the quantity of datapoints. Considering that the results are not

improving and the amount of time necessary for each training, the trial and error process is stopped at the third model.

The last solution to improve segmentation is based on the idea of switching the predefined VGG13 U-Net with a different model and doing some tries with a ResNet-50 U-Net and LinkNet, so what is practically made is, using TensorFlow API, do some change in the Isaac node that contains the hardcoded model, in order to try with a model, according to the literature Chaurasia and Culurciello [2017], more suitable for scene segmentation. However, even if the model is modified in the proper node, the complete Isaac app does not work, because of some mismatch errors in the nodes. This is an issue that involves the source C code of some specific nodes in the complete app, because of the lack of support of Isaac to Unity3D (support completely ends in 2022).

At this point, the decision to stop trying to achieve scene segmentation with Isaac is taken, mainly because of these remarks: from a setup point of view, there are problems among the nodes that are very difficult to solve; Isaac team is stopping supporting on Unity for its applications; the results obtained with the supported settings, such as the VGG13 U-Net model are not good enough; images collected in a synthetic environment, with this model and these hyperparameters, led to not sufficient scene segmentation results; an hypothetical trial and error approach for the correct tuning of hyperparameters takes high quantity of time and it doesn't seem to lead to any better result in terms of segmentation; the community related to this SDK is very small if compared to the ROS and ROS2 one, so it is more difficult to find user solutions to similar problems, detailed documentation or Isaac app developed by other users to handle similar tasks. So, what is observed is that Isaac, to achieve segmentation tasks offers some pre-developed solution, that is this specific case, for the reasons previously described, lead to segmentation results that are not good enough to achieve the thesis objective and so the decision to develop the segmentation module in a different way, that does not involve Isaac SDK, is taken.

4.2 Working Approach

As reported multiple times in the examined literature, obtaining a suitable dataset for scene segmentation-based navigation is a crucial topic. With an approach that can't refer to an SDK anymore, it's necessary to directly write the code for everything in the pipeline. For this reason and considering the poor results achieved in Isaac with a synthetic dataset, the choice is to avoid a collection in a virtual environment and go with a different approach.

Most of the works analyzed in paragraph 3.3, independently from the used model, prefer to train the backbone on a big and standard dataset for image classification, such ImageNet, that is not specifically built for indoor segmentation, but allows the network to learn the low and mid-level features of the images, such as object shapes and edges. Then, when this step of transfer learning is done, there is a fine-tuning one where the complete model is trained with the specific dataset, that is usually precisely built for the application. The poor availability of public big datasets for indoor scene semantic segmentation for navigation is the main reason why this way to operate is so popular Yao Yeboah and Farisi [2019]. Considering this, the decision to go with a similar approach is taken, so it is necessary to get the dataset for the fine-tuning phase. Before continuing with the description of the practical work, it is important to remind that the idea is to perform a binary segmentation, where the two classes are the navigable part and the non-navigable part, which includes any kind of obstacles, such as people, chairs, tables, boxes, the walls and the ceiling. As reported in Helland [2011], this choice has the advantage of higher possibilities to have a segmentation with high metrics values, than the multiclass

case. This is a crucial point, because the task to achieve is to perform navigation only on the base of the segmented images, so the better the model is to recognize the pixels of the travelable part, the more accurate is the navigation. Once the dataset for the fine-tuning is collected, the idea is to perform training with different models recurrent in literature, such as U-Net, Linknet, PSPNet or SegNet, with different backbones, like VGG, Resnet or mobilenet and find the best combination from a quantitative point of view, based on a metric value. For each model, a trial and error approach is thought for the hyperparameters tuning.

4.3 Dataset

A recurrent approach in the examined literature in paragraph 3.3, consists in building an ad-hoc dataset for the specific application for the fine-tuning step of the training. A binary segmentation database used only on a model pre-trained on a bigger dataset can be limited in terms of dimension: few hundreds of images with the corresponding true masks can be sufficient for a good enough RGB scene segmentation. In this sense, in this thesis the covered idea is looking for an available public suitable dataset for a similar application on the web and if the research is not successful, proceeding with the autonomous collection of images. Based on the works defining the state of the art, the characteristics to have a dataset to be suitable for this application must be the following:

- Binary masks: floor e non-floor classes
- Indoor scenarios
- Focus on the floor class
- Presence of obstacles
- Suitable size: from a few hundreds to some thousands

The best result of a research on the web in this sense, is a dataset built for binary segmentation with the following ID: “16yHVv2HIV2pqJj-Z1k8gRSb31enJeMdm”. The dataset was found on Google drive with the label: “Corridor segmentation dataset” and is not specifically related to a segmentation work, but considering the name and the content, it is reasonable to think that it was built to perform binary corridor segmentation. The total number of frames is 967 (each one with its correspondent mask), so based on what is reported in the investigated literature, there is a proper quantity of images to perform fine-tuning. All the frames represent indoor scenarios with a big presence of corridor images, with a variety of textures, scenarios, angles, height, light conditions and there is a recurrent presence of obstacles, such as chairs, bins and people. For what concerns the masks, it’s possible to notice very different shapes of the travelable parts (fig. 4.4). Comparing the features of the dataset with the ideal characteristics previously reported, it is possible to say that it can be a good candidate for the fine-tuning step in the scene segmentation module of the work.

4.4 Models

For the selection of a suitable model the starting point is the examined literature. In applications presented in paragraph 3.3 that involve deep learning, the recurrent models are the U-Net elinux.org [2022] Gupta et al. [2020], the LinkNet and the SegNet elinux.org [2022] with ResNet and VGG as the most recurrent backbones elinux.org [2022] Helland [2011] Gupta et al. [2020]. It’s important to remark that the chosen work approach is to do a pre-training of the backbone on the ImageNet dataset and then a fine-tuning using

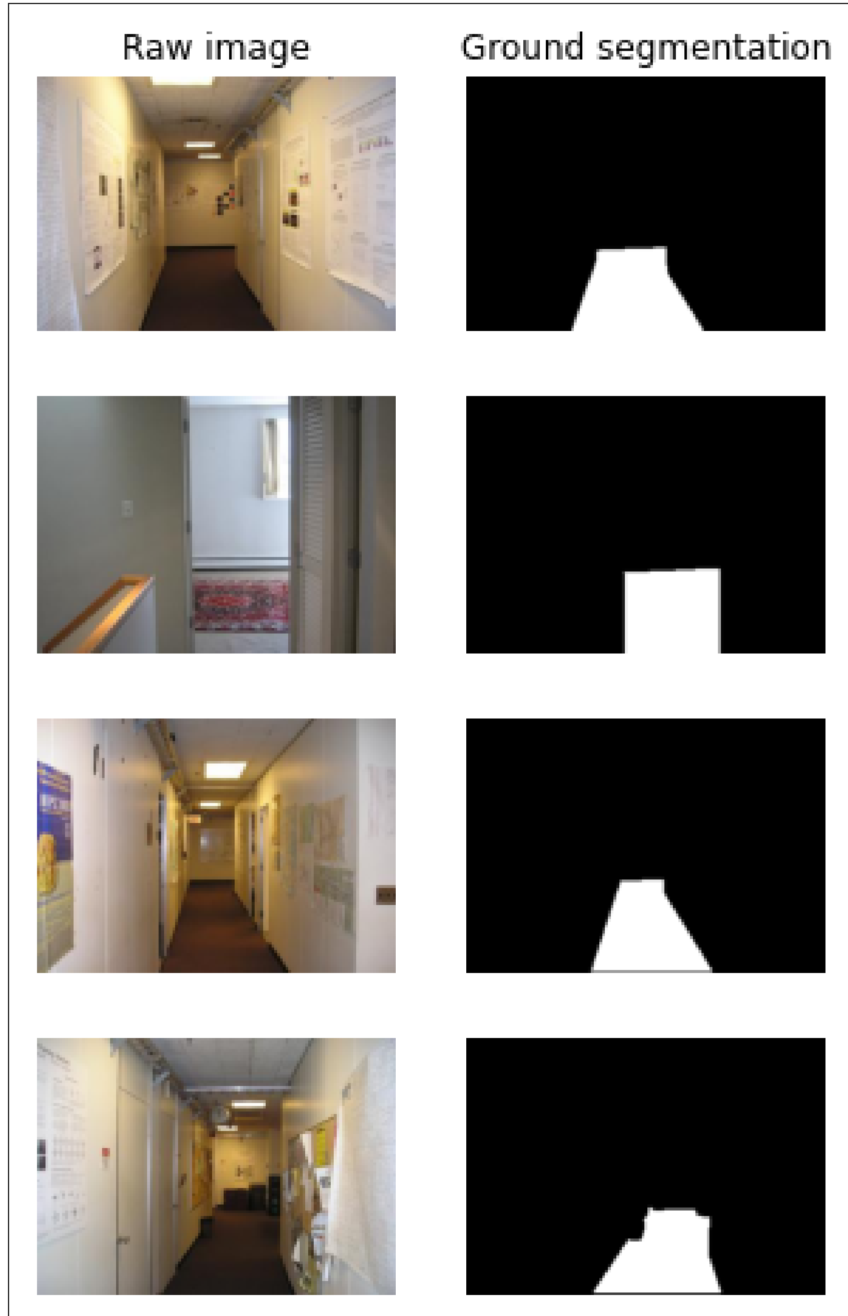

Figure 4.4. Dataset inspection.

the dataset of the paragraph 4.3.

Moreover, in order to obtain the best results with a certain model, the idea is to do an accurate tuning of the hyperparameters, via trial and error. For the reasons stated, it is necessary to select a network capable of obtaining binary segmentation with high performance and robustness while maintaining a small memory occupation in order to be compatible with the dedicated robot hardware (NVIDIA Jetson AGX Xavier) and save training time (for the hyperparameters tuning it is required to carry out the training multiple times). The working method is to attempt several models with different backbones,

setting the optimal hyperparameters for each of them, and then comparing them to find the best one for this particular application. The first implemented model is a hardcoded U-Net. According to the inspected papers, in particular Gupta et al. [2020], even if this specific model provides metrics results that are significantly lower than newer models such as SegNet (65% vs 90% on pixel accuracy), it is implemented as a first test for the segmentation pipeline. The following part of code shows how the model is implemented according to the TensorFlow API.

```
ef get_unet(input_img, n_filters=16, dropout=0.1, batchnorm=True):
    """Function to define the UNET Model"""
    # Contracting Path
    c1 = conv2d_block(input_img, n_filters*1, kernel_size=3,
                      batchnorm=batchnorm)
    p1 = MaxPooling2D((2, 2))(c1)
    p1 = Dropout(dropout)(p1)

    c2 = conv2d_block(p1, n_filters*2, kernel_size=3,
                      batchnorm=batchnorm)
    p2 = MaxPooling2D((2, 2))(c2)
    p2 = Dropout(dropout)(p2)

    c3 = conv2d_block(p2, n_filters*4, kernel_size=3,
                  batchnorm=batchnorm)
    p3 = MaxPooling2D((2, 2))(c3)
    p3 = Dropout(dropout)(p3)

    c4 = conv2d_block(p3, n_filters*8, kernel_size=3,
                  batchnorm=batchnorm)
    p4 = MaxPooling2D((2, 2))(c4)
    p4 = Dropout(dropout)(p4)

    c5 = conv2d_block(p4, n_filters=n_filters*16, kernel_size=3,
                  batchnorm=batchnorm)

    # Expansive Path
    u6 = Conv2DTranspose(n_filters*8, (3, 3), strides=(2, 2),
                  padding='same')(c5)
    u6 = concatenate([u6, c4])
    u6 = Dropout(dropout)(u6)
    c6 = conv2d_block(u6, n_filters*8, kernel_size=3,
                  batchnorm=batchnorm)

    u7 = Conv2DTranspose(n_filters*4, (3, 3), strides=(2, 2),
                  padding='same')(c6)
    u7 = concatenate([u7, c3])
    u7 = Dropout(dropout)(u7)
    c7 = conv2d_block(u7, n_filters*4, kernel_size=3,
                  batchnorm=batchnorm)

    u8 = Conv2DTranspose(n_filters*2, (3, 3), strides=(2, 2),
```

```
                padding='same')(c7)
    u8 = concatenate([u8, c2])
    u8 = Dropout(dropout)(u8)
    c8 = conv2d_block(u8, n_filters*2, kernel_size=3,
                batchnorm=batchnorm)

    u9 = Conv2DTranspose(n_filters*1, (3, 3), strides=(2, 2),
                padding='same')(c8)
    u9 = concatenate([u9, c1])
    u9 = Dropout(dropout)(u9)
    c9 = conv2d_block(u9, n_filters*1, kernel_size=3,
                batchnorm=batchnorm)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)
    model = Model(inputs=[input_img], outputs=[outputs],
                name='unet')
    return model
```

As it is possible to notice, there are 9 macro-layers, 5 for the encoder and 4 for the decoder and then a final layer is inserted for the binary classification with the sigmoid activation function. For the first four levels, the pattern is the same: a conv2d_block with batch normalization activated and the number of filters that increase from the basic value of the first layer to the 16-times greater of the last one, a 2x2 layer for max pooling and a dropout one to reduce chances of overfitting. conv2d_block is shown in the following part of code.

```
def conv2d_block(input_tensor, n_filters, kernel_size=3,
            batchnorm=True):
"""Function to add 2 convolutional layers """
"""with the parameters passed to it"""
# first layer
x = Conv2D(filters=n_filters, kernel_size=(kernel_size, kernel_size),\
        kernel_initializer='he_normal', padding='same') (input_tensor)
if batchnorm:
    x = BatchNormalization()(x)
x = Activation('relu')(x)

# second layer
x = Conv2D(filters=n_filters, kernel_size=(kernel_size, kernel_size),\
        kernel_initializer='he_normal', padding='same')(input_tensor)
if batchnorm:
    x = BatchNormalization()(x)
x = Activation('relu')(x)
return x
```

As one can see, there is a pair of 2D convolutional layers with a ReLu function and a settable kernel size. This way of implementation is basically a coding choice to easily set some hyperparameters like the actual kernel size. For the decoding part of the model, that involves the last 4 macro-layers, the repeated pattern is the following: a transposed convolution layer concatenated with the correspondent layer of the contracting part, a

dropout layer and a *conv2d-block* with number of filters starting form 16-times the original values and decreasing to it.

The model is trained directly on the indoor segmentation dataset in the Google Colab virtual environment, with no real concern for the quality of the segmentation (thus the lack of hyperparameter adjustment), but only to test the training code pipeline. The next paragraph contains additional information on model training as well as a description of the pipeline. The "Segmentation Models" package by qubvel is used to implement the different models. It is a python library with neural networks for image segmentation based on Keras (TensorFlow) framework. The following are the key characteristics of this library:

- API at a high level (just two lines to create NN)
- For binary and multiclass segmentation, there are four different model architectures to choose from (Unet, FPN, LinkNet, PSPNet)
- For each architecture, there are 25 backbones accessible
- Weights have been pre-trained in all backbones for faster and better convergence

Because the library is based on the Keras framework, the segmentation model that is constructed is simply a Keras model that can be easily created. You can vary the network design depending on the task by selecting backbones with fewer or more parameters and using pre-trained weights to initialize it. For each of the four different models, according to the API, it's possible to define the following parameters:

- backbone name: name of classification model used as feature extractor to build segmentation model (without last dense layers).
- input shape: shape of input data/image (H, W, C). According to the specific model, H and W of input images must be divisible by a certain number specified in the documentation.
- classes: number of output classes.
- activation function of the final model layer (e.g. sigmoid, softmax, linear).
- weights: path to model weights, which is optional.
- encoder_weights: one of None (random initialization) or ImageNet (pre-training on ImageNet).
- encoder_freeze: if True set all layers of encoder (backbone model) as non-trainable.
- encoder_features: a list of layer numbers or names starting from top of the model. Each of these layers will be concatenated with a corresponding decoder block.
- decoder_block_type: list of numbers of Conv2D layer filters in decoder blocks
- encoder_use_batchnorm: if True, BatchNormalisation layer between Conv2D and Activation layers is used.

The "Segmentation Models" library fits perfectly the requirements of the project, because allows to easily manage models that has wide use in segmentation fields and provides backbones already pretrained on the ImageNet dataset. This last feature allows to avoid to do the pre-training step manually, making the global learning process quicker. The drawback is that there are only four models, so some well performant networks (according to the examined literature), such as SegNet will be not tested, as well as other segmentation models described in chapter 2.

4.5 Training

The code flow for model training is examined in this paragraph. What is displayed is a sequence of code cells from a Google Colab notebook that has been used to conduct the

training. In order to speed up the global training process for all the models, some of them have been implemented in this environment, another part using Kaggle and the rest have been trained with local hardware. All the environments (both virtual and local) are GPU accelerated.

```
!pip uninstall keras -y
!pip uninstall keras-nightly -y
!pip uninstall keras-Preprocessing -y
!pip uninstall keras-vis -y
!pip uninstall tensorflow -y
!pip uninstall tensorflow-gpu -y
!pip install tensorflow==2.3.0
!pip install keras==2.4

! pip install -U segmentation_models
import segmentation_models
```

For a virtual environment, it is required to set the correct version of TensorFlow and Keras to allow the *Segmentation Models* library to work correctly.

```
# GET AND LOAD DATASET
get_cmu_corridor_dataset(dataset_path='./dataset')

X_train, X_test, y_train, y_test =
    load_cmu_corridor_dataset(
        dataset_path=/dataset/cmu_corridor_dataset',
        train_test_split=True,
        image_size=(240, 336),
        gray=False,
        scaling = True,
        verbose=True)
```

Here some basic python code is used to build a function to download and load the dataset into train and test images and labels. Note that it's possible to define the size of the frames. In the reported case the unusual 240x336 is chosen because of the constraint of the Pyramid Scene Parsing Network to ask for input to be divisible by 6 times the downsample factor, and 4 for this specific case that produces a total factor of 24. For the other networks (Unet, FPN, LinkNet) this constraint is different with the input that must be divisible by 32. In the implemented function, dataset split is included (80-20 in this specific case).

```
# MODEL
from segmentation_models import PSPNet
from segmentation_models import get_preprocessing
from segmentation_models.losses import bce_jaccard_loss
from segmentation_models.metrics import iou_score

BACKBONE = 'resnet34'
preprocess_input = get_preprocessing(BACKBONE)
x_train = X_train
```

```
x_val = X_test
y_val = y_test
# preprocess input
x_train = preprocess_input(x_train)
x_val = preprocess_input(x_val)

# define model
model = PSPNet(BACKBONE,
            encoder_weights='imagenet',
            input_shape=(240,336,3),
            classes=1,
            activation='sigmoid',
            psp_dropout=0.05,
            psp_conv_filters=16,
            psp_pooling_type='max',
            psp_use_batchnorm=True,
            encoder_freeze=True
            )
model.compile('Adam', loss='binary_crossentropy',
metrics=[iou_score])
```

In these lines the definition of the model is carried out: as one can notice it is directly imported by the qubvel library in the first line (PSPNet for the specific case) with the losses and the metrics. A predefined function to preprocess data is applied to images and labels of the training set and then the model is actually built. In the reported lines, it can be observed that the chosen backbone is a ResNet34 already trained on the ImageNet dataset (transfer learning phase); the input shape is set in accordance to the constraint of the net; number of classes is set to 1 because the API already considers the non-classes elements, by building the zero class, that in the case of this work is related with the non-floor elements.

Specific hyperparameters for the PSP model are then specified, such as the dropout, the type of pooling and the batch normalization. Last parameter tells that in the training process the weights of the encoder must not change, so the fine-tuning affects only the pyramid module and the output layers. Other models have a similar structure, as reported in the following APIs.

UNet

segmentation_models.Unet *(backbone_name='vgg16', input_shape=(None, None, 3), classes=1, activation='sigmoid', weights=None, encoder_weights='imagenet', encoder_freeze=False, encoder_features='default', decoder_block_type='upsampling', decoder_filters=(256, 128, 64, 32, 16), decoder_use_batchnorm=True, **kwargs)*

LinkNet

segmentation_models.Linknet*(backbone_name='vgg16', input_shape=(None, None, 3), classes=1, activation='sigmoid', weights=None, encoder_weights='imagenet', encoder_freeze=False, encoder_features='default', decoder_block_type='upsampling', decoder_filters=(None, None, None, None, 16), decoder_use_batchnorm=True, **kwargs)*

FPN

segmentation_models.FPN*(backbone_name='vgg16', input_shape=(None, None, 3), classes=21, activation='softmax', weights=None, encoder_weights='imagenet', encoder_freeze=False, encoder_features='default', pyramid_block_filters=256, pyramid_use_batchnorm=True, pyramid_aggregation='concat', pyramid_dropout=None, **kwargs)*

PSPNet

segmentation_models.PSPNet*(backbone_name='vgg16', input_shape=(384, 384, 3), classes=21, activation='softmax', weights=None, encoder_weights='imagenet', encoder_freeze=False, downsample_factor=8, psp_conv_filters=512, psp_pooling_type='avg', psp_use_batchnorm=True, psp_dropout=None, **kwargs)*

Few callbacks functions are implemented to be recalled during the training process for the fine-tuning. The first one is *EarlyStopping* which allows terminating the training before processing all the epochs, according to some criteria. In this specific case if there is no improvement of the validation loss for at least 20 consecutive epochs, the training is stopped early. The *ReduceLRonPlateau* callback is used to mitigate the danger of overfitting: if the validation loss does not decrease by at least a factor after a specified number of epochs (5), the learning rate is reduced by a factor (10). The possibility of being in a local minimum, which forces the loss to be too high, is lowered by this technique. Finally, there is a simply function to save the model with the best weights (*ModelCheckpoint*).

```
# CALLBACKS DEFINITION

    from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint,
    ReduceLROnPlateau, TensorBoard

    callbacks = [
        EarlyStopping( min_delta=0.01, patience=20, verbose=1),
        ReduceLROnPlateau(factor=0.1, patience=5, min_lr=0.00001, verbose=1),
        ModelCheckpoint('/content/models/current_model', verbose=1,
        save_best_only=True, save_weights_only=False),
        TensorBoard(log_dir='.logs')
    ]

    #fit model
    results = model.fit(
        x=x_train,
        y=y_train,
        batch_size=16,
        epochs=100,
        callbacks = [callbacks],
        validation_data=(x_val, y_val),
    )
```

The actual start of the training process is made by calling the fit method, where the last two hyperparameters, epochs and batch size, are defined. Then the learning curve is printed, as shown in fig. 4.5

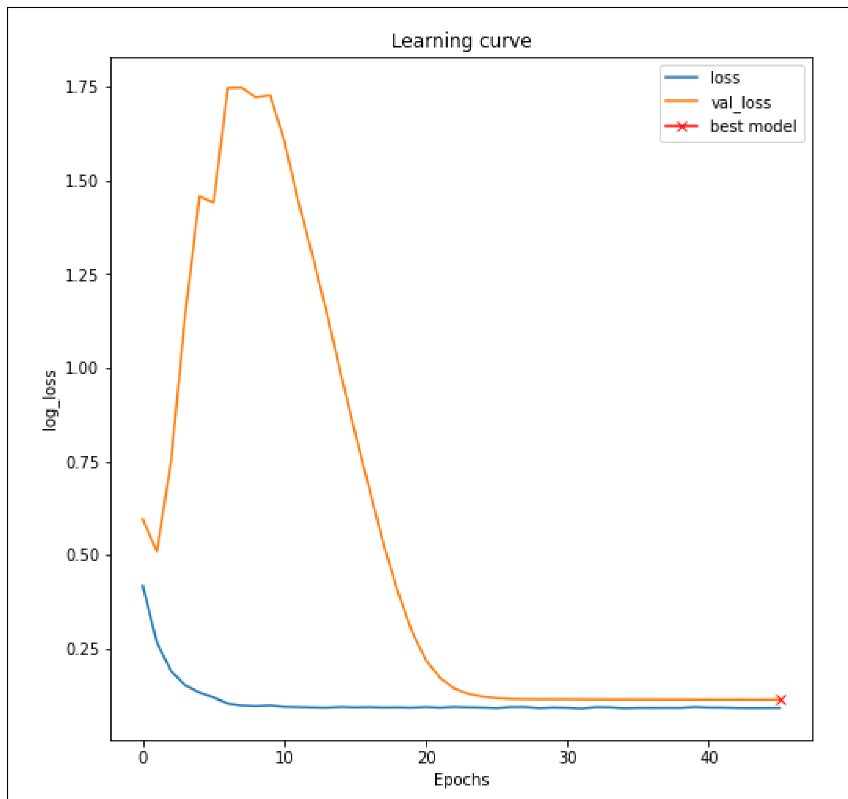

Figure 4.5. Learning curve example.

At the end of this training pipeline, some predictions on training, validation and test images are made, just for a preview of the qualitative performance of the network.

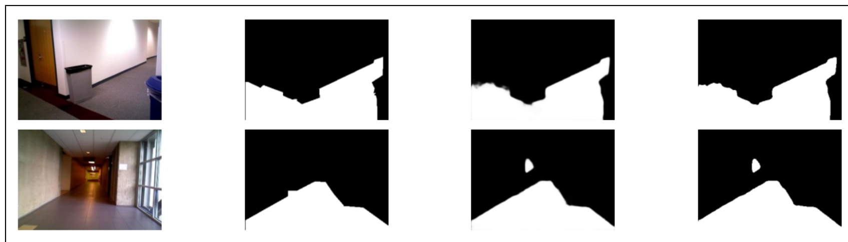

Figure 4.6. Image prediction examples.

4.6 Results

The results of the training of the models are reported in this section.

4.6.1 UNet

For the Unet the used metric is the mean intersection over unit score (mIoU) and the chosen loss is a binary cross entropy. According to the API shown in paragraph 4.5, all

the parameters that are not reported on the table are set on the standard value. The input shape is 224x320x3 for all the models. Adam is used as a solver.

Backbone	Decoder Filters	Encoder Weights	Encoder Freeze	Batch Normalization	Batch Size	Epochs	Callbacks	IoU Score
Vgg16	256, 128, 64, 32, 16	ImageNet	True	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	74.11%
Vgg16	512, 256, 128, 64, 32	ImageNet	False	True	32	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	62.96%
ResNet50	256, 128, 64, 32, 16	ImageNet	True	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	90.01%
ResNet50	256, 128, 64, 32, 16	ImageNet	False	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	87.46%
ResNet50	256, 128, 64, 32, 16	None	False	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	67.36%
ResNet34	256, 128, 64, 32, 16	ImageNet	True	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	90.73%
ResNet34	256, 128, 64, 32, 16	ImageNet	False	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	88.84%
ResNet34	256, 128, 64, 32, 16	None	False	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	68.84%
ResNet18	256, 128, 64, 32, 16	ImageNet	True	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	85.09%
ResNet18	256, 128, 64, 32, 16	ImageNet	False	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	79.12%
ResNet18	256, 128, 64, 32, 16	None	False	False	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	71.34%

Table 4.2. U-Net training results.

As one can notice the best result is achieved by a ResNet34 backbone with the encoder weight trained on the ImageNet and fixed during the fine-tuning. The stated findings are quite similar to those obtained on Teso-Fz-Betoño et al. [2020a], with the Vgg16 backbone performing much worse than the ResNet: in the article, Vgg achieved 68-76% accuracy, whilst ResNet guaranteed more than 90 percent accuracy. The results are much worse in cases 5,8,11, where the encoder weights are set to None (the transfer learning phase is skipped), than in cases with the same condition where ImageNet is used for pre-training (mIoU score more than 20% lower). Best hyperparameters configuration is highlighted in yellow.

4.6.2 LinkNet

For the LinkNet the input shape is fixed on 224x320x3 and Adam is used as solver. As well as the U-Net case, if a parameter in the table is not specified, it has to be considered as set on the default value according to the API.

Backbone	Decoder Filters	Encoder Weights	Encoder Freeze	Batch Normalization	Batch Size	Epochs	Callbacks	IoU Score
Vgg16	256, 128, 64, 32, 16	ImageNet	True	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	71.34%
ResNet50	256, 128, 64, 32, 16	ImageNet	True	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	82.26%
ResNet34	256, 128, 64, 32, 16	ImageNet	True	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 5)	83.13%
ResNet34	256, 128, 64, 32, 16	ImageNet	False	True	16	50	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	82.16%

Table 4.3. LinkNet training results.

In contrast to what was expected based on what was learnt in Chaurasia and Culurciello [2017], LinkNet performance is worse than what was observed in Unet, while remaining at a good level. This may be due to the fact that the U-Net structure fits a dataset like this better than the LinkNet one. Moreover, it is plausible to believe that the LinkNet hyperparameters configuration is not one of the best, but given the good results already acquired with U-Net, the choice is to not spend too much time tuning that. Finally, it's important to remark that even in this case the ResNet behaves significantly better than the Vgg, with more than 10% reached on the Jaccard score.

4.6.3 FPN

As well as in the U-Net and LinkNet cases the input shape is set to 224x320 and Adam is still the used solver, but differently from the previous models, here it's necessary to specify the number of classes (set to 1) and the activation function to sigmoid. Every

other parameter that is not specified in the table is fixed on the default value. Here the results are sufficient, but not good enough to be comparable with the best got on U-Net and here again it's possible to verify how the ResNet is performing better.

Backbone	Encoder Weights	Encoder Freeze	Batch Normalization	Batch Size	Epochs	Callbacks	IoU Score
Vgg16	ImageNet	False	True	16	60	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	66.68%
ResNet50	ImageNet	False	True	16	80	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	79.27%
ResNet34	ImageNet	False	True	16	80	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	77.14%
ResNet34	ImageNet	True	True	16	80	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	78.60%

Table 4.4. FPN training results.

4.6.4 PSPNet

The last experimented architecture is represented by the Pyramid scene Parsing Network. Although there are no specific cases of scene segmentation-based indoor navigation papers in which this model is used (as shown in paragraph 3.3), according to the theory Zhao et al. [2017], it can be considered a competitive option to achieve the desired performance. The solver of the training is once again Adam, the number of classes is set to 1 and sigmoid is put as the function of the last layer. A little difference from the other three architectures can be noticed in the input size: if in in U-Net, LinkNet and FPN, it's necessary to be divisible by 32 (so 224x320 was the solution), for the PSPNet, it must be equal to 6 times the downsample factor of the pyramid module. This last value is set to 4 for all the trained models (other values, according to quick experimentations led to worse results), so training images are resized to be divisible by 24, finally getting 240x336. Non-specified parameters are on the default value. Pooling type is set to 'max' and the dropout coefficient is set to 0.05.

The line highlighted in yellow represents the best hyperparameters configuration for the PSPNet but the best overall performance of all the trained models; the only one comparable is represented by the 90.73% of the U-Net. From a quantitative point of view the scores are very close, so they are both suitable to be the core of the segmentation module, but as images 4.7 and 4.8 report, there are little differences that makes the PSP module better form a qualitative point of view.

The PSP seems to be more suitable to segment images that constitute the base of autonomous navigation, mainly because of the lack of little lateral stains that sometimes the U-Net model produces.

Moreover, PSP behaves well even if the scenario is more complicated, i.e. cases of walls and floor of the same color, particular light conditions and reflections, resulting in more robustness. In other words, even if the achieved score is more or less the same, PSP is characterized by errors (FP and FN) at the boundaries of the segmented floor area, maintaining just two figures: the navigable area and the non-navigable area. By contrast, U-Net predictions are more often affected by holes or stains that are not connected to the

Backbone	Encoder Weights	Encoder Freeze	Batch Normalization	Batch Size	Epochs	Callbacks	IoU Score
Vgg16	ImageNet	False	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	69.12%
Vgg19	ImageNet	False	True	32	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	56.13%
ResNet18	ImageNet	False	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	83.81%
ResNet18	ImageNet	True	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	85.08%
ResNet18	None	False	True	16	60	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	51.12%
ResNet34	ImageNet	False	True	16	60	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	87.53%
ResNet34	ImageNet	True	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	91.26%
ResNet34	ImageNet	False	True	32	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	81.45%
ReNet34	None	False	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	51.58%
ResNet50	ImageNet	True	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	88.21%
ResNet50	ImageNet	False	True	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	85.09%
ResNet50	None	True	False	16	100	Early Stopping, Reduce LR on Plateau (Factor=0.1, patience 10)	79.65%

Table 4.5. PSPNet training results..

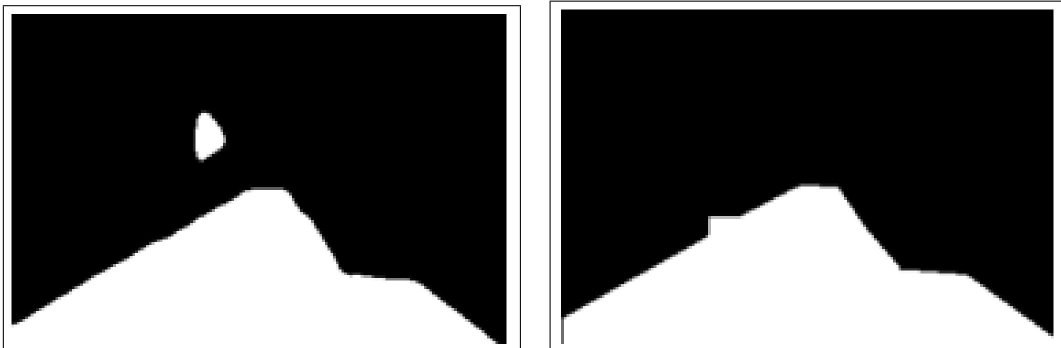

Figure 4.7. U-Net and PSPNet inference comparison

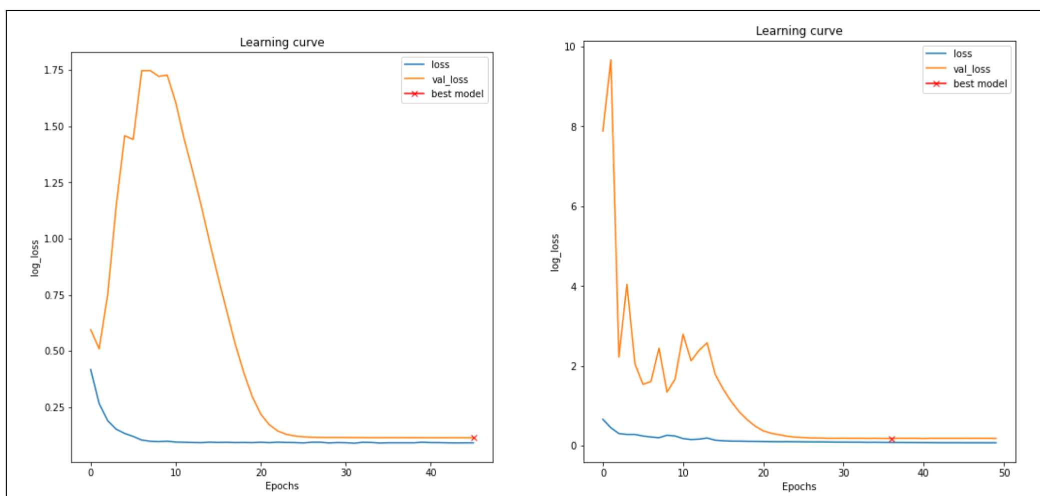

Figure 4.8. U-Net and PSPNet Learning Curve comparison

floor area, representing a bigger problem to handle with in a potential construction of a navigation algorithm.

Chapter 5

NAVIGATION

In chapter 4 about scene segmentation, a suitable and robust model is selected. The best candidate resulted in a Pyramid Scene Parsing Network with a ResNet34 as backbone, achieving more than 91% on the Jaccard metric. On the basis of this segmentation results the last challenge rises: building a navigation algorithm that allows an indoor service robot to navigate avoiding collisions, only on the base of the RGB segmented images taken by a camera.

In this chapter the presentation of similar works is done, then the actual code used to implement the image post processing and the traits of the final navigation algorithm are presented. The most inspiring work in this sense is represented by Daniel Teso-Fz-Betoño, Ekaitz Zulueta, Ander Sánchez-Chica, Unai Fernandez-Gamiz and Aitor Saenz-Aguirre with "Semantic Segmentation to Develop an Indoor Navigation System for an Autonomous Mobile Robot". In this paper, after the building of a deep learning module similar to the one implemented here, there is a 2 classes RGB mask of the scene for each image taken real-time by the camera. The idea is using the pixels related to the floor class to estimate the middle point for every i-th row of the image, applying a simple formula:

$$Middle_i = \frac{End_{location} - Star_{location}}{2} + Star_{location}$$

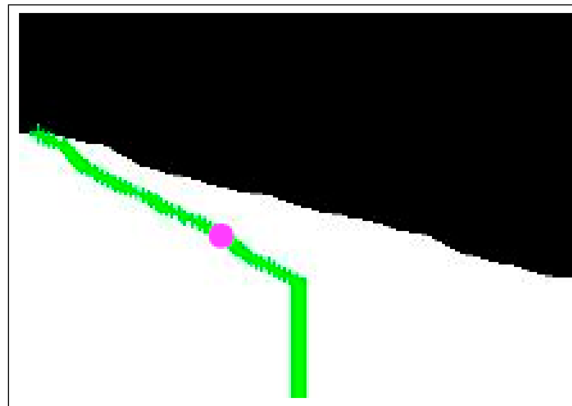

Figure 5.1. Middle path estimation.

This method produces good results when segmentation is performed correctly and when the pixels of the navigable class are all part of a unique block Teso-Fz-Betoño et al.

[2020b]. The drawback is the lack of robustness, which results clearly in cases where an obstacle is placed in the middle of the path: it's not possible to handle these scenarios with the above formula and a situation like the one in image 5.2 is obtained.

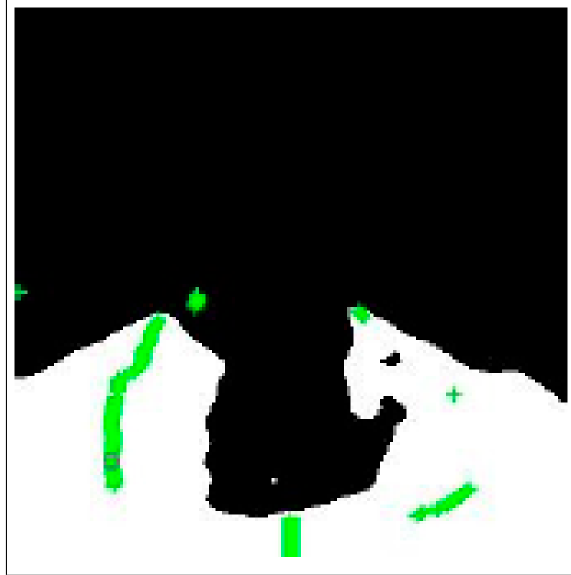

Figure 5.2. Example of bad detection when an obstacle is placed in the middle of the path.

In this thesis, the idea is to start from what is presented in this paper, but with the addition of other pieces in the image processing, in order to obtain a unique waypoint for each segmented image that can potentially represent the partial goal of the navigation. In a nutshell, what is described in detail in the next paragraphs, is a series of post-process steps on the segmented image obtained in chapter 4: middle path detection, segments identification, waypoint construction. Then on the base of the obtained waypoint, the navigation command (with the use of ROS), in terms of linear velocity and steering angle are calculated.

5.1 Post-Processing

In this paragraph all the functions defining the passages of the post processing are shown and analyzed. Let's consider image 5.3 as the output of the segmentation module as well as the input of the post-processing pipeline. As one can notice, there is a very smooth inference, with no stairs and a simple geometry.

Figure 5.3. Segmented Images.

```
def lasty(seg_array,flow=10):

  seg_array = np.around(seg_array, decimals=0)
  y_lim = seg_array.shape[0]
  x_lim = seg_array.shape[1]
  last = y_lim-1
  for y in range(y_lim-1,-1,-1):
    cc = 0
    for x in range(x_lim):
      if seg_array[y][x] == 1 and x < x_lim-1:
        cc = cc + 1
      if cc >= flow:
        last = y
        break

  return last
```

First function is only used to detect which is the last row where a floor (white) pixel is present, starting from the bottom of the image. The only parameter here is the so-called “flow”, that is a way to handle bad segmentations. Let’s consider an image just like the one used as example, but with a little stair on the top right corner, due to a bad inference. In this case it’s clear that, considering the last floor row, the one on the top right corner should be an error. With the flow parameter, the lateral dimension of the stair is checked and if it is less than the calibrated threshold, it is simply ignored. This a simple but effective method, based on the experimental observations, which reports the (not usual) segmentation errors to be little white stairs on random position of the images.

5.1.1 Middle-Point

```
def meanrow_definition(seg_array,flow=10,lasty=-1,span=2):
  if flow < 2*span:
    print("WARNING - meanrow_definition: flow must
        be at least 2*span, problems could be present")


  seg_array = np.around(seg_array, decimals=0)
  y_lim = seg_array.shape[0]
  x_lim = seg_array.shape[1]

  for y in range(y_lim-1,lasty,-1):
    cc = 0
    start = 0
    end = 0
    black = True

    for x in range(x_lim):
      if seg_array[y][x] == 1 and x < x_lim-1:
        cc = cc + 1
        if cc == 1:
          start = x
```

```
          black = False
        elif seg_array[y][x] == 0 and cc >= flow:
          end = x
          break

        elif seg_array[y][x] == 1 and cc >= flow and x == x_lim-1:
          end = x
          break
      if black == False and start < end
          and seg_array[y][round((end+start)/2)-span :
          round((end+start)/2)+span].all() == 1 :
          seg_array[y][round((end+start)/2)-span :
          round((end+start)/2)+span] = 0.5

  return seg_array
```

Meanrow_definition function is used to detect, for each row with floor pixels, the pixel in the middle position. The inputs are the last row to consider, obtained with the first function of the post-processing, *flow* and *span*. The last has only the visual utility to paint a number of pixels on the raw equal to that value, while *flow*, similarly for what was done with the last function, checks if a certain row of floor pixels is a good candidate to apply the middle pixel detection function. This is particularly useful in case of stairs or when a certain segmented frame has an obstacle in the middle, resulting in two or more branches of white pixels. If a branch is too narrow (so the number of white pixels is less than the tuned threshold), that branch is ignored and a larger one is considered. The result is shown in figure 5.4.

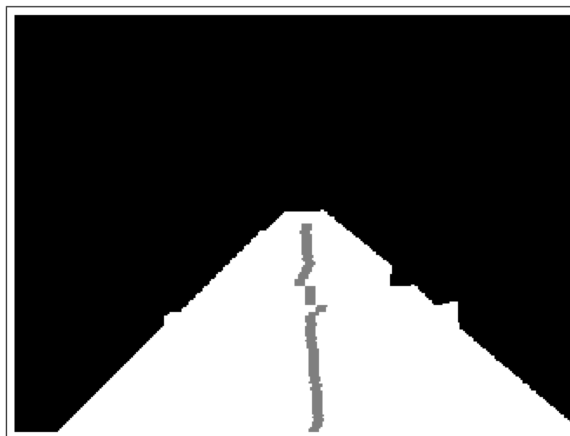

Figure 5.4. Image1 with meanrow function applied.

At this point, what is carried out in Teso-Fz-Betoño et al. [2020b] is achieved, but as previously discussed this can't be a robust method to build a navigation algorithm on top: it could be possible to let the robot moving, trying to follow the trajectory defined by the line, but it would be very problematic in cases like the ones shown in images 5.5 and 5.6.

Figure 5.5. Image2 with meanrow function applied.

Figure 5.6. Image3 with meanrow function applied.

5.1.2 Waypoints

The *meanrow_definition* function is used as the starting point for the selection of the waypoints, according to the *keypoints* function.

```
def keypoints(ppimage, span=2, lasty=-1, verticalflow=5,
            normalized=True, isarray=True):
  if isarray==False:
     ppimage = img_to_array(ppimage)/255.0
  if normalized == False: ppimage = ppimage/255.0
  y_lim = ppimage.shape[0]
  x_lim = ppimage.shape[1]
  upper = y_lim-1
  down = y_lim-1
  pointlist = []
  for y in range(y_lim-1,lasty-1,-1):
    for x in range(x_lim):
      if ppimage[y][x] == 0.5:
        #print(y)
```

```
      #print(x)
      if (ppimage[y][x:x+2*span-1] == ppimage[y-1][x:x+2*span-1]).any():
        #print("corr")
        upper = y-1
      else:
        #print("3")
        if down-upper >= verticalflow:
          pointlist.append(round(down/2+upper/2))
          for x in range(x_lim):
            if ppimage[round(down/2+upper/2)][x] == 0.5:
              pointlist.append(x+span-1)
              down = upper
              break
      break
  return pointlist
```

The aim of this step is the production of a list of points on the segmented image representing the intermediate goals that the robot should follow to carry out a collision free motion. The output of the *meanrow_definition* is put as input of the *keypoints* function with the objective of recognizing how many different segments are present in the image. For example, in image 5.4 there is a continuous single segment, while in image 5.5 four pieces can be recognized. This last value can change in the same image, according to the span parameter of the *meanrow_definition*: a thicker line has less possibilities to produce different pieces (and different keypoints) with respect to a slim one.

With a control on pixels, these line bits are converted into key-points, whose x and y coordinates produce a list used as output. Function parameters are the images with the processed line; the last value discussed previously; the span that defines how big is the range in which the continuity of line pixels between two rows of the image is checked. It is simply expressed in pixels and it's recommended to be consistent with the *meanrow_definition* span parameter. *Verticalflow* defines how many consecutive pixels must be found at minimum, in order to consider valid that piece of line to be converted into a waypoint. Finally there is a function with the only objective to make visible the obtained points.

```
def cross(seg_array, pointlist, thickness=2):

  y_cords = []
  x_cords = []
  for i in range(len(pointlist)):
    if i%2==0: y_cords.append(pointlist[i])
    else: x_cords.append(pointlist[i])
  for i in range(len(y_cords)):
    x = x_cords[i]
    y = y_cords[i]
    seg_array[y][x-thickness-1:x+thickness] = 0.5
    for i in range(thickness+1):
      seg_array[y-round(thickness/2)+i][x] = 0.5

  return seg_array
```

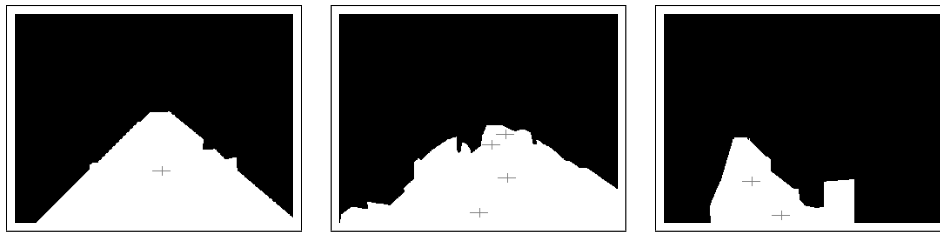

Figure 5.7. Waypoints calculation.

5.2 ROS Integration

This point of the development process consists in building the ROS app that runs on the Jetson hardware and make the robot able to carry out its function. The used framework is ROS2 with two different nodes: the first totally dedicated to image reading, segmentation and processing and the second where the motion command is calculated on the base of the post-processed data output of the first node.

5.2.1 Scene Segmentation Node

In this node, by using OpenCV library, the image is read using the camera sensor mounted on the robot, then with the PSP network previously trained, inference is carried out. The last step consists in the cascade of post-processing functions described in paragraph 5.1 with the final goal to obtain a list of waypoints. This is the information published by the node. In the code presented below, only the main class is reported. There are just some differences from what is already shown, in particular in the first lines that are dedicated to getting the images by camera using the OpenCV API. A 10ms timer period is set, defining the rate of acquisition. It's important to notice that based on the performance of the camera, this rate can be reduced by the technological limits of the sensor. In the case of the project a simple webcam is used, guaranteeing 30 images captured per second.

```
class SceneSegmentation(Node):

  def __init__(self):
    super().__init__("Scene_Segmentation")
    timer_period = 0.01
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.cap = cv2.VideoCapture(0)
    self.br = CvBridge()
    self.waypoint_publisher = self.create_publisher(
      Int16,
      '/waypoint',
      10)

  def timer_callback(self):
    ret, frame = self.cap.read()

    cv2.imshow("Camera Capturing", frame)
    current_image = cv2.resize(frame, (img_width, img_height))
```

```
current_image = np.array(current_image)
current_image = current_image/255.0
current_image = np.expand_dims(current_image, axis=0)
pred_misc = model.predict(current_image, verbose=1)
pred_misc = np.around(pred_misc[0])

cv2.imshow("Environment Segmentation", pred_misc*255.0)

last = lasty(pred_misc, flow=40)
seg_p = meanrow_definition(pred_misc, flow=40,
        lasty=last, span=6)
pointlist = keypoints(ppimage=seg_p, span=6, lasty=last,
            isarray=True)
print(pointlist)

waypoint_msg = Int16()
if len(pointlist)>0:
  waypoint_msg.data = pointlist[len(pointlist)-1]
else: waypoint_msg.data = 0

self.waypoint_publisher.publish(waypoint_msg)
cv2.waitKey(1)
```

5.2.2 Navigation Node

Here the navigation node is analyzed. A list of waypoints x and y pixels is received by the current node, operating as a subscriber of the segmentation one.

The starting idea consists in building a series of instructions that force the robot to sequentially reach the waypoints reported in the list to carry out its navigation. Some drawbacks feature the expressed strategy, in particular it's important to remind that the only present information is a list of pixels on a segmentation mask, without any kind of distance knowledge, so it would be necessary to define multiple reference systems (global and attached to the robot), in order to check the correct development of the motion. Moreover, until the robot reaches his final goal, it is virtually blind, ignoring eventual changes of the environment.

To handle this set of disadvantages, motion control strategy is changed to be logically simple, reliable and easier to implement: from the set of waypoints only one of them is extracted to be the actual goal. It expresses the hypothesis that the position of the robot where the frame is acquired and processed is the central bottom part of the image (for a 240x336 mask, the starting robot position is thought to be in y = 240, x=336/2 position). Starting from this information, the x coordinate of the pixel goal is used to calculate the steering angle of the robot, through a proportional control.

- Xr: actual x coordinate of the robot
- Xg: actual x coordinate of the goal pixel
- Y = Kp*(Xg-Xr)

Linear velocity is set to be constant. The proportional coefficient is calibrated in real-world tests to allow the robot to steer correctly and make the yaw rate consistent with the linear motion rate, set to 0.1 m/s. The best waypoint on the list is considered to be always the last one found, because it is the one that contains all the most complete

information about the scene.

It's salient to highlight that with this approach, there is no time of blindness for the sensor: camera acquires the scene, the image is segmented, processed and then published for the navigation node that calculates the motion command. As soon as the pipeline allows, the next segmentation of the scene and the next navigation command are obtained, modifying the motion of the robot in real-time, without waiting for it to actually be on the goal point.

A last piece of code is added for the navigation algorithm, specifying that in cases of void waypoints lists the linear velocity is decreased to be 0. In other words, this means that if the robot is too close to an obstacle, it stops the foreword motion and steers only.

```
lin_vel = 0.1
Kp = 0.005
lin_null = 0.0
class TurtlebotController(Node):
    def __init__(self):
        super().__init__("vehicle_controller")

        self.turtlebot_controller_publisher =
            self.create_publisher(Twist, "/cmd_vel", 10)
        self.waypoint_subscriber =
          self.create_subscription(Int16,"/waypoint,
          self.turtlebot_controller, 10)

    def turtlebot_controller(self, msg):
        waypoint_pixel = msg.data
        if waypoint_pixel>0:
            center_pixel = 168
            yaw_rate = -Kp*(waypoint_pixel-center_pixel)
            msg = Twist()
            msg.linear.x = lin_vel
            msg.angular.z =  yaw_rate
        else:
            msg = Twist()
            msg.linear.x = lin_null
            yaw_rate = 0.5
            msg.angular.z = yaw_rate

        print(yaw_rate)
        print(msg.linear.x)
        self.turtlebot_controller_publisher.publish(msg)
```

5.3 Implementation

The final stage of the development process is related to the definition and implementation of the ROS architecture, examined in the previous chapter. What is still missing at this stage of the work is the real-world implementation, parameters calibration and some practical tries to observe how good the entire pipeline works. In paragraph 3.1 the available hardware has been discussed in detail, so in this paragraph there is a description of how the robot is assembled to carry out its collision free motion.

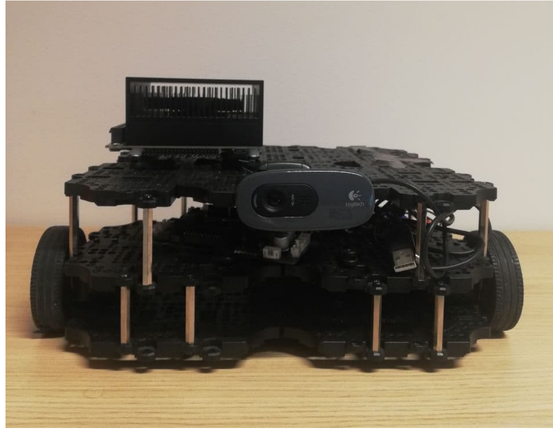

Figure 5.8. Robot front view.

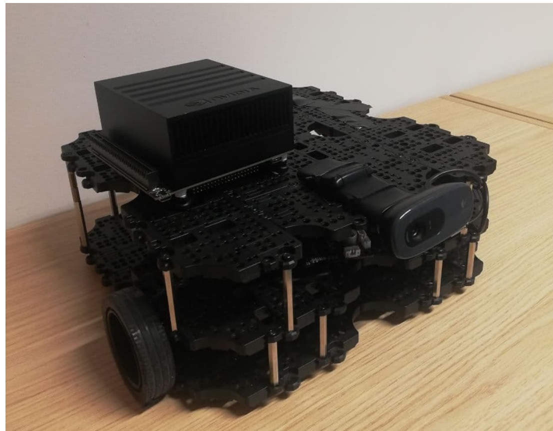

Figure 5.9. Robot lateral view.

Turtlebot3 Waffle with Raspberry Pi represents the moving part of the structure and on the top level of its chassis the Jetson module is fixed. Here is where the ROS2 application (developed on the workstation) runs, performing real-time segmentation and generation of navigation commands for the robot. By means of the *teleop* pre-defined node of ROS, the values of linear velocity and yaw rate are transmitted to the actuators (Turtlebot3 wheels). Finally, the perception sensor, represented by an RGB webcam, is fixed on the top front central part of the structure.

The position of the camera in terms of height and angle with respect to the ground is crucial. To get the best results from segmentation it's needed to make the acquired real-time images to have the point of view as similar as possible to the one of the training data.

The structure is completed with the battery for the robot and the Xavier. Moreover, a batch of cables is present to connect the Jetson module to the camera and to the electronic module of the robot.

At this point, as previously described, some parameters in the segmentation flow, like the *span* and the *verticalflow*, are tuned to obtain the best results from segmentation of a real-time real-world scenario. Furthermore, the proportional coefficient *Kp* of the navigation node is tuned to make the robot able to steer with a rate consistent with the

Figure 5.10. Robot during navigation.

set linear velocity. The best value for the coefficient with a 0.1m/s linear velocity is 0.005.

Linear velocity [m/s]	0.1
Proportional coefficient	0.005

5.4 Final Test

The complete pipeline is able to produce a navigation command for the robot (i.e. linear velocity and yaw rate) every 200ms, with approximately 50 spent to produce the segmented image and the rest for the post-processing operations and navigation commands definition. With a 5Hz rate the expectations are to have a robot that is comfortable to navigate in static scenarios, but also that behaves properly with scenes changing with a period lower than 200ms. Both static and dynamic environment will be used in the testing phase.

A series of tests is carried out in order to verify the correct behavior of the robot in an indoor environment. Test sessions take place in two main environments: a corridor and an office. The difficulty of the experiments is changed by modifying light conditions, presence of obstacles, as well as the number and the shape of them.

First test is carried out in a simple corridor scenario, with no obstacles at all and artificial lights. Floor and walls surfaces don't reflect light in a significant way and artificial illumination is present. The robot is positioned in the middle part of the corridor, so what is expected is it to go simply straight without modifying the trajectory. In the first try, the Turtlebot behaves as expected, maintaining the position in the center of the corridor for all the duration of the test.

Tests with the same conditions are then repeated, reporting a smooth behavior, with sometimes some little changes of trajectory, due to a small variability in the results of segmentation: this does not represent a problem, because the algorithm forces the robot to correct the trajectory with the successive results of segmentations. At the end of the

tries with this setup of the environment what is noticed is a 100% ratio of success over tries, with the robot neither hitting obstacles (represented only by walls) or even going close.

Second test is made in order to create harder conditions, but without going too deep yet. Conditions are the same as the previous setup, with the only difference of putting a box in the middle path of the corridor. The Turtlebot is able to avoid the obstacle, then proceeds its motion going towards the left wall and then adjusts its trajectory to be parallel to the barrier and continues the motions without other relevant changes. Test is repeated with different illumination conditions, turning off the artificial lights and letting the natural light in morning hours to be the only source. No relevant changes on the behavior are reported.

At this point, in order to complicate the scenario, the intention is to use a set of boxes to increase the number of obstacles and this is achieved using the same box and changing its position real-time during the motion of the robot. Even if in this case a dynamic situation is created, the following position of the box is chosen in order to make the scenario very similar to a static one, considering that the vertical distance between the box position is 3 meters and the robot proceeds with a velocity of 0.1m/s.

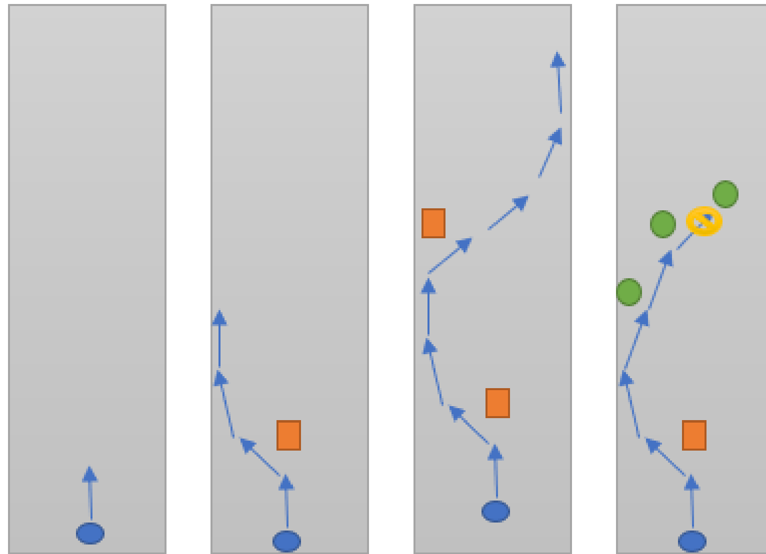

Figure 5.11. Test schemes.

In the third test the results are good again on the two replications carried out. The robot avoids the first obstacle in the same way of test 2 and in a similar way is able to not hit the second box on its trajectory, then continues its motion towards the right wall and when the distance becomes too little, the trajectory is adjusted in order to avoid it.

Last test in the corridor scenario results to be the one with the worst conditions. This time the box is positioned in the middle of the corridor for all the duration of the try and a person represents the successive obstacles. In this new scenario the different conditions are mainly two: first there is a human shape to segment, that is more complicated than a box with a rectangular section; second the person moves in the corridor, defining a dynamic scenario, increasing the probability of the robot to get confused.

The qualitative results of the test are the following: the robot is able to avoid the box and the person, reporting no big issues in recognizing it as an obstacle, and then, with

the human subject moving and representing another impediment, Turtlebot continues behaving correctly. The fault arises when the person moves into the third position, that is very close to the second one: the new command is not obtained fast enough to adjust the trajectory correctly.

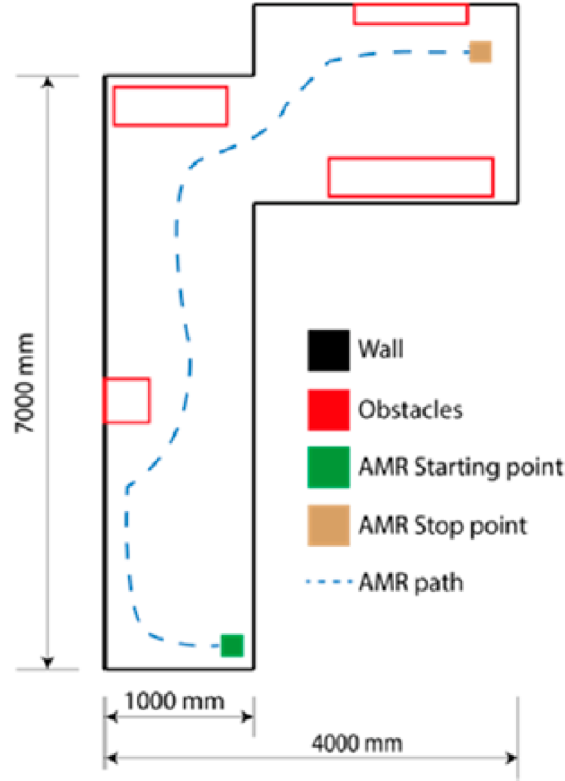

Figure 5.12. AMR test structure.

Test ID	Obstacles-Boxes	Obstacles-People	Outcome
1	None	None	Successful
2	1	None	Successful
3	2	None	Successful
4	1	3	Collision with the last person

Table 5.1. Corridor test recap

Chapter 6

CONCLUSION

The purpose of this thesis was the implementation of a solution for robot navigation in indoor environment, using computer vision, and deep learning for image understanding.

The proposed solution consists in two sections: the part related to find and train a suitable model for scene segmentation and the building of the navigation algorithm. For the computer vision part, a research about a proper model was made, with a phase of training of multiple models to get the a good enough network for the project needs. The final choice was on a PSP network, with the backbone (ResNet-34) pre-trained on the ImageNet dataset, upon which a fine-tuning is carried out, with a specific dataset for binary RGB indoor segmentation. For the creation of the navigation algorithm, a series of function which are directly related to the pixels of the segmented images are created with the goal to obtain a waypoint, in terms of pixel coordinates on the binary mask. A proportional controller based on the obtained coordinates is used to obtain the motion commands of the robot. With the use of ROS and a Jetson AGX Xavier Hardware, the algorithm has been tested on a Turtlebot3, with the results that testify that it's possible to achieve indoor navigation on the base of binary RGB scene segmentation. A future improvements for the algorithm can be the add of an actual goal to reach: in this configuration, the mobile robot navigates with the ability to carry out a collision free motion, but without a real point to achieve in the space, continuing its activity until stopped via software. Also, a point on which it's possible to get better results is the frequency with which the robot gets commands, for now at 5Hz. This improvement could be achieved working on a strategy a further optimization of the post processing code, reducing the number of operation and the pixels to be directly analyzed.

Bibliography

Abien Fred Agarap. Deep learning using rectified linear unit, Feb. 2019.

Morse Bryan S. Image understanding, 1998.

Abhishek Chaurasia and Eugenio Culurciello. Linknet: Exploiting encoder representations for efficient semantic segmentation. *2017 IEEE Visual Communications and Image Processing (VCIP)*, Dec 2017. doi: 10.1109/vcip.2017.8305148. URL http://dx.doi.org/10.1109/VCIP.2017.8305148.

elinux.org. Jetson agx xavier. 2022. URL https://elinux.org/Jetson_AGX_Xavier.

Surbhi Gupta, Sangeeta R, Ravi Shankar Mishra, Gaurav Singal, Tapas Badal, and Deepak Garg. Corridor segmentation for automatic robot navigation in indoor environment using edge devices. *Computer Networks*, 178, 2020. ISSN 1389-1286.

Aurèlian Gèron. *Hands on Machine Learning with Scikit-Learn, Keras and Tensorflow*. Oreilly & Associates Inc, 2019.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition multimed. 2015.

Y T. Helland. Rgb2gray. 2011. URL https://tannerhelland.com/2011/10/01/grayscale-image-algorithm-vb6.html.

Roberto Jacome, Miguel Realpe, Luis Chamba-Eras, and Marlos Santiago Vinan-Ludena. Computer vision for image understanding. *Conference: The International Conference on Advances in Emerging Trends and Technologies*, May 2019.

Asfullah Khan and al. A survey of the recent architectures of deep convolutional neural networks. *arXiv preprint arXiv:1901.06032*, 2019.

A. Krizhevsky, I. Sutskever, and G.E. Hinton. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

Yann LeCun and al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks Stat. Mech*, 1995.

Yann LeCun and al. Gradient-based learning applied to document recognition. page 2278–2324, 1998.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, and C. Lawrence Zitnick. *Microsoft COCO: Common Objects in Context*. Springer International Publishing, 2014.

Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.

Rajesh Kumar Riaz Khan, Emella Opoku Abogye. Evaluating the performance of resnet model based on image recognition. Nov. 2015.

Dennis Sayed. Riassunti reti neurali, 2018.

K. Simonyan and A. Zisserman. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

Karen Simonyan, Aleksander Zisserman, and al. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 1(60):11, 2014.

Daniel Teso-Fz-Betoño, Ekaitz Zulueta, Ander Sánchez-Chica, Unai Fernandez-Gamiz, and Aitor Saenz-Aguirre. Semantic segmentation to develop an indoor navigation system for an autonomous mobile robot. *Mathematics*, 8(5), 2020a. ISSN 2227-7390. doi: 10.3390/math8050855. URL https://www.mdpi.com/2227-7390/8/5/855.

Daniel Teso-Fz-Betoño, Ekaitz Zulueta, Ander Sánchez-Chica, Unai Fernandez-Gamiz, and Aitor Saenz-Aguirre. Semantic segmentation to develop an indoor navigation system for an autonomous mobile robot. pages 17–19, 2020b.

Cai Yanguang. Wei Wu Yao Yeboah and Zeyad Farisi. Semantic scene segmentation for indoor robot navigation via deep learning. pages 3–4, 2019.

Song Yuheng and Yan Hao. Image segmentation algorithms overview, 2017.

Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network, 2017.