POLITECNICO DI TORINO

Master Degree course in Data Science and Engineering

Master Degree Thesis

Assessing the Feasibility and Performance of Real-Time Semantic Segmentation in an Industrial IoT use case

Supervisors
Prof. Andrea Calimera

Candidate
Giacomo Zema

Academic Year 2021-2022

Acknowledgements

Alla mia famiglia, che da sempre mi fa sentire capace di raggiungere qualsiasi traguardo.
Ad Alessandra, il mio futuro.

Abstract

Semantic Segmentation is a computer vision task that consists in assigning a label to every pixel in an input image. A semantic segmentation model outputs a prediction of the same size as the input, in which each pixel is classified in a particular class. There are many applications for Semantic Segmentation such as scene understanding in autonomous driving or robot vision, land cover classification of satellite images, segmentation of medical images and others. Semantic Segmentation models are often very complex and require powerful hardware. This clashes with the usage of such models in a resource-constrained environment such as edge devices in an IoT network. For this reason, the standard approach when deploying semantic segmentation models in an IoT application is to offload both training and inference to a cloud server.

While performing training on a server equipped with a GPU is actually a smart choice, offloading the inference phase can be rather inefficient. The issue regards the transfer of data from the edge to the server and back, which is highly expensive in terms of energy and also poses critical issues in terms of scalability, inference time and data security. Many applications of semantic segmentation on edge devices rely on collecting image data from cameras or other sources and processing them right after, therefore the inference phase has real-time requirements that cannot be fulfilled by offloading it to a cloud server. So, for such applications, a clear solution is to perform the inference on the edge devices. There are many cases in the literature of real-time semantic segmentation models, but virtually none of them achieve real-time latency (lower than 33 ms) on less powerful hardware. The reason behind this circumstance is that these models are evaluated on really complex datasets, that favor networks with a larger number of parameters and more elaborate structures.

In this thesis, we explore different types of optimizations that can be applied to a network to improve its inference latency. The goal is to define an optimization pipeline that can be used to adapt the performance of a model to the requirements (IoU score and latency) of a specific task. We structured this pipeline according to the effort required by each step where with effort we intend both the time required by the operation and the difficulty of its implementation. The proposed optimizations are grouped into two main branches: Input Data and Model Topology. The first one is the most immediate and consists in changing the size of the input images in order to speed up the inference, it doesn't need any knowledge of the model and therefore requires the least effort. The second branch demands a deeper understanding of the underlying network to identify important tuning knobs related to its different layers and functional blocks.

In the experimental part of this thesis, we selected a specific industrial IoT use case and a baseline network amongst the several models reviewed in the background chapter. We ran the experiments on a Raspberry Pi 4 and we studied in detail every optimization approach and their combinations, gaining meaningful insights that allowed us to propose an optimization framework that accounts for different levels of effort.

Contents

Chapter 1

Introduction

Semantic Segmentation is a computer vision task that consists in assigning a label to each pixel in an image (dense prediction). Each label corresponds to a specific class. The result is an output segmentation map in which, every pixel belonging to an object in the input image, is assigned a value that represents the class that the object belongs to.

From the output of semantic segmentation, we can derive information about the position of an object in the image, its shape, size and orientation. Because of this, semantic segmentation models are used in a large variety of scenarios.

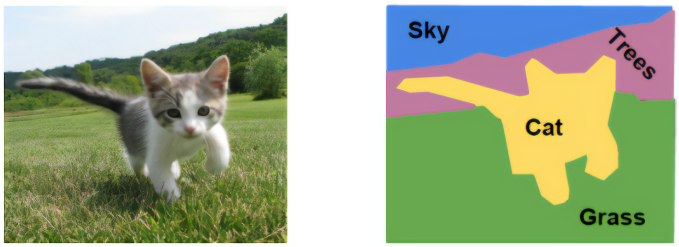

Figure 1.1. Semantic Segmentation - credit [29]

Scene Understanding is the main application. It attempts to analyze objects in context with respect to the 3d structure of the scene, its layout, and the spatial, functional and semantic relationship between the objects. The most popular application of semantic segmentation for scene understanding is autonomous driving. An autonomous vehicle must sense its surroundings and act safely to reach a certain target. Performing semantic segmentation is crucial for interpreting the visual data collected by the cameras in the vehicle, and this allows the control system to understand the environment and accordingly make the right decision.

Another application is in the field of robotics, Robot vision presents many possible applications for semantic segmentation, one of which is robotic grasping [1]. Automated grasping is the process of having a robot manipulator successfully grasp objects in a cluttered environment. The grasping procedure consists of 2 steps: the first is to model the scene and identify the objects and the second step consists in executing the action. For

Figure 1.2. Example ground truth map from the Cityscapes Dataset [9]

the first part to be successful, the shape and the location of the object must be known in advance, for this reason, thanks to its pixel-level output, semantic segmentation is a great tool for this task. Automated grasping is widely deployed in both manufacturing and service applications of robots.
Another notable application of semantic segmentation is Medical Image Segmentation, which involves the extraction of regions of interest from MRI or CT scans [31]. This operation has an important role in diagnosis, pre-operative planning and post-operative management of patients. In the medical domain, segmentation can also be used to identify areas of the anatomy required for a particular study, for example, to simulate physical properties or virtually position CAD-designed implants within a patient.
Semantic segmentation can be used in the screening of microscopic slides, where it refers to the process of finding the boundaries of cells, cell nuclei and histological structure with adequate accuracy of images of stained tissue with different markers. Performing an accurate segmentation of each cell is a difficult task because of the diversity of structures contained in the images, the intense variation of background, and other issues.
Semantic segmentation can be also used on satellite images. One of the most common applications is land cover classification [34] which takes as input a multi-spectral satellite image of an area and outputs the land cover map of the area. Similar tasks are performed in the context of precision agriculture, where semantic segmentation can be used to isolate areas like agricultural parcels.

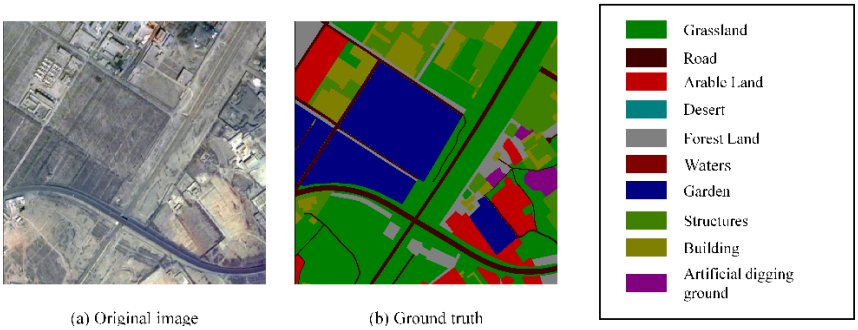

Figure 1.3. Land Cover Classification on Satellite Images [28]

Facial recognition is another possible application of semantic segmentation. Semantic segmentation of faces involves categories such as eyes, nose and mouth in addition to skin, hair and background. Detailed segmentation of facial features can then be used to train other computer vision applications to distinguish an individual's ethnicity age and expression. Another use for it can be the enhancement of photographic portraits through depth of field simulation and other processing techniques that require a precise separation of the face from the rest of the image.
For these applications and many more, the literature presents several datasets that can be used to train and evaluate semantic segmentation models.

A subset of the applications of semantic segmentation is made of those that require real-time inference. These are the scenarios in which there is the need to process an image right after its collection. The real-time requirement reflects constraints regarding the inference latency. Ideally, the lower the latency the better, but there are certain cases in which such requirements are stricter. The most immediate example is autonomous driving. The segmentation of the images is only a part of the pipeline that makes up this task, so the inference latency of the segmentation model needs to be lower than the latency constraint of the entire task. Conventionally we define as real-time any operation that is capable of producing more than 30 frames per second (33ms latency), but there are some cases in which this value can be too high or way too low. Coming back to autonomous driving, it would probably be better to have more than 100 fps (10ms latency) in order for the decisions to be made on images updated very frequently, especially as the speed of the vehicle increases.

The need for a lower latency is not only limited to real-time applications, but also for energy-constrained environments. In fact, latency is strictly correlated with power consumption. Before data and instruction parallelization, the indicator of power consumption of an application was entirely dependent on the number of multiplication-accumulation operations (MACs) that it required. But when more instructions can be executed at the same time and more data points can be loaded into the CPU memory at once, there can be cases in which models that require more MACs but better exploit the parallel architecture, achieve the same or even lower power consumption compared to a model that does not.
Therefore, a model that is capable of lower inference latency, will draw less power, which makes it more suitable for deployment in applications for which energy efficiency is critical.

The domain on which we will focus in this thesis is IoT networks. An IoT network can have several different configurations, but the most common consists of a large number of low-power devices collecting data at the edge of the network and a server at the center of the network, or in the cloud, that gathers the information coming from the edge-devices and makes decisions. Server and edge-devices are connected through an internet connection.

Edge-devices can have very different characteristics. They can have a Micro Controller Unit (MCU), an example of which are ARM Cortex-M series; some of them have proper

Figure 1.4. ST Microelectronincs Smart Camera Module with the STM32H747 MCU

CPUs, like the 64bit ARM Cortex A72 in the Raspberry Pi (used in this thesis). In some cases, edge devices can be equipped with an embedded GPU, like the NVIDIA Jetson family, or custom accelerators specific for deep learning inference, which typically use spatial or systolic architectures to optimize the highly-parallel MAC kernels that are at the core of most key neural network layers.

Computer vision models, such as semantic segmentation, are often very complex and require a large number of computations. This aspect collides with their deployment in resource-constrained environments such as edge-devices in IoT networks. For this reason, the standard approach for deep learning in an IoT context is to offload both training and inference phases to a remote server (or cloud).

Training is an offline task and it makes sense to perform it once in a while on a server equipped with a GPU, while offloading inference can be problematic. The issues regard the network connection. Firstly, transferring image data to the server and transferring back the prediction map is highly energy expensive, which is critical since many IoT devices are battery-powered. Additionally, image data can contain sensitive information and this can raise privacy concerns when transferring them over the network. Finally, offloading the inference phase leads to unpredictable latencies when devices have a slow or intermittent connection. Therefore, if an application has real-time requirements, offloading the inference would make it impossible to meet such requirements, while bringing the inference to the edge is a way of achieving predictable latencies.

The first step toward real-time semantic segmentation is to choose an adequate model. There are plenty of such networks in the literature, which employ different design-time approaches to lower the inference latency. Thanks to these techniques, these models will be faster compared to not optimized counterparts across different hardware. The issue is that, obviously, they will not achieve real-time inference on all of the devices on which

they can be deployed.

For this reason, there are further optimizations that can be implemented after the model has been designed to bring its performance closer to the requirements of the task, on the available hardware platform. This is a very vast family of optimizations, ranging from the manipulation of the model structure through its simplification or its compression [15], to the quantization of the parameters and the activations to lower bit representations [21] and many others.

In this thesis, we face the challenge of achieving real-time inference in a semantic segmentation task on an IoT device.
Computer vision tasks on edge-devices, often rely on the collection of image data from cameras and processing them right after. This means that the inference phase has real-time requirements, but there aren't any models that are able to meet such requirements on less powerful devices. This is because real-time semantic segmentation models are evaluated on very complex datasets, like Cityscapes [9] or CamVid [3], which present a high number of classes and images with many objects at different scales. While autonomous vehicles are the main example of the need for real-time segmentation, they are not representative of a realistic IoT use case. As a matter of fact, modern "self-driving" cars are equipped with very powerful hardware.

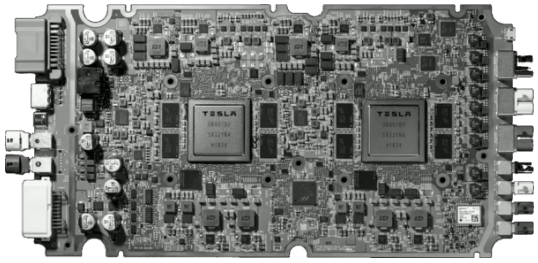

Figure 1.5. Tesla's latest Full Self Driving (FSD) chip [43]

Even smaller models designed to achieve real-time performance need to have enough parameters to accurately capture context and detail information from these elaborate input images.
Typical IoT applications won't be as demanding as the datasets that these models are evaluated on. Therefore, there is a large headroom for optimizing models in order to meet the requirements of the task.
Examples of less demanding semantic segmentation applications can be delivery drones [2], as the segmentation of aerial views is far less elaborate than busy urban driving scenes. Other examples are in the robotic domain like grasping [1] or inspection [36]. Both of these tasks do not operate on very complex input images and are performed on hardware that is resource-constrained in terms of energy or computational power.

In this thesis, we will focus on methods to bring the inference latency of an established

model under the requirements of a given task on a given dataset.
Starting from an existing model is easier than proposing a brand new one. It does not require the validation of the new architecture and, most importantly, it makes much more sense to modify an existing network to adapt it to a specific task, rather than designing an ad-hoc model just for that task. The amount of work that goes into designing and testing an original model is not justified if its purpose is to be deployed on just one dataset.

In real-world applications, it is common to take an existing network and deploy it on the desired data.

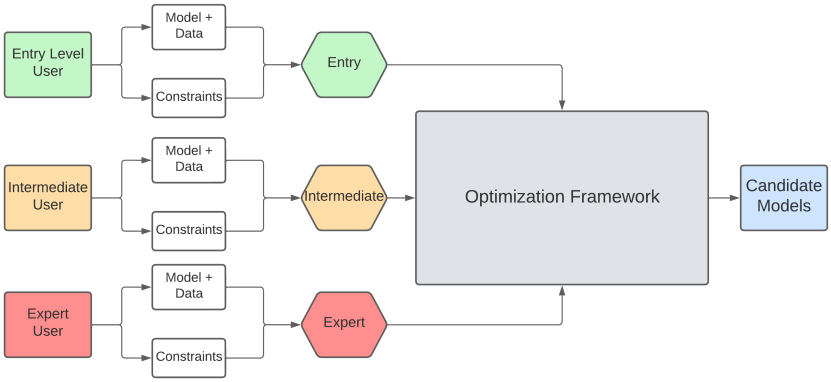

Figure 1.6. A Representation of the Optimization System

Our goal is to provide a system of optimizations that allows developers to manipulate the behavior of a network to meet certain constraints in terms of inference latency while keeping acceptable accuracy. We structured the systems according to the effort required by the optimizations it implements. In this context, the term "effort" describes both the expertise required to implement an optimization technique and the time required to execute it. In particular, the proposed automatic optimization framework accounts for different levels of expertise by allowing users to enter from different points based on their ability to understand the structure of the network. Moreover, users with different amounts of available time can exit the network at different points. It is not easy to define time constraints because different baseline models will result in different execution times for each optimization, therefore there is no way of knowing in advance how much time the optimization process will take. Time is an important factor especially when the models are trained on a cloud computing platform whose billing mechanism is based on the duration of the execution.

In the experimental part of this thesis, we will focus on finding a pipeline of optimizations that can make up the inner working of such a system. We selected a dataset that represents a realistic use case for real-time semantic segmentation and a baseline model from those reviewed in the background chapter. The hardware platform chosen to represent an edge-device is a Raspberry Pi 4 model-B with 4GB of RAM.

Out of the possible ways to reduce the inference latency of a network we chose to analyze two main approaches: simplifying the input data and simplifying the structure of the network. These are completely different in terms of effort, since the first one only requires handling the input images, while the second requires a deeper understanding of the baseline architecture in order to feed the system a parametrized network and a set of hyperparameters and their values. Changing the size of the input is mentioned in the literature of some real-time segmentation models [35], with the purpose of achieving better latency on the same network structure while losing some accuracy. Network simplification, on the other hand, is often mentioned as the most effective way of improving latency without severely impacting accuracy [10], but it is not very well documented, except in papers presenting efficient network architectures as an improved version of bigger and slower models [8] [22] [20].

Each optimization in the analysis corresponds to a version of a baseline model chosen beforehand, which will feature different input characteristics, a modified structure, or both. Every one of these models will be evaluated in terms of inference latency and intersection over union score (IoU), which is the most effective way to evaluate the output quality of a semantic segmentation model. The results gathered from the evaluations will be used to make important considerations on the tuning knobs of each optimization stage and on the whole optimization pipeline.

In the end, we will be able to understand the way in which each optimization affects the behavior of the model and how to combine them in order to meet given accuracy and latency constraints.
Having gone through the optimization pipeline, we will be presented with several variations of the baseline model that respect the pre-defined accuracy and latency constraints. The choice of one model will depend on every specific task: if it prefers faster inference or better accuracy.

As a future development, from the set of experiments performed in this thesis, particularly the network simplification part, we can gather useful guidelines that can be used to design a brand-new network architecture. As a matter of fact, especially in the feature extraction part, many convolutional neural networks share components if not entire backbones. Therefore, understanding how these components react to different simplifications, can be useful when integrating them into the structure of a model.

Chapter 2

Background

In the previous chapter, we have introduced what semantic segmentation is and mentioned some of its applications in different domains and environments. Now we can explore in more detail which are the most relevant datasets and explore the defining characteristics of semantic segmentation models by reviewing the state of the art.

At the end of this chapter, we will also present the leading design-time optimizations for real-time semantic segmentation models.

2.1 The Datasets

A dataset for semantic segmentation consists in images and the corresponding segmentation maps.
Each dataset presents a certain number of categories and in each image there can be objects belonging to different classes as well as multiple instances of the same class.
Surveying the literature reveals that the principal datasets can be grouped into the following categories:

- Scene understanding
- Urban driving
- Medical imaging
- Aerial images (satellite and drones)
- Others

2.1.1 MS COCO

The MS COCO (Microsoft Common Objects in COntext) dataset [25], belongs to the scene understanding category. It contains annotations for several deep learning tasks, including semantic segmentation. It contains very heterogeneous images that portray scenes from various context.

Figure 2.1. Samples from the MS COCO Dataset

The dataset presents 91 classes that form a total of 2.5 million instances in 328k images. The 91 classes can be grouped into 11 macro-categories:

- person and accessories
- animal
- vehicle
- outdoor objects
- sports
- kitchenware
- food
- furniture
- appliance
- electronics
- indoor objects

2.1.2 Pascal VOC

Pascal Visual Objects Classes (VOC) [12] is another dataset for scene understanding that features annotations for semantic segmentation and other computer vision tasks. It presents similar characteristics to COCO, but it is less complex. The segmentation image set contains 1464 training samples and 1449 testing and validation samples.
Pascal VOC presents 20 classes organized hierarchically, that form 4 macro categories:

- Vehicles
- Household
- Animal
- Person

2.1.3 ADE20K

ADE20K [50] is another scene understanding dataset that has similarities with COCO and Pascal VOC in ths sense that it represents a wide variety of scenes and a large number of object categories.
It contains 20,210 images in the training set, 2,000 images in the validation set, and 3,000 images in the testing set. Similarly to COCO and Pascal, the categories are organized in a hierarchical structure as objects and parts. All the images are exhaustively annotated with objects and many objects are also annotated with their parts. Parts can have parts too: for example, the 'rim' is a part of a 'wheel', which in turn is part of a 'car'. A 'knob' is a part of a 'door' that can be part of a 'cabinet'. The total part hierarchy has a depth of 3. Overall, it contains 150 object categories, not including parts.
Any image in ADE20K contains at least 5 different objects and up to 273 instances, which become 419 when counting parts as well. This makes it one of the most complex datasets.

Figure 2.2. Samples from the ADE20K Dataset

2.1.4 Cityscapes

We are treating urban driving scene datasets as a separate category, but they are a subset of scene understanding datasets. In fact, some of the images from COCO and Pascal are from driving scenes. The reason why we treat this category separately, is that these datasets only contain images from the point of view of a vehicle and only present urban driving scenes, contrary to the huge variety of contexts present in COCO and Pascal.
Cityscapes [9] is comprised of a large, diverse set of stereo video sequences recorded in streets from 50 different cities across several months. It presents 2 sets of images, the first consists in 5000 images with high quality pixel annotations, the second consists in 20000 images with coarse annotations. All of the images were shot in good weather conditions, because the researcher believed that segmentation of driving images with adverse weather is a task that requires specific measures and a different approach.

It contains 30 classes belonging to 8 macro-categories:

- Flat
- Human
- Vehicle
- Construction
- Object
- Nature
- Sky
- Void

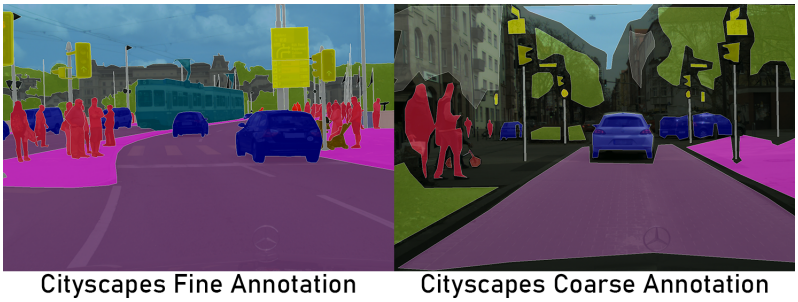

Figure 2.3. Samples from the Cityscapes Dataset

2.1.5 CamVid

Cambridge-driving Labeled Video Database (CamVid) [3] was introduced as the first collection of videos with semantic annotations. The database provides ground truth labels that associate each pixel with one of 32 semantic classes. It contains 367 in the training set, 101 in the validation set and 233 in the test set. Differently from the datasets introduced above, in CamVid, classes are not organized hierarchically.
The dataset contains 4 video sequences that were recorded with an HD camera mounted to the dashboard of a car. One of the sequences was shot at dusk, while the others were shot in the daylight with sunny weather conditions. The total duration of the sequences is 22 minutes and 14 seconds.

2.1.6 SYNTHIA

SYNTHIA [38] is an urban driving scene dataset that has a peculiarity: it is a collection of photo-realistic frames rendered from a virtual city created with the Unity development

platform.
The dataset contains pixel level annotations for 13 semantic classes:

- misc
- sky
- building
- road
- sidewalk
- fence
- vegetation
- pole
- car
- sign
- pedestrian
- cyclist
- lane-marking

It is divided into 2 complementary set of images, the first one contains 13400 images from random point of view across the city, while the second one contains 200000 images captured from the perspective of a car driving in the streets of a virtual city. This dataset is rarely used as a benchmark for new models, like cityscapes or camvid, but its inclusion in the training stage has proved to significantly improve performance on datasets that contain real-life images.

Figure 2.4. Samples from the SYNTHIA Dataset

2.1.7 BRATS 2015

The multimodal BRAin Tumor Segmentation (BRATS) [31] dataset contains two types of images: clinical image data and synthetic image data.
The Clinical Image Data consists of 65 multi-contrast MR scans from glioma (the most frequent primary brain tumor in adults) patients, out of which 14 have been acquired from low-grade glioma patients and 51 from high-grade glioma patients. The images represent a mix of pre and post-therapy brain scans.
The Synthetic Data consists of simulated images for 35 high-grade and 30 low grade gliomas that exhibit comparable tissue contrast properties and segmentation challenges as the clinical datasets.

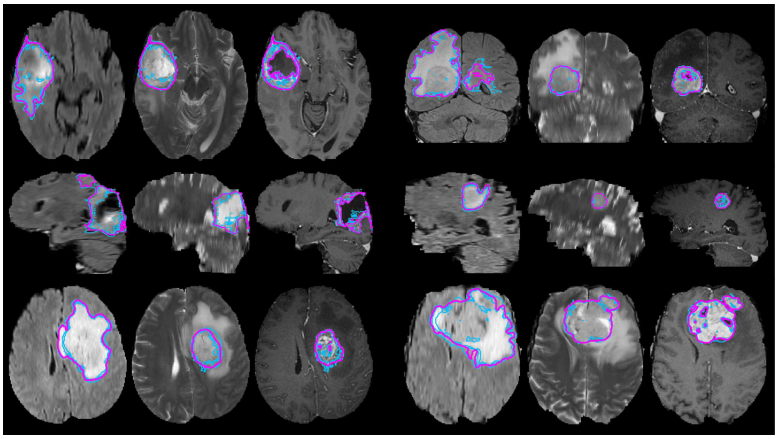

Figure 2.5. Samples from the BRATS 2015 Dataset

The simulated images are annotated by the software that generated them, while the clinical images require manual annotations. The annotations define a "tumor area" that presents 4 types of intra-tumoral structures: edema, non-enhancing solid core, necrotic core and non-enhancing core. Because, in many cases, tumor boundaries are difficult to define clearly, each image has been labeled by several experts and the results were subsequently fused to obtain a single "consensus segmentation".

2.1.8 Dubai UAE Dataset

The Dubai UAE dataset as an example of a dataset for semantic segmentation of aerial imagery. It is a joint project by Humans in the Loop and the Mohammed Bin Rashid Space Center in Dubai.
The dataset consists in aerial images of Dubai obtained by MBRSC satellites. The images are annotated for semantic segmentation and contain objects belonging to 6 total classes:

- Buildings
- Land
- Road
- Vegetation
- Water
- Unlabeled

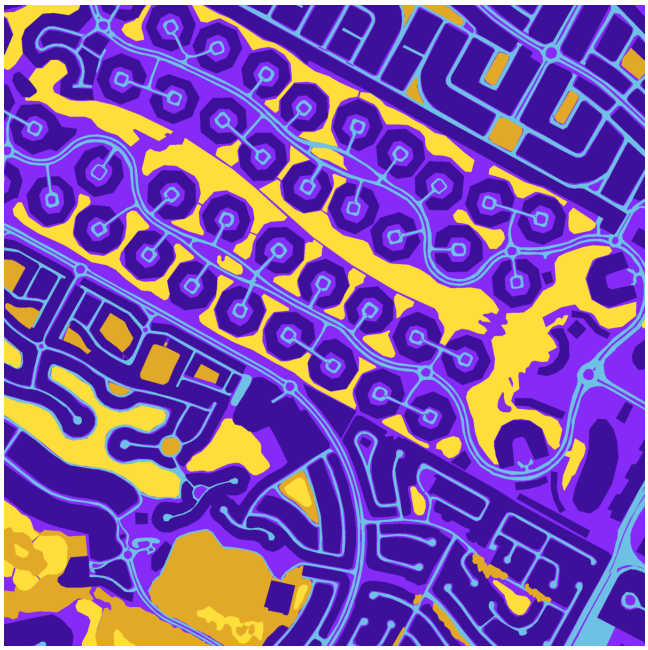

Figure 2.6. A sample annotation from the Dubai UAE Dataset

The total volume of the dataset is 72 images grouped into 6 larger tiles.
Differently than the other domains, there isn't a predominant dataset for aerial images. Nonetheless this is one of the most requested applications and one of the most challenging. In fact, especially for satellite images, there are so many factors at play, that it is very hard to achieve good accuracy on all classes.

2.1.9 Other Datasets

Semantic segmentation is not limited to the applications described above. There are many popular datasets belonging to other categories.

Person-In-Context (PIC) [26] is a dataset for human-centric relation segmentation, which aims to predict the relations between the humans and the surrounding entities and identify relation correlated human parts.

The CalCROP21 [13] dataset contains satellite data (at 10 spatial resolution) of the Central Valley region in California/ The semantic categories are the crop types grown in that area.

SUIM (Segmentation of Underwater IMagery) [23] contains over 1500 images with pixel level annotations for 8 categories of underwater objects (fish, reefs, plants, wrecks/ruins, robots, sea floor). With a variety of use cases, this dataset opens up promising opportunities for future research in underwater robot vision.

LIP (Look Into Person) [14] is a dataset for Human-Parsing. It contains over 50000 annotated images, with 19 semantic classes. In a human parsing task, the classes represent different parts of the human anatomy and clothing. MHP (Mulitple-Human Parsing) [24] is also a dataset for human parsing, but, differently from others, it contains images in which there is always more than one human.

FoodSeg [44] is a large dataset for food image segmentation. It contains 9490 images annotated with 154 ingredient classes. A dataset like FoodSeg lends itself to many use cases, like estimation of calories from the picture of a prepared dish, automated checkout at a self-service restaurant and so on.

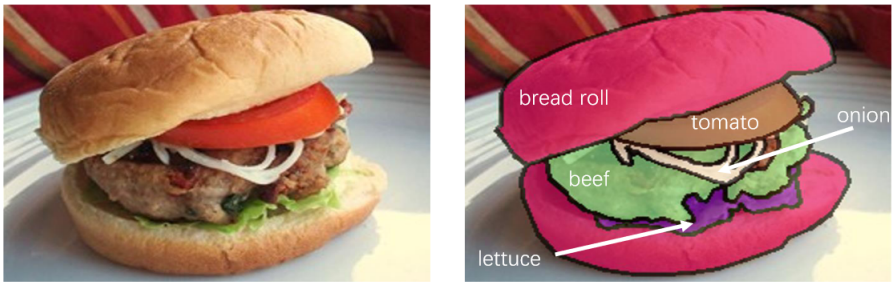

Figure 2.7. A sample from the FoodSeg Dataset

2.2 The State of the Art

Before we analyze what are the state-of-the-art approaches and models for semantic segmentation, it is important to explain which are the challenges of this task and highlight the difference with other computer vision applications.
In introducing these challenges of segmentation, we need to consider the output of the models. The fact that the model outputs a prediction map instead of a one-hot encoded vector means that the features required to produce a prediction need to carry additional information.

In convolutional models, context information is vital to make accurate predictions. For tasks like classification, context information is acquired by gradually subsampling the input image across the structure of the network.
At this point it is useful to introduce the concept of receptive field. This is one of the basic concepts in convolutional neural networks. The receptive field of a feature in a CNN is the region in the input space that is used to compute the value of that particular feature. Therefore, if the receptive field of a feature is small, it means that that feature does not contain much context information. On the contrary, if the receptive field is large, it means that a big region of the input was used to compute the value of that feature and that it embeds more context.
Although subsampling layers are helpful to expand the receptive field, they completely ignore feature resolution, which is crucial for a pixel-level task like semantic segmentation.

Another challenge in semantic segmentation models is the presence of objects belonging to the same class represented at different scales within the same image. One example of this phenomenon can be a Cityscapes image in which there are two vehicles at different distances from the point of observation, the further vehicle will appear smaller than the one that is closer while belonging to the same semantic category. Building a model that is resilient to scale variations is not trivial, especially considering that most classification models, upon which semantic segmentation networks are based, rely on scale information to make predictions

The third challenge in the application of convolutional neural networks to semantic segmentation si the reduced localization accuracy. This issue relates to the fact that an object-centric classifier requires invariance to spatial transformations, inherently reducing the spatial accuracy. In practical terms, this means that score maps of deep convolutional networks can predict the presence and the rough position of objects, but cannot clearly delineate their borders. This happens because the information about the precise (pixel-level) location of an object in an image is lost as the input image travels through the layers of the network.

2.2.1 Fully Convolutional Networks

Fully Convolutional Networks (FCN) are one of the earliest approaches to semantic segmentation.
The idea behind FCNs is to modify existing CNN architectures by replacing all fully-connected layers with convolutional layers, so that the output is a spatial segmentation map and not an array of classification scores.

It has been observed how important skip connections are in this kind of models, thanks to skip connections feature maps from the final layers are up-sampled and fused with feature maps of earlier layers so that the model combines semantic information from deep layers with appearance information from shallow layers.
One of the limitations of a simple FCN model is that it does not take into account global context information as well as some more advanced architectures. This is the reason why later models integrated particular layers and functional blocks into FCNs.

Considering just the basic definition we can say that all of the models analyzed in this chapter are Fully Convolutional Networks. In fact, none of them features Fully Connected layers.

An interesting consequence of eliminating the Fully Connected layers is that there are no restrictions to the dimensions of the input. In fact, a FCN model that was trained on 512x512 samples can be fed a 256x256 image and it will return a 256x256 prediction map without functional issues. This does not mean that the output quality will be the same when changing the input resolution, but that is a topic for later.

An example of a simple Fully Convolutional Networks was introduced in [29].
The main idea is to combine layers of the features hierarchy in order to refine the spatial precision of the output. To do so, skip connections are added from shallower to deeper layers.
The purpose of skip connections is that shallower layers have smaller receptive fields and

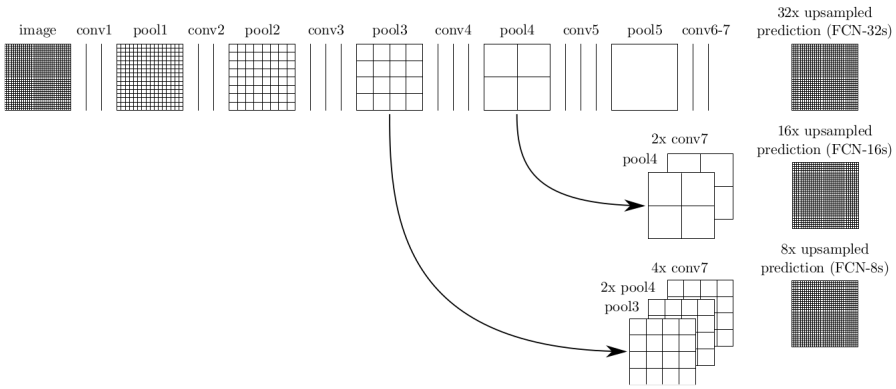

Figure 2.8. Skip Connections in the FCN model [29]

therefore are more suited for local predictions, while layers further down the structure of the network are able to capture more context information.
The feature maps from the different skip connections are combined at the end of the network, where they are brought to scale agreement by upsampling the lower resolution maps, which are then cropped to be aligned with the other layers (portions of the upsampled layer that extend beyond the other layers because of padding are removed).
The layer fusion operation is done througn the concatenation of the feaure maps, after which the prediction map is produced by a pointwise convolution (1x1 kernel size).

2.2.2 Encoder-Decoder Models

In a typical convolutional network, the height and the width of the input image is gradually reduced with convolutional layers and pooling layers. The reduction of these dimensions allows deeper filters to focus on a larger receptive field (same size filter, smaller input image). Moreover, alongside the reduction in height and width there is an increase in the number of channels, which means that more complex features are extracted from the image. As we have said before, this comes at the cost of localization accuracy and feature resolution.
Encoder-decoder models perform downsampling using a classification backbone (encoder) and then try to reconstruct the lost information in the second module of the network, the decoder. The decoder's job is to bring the resolution of the feature maps to the resolution of the input image and in doing so integrate pixel-level detail information.

Due to the shape of the feature maps across the layers of the network, the model that introduced the encoder-decoder architercture is called U-Net [37].
The encoder in U-Net consists in a series of convolutional and pooling layers and, in

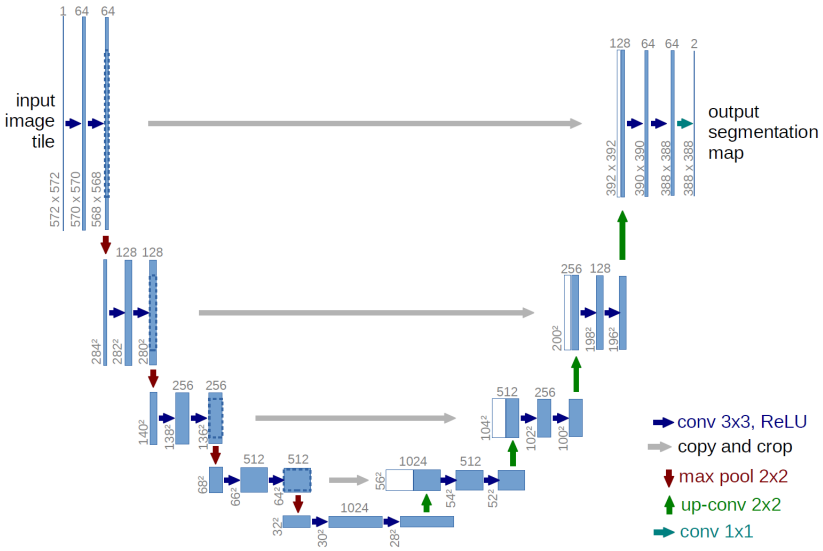

Figure 2.9. The U-Net structure [37]

some variation of this model, it consists of a classification backbone like ResNet [16] or Inception [41]. The fact that the encoder is obtained from a classification model, means that this portion of the network can be pretrained on a classification dataset like ImageNet to facilitate the training of the complete network on a segmentation task.

The decoder in U-Net has the purpose of enabling precise localization when upsampling the output of the encoder. The upsampling can be done either by un-pooling layers (bilinear upsampling) or transposed convolutions. The difference between the two is that transposed convolutions are learnable.

An important feature of U-Net is the presence of skip connections between the various blocks of the encoder and the corresponding blocks in the decoder, whose purpose is to keep alive information that is important to the segmentation output. This way, detail information that is present in the feature maps closer to the input, is brought back in the decoder.

2.2.3 Dilated Convolutions

While it is not a category of models by itself, dilated convolution is one of the core components of many semantic segmentation models.
A dilated convolutional layer "inflates" the kernel by inserting holes in between its elements. The dilation rate l is the hyperparameter that regulates how much the kernel is widened. The space between two adjacent elements of the kernel after dilation is l-1.
The effect of dilated convolution is that with the same kernel size, it is able to reach a

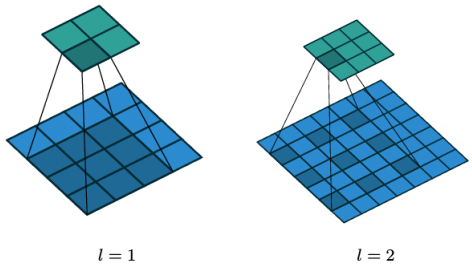

Figure 2.10. Dilated Convolution [48]

wider portion of the input (wider receptive field), thus capturing more context. It is similar to what is achieved by subsampling the input and then performing the convolution. Only in that case, instead of widening the filter, the input image is shrunk so that the same filter can cover a wider context.

So, the use of dilated convolution is introduced in semantic segmentation models in order to widen the receptive field, without loosing resolution by subsampling the input. In [30] the researchers define the concept of Effective Receptive Field by computing, with partial derivatives, the impact that each pixel belonging to the theoretical receptive field

actually has on the output of the layer. In doing so they compare the ERF of Subsampling, Dilated Convolutions and Simple Convolution. It is clearly visible how dilated convolution presents a larger effective receptive field compared to simple convolution.

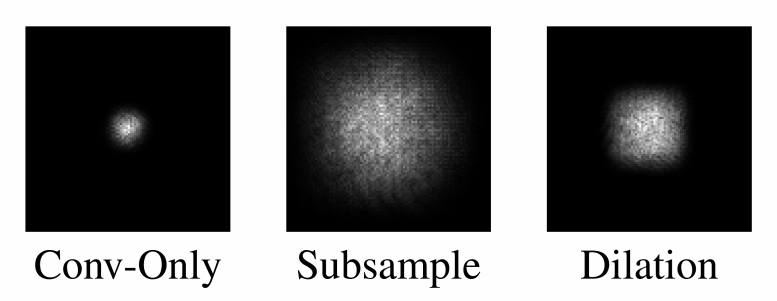

Figure 2.11. Effective Receptive Field visualization [30]

2.2.4 Pyramid Pooling

We have mentioned that one of the problems with segmentation is the presence of multiple instances of the same object at different scales.

The Spatial Pyramid Pooling module introduced in [17] partitions the image into divisions from finer to coarser levels and aggregates local features into them. Such a module allows to extract information from different scales. In fact, each pyramid level corresponds to a different scale.

While SPPNet introduced the concept of Pyramid Pooling for classification and object detection, it was PSPNet [49] to bring it to semantic segmentation. This network uses a ResNet18 [16] classification backbone (pre-trained on ImageNet) and modifies some of its convolutional layers by adding a dilation rate. The backbone generates an output feature map that has 1/8th of the resolution of the input image. This output feature map is then fed to the Pyramid Scene Parsing Pooling Module that extracts 4 different Pyramid scales from it. The coarsest scale is obtained through global average pooling (generates a feature map of resolution 1x1), while the other scales are obtained by dividing the input feature map into regions (pooling windows) and considering one value representative for the entire region (max or average). After the pooling operations, the 4 scales are fed to a pointwise convolutional layer with the purpose of reducing their number of channels to 1/4th of the original number. Then, the 4 maps are upscaled to the resolution of the input of the module and are concatenated. The input map of the module is also concatenated to the upsampled pyramid scales, obtaining double the number of channels, which is brought back to normal by another pointwise convolution.

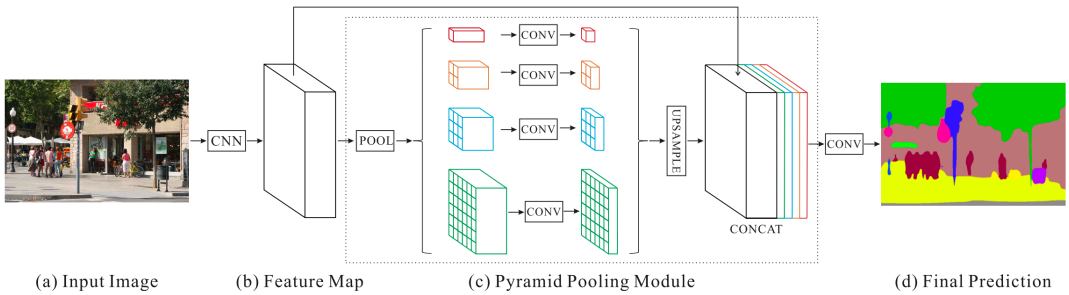

Figure 2.12. Pyramid Scene Parsing Pooling in PSPNet [49]

The Pyramid Scene Parsing Pooling Module produces a final feature map that contains both local and global context information and is also robust to multi-scale representations of the same object.

2.2.5 DeepLabV3 and DeepLabV3+

DeepLab [5] is a model that has gone through several iterations. In its first form, it introduced the use of Conditional Random Fields as a post-processing tool to improve the localization performance.

In its third iteration, DeepLabV3 [6] abandoned Conditional Random Fields and introduced a revised architecture that featured Dilated Convolutions and its own flavor of a Pyramid Pooling Module.

DeepLabV3 uses dilated convolutions to increase the receptive field of a convolutional kernel, without increasing the output stride of the layer (the ratio between the resolution of the input image and the resolution of a feature map). Increasing the output stride results in a reduction in resolution that is detrimental to the quality of the segmentation output. Employing dilated convolutions allows the network to keep high resolution feature maps, while still extracting the same amount of context information.
In order to tackle the problem of multi-scale objects, DeepLabV3 uses a modified Pyramid Pooling Module, called Atrous Spatial Pyramid Pooling module. This module uses 2 parallel branches whose outputs are then concatenated.

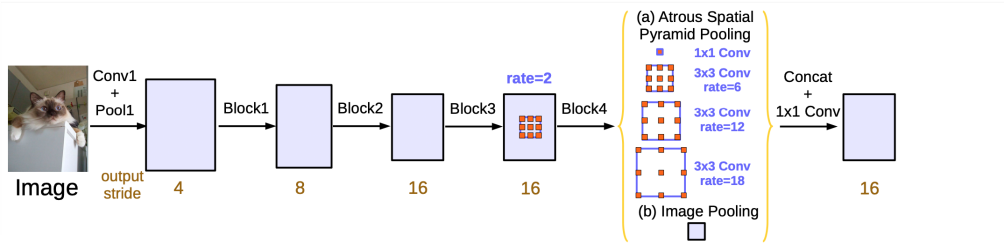

Figure 2.13. Scheme of the structure of DeepLabV3 [6]

The first branch concerns image-level features, while the second is the actual pyramid pooling step.
In the first branch, the last feature map of the network is taken, global average pooling is applied to it, then it is fed to a pointwise convolution an finally it is upsampled to its original resolution.
In the second branch, a pointwise convolution is applied to the last feature map of the network, then three 3x3 dilated convolutions, with 3 different dilation rates (varying on the output stride, ie. [6, 12, 18] with output stride = 16), are performed to the same map. The inputs of the 4 convolutions are then concatenated and fed to a pointwise convolution. Finally the outputs of the 2 branches are also concatenated and fed to a pointwise convolution.

DeepLabV3+ [7] extends DeepLabV3 by adding a simple, yet effective, decoder module to refine the segmentation output, especially around object boundaries.

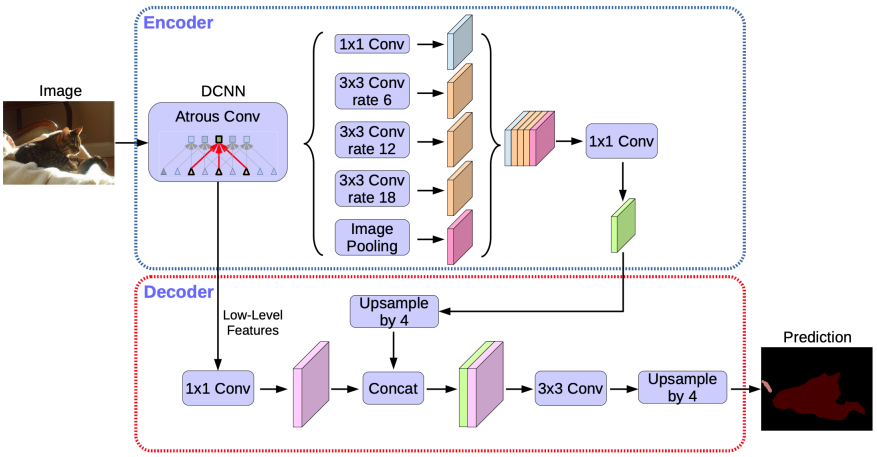

Figure 2.14. Scheme of the structure of DeepLabV3+ [7]

So this network becomes a combination of all the approaches described in this section: it is a fully convolutional network, it uses both dilated convolutions and a spatial pyramid pooling module, and, in its latest revision, it has an encoder-decoder structure.

Furthermore, DeepLabV3+ introduces Depthwise Separable Convolution as an alternative to traditional strided convolution in order to reduce the computation cost and the number of parameters. We will look more deeply into depthwise separable convolution in the following section.

2.3 Resource Optimization Approaches

In this section we will describe the main design-time optimizations that are integrated into semantic segmentation models to reduce their computational complexity without impacting the accuracy of the prediction.

2.3.1 Depthwise Separable Convolution

In standard convolution a convolution between an input of size $D_f \times D_f \times M$ and a filter of size $D_k \times D_k \times M$ is performed N times, where N is the number of filters, generating N output maps of size $D_g \times D_g$, so that the shape of the output is $D_g \times D_g \times N$.
So the number of multiplications required is: $D_k \times D_k \times M \times D_g \times D_g \times N$

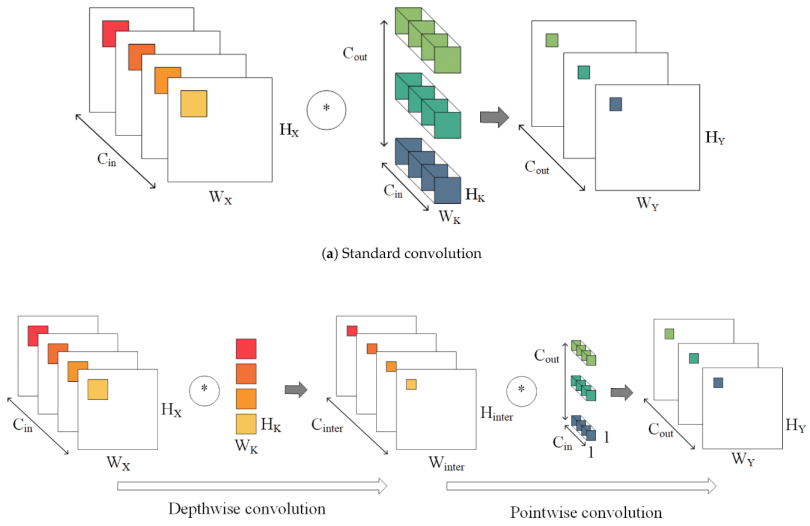

Figure 2.15. Depthwise Separable Convolution

With Depthwise Separable Convolution the standard convolution operation is divided into 2 steps.

The first step is Depthwise Convolution in which the size of the input feature map is $D_f \times D_f \times M$, and there is one filter of dimension $D_k \times D_k$ for each channel of the input map (M filters in total). Each filter is convolved over the corresponding channel of the feature map generating an output map with shape $D_g \times D_g$, so depthwise convolution yields an output map of shape $D_g \times D_g \times M$.

The second step is Pointwise Convolution. This operation is equivalent to performing the linear combination of the M channels of the input. More in detail, the input is the output of the depthwise convolution (of size $D_g \times D_g \times M$), there are N filters of size $1 \times 1 \times M$. The output is obtained by stacking the outputs of all the convolutions between

the input maps and the filters. The final output size is the same as standard convolution, $D_g \times D_g \times N$.

The number of multiplications required is given by the sum of the number of multiplications of the previous steps: $D_k \times D_k \times D_g \times D_g \times M + D_g \times D_g \times M \times N$. The ratio between the number of multiplications of depthwise separable convolution and traditional convolution is: $(D_k \times D_k + N)/(D_k \times D_k \times N)$, so as N increases, so does the difference in efficiency between the two operations.

2.3.2 Grouped Convolution

Grouping takes into account the channel dimension, which increases very rapidly as we go deeper in the network. In fact, the spatial dimensions still have effect, but as we go deeper, the channel dimension is the most concerning.

The idea is to divide the feature maps and the filters in g groups. The value of g is an hyperparameter that has to be chosen experimentally.

The size of the input feature map has shape $D_f \times D_f \times M$ and we have N filters, each of which has shape $D_k \times D_k \times M$. We split the M channels of the input feature map into g groups, each of which has M/g channels. We perform the same operation with the N filters splitting their channels into g groups. Then, when performing the convolution, for each one of the N filters, we will consider only one of its splits and convolve it on the corresponding group of input channels.

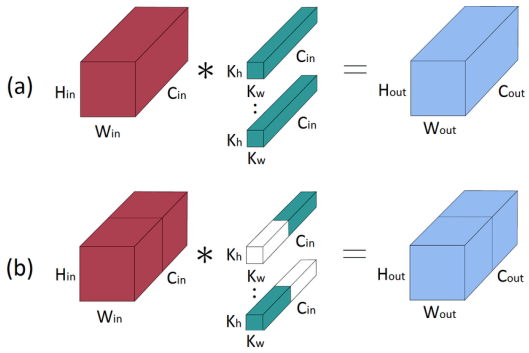

Figure 2.16. Standard vs Grouped Convolution

So, in the end, we perform g groups of convolutions between smaller inputs and filters. In particular, the N filters can be divided into g groups, each of which presenting N/g filters, and every group is convolved on the corresponding group of input channels. The g groups of output maps, each of which presenting N/g channels, are concatenated, resulting into an output of shape $D_g \times D_g \times N$.

The gain in efficiency is in the number of parameters used for the convolution, as the

total number of parameters is reduced by a factor of g.

2.3.3 Channel Shuffling

A Convolutional Neural Network that makes use of grouped convolution, consists of repeated building blocks with the same structure.
A limitation of models like this is that they take advantage of group convolution for 3x3 or bigger filters and do not consider 1x1 convolutions. In smaller networks (and especially in the domain of semantic segmentation) 1x1 convolutions are very common and require quite a large number of computations.

A way to make these 1x1 convolutions more efficient is to use 1x1 group convolutions instead. The problem with grouped convolution is that it prevents channels from mixing across groups, outputs from a certain group only relate to the inputs within the group. In order to overcome such side effect, a channel shuffling operation is introduced as a way of helping information flow across channels.
The shuffle operation consists in dividing the existing groups into several subgroups and

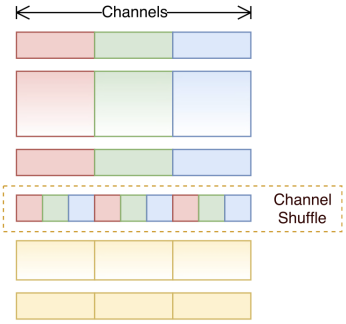

Figure 2.17. Channel Shuffling [47]

then feed each group in the next layer with different subgroups.
The ShuffleNet [47] introduces a building block for a network that takes advantage of this type of group convolution. However the idea behind channel shuffling can be generalized to any type of group convolution, in fact the disadvantage of group convolutions preventing the information flow across channels is well known and in most architectures is countered by adding a 1x1 convolution. With channel shuffling this operation can be avoided, reducing the number of parameters and computations.

Moreover, channel shuffling is differentiable, which means that it can be added into network structures without any modifications to the training procedure.

2.3.4 Early Downsampling

It is clear that processing large input frames is very computationally expensive, so the early stages of the network can be very intensive in terms of computations.

Moreover, the primary role of the initial network layers should be that of feature extraction and the preprocessing of the input data for the later stages of the architecture, rather than contributing directly to the prediction.

In the ENet [33] paper, the assumption is that visual information in the input image is highly spatially redundant and thus it can be compressed into a more efficient smaller representation without losing vital information.
ENet uses early downsampling as its first two blocks heavily reduce the input size.

The first block takes a 512x512 input image and uses a max pooling operation with a 2x2 window and no overlapping in parallel with a convolutional layer with 13 3x3 filters and stride 2, the output of the 2 operations are concatenated in a 256x256x16 feature map
The second block is a bottleneck block which carries out a further downsampling with a 2x2 convolution with stride 2 followed by a 3x3 and a 1x1 convolutions (which do not alter the size of the feature maps), in parallel with a max pooling operation, the outputs are then summed to obtain a 128x128x64 feature map.
We can observe how in the first 2 building blocks of the architecture the resolution is reduced to a quarter of the original image size.

What we can keep in mind from this is that performing the downsampling early in the network results in lower complexity of the layers that operate on the smaller inputs and overall the complexity of the network is going to be lower.

2.3.5 Small Size Decoder

Encoder-Decoder architectures can present a symmetric structure, but it is quite easy to understand how the jobs of the encoder and the decoder are vastly different.

In fact, while the encoder's job is to pre-process the input, extract features from it and capture context and scale information (as it happens in classification models), the decoder only has to scale up the feature maps produced by the encoder recovering fine detail information.

Following this intuition, in order to save resources, we can reduce the size of the decoder, resulting in an asymmetric network architecture that, in most cases can yield results that are comparable to its symmetric counterpart, keeping the computational cost down.

2.3.6 Factorization of Large Filter Convolutional layers

Total computational cost increases with the square of the kernel size.
The idea, introduced in [42], is to approximate the output of a convolutional layer with a large filter size using a multi-layer convolutional network with smaller filter size, thus with fewer parameters.

Factorizing a convolutional layer with a large kernel into multiple convolutional layers with smaller filter size is preferable for two reasons:

- Reducing the number of parameters : for example, by stacking three 3×3 convolutional layers correspond to the same effective receptive field of a 7×7 layer while reducing the number of parameters to almost half.
- The decision function is made more discriminative by incorporating multiple nonlinear rectification layers instead of a single one

2.3.7 Two-Branch Networks

We have seen how downsampling can lead to improvements in inference time, but causes a loss in spatial detail, which is crucial in semantic segmentation.

The idea behind two branch networks is to use one shallow branch to capture the spatial details keeping high resolution feature maps and a deeper branch that uses downsampling to efficiently extract context information and features useful for the classification output. A feature fusion operation is performed before the output layer.

Further details of the implementation of this approach are peculiar of the proposed architectures that adopt such approach.

2.4 Real-Time Semantic Segmentation Models

In this section we will explore some of the most popular real-time segmentation models, highlighting their utilization of the approaches described in the previous section.

2.4.1 DDRNet

The structure of DDRNet [19] consists of two main components: the Deep Dual Resolution Network and the Deep Aggregation Pyramid Pooling Module.

The Deep Dual Resolution Network is built by adding a high-resolution branch to a ResNet [16] backbone. This makes it a Dual-Branch architecture.
The high-resolution branch is appended to the convolutional stage in the backbone in which the resolution of the output feature maps is 1/8th of the input image.
Moreover, the input stem of the original ResNet [16] is modified by factorizing the 7x7 convolutional layers with two 3x3 layers.

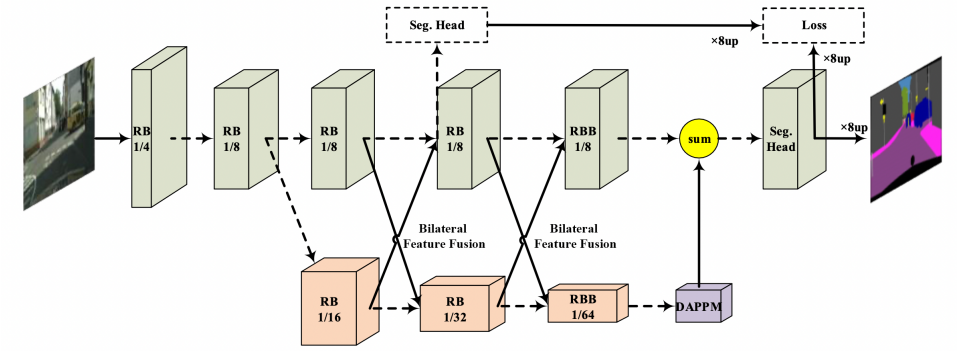

Figure 2.18. The DDRNet Model [19]

Two Bilateral Feature Fusion Blocks are added at different points of the network to exchange information between the two branches. They work by upsampling (bilinear upsampling) the feature maps from the low-resolution branch and then summing them with the feature maps of the high-resolution branch. The same is done after downsampling (strided convolution)the feature maps from the high-resolution branch.

The Deep Aggregation Pyramid Pooling Module is used to extract contextual information from low-resolution feature maps. Taking as input feature maps of 1/64th the resolution of the input image, large pooling windows with exponential strides are used to generate feature maps of 1/128th, 1/256th, 1/512th of the input image resolution. The input of the module and image-level information generated by global average pooling are also utilized. All these feature maps are upsampled and fed to 3x3 convolutional layers to fuse contextual information of different scales in a hierarchical residual way. The feature maps are then concatenated and fed to a pointwise convolution before summing them to

the input of the block.

DDRNet, in its smallest configuration, has 5.7 million trainable parameters. The possible configurations present different numbers of filters at each convolutional layer.

2.4.2 Fast-SCNN

Fast Segmentation Convolutional Neural Network (Fast-SCNN) [35] is based on the two-branch architecture and introduces the Learning to Downsample Module.
The building blocks of the network architecture are: the Learning to Downsample Module, a Global Feature Extractor and a Segmentation Head.

The Learning to Downsample Module consists of three layers, which are a standard convolution and two depthwise separable convolutional layers, all three have 3x3 convolutional kernels.

The Global Feature Extractor directly operates on the output of the Learning to Downsample module. This is supposed to save computational resources by sharing the downsampling performed in the Detail Branch, instead of downsampling in both branches. The GFE itself is composed by three Inverted Bottleneck Blocks borrowed from MobileNetV2 [39], which employ depthwise separable convolutions.

A Pyramid Pooling Module, built on the example of PSPNet [49], is added to the end of the Global Feature Extractor.

The Feature Fusion Module upsamples the output of the Global Feature Extractor to the same resolution of the output of the Learning to Downsample module, performs a depthwise separable convolution and then sums the feature maps together.

The Segmentation Head employs two depthwise separable convolutional layers and one pointwise convolution that brings the number of channels to the number of classes of the dataset.
In its only configuration, FastSCNN presents 1.12 million trainable parameters.

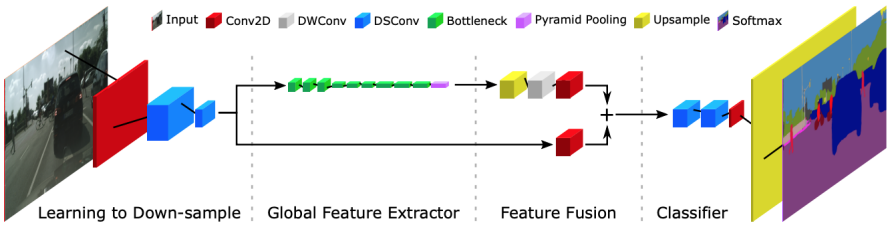

Figure 2.19. The FastSCNN Model [35]

2.4.3 BiSeNet

The Bilateral Segmentation Network (BiSeNet) [46] model is a dual branch architecture with a Spatial Path and a Context Path.

The purpose of the Spatial Path is to preserve the resolution of the input image to encode enough spatial information. The Spatial Path is composed by three layers, each of which contains a convolution with stride=2 followed by Batch Normalization and a ReLU activation layer. Therefore, this path produces an output feature map whose resolution is 1/8th of the input image resolution.

The Context Path is designed to provide a sufficient receptive field. It employs a lightweight model, like Xception [8], as a backbone. This model can downsample the input image to obtain a large receptive field. A global average pooling layer is added at the end of the lightweght model. Then, the upsampled output of the global average pooling layer is combined with the output of the lightweight model.

BiSeNet introduces an Attention Refinement Module to refine the features at each stage of the Context Path. The Attention Refinement Module employs global average pooling to capture global context and computes an attention vector to guide the feature learning.

At the end of the network, the features from the Spatial and the Context path are fused together to make a prediction.

In its smallest configuration, BiSeNet presents 5.8 million trainable parameters. The

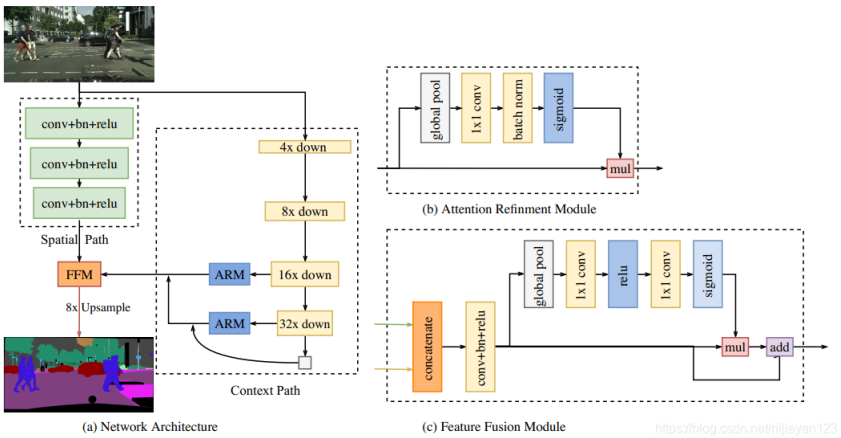

Figure 2.20. The BiSeNet Model [46]

configurations vary from each other according to the backbone used.

2.4.4 BiSeNetV2

BiSeNetV2 [45] is a dual-branch network that consists of a Detail Branch and a Semantic Branch, that are merged by the Bilateral Aggregation Layer.

The Detail Branch is responsible for the spatial details, therefore it presents a shallow

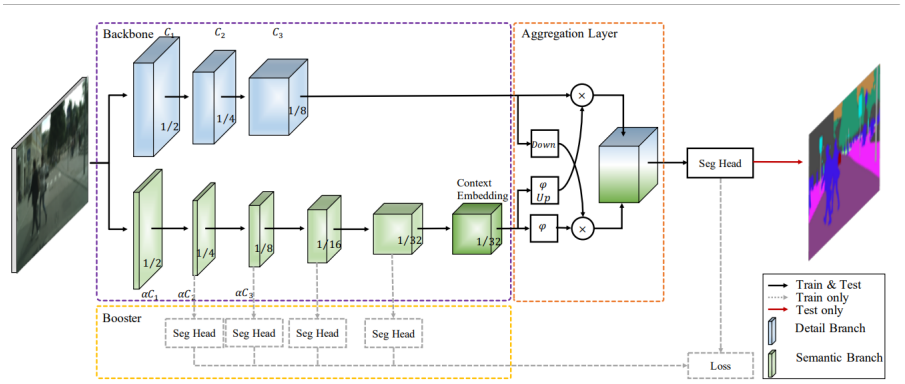

Figure 2.21. The BiSeNetV2 Model [45]

structure with wide channels: high resolution feature maps and low number of channels. On the contrary, the Semantic Branch is designed to capture high-level semantics (context information), which require a large receptive field.

The Detail Branch contains three stages, each of which consists of a convolutional layer, followed by batch normalization and a ReLU activation. This way, the branch produces feature maps that are 1/8th the resolution of the input image.

The Semantic Branch is more complex and it is composed by the following building blocks: Stem Block, Gather and Expansion Blocks, Context Embedding Block. The Stem Block is inspired to the Inceptionv4 [40] model and it is used to shrink the feature representation.

The Gather and Expansion Block is inspired to MobilenetV2's [39] Inverted Bottleneck Block and it consists of: a 3x3 convolutional layer, a 3x3 depthwise convolution, a 1x1 pointwise convolution. This block can be used with either stride=1 or stride=2. In the second case, two 3x3 depthwise convolutions are added and one 3x3 depthwise separable convolution is performed in the shortcut connection. Differently from the Inverted Bottleneck Block, the GE block presents one more 3x3 convolution.

The Context Embedding Block uses global average pooling and a residual connection to embed the global contextual information efficiently. This module can be thought of as a Pyramid Pooling module with only one scale.

The Bilateral Aggregation Layer is used to merge the complementary information from the two branches. It does so by upsampling the output feature map of the Semantic Branch to match the output of the Detail Branch. The feature fusion is then carried out

as the element-wise product of the feature maps.

The output of the model is generated by the Segmentation Head, which performs a 3x3 convolution, followed by batch normalization and a ReLU activation layer. I then uses a 1x1 convolution to bring the number of channels to the number of classes of the dataset, and finally uses bilinear upsampling to bring the prediction maps to the input image resolution.

BiSeNetV2 presents 3.4 million trainable parameters.

Chapter 3

Methodology

After having reviewed the literature on the state-of-the-art for semantic segmentation and real-time semantic segmentation, in this chapter we are going to introduce our experimental framework, the challenges we are setting up to face and how we plan on doing so.

3.1 Objective and Scope of the Experiments

In reviewing the literature we have analyzed many models that achieved real-time performance on powerful hardware. This means that if there is the need to perform real-time segmentation, we can do it as long as the hardware is powerful enough. However there are many instances in which we could benefit from real-time performance on more resource constrained devices. So the challenge we want to face is indeed bringing real-time performance on "IoT" grade devices. With the term IoT we generally indicate devices that consume little power and have limited memory and computational resources. In order to provide an example of such devices we used a Raspberry Pi 4. With the term real-time inference we indicate inference latencies that are lower than 0.0334, which results in 30 frames per second. Of course this is purely conventional, the number of frames per second required vary greatly with the task on which the model is deployed. Also in some cases we aim at the lowest possible inference speed, not because we need a high number of frames per second, but because we want to draw less power and latency is strictly correlated with power consumption. In an IoT use case, power draw is crucial, since devices are often battery-powered.

In order to face this challenge we need to create a framework on which our experiments are going to be run. Such framework consists in a Dataset, a Training Pipeline and a Baseline model. In building such framework we will be explaining many critical issues arising when trying to perform semantic segmentation on IoT devices.

After building the framework we will introduce the methodology of the experiments. With this thesis we want to propose a pipeline of optimizations that presents incremental steps with increasing levels of effort. We use the word "effort" with a specific meaning,

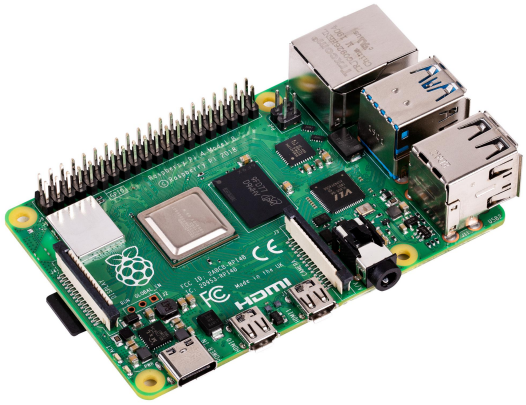

Figure 3.1. The Raspberry Pi 4 model B

we intend an effort in both time and difficulty of understanding and implementation. In particular saying that a particular optimization requires more effort than another might mean that it takes longer to implement or that it requires a higher level of expertise and understanding of the underlying network or all of the above. This pipeline of optimizations can be "packaged" as an automatic process that takes as input a model with some constraints and a set of parameters to optimize that are provided by a user. The amount of parameters and their type will vary according to the level of effort decided by the user.

In the following sections we will cover in detail every step mentioned above.

3.2 The Dataset

The dataset in analysis was downloaded from Kaggle at the following link [https://www.kaggle.com/datasets/christianvorhemus/industrial-quality-control-of-packages]. It was originally intended for defect detection a classification task. In order to make the dataset suitable for semantic segmentation we had to annotate each image and to do so we used the label-studio API [https://labelstud.io/]. The images of the dataset are RGB and represent a package that is being transported on a conveyor belt, each image contains only one package at a time. The only classes that we annotated are "package" and "background", which makes this a binary segmentation task.

The dataset contains 4oo images, 200 of which show the package from a side angle, while the rest are taken from a top down angle. Half of the images also contain packages with defects, but in our segmentation stage these defects appear on the ground truth mask only when they affect the actual shape of the package, otherwise they do not have any effect. The resolution of the images is 960x540 pixels and of course the ground truth maps have the same resolution, but only have one channel.

The reason why our choice fell on this particular dataset is that it represents a very

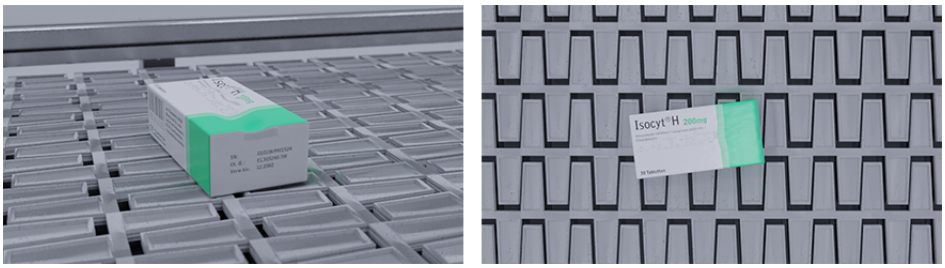

Figure 3.2. Samples from the Dataset

realistic use case for real-time semantic segmentation on an edge device. In the literature explored so far the most recurring dataset when speaking of real-time performance is either Cityscapes [9] or CamVid [3], which are both urban driving scene datasets. It is understandable how driving scene segmentation is a task that needs real-time performance in the context of autonomous driving: in order to make decisions the computer needs many images per second and the segmentation model needs to keep up, especially as the speed of the vehicle increases. However cars that feature this type of technology are equipped with very powerful hardware that is used to perform the inference. It would be realistic to force an edge-device, comparable to the raspberry pi used in this the to perform the same task. A dataset like Cityscapes is too complex with its 19 classes and cluttered images. State of the art real-time segmentation models already struggle with it, so it would be difficult to introduce further optimizations without dramatically loosing accuracy.
For this reason we needed to consider a dataset that represented a scenario in which the need to achieve real time performance on an IoT device is actually acceptable. Coming back to the dataset in analysis, it lends itself to many different applications in an industrial context. From simply monitoring packages moving on a conveyor belt, to supplying information about the shape, orientation and position of the package to a robotic arm that operates on the same conveyor belt.

We have just explained why we chose this dataset, but it is important to keep in mind its shortcomings in order to ponder the results that we will get in the experimental part of this thesis. As said before, the dataset is quite simple, by featuring only one item per image and only 2 classes. This is both a strength and a weakness, because if on one hand we need it to be simple in order to apply more aggressive optimisations, if it is too simple our experiments could be inconclusive.

So, in order to go further with our analysis with this dataset, we need to address these problems making some important clarifications. The first is that in this thesis we are not trying to find the best possible model for this dataset, but rather we explore the effects of different possible optimization strategies on the latency and accuracy of the model and we use this dataset as an example application of real-time semantic segmentation on edge devices. We are not interested in the actual performance numbers as much in the trends formed by this numbers and the difference among models resulting from different

optimizations. The second thing to note is that if we were to base our analysis on a more established dataset we would lose the reason why the inference is performed on an edge device, while with this dataset it is very easy to imagine different real world applications for a device like the Raspberry Pi used in the experiments with a camera attached to it, monitoring a conveyor belt or using the output of the semantic segmentation stage as the base for further actions.

3.3 The Framework

After the dataset has been chosen, since it was intended for image classification, it is necessary to create the target masks to train the model. To perform this operation we used the label-studio application, that allowed us to draw polygons corresponding to the packages portrayed in the image.

3.3.1 Data Augmentation

In order to create a strategy for our task it is important to understand how the various transformations work and what transformations are suited to each task.

Of course we need to keep in mind that for semantic segmentation the ground truth is an image itself, so when applying transformations to an input image, the effect of some of these transformations on the target mask needs to be accounted for. There are pixel-level transforms which concern color, contrast, brightness and so on, that would not make any sense if applied to the truth masks because the pixel information of the masks refer to the class to which that pixel belongs to, while the pixel information of the input image contains color channel (RGB) information.

The data augmentation API used in this thesis is Albumentations [4]. This library contains 2 types of transforms: pixel-level transforms and spatial-level transforms. Pixel-level transforms can be applied to any target and the transform will change only the input image and leave any other input target unchanged. Spatial-level transforms will change both the input image and the target mask (crop, pad, elastic transform, rotate, transpose, flip, resize ...).
In the state-of-the-art models reviewed in the previous chapter, we have observed the following data augmentation techniques:

- Random horizontal flip
- Random scale
- Random crop
- Normalization

It is important to point out that the data augmentation techniques to apply depend son the type of dataset that the model is going to be trained on. The transforms mentioned

before mostly refer to city driving scenes datasets, while for example a satellite images dataset could also benefit from random rotations and vertical flipping. For biomedical image analysis color transformations have shown to help deep networks to generalize better, along with operations like grid distortions and elastic transforms, since medical imaging is often dealing with non-rigid structures that have shape variations.

For our particular task, while we could have experimented with many different transforms, we decided to not introduce too many variables and so we have taken a more conservative approach with the following transforms:

- Random Crop (Center Crop in val dataset)
- Shift Scale Rotate
- RGB Shift
- Random Brightness and Contrast
- Normalization

The idea is to introduce a bit more variability to our dataset in which all the packages are of the same colour and approximately of the same size according to the angle the picture was taken at. With the shift scale and rotate we aimed at creating some variability regarding the position and the scale of the items in the image.

Figure 3.3. The RGBShift transformation

The last step in dataset preparation regards the handling of the dimensions of the input images and the different strategies that can be followed. [https://albumentations.ai/docs/examples/pytorch_semantic_segmentation/].
Differently from general convolutional neural networks, most semantic segmentation models are Fully Convolutional Networks, which means that they do not present any fully connected layers. This characteristic allows these networks to accept inputs of different sizes, eliminating the need to have fixed square input images. However PyTorch requires all images in a batch to have the same dimension. We decided to use the most used approach of training the model on a square random crop of size 512x512 pixels. For the inference instead of cropping the image, we resize it to the same 512x512 resolution, so the model will output a 512x512 predicted mask which we will resize back to the original

resolution of 960x540 before computing metrics or visualising the image. We did this by incorporating information about the original dimensions of the images in the test dataset class, so when loading images from the test dataset we get the resized image, the resized mask and a tuple containing the original height and width. We will later use this tuple to revert the resize operation.

3.3.2 Training Pipeline

The next step in the creation of the framework is setting up a training and testing pipeline. We will use this very pipeline to train and evaluate every model considered in the experimental part of the thesis. We used the PyTorch API.

For the actual training of the model we used the Adam optimiser. The loss function is Binary Cross Entropy Loss, which is the standard in semantic segmentation applications.

$$loss_{BCE} = -(log(p) + (1-y)log(1-p))$$

Additionally, we implemented a Polynomial Learning Rate Scheduler, which was a constant in the implementations of many of the models reviewed in the previous chapter, like Parsenet [27] or Deeplab [5]. Using this scheduler, the learning rate for the given epoch is given by the formula:

$$lr = lr_0 * (1 - \frac{i}{T_i})^{power}$$

As far as evaluation metrics go, we will be evaluating the model based on the mean intersection over union (IoU) score, which is more resilient to class imbalance in the input images.
Class imbalance is an inherent characteristic of semantic segmentation, because objects belonging to different classes have different sizes. In order to counter this phenomenon, we can use metrics that take it into account.
The Jaccard index or intersection over union score (IoU) is computed by dividing the area of the overlap between the predicted segmentation mask and the area of the ground truth mask, by the union of the two areas.

$$IoU = \frac{A \cap B}{A \cup B} = \frac{TP}{TP + FP + FN} = \frac{\sum(y_i \cdot \hat{y}_i)}{\sum(y_i + \hat{y}_i) - \sum(y_i \cdot \hat{y}_i)}$$

The perfect segmentation happens when the intersection and the union of the two masks coincide, in that case the IoU will be equal to 1. On the contrary, in the case in which there is no overlap between ground truth and prediction, the intersection will be equal to 0 and so will be the IoU score.
An IoU score is computed for each class and then a mean IoU score is used to evaluate the performance of the model as a whole. It is interesting, in cases with more than 2 classes, to observe the scores across all the classes, for better understanding of the model's behavior.
Of course the other metric that we use to evaluate a model in this thesis is the inference latency. As a matter of fact our analysis can be seen as an multi-objective optimization

problem in which we want to achieve the lowest possible latency and the highest possible IoU score.

Latency is actually a tricky subject, since it heavily depends on the runtime used to run the inference. In building this framework we decided to export the model in the ONNX exchange format and use the ONNX Runtime for the inference. The reason behind this choice is that inference latency obtained using ONNX Runtime was almost twice as fast compared to the integrated PyTorch runtime.
ONNX is an exchange format designed to allow interoperability between deep learning frameworks such as PyTorch, Tensorflow, etc... ONNX represents a model as a computational graph in which nodes represent operations and the edges are the pathways of the data. ONNX Runtime is an optimized inference engine that can run across different hardware platforms, implementing optimizations techniques specific for each hardware platform.
ONNX Runtime parses through the model to identify optimization opportunities. It abstracts custom accelerators and runtimes to maximise their benefits across an ONNX models. In particular it partitions the ONNX model graph into subgraphs that align with available custom accelerators and runtimes and when operators are not supported by available custom accelerators and runtimes, ONNX provides a default runtime that is used as a fallback execution, so that any model will run. The hardware architecture of the Raspberry Pi used in this thesis is ARM64 (Cortex-A72 64bit ARM 8 CPU).

3.3.3 Model Selection

Now that the training and testing framework is established it is time to use it to train some models on our dataset and get an understanding of what the performance is like on the Raspberry Pi. In this stage we will be training some of the models introduced in the previous chapter and then we will be selecting the baseline model to be used in the experimental part.
The architectures that we considered in this model selection phase are:

- DeepLabv3
- BiSeNetV2
- FastSCNN

The reasoning behind this selection is that DeepLabv3 represents the state-of-the-art of semantic segmentation. We specifically selected the version with the MobileNetV2 backbone. We then chose BiSeNetv2 [45] and FastSCNN [35] because they achieved a good balance between IoU score and inference latency on the Cityscapes dataset according to several surveys [32] [18]. In particular FastSCNN is one of the networks with the fastest inference, while BiSeNetV2 was able to achieve impressive IoU performance for a real-time model.

We trained all three networks for 400 epochs on the same training set and evaluated them measuring the IoU score on the same test set and measuring the average latency on each single image belonging to the test set.
We can clearly see how, on the Raspberry Pi, the inference latency of DeepLab and

Model Name	**latency (s)**	**frame rate**	**IoU score**
DeepLabV3 (mobilenet)	1.6006	0.6248	0.9788
FastSCNN	0,2200	4.5455	0.9776
BiSeNetV2	1.7427	0.5738	0.9823

BiSeNet is too high, while the IoU score of the 3 models is very similar. For this reason we decided to use FastSCNN as the baseline for our experiments. A consideration that we can make is that the ARM processor of the Pi does not feature any Deep Learning specific hardware acceleration, if we were to consider a different hardware platform our choice of baseline could be different.

The code regarding the implementation of this framework is available on the GitHub repository of this thesis. https://github.com/xolotl18/master_thesis

3.4 Introduction to the Experiments

In the previous sections we have built a framework that we can use to train and evaluate models and measure their inference latency on IoT hardware. Now we are going to introduce in more details the methods that we intend to apply to face the challenge that we introduced earlier.

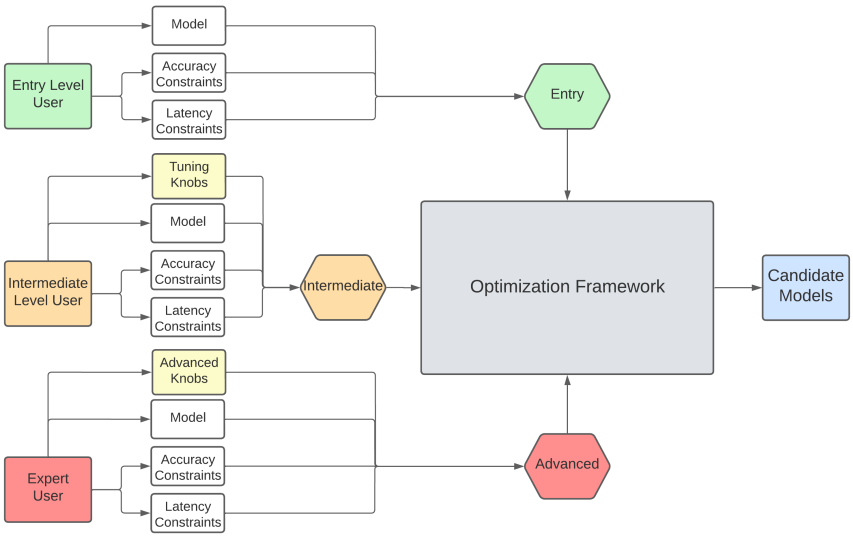

Figure 3.4. Scheme of the Automatic Optimization System

The series of experiments that we are going to introduce in the next sections can be summed up in an automatic system of optimizations. We have not implemented such system in this thesis, but we thought it would be an effective way of presenting the contribution of this work.
The proposed automatic system can be used by an end user as a tool that operates on an existing model and optimizes it by bringing its inference latency under a desired threshold.

We have already proposed this concept alongside the idea of a varying level of effort and we characterized effort in terms of both time available and user expertise . The way the effort level can be implemented in the system is to account for different levels of user expertise in the input stage, so that users that have different levels of understanding of the structure and inner working of the model will input different data into the system that will perform different optimizations. The idea behind this is that a more advanced user is going to be able to identify and feed to the system more tuning knobs, thus generating a bigger solution space and having better chances of reaching a satisfactory result. On the other hand, the time aspect of effort is taken care of in the internally, so that users that do not have as much time as others can exit the optimization process at different points in time.

The output of this system is a series of candidate models that feature different values of the tuning knobs identified by the users. Each of this model will present advantages and disadvantages and it will be up to the end user to decide which one to implement according to the requirements of the application at hand.

The concept of an automatic system that explores a series of possible optimizations in order to find the perfect combination of parameters for a specific task is very vague, but it is supposed to. We presented the idea of such system in order to introduce the experimental part of this work. In particular we set out to explore what we think are some of the optimizations that could be implemented under the hood of the system that so far we treated like a black box. In the following sections we will introduce what are the strategies that we can follow in order to bring the inference latency of semantic segmentation models to real-time values.

3.5 The Optimization Pipeline

The optimizations analyzed in this thesis can be divided in two branches: Input Data and Model Topology. As the name suggests Input Data Optimization focuses on modifying the input images, thus feeding the model smaller data that will result in faster inference. Network Topology Optimization operated directly on the structure of the model by changing the value of different tuning knobs, producing a smaller model that will occupy less memory space and run faster.

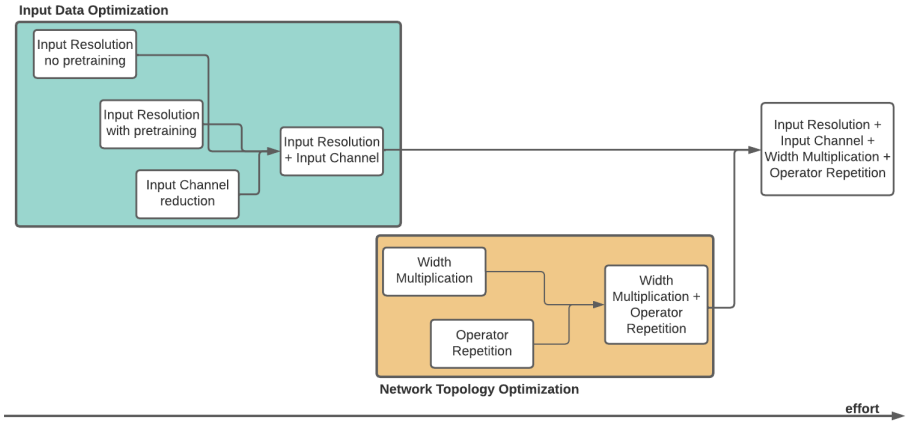

Figure 3.5. The Optimization Pipeline

We said that the pipeline of optimizations will follow an order defined by the level of effort that they require. The two main branches differ greatly in effort, as it is clear that operating on input data does not require any understanding of the structure of the model, while in order to identify and select a range of possible values for the tuning knob in the branch of Network Topology Optimization requires a much deeper knowledge of the baseline network and its functioning blocks.

In our analysis we will present the various stages of optimization belonging to each branch, following the order dictated by the effort required by each stage.

3.5.1 Input Data Optimization

Before we dive into the description of the actual experiments regarding Input Data Optimization, it is important to make a brief disclaimer on one crucial operation on the model structure that allowed us to proceed: we had to remove the Pyramid Pooling Module from the FastSCNN baseline in order to perform these experiments.

We talked about this module in the previous chapter 2. This module was introduced in the SPPNet paper [17], in order to integrate multi-scale context information in the

features extracted by the network. Such module works by taking as input feature maps of a certain size (h*w*c), using average or max pooling layers to change the height and width of these feature maps to desired Pyramid Scales. Then it performs point-wise convolutions on each scale in order to bring down the number of channels to 1/(number of pyramid scales), it upscales each scale to the original height and width, concatenates the different feature maps and the original input and finally perform a point-wise convolution to bring back the number of channels to the original number, so that the dimensions of the output of the pyramid pooling module are the same as those of its input.

In order to implement such module into a network using the PyTorch API and accept inputs of different sizes without altering the network for every different input size, we must implement the AdaptiveAveragePooling layer [https://pytorch.org/docs/stable/generated/torch.nn.AdaptiveAvgPool2d.html]. This module always returns feature maps of a certain size regardless of the dimension of the input feature maps. An issue manifested itself when trying to convert the torch model into the ONNX computational graph. Unfortunately ONNX does not support the AdaptiveAvgPool2D operator yet because its output dimension is not a function of the dimension of the input. So in order to keep this functional block in our baseline network we implemented it using traditional AvgPool2D layers with fixed window sizes. This means that if we need to change the height and the width of the input images we also need to change the size of the pooling windows every time.
So we decided to omit this module entirely from the network for two main reasons.

Firstly if we consider where in the network the module is placed, starting with a baseline input resolution of 512x512, the size of the feature maps that enter the module is 16x16x128 and the pyramid scales considered are of size: 1x1, 2x2, 4x4, 8x8. There are no issues with input images of resolutions up to 256x256, but if we want to consider a smaller input resolution, which is absolutely justifiable, we would have to sacrifice some of the pyramid scales. As a matter of fact, if we were to feed the model an image with resolution 128x128, the input of the pyramid pooling module would be of resolution 4x4, which is lower than 8x8 which is the resolution of one of the pyramid scales.

The second reason for removing this module for this part of the analysis regards its utility. Pyramid Pooling was introduced in SPPNet [17] to improve the model's ability to identify items in the image that belong to the same class but are portrayed in different scales at the same time. One glaring example is the presence of more than one car in a Cityscapes image, except that one car is further away than the other and so it looks smaller even if it is still a car. In particular the goal of Pyramid Pooling in SPPNet was to generate feature representations of objects which had a fixed size regardless to the dimension of the object in the input image, thus making the network more robust to size variations in objects belonging to the same class. The idea is that each pyramid scale would represent a different size of sub-images, so that an object that is smaller that another object belonging to its same category, will look the same when changing the pyramid scale used to represent it.

We question the useless of this module in our analysis because of the characteristics of the dataset. The packages portrayed in the images of our dataset are roughly of the

same size and there is always only one package per image, which makes things easier. Having said this, when operating on the resolution of the input data, we will not be including the Pyramid Pooling Module in our model. This is a necessary action for this specific baseline and should not be a problem with other network architectures. In fact we noticed that BiSeNetV2 [45] had no problem in the ONNX conversion because it does not feature pyramid pooling. Still we decided to keep FastSCNN as our baseline because the difference in inference latency with BiSeNetV2 was too large.

Resolution Reduction

For a classification model, changing the resolution of the input images can be a very tricky subject. On the contrary, for semantic segmentation it is quite a common practice at it is even mentioned in the FastSCNN paper [35] as a viable option to reduce the inference latency. The reason why it is possible is that most semantic segmentation models are Fully Convolutional Networks 2. This means that the absence of Fully Connected layers is what allows us to change the input resolution without issues in most networks. Of course there are limits to how far we can go, because feature maps undergo a series of reduction in height and width and increase in number of channels as they travel across the network, so the original resolution must be dividable by the various scaling factors. For example, in our baseline the input image gets reduced by a factor of 32 when it gets to the smallest feature map, so our input image must be dividable by 32 and also larger than 32x32 which would make the lowest candidate input resolution 64x64.

But how does the input resolution affect the inference latency of a model? To answer this question we need to understand the different types of data present in a convolutional neural network such as our baseline.
We can divide the components of the network into parameters and features. Our network being a Fully Convolutional Network means that its parameters are the filters of the convolutional layers (trainable parameters) and these are the part of the model that is saved to disk. The feature maps or activations are the other component. The first convolutional layer receives an image as input it outputs a number of feature maps equal to the number of convolutional filters of the layer. The height and width of the feature maps is determined by the stride and the kernel size of the convolutional layer. The feature map that was produced by the convolutional layer is then fed as input to the next convolutional layer.
Changing the resolution of the input we are going to change the size of all feature maps in the models. Comparing the same model with different input sizes we can clearly observe how this reduction propagates through the network.

In practice, smaller feature maps mean that the convolutional filters will slide fewer times across each image which will result in less time spent for each layer of the network.

Going back to the idea of effort that we introduced earlier, changing the input resolution is rather simple. It is a matter of resizing the input images during data preparation. However we can differentiate two different approaches that differ in the time spent to generate the candidate models.

Block	Input Size	Output Channels	Stride
Conv2D	512x512x3	32	2
DSConv	256x256x32	48	2
DSConv	128x128x48	64	2
bottleneck	64x64x64	64	2
bottleneck	32x32x96	96	2
bottleneck	16x16x128	128	1
Pyramid Pooling	16x16x128	128	-
Feature Fusion	16x16x128	128	-
DSConv	64x64x128	128	1
Conv2D	64x64x128	1	1

Table 3.1. Model with 512x512 Input Resolution

Block	Input Size	Output Channels	Stride
Conv2D	128x128x3	32	2
DSConv	64x64x32	48	2
DSConv	32x32x48	64	2
bottleneck	16x16x64	64	2
bottleneck	8x8x96	96	2
bottleneck	4x4x128	128	1
Pyramid Pooling	4x4x128	128	-
Feature Fusion	4x4x128	128	-
DSConv	16x16x128	128	1
Conv2D	16x16x128	1	1

Table 3.2. Model with 128x128 Input Resolution

The fastest approach is to feed inputs of different sizes to the network without retraining it. So we would take the model that was trained on 512x512 training images and run inference on smaller images. The network is going to output maps of the same size as the input and the inference is going to be faster because of what we have explained earlier. We are probably going to see some peculiar behavior from a network that is fed a smaller image after being trained on larger ones. If this approach were to produce acceptable results in terms of IoU, it would be the fastest possible way to achieve faster inference on an existing model.

If the no retraining approach were to produce outputs of insufficient quality, one possibility could be retraining the network every time we switch to a lower resolution. This means that we are going to spend time and computational resources every time we try a

different input resolution. However our baseline model is fairly compact and the training phase took about 30 minutes on our hardware. A possible thing that could be done in order to reduce complexity could be to freeze all the parameters involved in the feature extraction portion of the network and retrain only the segmentation head. This is a very compelling strategy when handling bigger models, like DeepLab, that has a very large number of parameters and requires a lot of graphic memory when training the entire network.

Reduction of the Number of Input Channels

The other optimization to perform on input data is reducing the number of channels of the input image. We borrow this idea from super resolution models [11].

The idea is to have an input image of size 512x512x1. In order to reduce the number of channels to 1, we need to convert the image to the Y'CbCr color space and then select only the Y' channel. Y'CbCr is used to separate out a luma signal Y' and CB and CR which are the blue-difference and red-difference chroma components. Y' contains the most information and is obtained from the RGB channels in the following way:

$$Y' = K_r * R' + K_g * G' + K_b * B'$$

where KR, KG, and KB are ordinarily derived from the definition of the corresponding RGB space, and required to satisfy $K_r + K_g + K_b$=1. Also the ' symbol indicates a gamma correction, thus R', G' and B' nominally range from 0 to 1, with 0 representing the minimum intensity and 1 the maximum so the resulting luma value Y' will have a nominal range from 0 to 1. The first layer of the network needs to be modified in order to account for the reduced number of input channels.

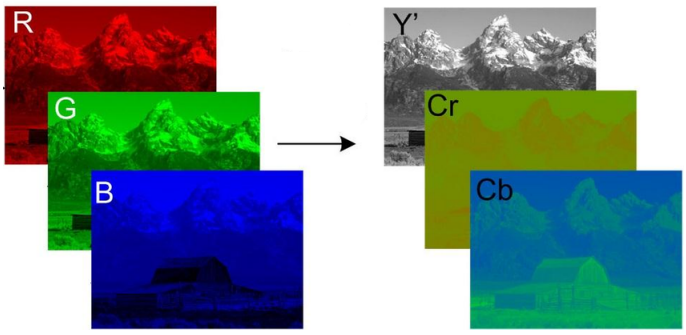

Figure 3.6. Conversion to Y'CbCr color space

Reducing the number of input channel should simplify the first convolutional layer, which will perform less operations. However the output of the first layer is still going to be of the same size as the output of the network with 3 input channels, because the number of channels of a feature map depends on the number of filters present in the convolutional

layer that generated such feature map. This means that the impact of this optimization will probably be much lower than reducing the height and the width of the input image.

3.5.2 Model Topology Optimization

Modifying the input data is a simple operation and does not require any modification of the model. However, if we only change the size of the input data, the number of parameters of the model won't alter the number of parameters of the network, which usually is pretty high. Neural Networks are often over-parametrized and in many cases it has been shown that big architectures can be greatly scaled down with negligible accuracy loss, sometimes even on the same application [22] [20].

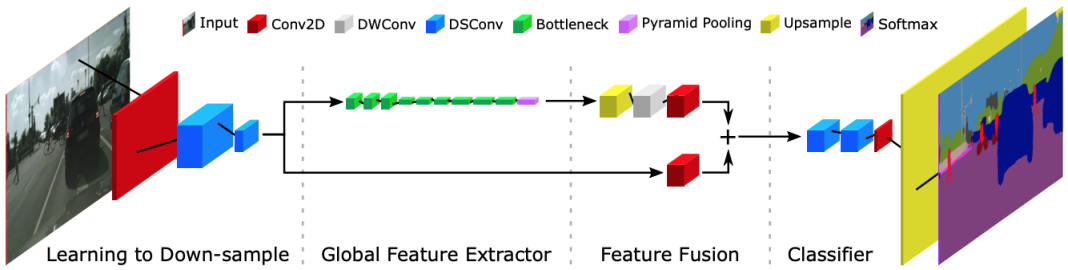

Figure 3.7. Illustration of the structure of our baseline model [35]

For this reason we introduce the second branch of optimizations that operate on the actual structure of the network. Going from the previous branch to this one is a big jump in terms of "effort". In fact, in order to modify the structure of a model, one must be able to understand the function of each building block and to identify which are the possible tuning knobs for such optimization. As far as the time aspect of the effort, the possible combinations of tuning knobs can be many and each combination results in a model that needs to be trained from scratch, which as we have already said, takes quite some time. Even if the optimizations belonging to this branch require more effort, they are a promising method of improving the latency while preserving the intersection over union score. The goal is, by removing some parts of the network and then retraining the model, to remove redundant information that is not crucial when producing the output. This is quite an intuitive strategy, which is validated by the fact that the dataset in our analysis is much simpler than datasets like CamVid [3], Cityscapes [9], COCO [25] and others, that are used to evaluate semantic segmentation models. Our packages dataset contains a lot less features, thus it should need less network parameters to extract them.

In the background chapter 2 we have spoken about optimizations that are supposed to help bring semantic segmentation models closer to real-time performance. The difference between the techniques introduced then and those described in this section is that in the background chapter we discussed about design-time optimizations, while here we are dealing with ways to simplify the structure of an existing model. This does not mean that

there cannot be any cross-over between the two categories, in fact in the MobileNet paper [20] an optimization proposed was to substitute standard convolutions with depthwise separable convolutions. However in our analysis we will focus on modifying the layers of the network instead of substituting them with variations, because we are starting from an already resource optimized model that already implements some of the techniques discussed in the background chapter.

There are many ways we can operate on the structure of the network and in this section we will talk about a few strategies that fit our baseline model FastSCNN. In the literature there are many times in which Model Simplification is mentioned as the most effective way to improve inference latency [10], but it is very hard to find actual examples of it. In this thesis we will explore some general strategies that can be applied to many semantic segmentation models. These strategies consist in identifying some hyper-parameters that regard certain aspects of the structure of the network and studying the impact of different values of these hyper-parameters on the performance of the model.

It is clear how this branch of optimization will have a varying impact depending on the structure of the network. Different architectures present different tuning knobs and bigger networks have more redundant parameters that can be removed without impacting the quality of their output.

Width Multiplication

The first type of Network Topology Optimization is Width Multiplication. We borrowed the name from the analogous operation performed in the MobileNet paper [20]. This optimization consists in reducing the number of output channels of each convolutional layer in the network by a multiplicative factor "a", which we call width multiplier.

In this stage we also included the hyper-parameter "t", which we called expansion rate. The expansion rate is the scalar that is multiplied to the number of channels of the input feature maps of the bottleneck block. The expansion rate was introduced in MobileNetV2 [39] with the Inverted Bottleneck Block, in which in the first convolution, instead of reducing the number of channels like in the ResNet paper [16], expands the number of channels of the input by a factor t and brings it back to the original number after the residual connection.

These two tuning knobs work on our baseline model, but they are also suitable for many other networks. In fact, width multiplication (a) can be performed on every network that presents a convolutional layer and those are very common since most state-of-the art models are Convolutional Neural Networks. The expansion rate (t), on the other hand, is specific to models that feature the Inverted Bottleneck Block. This makes it less widely applicable than the width multiplier, but still relevant because, thanks to the success of MobileNetV2, the Inverted Bottleneck Block is very popular in real-time semantic segmentation models [35] [45] and also in bigger networks that can be configured with MobileNet as a backbone [6].

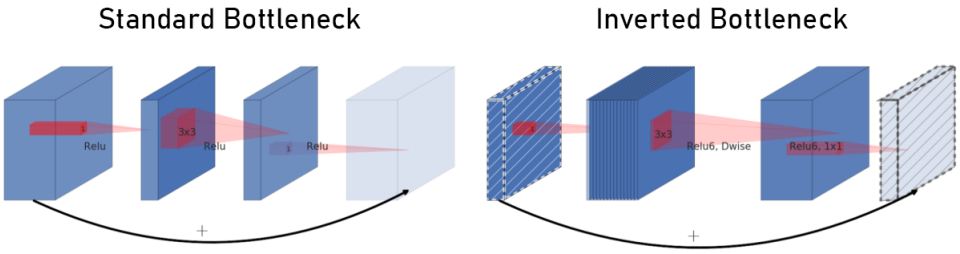

Figure 3.8. The Inverted Bottleneck Block [39]

The effect of "a" on the structure of the network concerns both the number of trainable parameters and the size of the feature maps.
In fact, reducing the number of output channels of a convolutional layer is done by removing some of its filters, therefore the number of parameter of that layer is reduced. Moreover, this operation will produce smaller feature maps (less wide), which will be the input for the next layer, which consequently will have smaller filters. The effect of "t" is similar, but it is limited to the Inverted Bottleneck Blocks, in which the number of channels is expanded by a factor t, a depth-wise separable convolution is performed and then a point wise convolution is used to bring the number of channels back to their original number.

Operator Repetition

In the Operator Repetition phase we identify the layers or modules in the network that are repeated multiple times or that can be removed safely without affecting the functionality of the network.

In order to identify possible tuning knobs fro this phase, it is necessary to have a good understanding of the structure of the baseline and the function of each layer. Some blocks might be repeated more times, but before removing some of them it is necessary to understand the consequences of their removal. In the case of the baseline, there are 3 bottleneck operations that present 3 Inverted Bottleneck Blocks each, in the Context Branch of the network. So there are 9 bottleneck blocks in sequence. One could think that the bottleneck block is repeated 9 times and choose to choose this number as an hyperparameter for the operator repetition stage. However, looking closer, we understand that there are 3 groups of bottleneck blocks because each group performs a reduction of the height and width dimensions of the network with one convolutional layer with stride=2 and kernel size 3x3. This means that if the wrong bottleneck block was removed, the output resolution would not match the input's and it would not make sense. So the right approach is to select the number of blocks inside each group as a tuning knob (r), where the baseline is r=3.

Sometimes a layer or a functional block can be removed safely, but this does not always mean that it should be. In the case of our baseline, the Pyramid Pooling Module is placed at the end of the Context Branch and the dimensions of its input are the same as those of its output. If we were to remove it, the size of the output of the network would stay the same. In this case it is necessary to decide whether the purpose of the block is too important for it to be removed. As we had to explain previously, due to its incompatibility with the ONNX export, the purpose of the Pyramid Pooling Block is not relevant on our dataset, so the block can actually be removed without problems. By choosing pp = [0, 1] we can analyze if our hypotheses are correct and let the results decide if pyramid pooling improves the quality of the output or not.

The tuning knobs in this stage are very specific to our baseline, because different models will present different building blocks. The ideas behind them, however, are widely relevant. In fact, neural networks are repetitive by design and all models contain redundant layers. This optimization stage consists in identifying elements that can be removed and understand if doing so has any effect on the quality of the output or on the inference latency on a specific dataset. The goal of this stage is to reduce the baseline to a structure that is essential for the task at hand.
Comparing the effect on the structure of the network of these tuning knobs to "a" and "t" from the previous stage, we can make a few considerations. In the case of "r", there is no effect on the feature maps flowing through the layers. By changing the value of this hyper-parameter we simply remove a set of layers from the network, therefore the total number of parameters is reduced by the amount of parameters that made up those layers, and we can say the same for "pp".
It is very easy to see how a bigger model would impacted in a much more significant way, because the number of superfluous parameters would be much bigger.

3.6 The Analysis

Summing up, the experiments will following the order:

- Resolution Reduction with no re-training
- Resolution Reduction with re-training
- Input Channel Reduction
- Resolution Reduction and Input Channel Reduction
- Width Multiplication
- Operator Repetition
- Width Multiplication and Operator Repetition
- Input Data Optimization and Model Topology Optimization

Each of the experimental stages consists in a series of models that we will evaluate on the same test dataset, running inference on a Raspberry Pi 4. For each stage we will visualize the summary for each model, which contains information about the total number of parameters, the input size, the size of the forward pass and the number of multiplication-accumulation operations. Then we record the IoU score and the mean latency of each model and represent it in a table and in a Pareto frontier plot. We used Pareto frontiers because the choice of a model is a multi-objective optimization problem where the objectives are inference latency and intersection over union score. Therefore visualizing the performance of each model as a point in a bi-dimensional space gives us a clear understanding of the effect of the optimizations that we are trying to study.

So with our experiments we are actually exploring a solution space in which every combination of optimizations makes up a model and we are trying to identify the best models in terms of accuracy-latency trade-off. However, in analyzing the experiments, we won't be obsessing over the performance numbers, because we are not actually deploying a model on this task in the real world. We are interested in studying each optimization and its effect on the behavior of the model and we want to deduce some general guidelines about our pipeline.

Throughout the course of this chapter we have mentioned the applicability of the different optimizations to baselines different from ours. Our analysis would have little to no relevance if the various optimizations were usable for only one specific model. In order to rest this argument, we will propose a set of optimizations for a different model: BiSeNetV2.

As we said before, when dealing with Fully Convolutional Networks, it is always possible to change the input resolution, so that strategy can be applied to BiSeNet without issues. And also the reduction of the input channels can be performed easily, the only thing to change is the number of input channels of the first layer of the network, which

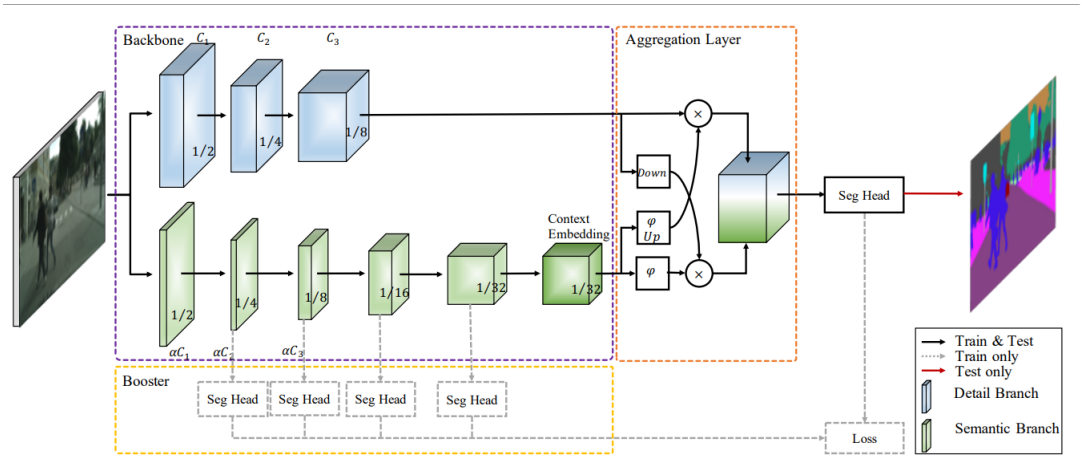

Figure 3.9. Illustration of the structure of BiSeNetV2 [45]

is usually a parameter.

For the Model Topology Optimization, we need to dig deeper into the structure of the network to identify the proper tuning knobs. BiSeNetV2 has almost three times the number of the parameters of FastSCNN, so these optimizations will be more powerful. The width multiplication hyper-parameter "a" can be used for this network as well. There are many convolutional layers, so this optimization is applicable to this model. The paper actually suggests changing the number of channels of the layers in the Segmentation Head as a way of reducing its complexity. The tuning knob "t" is also applicable in this model, since it feature its own version of the Inverted Bottleneck Block called the Gather and Expansion block. This block is used many more times than the bottleneck block in FastSCNN, so the value of "t" would likely have a bigger impact on the performance of this network.

For Operator Repetition, we can see how there are some GE blocks that are repeated more than one time in the Semantic Branch, which can be easily removed without affecting the output and the same thing can be said for some of the Conv2D layers in the Detail Branch. BiSeNetV2 features its own interpretation of the Pyramid Pooling Block as the Context Embedding Block, which can be removed, but it is probably not worth doing so, because it only presents one pyramid scale and one point-wise convolution, so its impact on performance is really minimal.

Chapter 4

Experimental Results

4.1 Experimental Setup

For each model presented in this analysis we used the following procedure. Each model was trained for 400 epochs on the same train set and then tested on the same test set. The actual train val test split is 75% 10% 15% respectively.
The best epoch is selected according to the value of the loss computed on the validation dataset.

The IoU and latency values are obtained by running the model in ONNX format on the test dataset using ONNX runtime on a Raspberry Pi model 4 (4GB of RAM). Specifically the inference on the whole test dataset is performed 10 times, the latency is measured on each single image, then the final latency value for the model is obtained by computing the mean across all values. The IoU score is computed only on the first of the 10 runs. The reason we ran the inference for multiple runs is to decrease variability from one experiment to the other when measuring latency.

In order to visualize the result data, we will use tables with the values of inference latency, frame-rate (computed as the inverse of the latency) and IoU score. We will also include tables that describe the changes in terms of number of parameters, size of the forward/backward pass (which contains information about the system memory occupied by both the feature maps and the parameters), number of Multiplication-Accumulation operations (MACs), from one model to the other. We will also display Pareto Frontier plots in which all the models are represented as points in a bi-dimensional space (IoU, frame-rate) and a frontier consisting of the models with either higher IoU or frame-rate is displayed. The choice of using the frame-rate instead of the latency for the graph is simply aesthetic, this way for both quantities in the graph the higher value is better and this affects the shape of the ideal frontier.

4.2 The Baseline

Model Description The baseline model achieves an acceptable latency of 0.22 seconds on the raspberry pi, such latency allows it to perform the segmentation task smoothly but it is far from the real time goal of 30 fps or 0.0334 seconds.

For this reason in the following stages we will apply different strategies with increasing effort level to bring the performance closer to real time trying to keep the intersection over union score as close as possible to that of the baseline.

Block	**Output chans**	**Stride**	**Repetitions**	**Exp rate**
Conv2d	32	2	1	-
DSConv	48	2	1	-
DSConv	64	2	1	-
bottleneck	64	2	3	6
bottleneck	96	2	3	6
bottleneck	128	1	3	6
Pyramid Pooling Module	128	-	1	-
Feature Fusion Module	128	-	-	-
DSConv	128	1	2	-
Conv2d	num_classes (1)	1	1	-

Table 4.1. Model Structure

model	**# params**	**input (MB)**	**fwd/bwd pass (MB)**	**MACs (M)**
FastSCNN	1,122,673	3,15	209,88	830,67

Table 4.2. Model Summary

model	**mean IoU**	**inference latency (s)**	**frame rate**
FastSCNN (baseline)	0.9776	0.22000	4.5454

Table 4.3. Model Performance

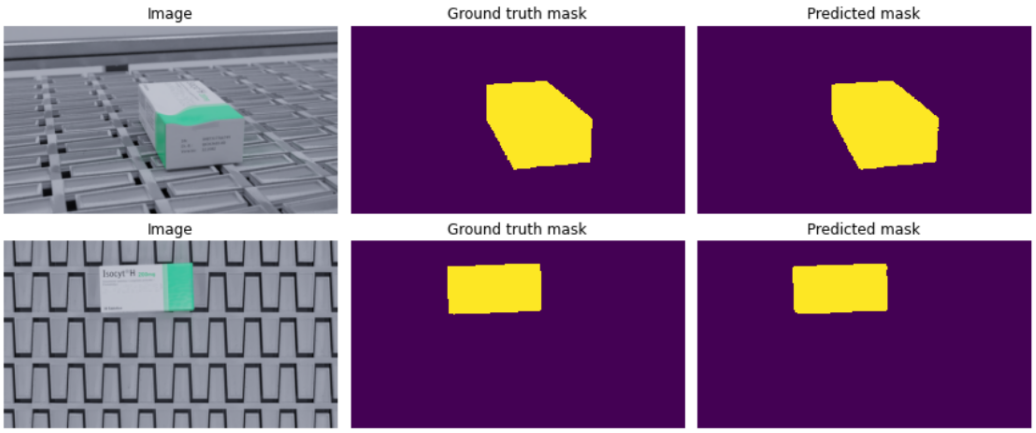

Figure 4.1. Output of the Baseline Model

4.3 Input Data Optimization

4.3.1 Stage 1 - Input Resolution Reduction without retraining

Block	Output chans	Stride	Repetitions	Exp rate
Conv2d	32	2	1	-
DSConv	48	2	1	-
DSConv	64	2	1	-
bottleneck	64	2	3	6
bottleneck	96	2	3	6
bottleneck	128	1	3	6
Pyramid Pooling Module	128	-	0	-
Feature Fusion Module	128	-	-	-
DSConv	128	1	2	-
Conv2d	num_classes (1)	1	1	-

Table 4.4. Model Structure

Details on the Experiment For this experiment we have selected 7 possible input resolutions, starting from the full resolution input that we have observed in the baseline.

As it was previously stated the images in our dataset have a resolution of 960x540 which is cropped to 512x512 patches for training and resized to 512x512 patches for testing. From this full resolution we started scaling it down to half (256x256) and then proceeded in smaller increments. We started with a bigger reduction in the beginning and then reduced the jump from one resolution to the next after that. The idea is to cover an exhaustive range of resolutions and push the reduction to its limit, we do so exploring

more resolutions closer to the bottom end, where we will see the biggest drop in accuracy and the biggest jump in frame-rate. Ideally we could have explored all 14 resolutions from 512x512 to 96x96, but it would have required too much training time, especially considering the upcoming optimization stages. The most important thing to keep in mind when choosing the input resolution is the scaling that the feature maps will endure passing across the different layers in the network. In the case of FastSCNN the size of the smallest feature map is 1/32th of the original input resolution, this means that an input resolution of 96x96 will result in a smallest feature map of size 3x3. This number may appear too small, but the idea is that the number of feature maps at that point in the network will be 128, so having a small h and w dimension should be compensated by the number of channels and the overall amount of data should still be enough to perform segmentation on a less demanding dataset like the one in analysis. As it was stated and justified in the Methodology chapter, for the input data optimization stage we are not going to include the Pyramid Pooling Module in the structure of the model [Table4.4]

input res	**# params**	**input (MB)**	**fwd/bwd pass (MB)**	**MACs (M)**
512x512x3	1,122,673	3.15	200,97	826,65
256x256x3	1,122,673	0,79	50,24	206,68
224x224x3	1,122,673	0,6	38,47	158,24
192x192x3	1,122,673	0,44	28,26	116,27
160x160x3	1,122,673	0,31	19,63	80,75
128x128x3	1,122,673	0,2	12,56	51,69
96x96x3	1,122,673	0,11	7,07	29,08

Table 4.5. Model Summaries

In this stage, we explore the optimization that requires the lowest effort. The only thing that we change from the baseline is the data transformation step for the test dataset class, in which we resize the input images to the desired resolution. The effort is low because the training is only performed once, so there is going to be a big reduction in the time required to run this optimization step, compared to the next step in which we are going to train the model from scratch for each input resolution.

Before looking at the performance result, we can see from the model summaries [Table 4.5] that the number of parameters stays the same, while the size of the forward/backward pass decreases with the input size and so does the number of MACs. This is because we are not touching the actual structure of the network, but by operating on the size of the input samples, we are affecting the size of the feature maps at each layer of the model. We expect the performance to degrade greatly as the resolution decreases because the model used in these experiments is always the same, which has been trained on 512x512 training samples.

input resolution	inference latency (s)	frame rate	mean IoU
512x512x3	0.2169	4.6104	0.9792
256x256x3	0.0611	16.3667	0.5440
224x224x3	0.0467	21.4133	0.4361
192x192x3	0.0355	28.1690	0.2583
160x160x3	0.0265	37.7358	0.2137
128x128x3	0.0164	60.9756	0.2208
96x96x3	0.0117	85.4701	0.1602

Table 4.6. Performance Data

Comments on the Results The first thing we can observe is the fact that the inference latency actually decreases far past what we would consider real-time performance (30 fps or 0.0334 seconds).

So we can see that changing the input resolution is promising in terms of inference speed. The clear issue is that the drop in intersection over union score is very critical. By decreasing the input resolution from the baseline 512x512 to 256x256 the IoU score goes from 0.9792 (it is higher than the baseline because we did not include the pyramid pooling module) to 0.5440. At this point the IoU score is already too low and the model is not usable, while the inference latency is still too high at 0,0611 seconds to be considered real-time. We get real-time performance only at an input resolution of 160x160 and at this point the IoU score is only 0,2137, which is far from acceptable and in practical terms it means that the output of the model has almost nothing to do with the target mask.

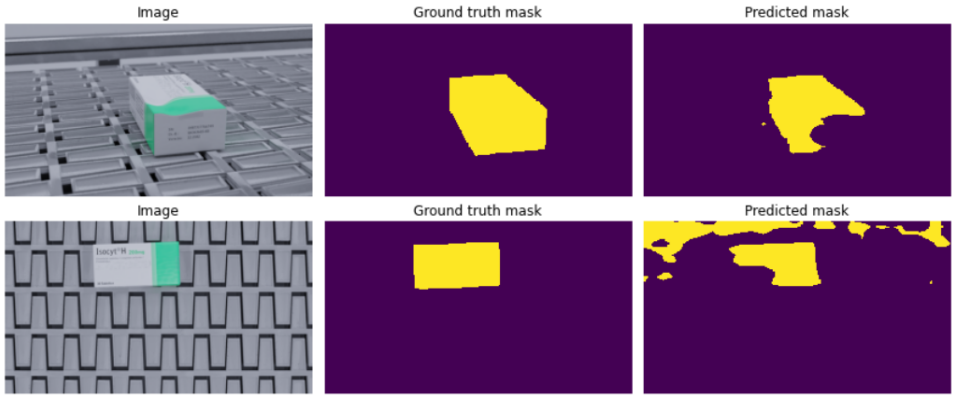

Figure 4.2. Output with 256x256 input without retraining

These results allow us to understand the potential of changing the input resolution in terms of latency gains, but also bring to the conclusion that the accuracy of the output

is not sufficient and opens the door to the second stage of our analysis which is changing the input resolution and re-training the model at every change.

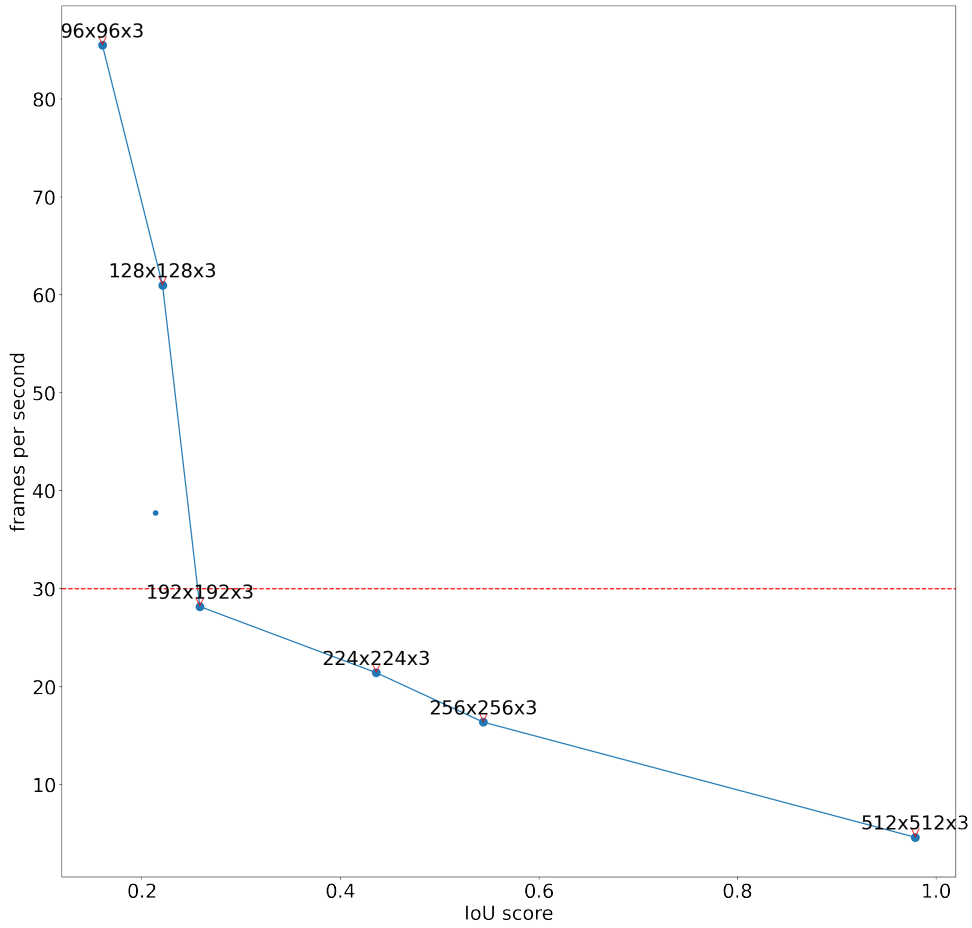

Figure 4.3. Pareto Frontier Plot - Stage 1

4.3.2 Stage 2 - Input Resolution Reduction with retraining

Details on the Experiment From the previous stage we have learned that simply changing the input resolution keeping the same model trained on bigger samples leads to unacceptable output quality. In this experiment we will have all the implications from the previous stage [Tables 4.4 4.5], but we expect an increase in the quality of the predictions thanks to the re-training of the network.

In literature, for bigger models, there are examples in which in order to use the same model with input sizes different from the standard input, the only part of the network that is retrained is the classification head, while the rest of the model, which will act as a feature extractor, is "frozen". In the case of FastSCNN we are going to retrain the entire network, since the number of parameters allows us to do so and we will probably have some gains in terms of performance. However it is something to keep in mind if the approach proposed in this thesis is going to be used starting from a baseline with more parameters.

In the FastSCNN paper, in the last section, it is proposed an input resolution reduction step in which ½ and ¼ of the original input size are analyzed and the obtained latency gains and IoU drop are rather interesting, so we are confident that the results obtained in this experimental stage will be comparable to those obtained in the paper.

input resolution	inference latency (s)	frame rate	mean IoU
512x512x3	0.2177	4.5934	0.9788
256x256x3	0.0639	15.6495	0.9729
224x224x3	0.0467	21.4133	0.9708
192x192x3	0.0350	28.5714	0.9723
160x160x3	0.0260	38.4615	0.9641
128x128x3	0.0165	60.6061	0.9564
96x96x3	0.0119	84.0336	0.9393

Table 4.7. Performance Data

Comments on the Results The situation dramatically changes when re-training is performed. Now the lowest IoU score is 0.9393 for the input resolution of 96x96. At 192x192 we get very close to real-time performance with an inference latency of 0.0350 (28.5714 fps), while achieving 0.9723 IoU score which is still very close to the baseline of 0.9792. At the next resolution of 160x160 we achieved way past our real time goal with an inference latency of 0.0260, while the IoU score drops to 0.9641, which is lower than the baseline but still way higher than the 0.2137 achieved without retraining the model

in stage 1 [4.3.1].

These results bring us to the first consideration of our analysis: retraining is necessary when changing input resolutions. Another important thing that we can observe is that, except for the resolution 224x224, IoU scores always decrease when jumping to a lower resolution and also inference latency decreases (frames per second increase).

For the dataset in question we observe quite respectable IoU even at the lowest resolution of 96x96. This might not be always the case and the only way of knowing if a particular score actually corresponds to acceptable output quality is to visualize a few predicted masks alongside the ground truth and visually verify if the numbers reflect the actual output quality.

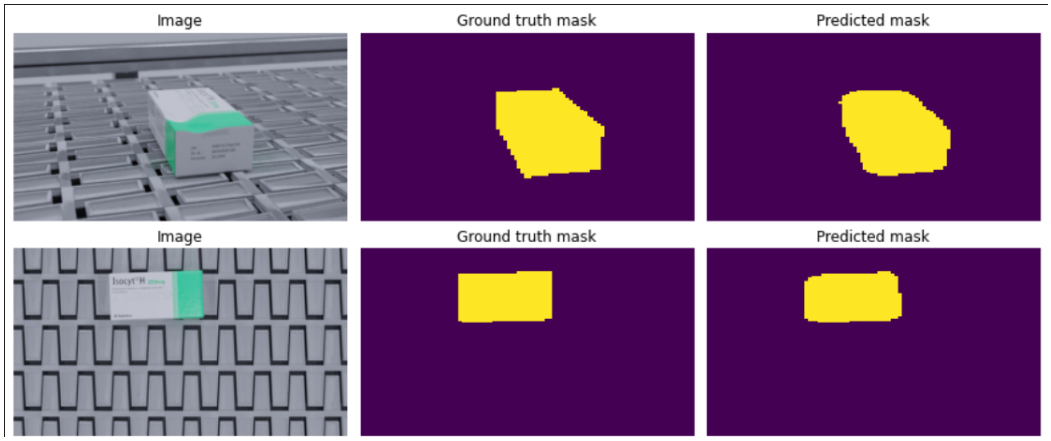

Figure 4.4. Output with 96x96 input with retraining

In conclusion, retraining a network or part of it when jumping from an input resolution to a lower one is a necessity. While keeping the same model would mean saving the training time and the hardware resources, the output would simply be unusable. A way to make this stage less time and computationally expensive would be to reduce the number of "candidate resolutions" that make up the solution space, this way we run the risk of not finding the ideal input resolution, but we can still achieve real-time performance.

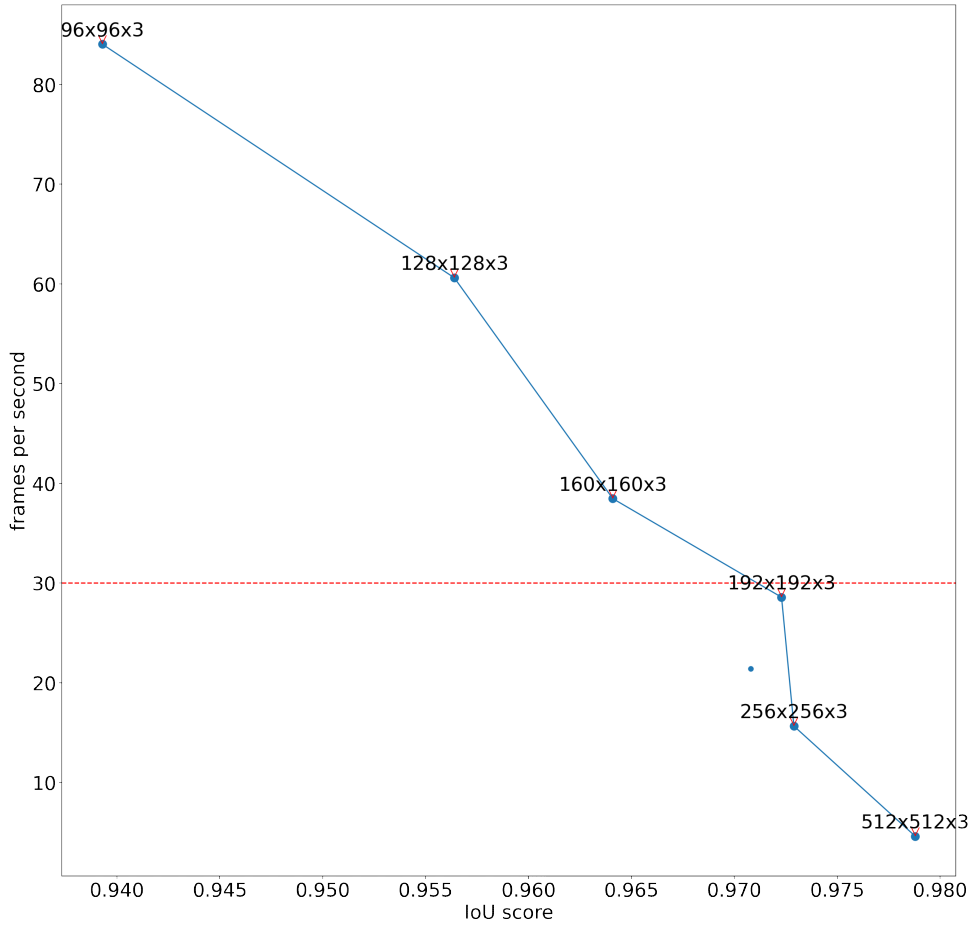

Figure 4.5. Pareto Frontier Plot - Stage 2

4.3.3 Stage 3 - Input Channels Reduction

Details on the Experiment In this stage of the experiment we first compare the baseline model whose input images have dimension 512x512x3 with a model presenting the same inner structure [Table 4.4], but that accepts an input of dimension 512x512x1. This is done by converting the original image to the YCbCr color space and utilizing only the Y channel, as was explained in the Methodology chapter [3].

After analyzing the difference with the baseline, we go on and inspect the differences with all the other resolutions from the previous steps in order to assess the difference in latency and accuracy with this new strategy.

From the model summaries we can observe that in fact the number of MACs is consistently lower in the case of the one-dimensional input, but not by much, of course this same behavior is reflected in the forward/backward pass size. As previously stated the number of parameters does not change because we are still not touching the internal structure of

input res	# params	input (MB)	fwd/bwd pass (MB)	MACs (M)
512x512x1	1,122,097	1,05	200,97	788,9
256x256x1	1,122,097	0,26	50,24	197,24
224x224x1	1,122,097	0,2	38,47	151,02
192x192x1	1,122,097	0,15	28,26	110,96
160x160x1	1,122,097	0,1	19,63	77,06
128x128x1	1,122,097	0,07	12,56	49,33
96x96x3	1,122,097	0,04	7,07	27,76

Table 4.8. Model Summaries

the network.

Comments on the Results In the results of this stage we can see the same trends that we experienced in the previous stage. When jumping from one resolution to a lower one, the inference latency always decreases and so does the IoU score.

input resolution	inference latency (s)	frame rate	mean IoU
512x512x1	0.2068	4.8356	0.9715
256x256x1	0.0553	18.0832	0.9542
224x224x1	0.0435	22.9885	0.9458
192x192x1	0.0343	29.1545	0.9335
160x160x1	0.0249	40.1606	0.9227
128x128x1	0.0158	63.2911	0.9219
96x96x1	0.0116	86.2069	0.8974

Table 4.9. Performance Data

Comparing the result of the baseline to the 512x512x1 input size we can observe that the latter presents indeed faster inference, but if we consider the IoU score it achieves 0.9715 which is lower than the score of 0.9728 achieved by the model with 192x192x3 input size. This is a trend that is constant across the other dimensions. While there always is a decrease in inference latency going from three channels to one input channel, the corresponding drop in intersection over union score is much bigger and evaluating a trade-off between accuracy and latency we can observe that changing the height and the width of the input images is always a better alternative.

As a matter of fact, analyzing the Pareto frontier with the models from the previous stage and the models form the current stage, we can observe that the only model belonging to the frontier with one channel is 96x96x1 because it achieved the lowest latency, but the

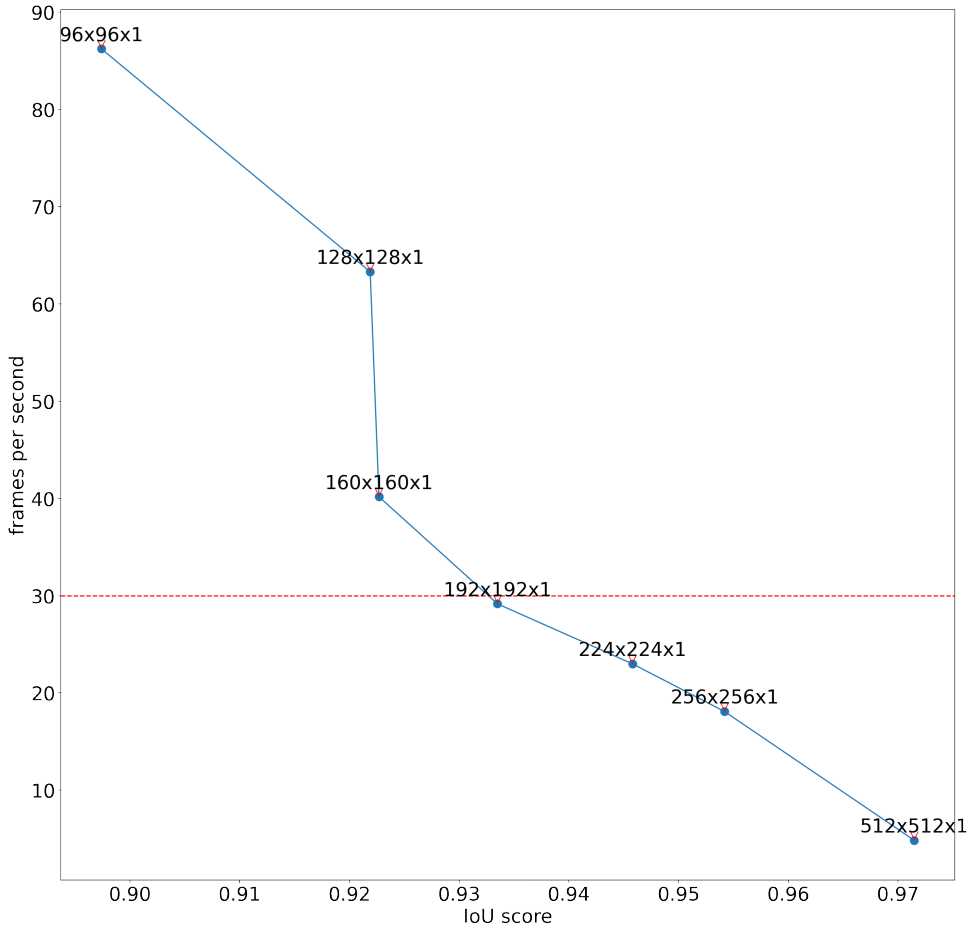

Figure 4.6. Pareto Frontier Plot - Stage 3

rest of them all present 3 input channels, because the slight decrease in latency of the 1-channel inputs is always countered by a bigger drop in intersection over union score compared to its 3 channel counterpart. So we can observe that 96x96x3 yields much lower latency compared to 128x128x1, while also resulting in a higher IoU score, it is also more accurate than the model with input size 160x160x1 which is more than twice as slow.

So from these results we can derive the next consideration of our analysis: decreasing the number of input channels damages the accuracy of the model more than it decreases its inference latency. For this reason I would only consider this optimization if we were searching for the fastest possible network with very low accuracy demands, because for any other use-case it is always preferable to just lower the input resolution keeping 3 input channels and the RGB color space.

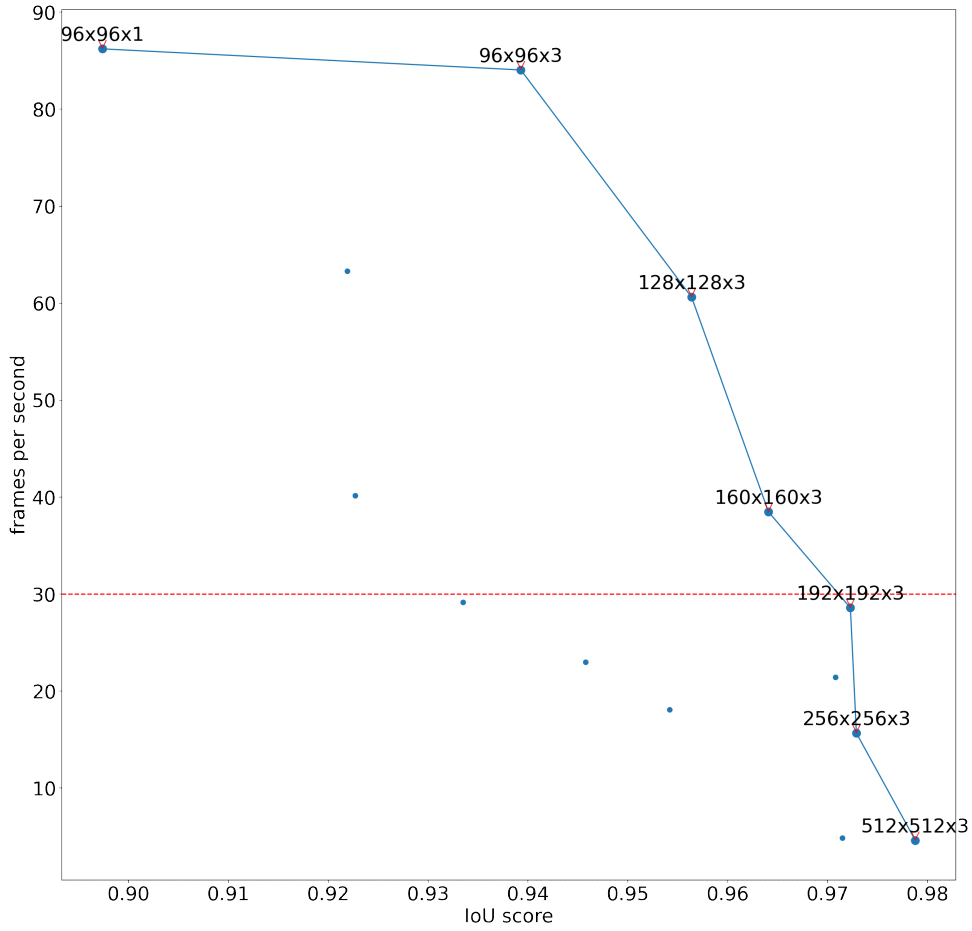

Figure 4.7. Pareto Frontier Plot - Stages 2 and 3

4.4 Network Topology Optimization

4.4.1 Stage 4 - Width Multiplication

Details on the Experiment This is the first stage in which we start manipulating the structure of the network, we are entering the second macro branch of our pipeline. With the term Width Multiplication we refer to the number of channels of any convolutional layer (methodology chapter), so in the case of FastSCNN we identified 2 tuning knobs for this particular simplification. The tuning knob a is the actual width multiplier, while t is the expansion rate of the bottleneck blocks. As previously explained in the methodology chapter, the expansion rate determines the number of channel of a convolutional layer that is located inside the bottleneck block, in particular t is multiplied by the number of channels of the input of the bottleneck block, so it is itself a width multiplier.

We chose a 4 values of a=[0,75, 0.50, 0.25 0.125], the baseline is a=1.0. We sacrifice some of the combination by starting from 0,75 and not from 1.0 because the effect of

Block	Output chans	Stride	Repetitions	Exp rate
Conv2d	32*a	2	1	-
DSConv	48*a	2	1	-
DSConv	64*a	2	1	-
bottleneck	64*a	2	3	t
bottleneck	96*a	2	3	t
bottleneck	128*a	1	3	t
Pyramid Pooling Module	128*a	-	1	-
Feature Fusion Module	128*a	-	-	-
DSConv	128*a	1	2	-
Conv2d	num_classes (1)	1	1	-

Table 4.10. Model Structure

t is not as important as that of a and we wanted to save some training time. We can observe the effect of varying the value of t for all other values of a. We selected the values [6, 4, 2] for t. We start from the baseline value of 6 and we stop at 2. We could have gone further and tried the value 1, which means that there is no channel expansion in the bottleneck block, but we decided to avoid this because it would go against one of the main defining characteristics of the bottleneck block. Two in an acceptable reduction from 6 and allows us to keep intact one the functionalities of the bottleneck block.

In this stage, looking at the model summaries, we notice that the input size stays constant, while the number of parameters changes as we change a and t. This is because we are deleting actual model parameters from the network instead of manipulating the input data as we did in the previous stages. The size that the saved model takes up in memory is much lower because the saved parameters are a lot less. The number of MACs decreases as we decrease both the values of a and t, as we would have expected.

In this stage we are keeping the pyramid pooling module, because this time it does not interfere with the tuning knobs and in the following stage we will assess its importance in the model. Out of the model topology optimizations, width multiplication is the most universally applicable, since the great majority of segmentation models features convolutional layers and with this optimization we are operating on the number of channels in the convolutional layers.

Comments on the Results Analyzing the behavior of the model when changing the values of the tuning knobs a and t we can observe that changing the value of a yields the biggest decrease in inference latency. Changing from one value of a to the next always leads to an improvement in latency. Keeping the value of t at the baseline 6, we notice that we can get over the real-time constraint of 30 fps with a latency of 0.0265 when a=0.125, while the IoU stays at a value of 0.9711. If we compare only this first step to the previous branch of optimizations we achieve similar latency performance with an input size of 160x160x3 which resulted in a lower IoU score of 0.9641.

model	a	t	r	pp	#params	input (MB)	fwd pass(MB)	MACs (M)
baseline	1	6	3	1	1,122,673	3,15	209,88	830,67
1	0,75	6	3	1	644,917	3,15	157,42	485,61
2	0,5	6	3	1	298,553	3,15	104,96	232,14
3	0,25	6	3	1	83,581	3,15	52,49	70,27
4	0,125	6	3	1	25,367	3,15	26,26	23,69
5	0,75	4	3	1	450,757	3,15	134,61	396,21
6	0,5	4	3	1	208,793	3,15	89,75	191,35
7	0,25	4	3	1	58,541	3,15	44,89	59,28
8	0,125	4	3	1	17,807	3,15	22,46	20,54
9	0,75	2	3	1	256,597	3,15	111,8	306,81
10	0,5	2	3	1	119,033	3,15	74,55	150,56
11	0,25	2	3	1	33,501	3,15	37,29	48,29
12	0,125	2	3	1	10,247	3,15	18,66	17,4

Table 4.11. Model Summaries

Changing the value of t always results in lower inference latency but in some cases it resulted in even higher intersection over union. However the difference in latency and IoU score is not as pronounced as it was for a. From the model summary we see that the number of MACs gets significantly lower from one value of t to the next, as does the total number of parameters, but this does not reflect as strongly on the value of the IoU score. In the following stages we will explore the interaction of different values of t with other tuning knobs.

In this first stage we start to get an understanding on how operating on the structure of the network is more effective in terms of intersection over union. As a matter of fact the combination a=0.125 and t=2 yields a frame-rate of 47.1698 and an IoU score of 0.9686, which is still higher than the IoU of the 160x160x3 model. On the other hand, the inference latency achieved with lower resolutions is still lower than the lowest achieved changing just a and t. While 47 frames per second are way past the real time requirements, in some cases a faster inference could be required, for example in cases in which the scene changes very fast and an interval of 0.0334 seconds between one frame and the next could lead to missing important information.

model name	a	t	r	pp	inference latency (s)	frame rate	mean IoU
baseline	1	6	3	1	0.2204	4.5372	0.9777
1	0,75	6	3	1	0.1648	6.0679	0.9771
2	0,5	6	3	1	0.1000	10.0000	0.9782
3	0,25	6	3	1	0.0485	20.6185	0.9762
4	0,125	6	3	1	0.0265	37.7358	0.9711
5	0,75	4	3	1	0.1384	7.2254	0.9778
6	0,5	4	3	1	0.0871	11.4810	0.9773
7	0,25	4	3	1	0.0420	23.8095	0.9738
8	0,125	4	3	1	0.0238	42.0168	0.9716
9	0,75	2	3	1	0.1123	8.9047	0.9788
10	0,5	2	3	1	0.0726	13.7741	0.9774
11	0,25	2	3	1	0.0354	28.2485	0.9740
12	0,125	2	3	1	0.0212	47.1698	0.9686

Table 4.12. Performance Data

4.4.2 Stage 5 - Operator Repetition

Details on the Experiment The term operator repetition indicates a series of tuning knobs related with layers or functional blocks of the network that are repeated across the network structure and that can be removed without consequences other than the reduction of the number of parameters.
In the case of our baseline, FastSCNN, we identified the tuning knobs in r, which is the number of times the bottleneck block is repeated in the Context Branch and t (0 or 1), which is the presence of the pyramid pooling block.

From what was already stated in the methodology chapter, we do not expect the presence of the pyramid pooling block to make a big difference on the behavior of the network in the case of our dataset. On the other hand the effect of the tuning knob r, for which we are considering the values [3, 2, 1], can have a strong effect on the performance of the model, however in the case of our baseline there aren't many functional blocks to start with, so changing the value of r is not going to have the same effect as it would in the case of a bigger and more complex baseline.

From the model summaries we can see that eliminating the pyramid pooling module only slightly changes the number of MACs and does not change the number of trainable parameters, while the number of repetitions of the bottleneck block has a much more sizable effect. However, we can already notice that these quantities are much bigger than

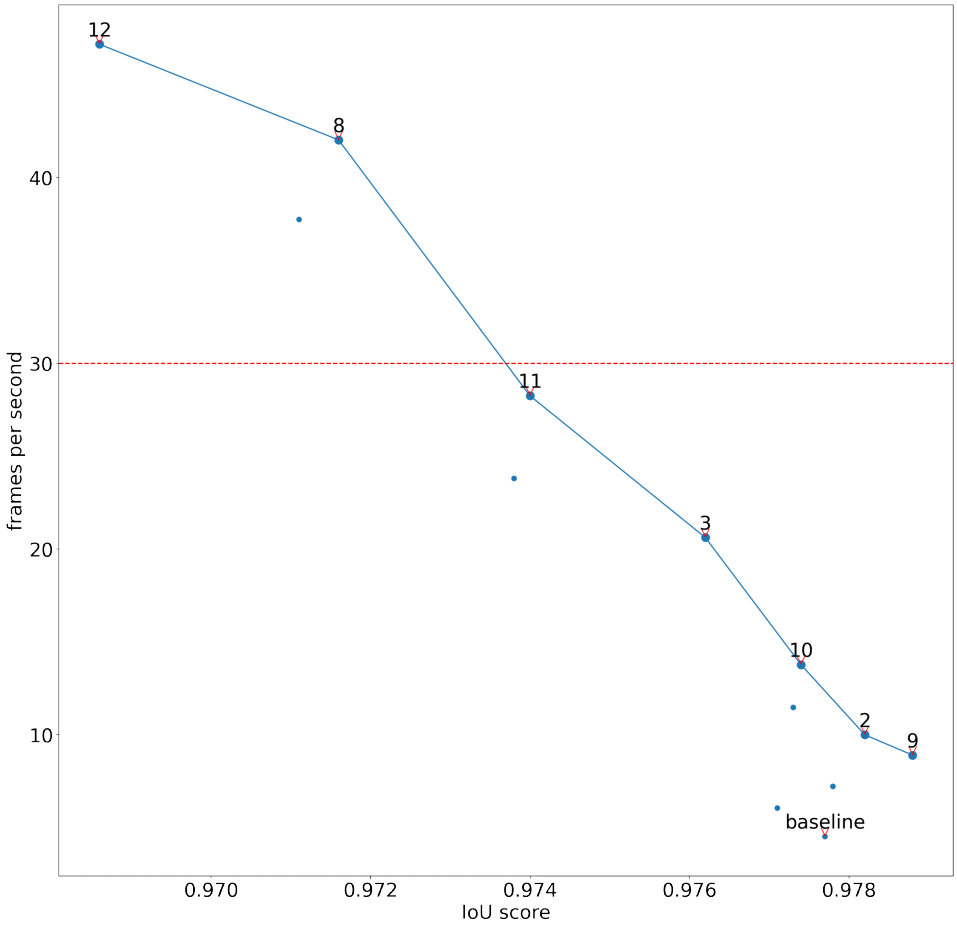

Figure 4.8. Pareto Frontier Plot - Stage 4

Block	**Output chans**	**Stride**	**Repetitions**	**Exp rate**
Conv2d	32	2	1	-
DSConv	48	2	1	-
DSConv	64	2	1	-
bottleneck	64	2	r	6
bottleneck	96	2	r	6
bottleneck	128	1	r	6
Pyramid Pooling Module	128	-	pp	-
Feature Fusion Module	128	-	-	-
DSConv	128	1	2	-
Conv2d	num_classes (1)	1	1	-

Table 4.13. Model Structure

model	a	t	r	pp	#params	input (MB)	fwd pass(MB)	MACs (M)
baseline	1	6	3	1	1,122,673	3,15	209,88	830,67
13	1	6	3	0	1,122,673	3.15	200,97	826,65
14	1	6	2	1	743,281	3,15	184,32	695,05
15	1	6	2	0	743,281	3,15	175,41	691,03
16	1	6	1	1	363,889	3,15	158,76	559,43
17	1	6	1	0	363,889	3,15	149,85	555,41

Table 4.14. Model Summaries

in the previous stages.

Comments on the Results The first thing that we notice is that the latency achieved in this step, while lower than the baseline, does not get close to the real-time goal. However, the intersection over union score stays very high throughout the range of values of the tuning knobs, even achieving the best score observed so far of 0.9802 in the smallest model for this stage with r=1 and pp=0.

model name	a	t	r	pp	inference latency (s)	frame rate	mean IoU
baseline	1	6	3	1	0.2204	4.5372	0.9777
13	1	6	3	0	0.2391	4.1823	0.9792
14	1	6	2	1	0.1973	5.0684	0.9741
15	1	6	2	0	0.1942	5.1493	0.9750
16	1	6	1	1	0.1643	6.0864	0.9778
17	1	6	1	0	0.1663	6.0132	0.9802

Table 4.15. Performance Data

Thanks to this stage we understand that the presence of the pyramid pooling module is useless for the dataset in question and at times even detrimental. The difference in intersection over union is practically negligible but the model without the pyramid pooling module always performed better. In terms of latency the model without the pyramid pooling module performed better 2 times out of 3, but the difference is always very small. From this data we have the confirmation of what we assumed in the methodology chapter based on the characteristics of our dataset.

The repetition of the bottleneck block has much more evident effects on inference latency. From one value of r to the next there is always a reduction in latency. The intersection

over union score does not follow the same trend, keeping a high value for every model tested in this stage and at times being higher with a lower value of r. A possible justification of this behavior is that the models analyzed in this stage feature a large number of parameter and even when reducing the number of bottleneck blocks there are enough parameters to grant sufficient performance. An interesting analysis that we are going to carry out in the following stages is regards the interaction of this tuning knob with models with fewer parameters, combining it with the other tuning knobs that we have introduced in the previous stages.

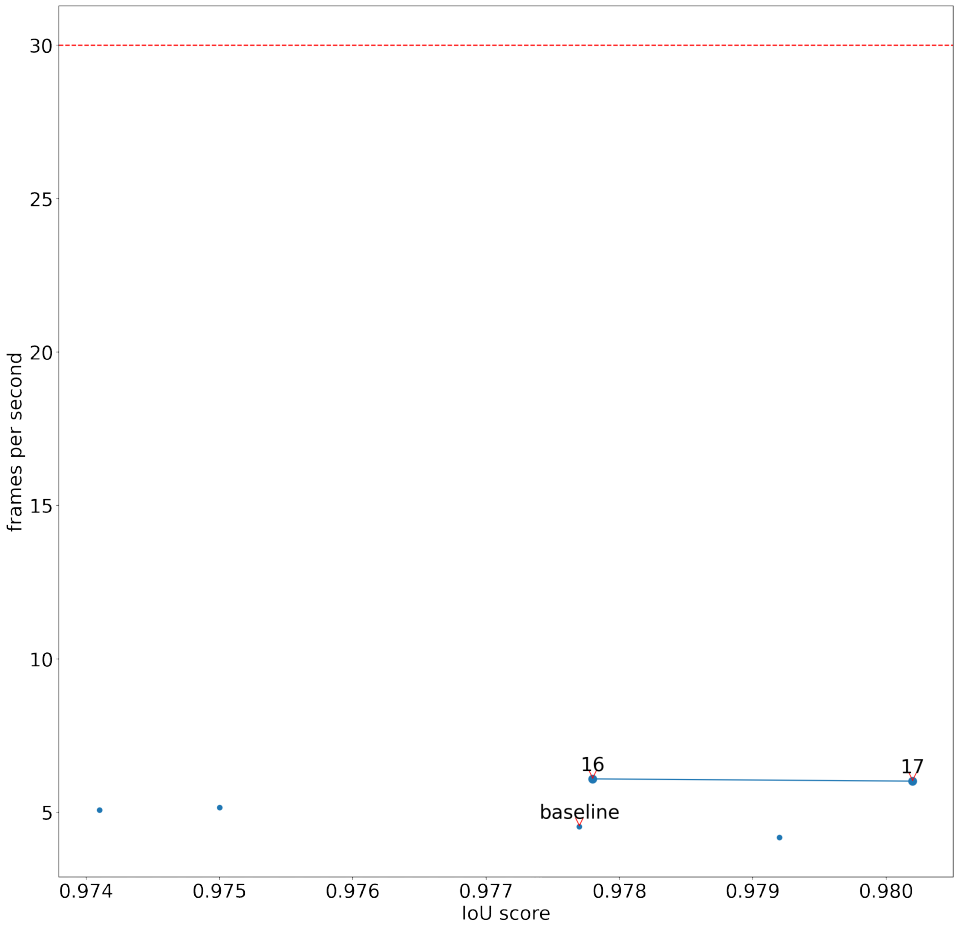

Figure 4.9. Pareto Frontier Plot - Stage 5

4.4.3 Stage 6 - Width Multiplication and Operator Repetition Combination

width multiplier	expansion rate	operator repetition	pyramid pooling
0.25	6	2	0
0.25	6	1	0
0.25	4	2	0
0.25	4	1	0
0.25	2	2	0
0.25	2	1	0
0.125	6	2	0
0.125	6	1	0
0.125	4	2	0
0.125	4	1	0
0.125	2	2	0
0.125	2	1	0

Table 4.16. Solution Combination - Parameter Matrix

Details on the Experiment Up until this stage we have explored all the possible combinations of the tuning knobs that were considered at each stage. Now we need to combine the tuning knobs considered in the previous two stages and we are faced with the problem of the number of possible combinations. As we have already stated, training a model requires time and computing resources, so, in order to reduce the amount of resources used in the experimental part of this thesis we explored a subset of the solution space.

Block	Output chans	Stride	Repetitions	Exp rate
Conv2d	32*a	2	1	-
DSConv	48*a	2	1	-
DSConv	64*a	2	1	-
bottleneck	64*a	2	r	t
bottleneck	96*a	2	r	t
bottleneck	128*a	1	r	t
Pyramid Pooling Module	128*a	-	0	-
Feature Fusion Module	128*a	-	-	-
DSConv	128*a	1	2	-
Conv2d	num_classes (1)	1	1	-

Table 4.17. Model Structure

This subspace is defined by selecting a subset of the original set of values of the tuning knobs from the previous stages. In particular, for the width multiplication we used only the values of 0.25 and 0.125 for a because they were the only values that achieved performance over the real-time objective. For t we kept all 3 original values of [6, 5, 2], for r we kept the values 2 and 1 and discarded r=3, and we only considered models without the pyramid pooling module since we assessed its uselessness for the task at hand.

The purpose of this set of experiments is to investigate the interaction of the tuning knobs that operate on the topology of the network. In the previous stages we have observed how the number of parameters of the network and multiplication-accumulation operations change with these optimization strategies and in this stage we want to understand how the behavior of the network changes when we operate on both width multiplication and operator repetition at the same time.

model	**a**	**t**	**r**	**pp**	**#params**	**input (MB)**	**fwd pass(MB)**	**MACs (M)**
18	0,25	6	2	0	55,549	3,15	43,88	61,19
19	0,25	6	1	0	27,517	3,15	37,49	51,517
20	0,25	4	2	0	39,805	3,15	38,24	53,44
21	0,25	4	1	0	21,069	3,15	33,82	46,95
22	0,25	2	2	0	24,061	3,15	32,6	45,68
23	0,25	2	1	0	14,621	3,15	30,15	42,44
24	0,125	6	2	0	16,919	3,15	21,95	21,3
25	0,125	6	1	0	8,471	3,15	18,76	18,45
26	0,125	4	2	0	12,151	3,15	19,14	19,1
27	0,125	4	1	0	6,495	3,15	16,92	17,2
28	0,125	2	2	0	7,383	3,15	16,32	16,91
29	0,125	2	1	0	4,519	3,15	15,09	15,96

Table 4.18. Model Summaries

From the model summary we can observe the lowest number of parameters so far of 4519 which is almost 250 times smaller than the original number of parameters of the baseline (1.12M). Such an extreme reduction is very useful to understand how big the model needs to be to perform well on the selected task. Combining the two strategies in the model topology optimization branch it become clear how the effort required by this branch is much higher not only because of the time required to train all the models, but also because it requires a deep understanding of the network structure and of the characteristics and demands of the dataset. In fact, identifying the tuning knobs requires

a deep knowledge of the architecture of the network and its functionality, while selecting the possible values of this tuning knobs has more to do with the complexity of the dataset, which determines how far the simplification of the network can go.

Comments on the Results The results of these experiments are very interesting. We can immediately see that the width multiplier knob a is the most effective out of the 4, as we anticipated in the width multiplication stage. As a matter of fact even when combined with the other 3 tuning knobs, it is still the one that grants the biggest improvement in inference latency.

model name	a	t	r	pp	inference latency (s)	frame rate	mean IoU
18	0.25	6	2	0	0.0435	22.9885	0.9744
19	0.25	6	1	0	0.0383	26.1096	0.9747
20	0.25	4	2	0	0.0380	26.3158	0.9711
21	0.25	4	1	0	0.0352	28.4090	0.9733
22	0.25	2	2	0	0.0344	29.0697	0.9733
23	0.25	2	1	0	0.0320	31.2500	0.9702
24	0.125	6	2	0	0.0243	41.1522	0.9739
25	0.125	6	1	0	0.0224	44.6428	0.9701
26	0.125	4	2	0	0.0219	45.6621	0.9717
27	0.125	4	1	0	0.0203	49.2610	0.9614
28	0.125	2	2	0	0.0209	47.8468	0.9706
29	0.125	2	1	0	0.0201	49.7512	0.9677

Table 4.19. Performance Data

Changing the value of t and r had a similar effect on the performance of the network. The drop in latency was not as large as for a, but it was still noticeable and the IoU score stayed at an acceptable value throughout the experiment. However, the behavior of the intersection over union score was more predictable when going from higher to lower values of t an r. In the previous stage, changing the value of r to a lower value sometimes yielded a better IoU score, while this time it always led to a small drop. We explained this behavior as the network having many parameters to compensate for the simplification of the context branch by removing some of the bottleneck blocks. In that case having less parameters could have led to advantages in the training stage a so the performance slightly improved. In this stage we are starting from models that were already greatly simplified by reducing the number of channels of the convolutional layers. So, even if the performance is still more than acceptable, we start to witness the limits of the simplifications and having less parameters implicates that the network extracts less knowledge

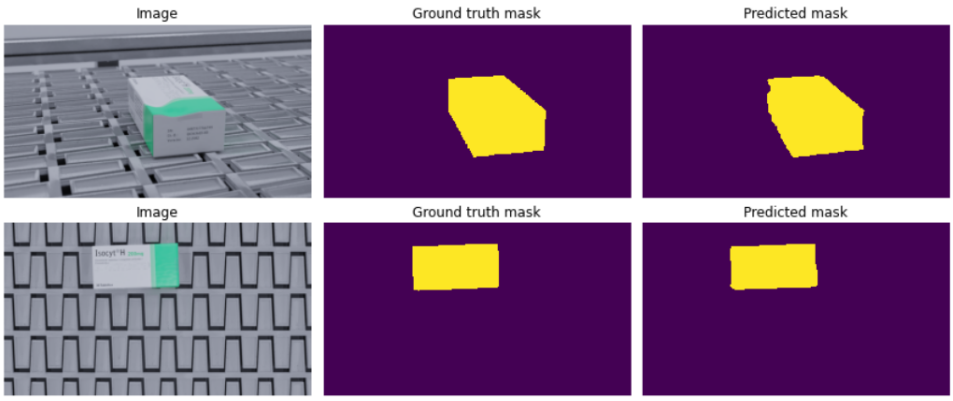

Figure 4.10. Output with a=1.0 t=2 r=1 pp=0

and the quality of its predictions suffers from it.

Since "a" is the most effective tuning knob by a great margin, the best results achieved in this stage are not that far from the results obtained in the width multiplication stage. However, by changing the other knobs we obtain further control on the performance of the network. For example the combination a=0.125, t=6, r=1 achieves an inference latency of 0.0243 (41.152 fps) while keeping an IoU score of 0.9739 which in the previous stages was paired with much slower inference. This trend is well portrayed in the Pareto frontier plot in which we included all the models considered in the 3 network topology optimization stages. While there is not a great jump in inference latency, most of the models on the frontier are models from this third stage, the most relevant being: [a=0.125, t=2, r=1][a=0.125, t=2, r=2][a=0.125, t=4, r=2][a=0.125, t=6, r=2], because they all achieve an inference latency over the real-time objective and also keep a high intersection over union score.

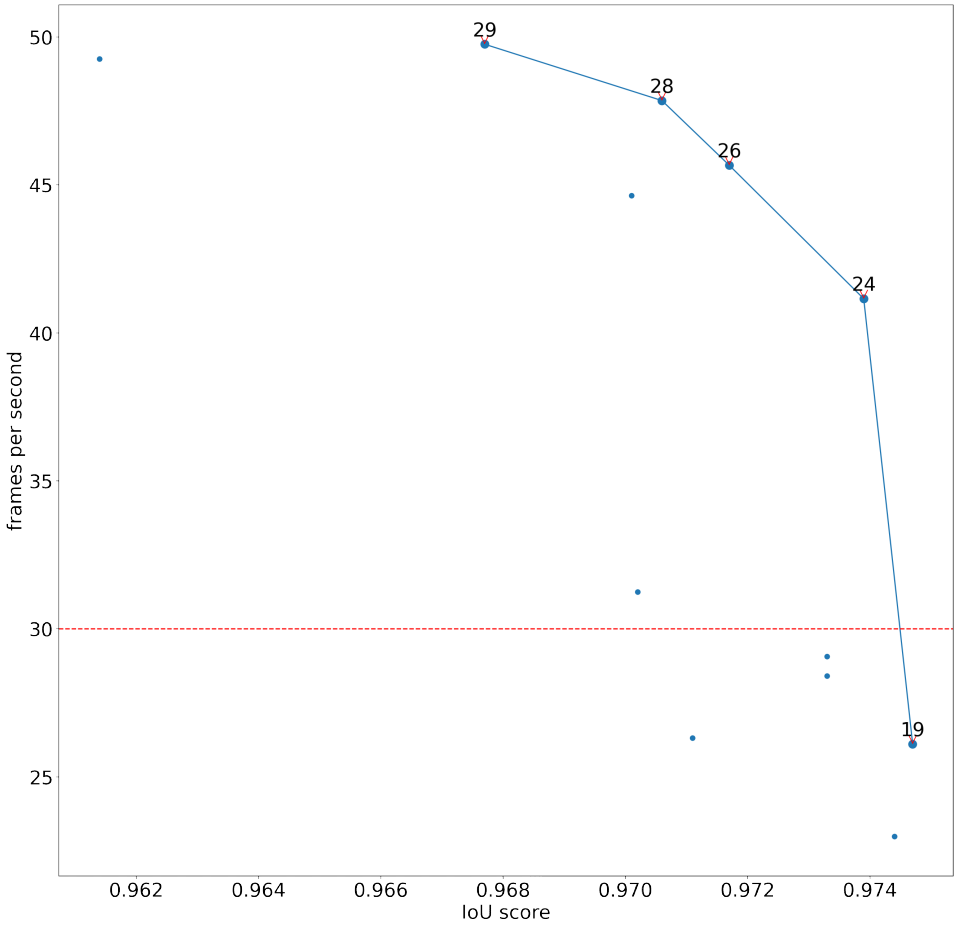

Figure 4.11. Pareto Frontier Plot - Stage 6

4.5 Combinations of the Two Macro Branches

Comparison of the Two Macro Branches From the experiments carried out so far we can make some important considerations between the two macro-branches and the models obtained during the analysis.

In the Pareto frontier plot above we can clearly notice that the models belonging to the Input Data Optimization step present on the frontier are located in the left and upper part of the plot. This means, as we have anticipated, that the fastest overall inference latency is achieved by using a reduced input resolution. It also means that these models achieved the worst intersection over union scores out of the models belonging to the frontier.

The second consideration is that the models belonging to the Network Topology Optimization branch are located to the right of the plot. This means that these simplifications

managed to achieve higher intersection over union scores. The reasoning behind this behavior is that instead of eliminating important knowledge from the input they reduce the number of parameters in the network, generating a much more essential model that has an output comparable to that of the full model because of the less demanding nature of the dataset. It is true that a more complex dataset could present more challenges when operating on the topology of the network, but it is also true that the very definition of neural networks accounts for a larger than necessary number of parameters.

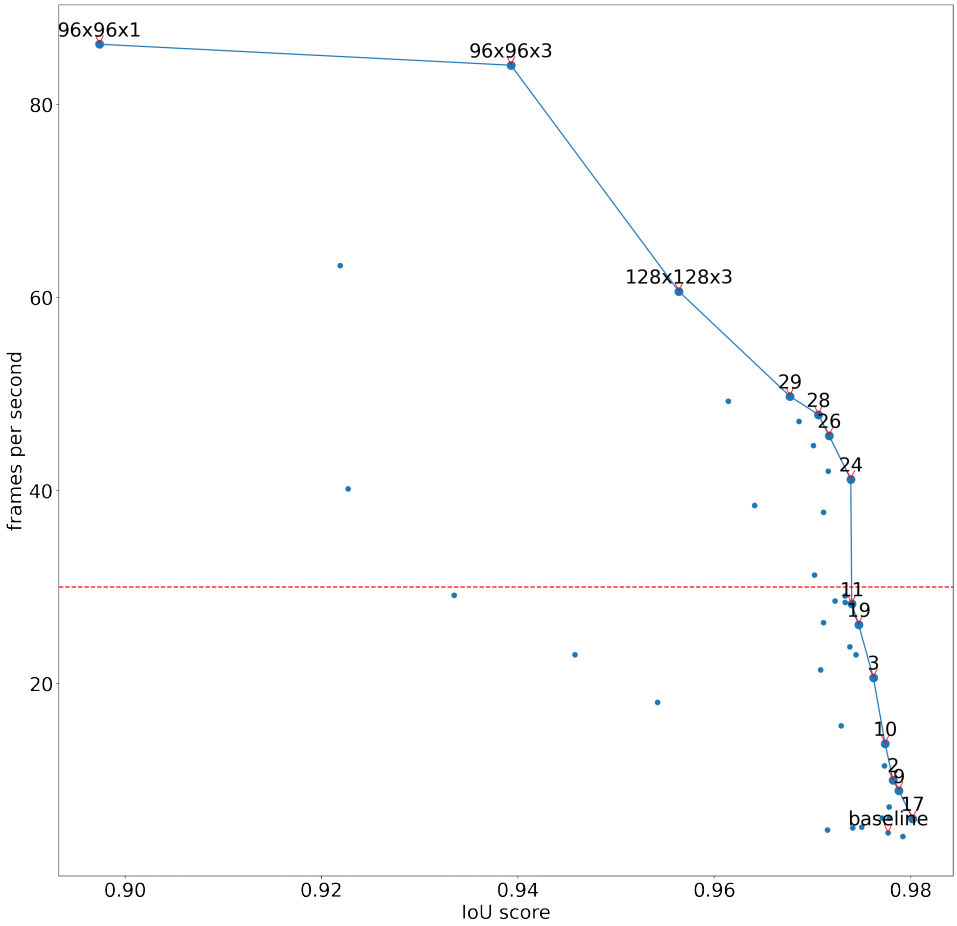

Figure 4.12. Pareto Frontier Plot - Stages 2 to 6

Another important consideration about Network Topology Optimization is that the models produced in this branch are much closer together on the frontier, which means that operating on the tuning knobs allows for finer control over performance, which cannot be said for the models belonging to the Input Data Optimization branch, that are much further apart with large drops in latency always followed by large drops in IoU score. This means that in cases in which the window created by latency and accuracy constraints is very narrow, operating exclusively on input resolution might bring to missing the window,

while operating on the structure of the network has more chances of finding acceptable solutions.

Finally, from the plot we can see how one can opt for one or the other strategy according to the constraints of the task. In fact, if the constraint on latency is stricter than the constraint on accuracy, the most effective strategy is changing the input resolution. On the other hand, if the accuracy constraint is the stricter manipulating the structure of the network with the techniques illustrated in these experimental stages is the best way to keep a higher intersection over union score and also reduce the inference latency by a considerable amount.

4.5.1 Stage 7 - Input Data Optimization and Network Topology Optimization Combination

Details on the Experiment This was the most elaborate stage of experiments. Combining models from all of the previous stages resulted in a large number of possible solutions. If we were to analyze each possible combination of the tuning knobs from the Network Topology Optimization branch with all of the resolutions analyzed in the Input Data Optimization branch, we would have had to train 490 models, which would have resulted in a very large amount of time spent training such models and a large amount of computational and energy resources used in the process. In order to obtain meaningful information out of the following analysis without going through so many models, we selected the best performing models from stage 6 and the best performing resolutions from stage 2. The structure of the network is described in Table 4.17.

model	input res	a	t	r	pp	#params	input	fwd pass	MACs (M)
1	512x512	1	6	1	0	363,889	3,15	149,85	555,41
2	512x512	0,25	6	1	0	27,517	3,15	37,49	51,46
3	512x512	0,125	6	2	0	16,919	3,15	21,95	21,3
4	512x512	0,125	4	2	0	12,151	3,15	19,14	19,1
5	512x512	0,125	2	2	0	7,383	3,15	16,32	16,91
6	512x512	0,125	2	1	0	4,519	3,15	15,09	15,96
7	256x256	1	6	1	0	363,889	0,79	37,46	138,86
8	256x256	0,25	6	1	0	27,517	0,79	9,37	12,87
9	256x256	0,125	6	2	0	16,919	0,79	5,49	5,33
10	256x256	0,125	4	2	0	12,151	0,79	4,78	4,78
11	256x256	0,125	2	2	0	7,383	0,79	4,08	4,23
12	256x256	0,125	2	1	0	4,519	0,79	3,77	3,99
13	192x192	1	6	1	0	363,889	0,44	21,07	78,11
14	192x192	0,25	6	1	0	27,517	0,44	5,27	7,24
15	192x192	0,125	6	2	0	16,919	0,44	3,09	3
16	192x192	0,125	4	2	0	12,151	0,44	2,69	2,69
17	192x192	0,125	2	2	0	7,383	0,44	2,38	2,38
18	192x192	0,125	2	1	0	4,519	0,44	2,12	2,24

Table 4.20. Model Summaries I

model	input res	a	t	r	pp	#params	input	fwd pass	MACs (M)
19	160x160	1	6	1	0	363,889	0,31	14,63	54,25
20	160x160	0,25	6	1	0	27,517	0,31	3,66	5,03
21	160x160	0,125	6	2	0	16,919	0,31	2,14	2,08
22	160x160	0,125	4	2	0	12,151	0,31	1,87	1,87
23	160x160	0,125	2	2	0	7,383	0,31	1,59	1,65
24	160x160	0,125	2	1	0	4,519	0,31	1,47	1,56
25	128x128	1	6	1	0	363,889	0,2	9,37	34,72
26	128x128	0,25	6	1	0	27,517	0,2	2,34	3,22
27	128x128	0,125	6	2	0	16,919	0,2	1,37	1,33
28	128x128	0,125	4	2	0	12,151	0,2	1,2	1,2
29	128x128	0,125	2	2	0	7,383	0,2	1.02	1,06
30	128x128	0,125	2	1	0	4,519	0,2	0,94	1
31	96x96	1	6	1	0	363,889	0,11	5,27	19,53
32	96x96	0,25	6	1	0	27,517	0,11	1,32	1,81
33	96x96	0,125	6	2	0	16,919	0,11	0,77	0,75
34	96x96	0,125	4	2	0	12,151	0,11	0,67	0,67
35	96x96	0,125	2	2	0	7,383	0,11	0,57	0,6
36	96x96	0,125	2	1	0	4,519	0,11	0,53	0,56

Table 4.21. Model Summaries II

In particular we chose 6 models from stage 6, all of which belong to the Pareto frontier for that stage. The first combination is [a=1.0, t=6, r=1] which is the one that achieved the best overall intersection over union score; then we have [a=0.25, t=6, r=1], [a=0.125, t=6, r=2], [a=0.125, t=4, r=2], [a=0.125, t=2, r=2], [a=0.125, t=2, r=1]. The latter corresponds to the lowest inference latency recorded in the Network Topology optimization branch. The first two models did not achieve real-time latency, but including them in the analysis can still be beneficial for observing how changing the input resolution affects the simplification of the structure of the model.

We did not include any of the models on the frontier that featured the pyramid pooling module because of the incompatibility with the input resolution reduction and also because we assessed that the difference between a model with or without pyramid pooling is negligible [Table 4.15]. We analyzed all resolutions except for 224x224 because it had lower accuracy than lower resolutions while having slower inference latency.

In this "combination" step we chose model structures that appeared on the Pareto curve in stage 6 [4.4.3] instead of selecting ranges of values for the tuning knobs. The reason behind this is that in stage 6 we were interested in understanding the way different values of a, t and r interacted with each other, while now we already know important information about the behavior of these tuning knobs and are interested in understanding how the two branches of optimizations interact with each other and how far we can push the performance of the baseline combining the two strategies.

Looking at the model summaries we see that lowest number of MACs is reached in the last model. This value is significant because its value is affected by changes in both the input data and the structure of the model, while the number of trainable parameters only changes when the structure of the network changes and the input size changes only when the input resolution changes. It is also important to understand how also the size of the forward/backward pass, which describes the amount of system memory required to run the model, reaches its all time low in this stage. It is remarkable because with a disk occupation of 0.01MB and a system memory occupation of 0.53MB, the smallest model is way below the system specifications of our device (raspberry pi 4) and probably could run on much lower end hardware. This last observation is true for many of the models in this stage and opens the floor to the discussion on how the combination of all the possible optimizations in the pipeline can be used to optimize the model to an extreme that might not be needed for a task like the one presented in this thesis, but could prove useful for simpler tasks (even outside of the domain of semantic segmentation) running on less powerful hardware.

Comments on the Results The first thing we can see from these results is that at least one model for each input resolution appears on the Pareto frontier. This means that each time we go from a resolution to the next there is always a gain in inference latency, no matter the network simplifications occurring at the same time.

The second thing that we notice is that we were able to achieve incredibly high framerates. There aren't many realistic use-case scenarios for a model that runs at over 800 frames per second, but such a fast latency can be interpreted not only in the context of how many operations can be performed in a time unit, but also in the context of energy consumption. In fact, with parallelized computing, the amount of energy consumed when performing a task is no longer dependent on the number of multiplication-accumulation operations contained in the task, but it is strictly correlated with the latency, so a model that can perform inference at a very low latency will also consume very little power. If we put this in the context of a battery-powered edge-device, it means that performing inference on the same stream of images, the model with the fastest inference will consume a lot less energy and its battery life will be much longer. Another reason why we would want faster inference is that, how we have stated in other parts of this thesis, semantic segmentation is not an end to itself. It is a way for a computer to understand a particular scene, but its output is almost always the input for some other operation, which will take some time. So depending on the complexity of these additional operations and

the nature of the task, it may be necessary to achieve a latency that is much lower than what we have so far defined as real-time. However while it seems having such a low latency is a good thing, it also brings attention to the importance of an accuracy constraint.

model	input res	a	t	r	pp	inference latency (s)	frame rate	mean IoU
1	512x512	1	6	1	0	0,1663	6.0132	0,9789
2	512x512	0,25	6	1	0	0,0381	26.2467	0,9733
3	512x512	0,125	6	2	0	0,0244	40.9836	0,9679
4	512x512	0,125	4	2	0	0,0223	44.8430	0,9717
5	512x512	0,125	2	2	0	0,0207	48.3092	0,9703
6	512x512	0,125	2	1	0	0,0205	48.7805	0,9651
7	256x256	1	6	1	0	0,0431	23.2019	0,9711
8	256x256	0,25	6	1	0	0,0091	109.8901	0,9636
9	256x256	0,125	6	2	0	0,0065	153.8462	0,9613
10	256x256	0,125	4	2	0	0,006	166.6667	0,9524
11	256x256	0,125	2	2	0	0,0056	178.5714	0,9541
12	256x256	0,125	2	1	0	0,0053	188.6792	0,9391
13	192x192	1	6	1	0	0,0237	42.1941	0,9703
14	192x192	0,25	6	1	0	0,0053	188.6792	0,9588
15	192x192	0,125	6	2	0	0,004	250.0000	0,9492
16	192x192	0,125	4	2	0	0,0037	270.2703	0,9493
17	192x192	0,125	2	2	0	0,0033	303.0303	0,9476
18	192x192	0,125	2	1	0	0,0031	322.5806	0,9275

Table 4.22. Performance Data I

Speaking of intersection over union, we see interesting values for the resolutions of 512x512 and 256x256, while for the others we start seeing lower values. The reason for this behavior is that we have reached a point at which we have so few parameters that every time we remove something from the network we run the risk of losing crucial information. Before combining these two main optimization branches we were able to achieve interesting IoU values because in one case the lower input resolution was countered by the amount of parameters of the network and in the other case the reduction of parameters of the network did not affect the output quality a lot because most of the parameters were redundant and a full resolution input image is very rich in information. When combining these strategies we get closer and closer to the point in which, even for a simple dataset

model	input res	a	t	r	pp	inference latency (s)	frame rate	mean IoU
19	160x160	1	6	1	0	0,0161	62.1118	0,9621
20	160x160	0,25	6	1	0	0,0039	256.4103	0,9487
21	160x160	0,125	6	2	0	0,0031	322.5806	0,9386
22	160x160	0,125	4	2	0	0,0028	357.1429	0,9327
23	160x160	0,125	2	2	0	0,0025	400.0000	0,9379
24	160x160	0,125	2	1	0	0,0023	434.7826	0,9317
25	128x128	1	6	1	0	0,0107	93.4579	0,957
26	128x128	0,25	6	1	0	0,0027	370.3704	0,9397
27	128x128	0,125	6	2	0	0,0022	454.5455	0,9143
28	128x128	0,125	4	2	0	0,002	500.0000	0,9255
29	128x128	0,125	2	2	0	0,0018	555.5556	0,9125
30	128x128	0,125	2	1	0	0,0017	588.2353	0,93
31	96x96	1	6	1	0	0,0069	144.9275	0,9387
32	96x96	0,25	6	1	0	0,0019	526.3158	0,9101
33	96x96	0,125	6	2	0	0,0016	625.0000	0,9053
34	96x96	0,125	4	2	0	0,0015	666.6667	0,9051
35	96x96	0,125	2	2	0	0,0014	714.2857	0,9027
36	96x96	0,125	2	1	0	0,0012	833.3333	0,9054

Table 4.23. Performance Data II

like ours, the information flowing through the network is all essential to the task and reducing it by simplifying the network structure or resizing the input images results in a substantial loss of output quality.

Another notable thing is that the lowest value for the intersection over union score achieved in this stage is still higher than the lowest value achieved when using only one input channel, which confirms our decision to exclude that particular type of optimization from the pipeline.

Analyzing these results more closely we can identify 3 clusters of models in the Pareto frontier plot. The first one is on the bottom right and corresponds to the models with the full-size input. The IoU is very high, but the latency is the highest among the clusters. The second cluster consists in a group of models with input sizes varying from 256x256 to 192x192 that have frame-rates that space from 110 fps (0.0091s latency) to 303 fps

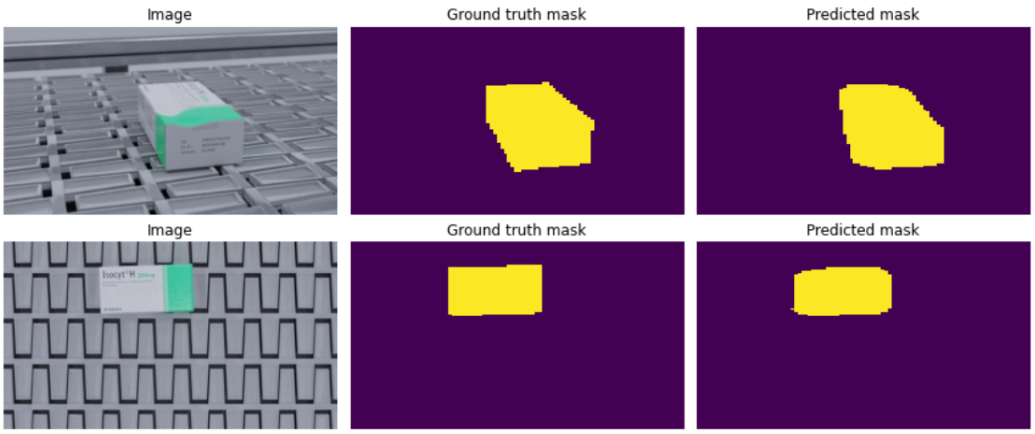

Figure 4.13. Output of the fastest model configuration (n. 36)

(0.0033s latency) and with IoU scores that vary from 0.9636 to 0.9476. This is probably the most interesting of the clusters because the latency is very low and the IoU score is still acceptable. The third cluster contains all of the other models, which achieve very low inference latency values, but also present the most noticeable degradation in intersection over union. In some way these 3 clusters of models represent the possible preferences in terms of latency or accuracy. The middle cluster is the most versatile, but there can be situations in which one of the two extremes may be necessary. And in those cases one can choose among the different models in the selected cluster in order to further tune the model for the task at hand.

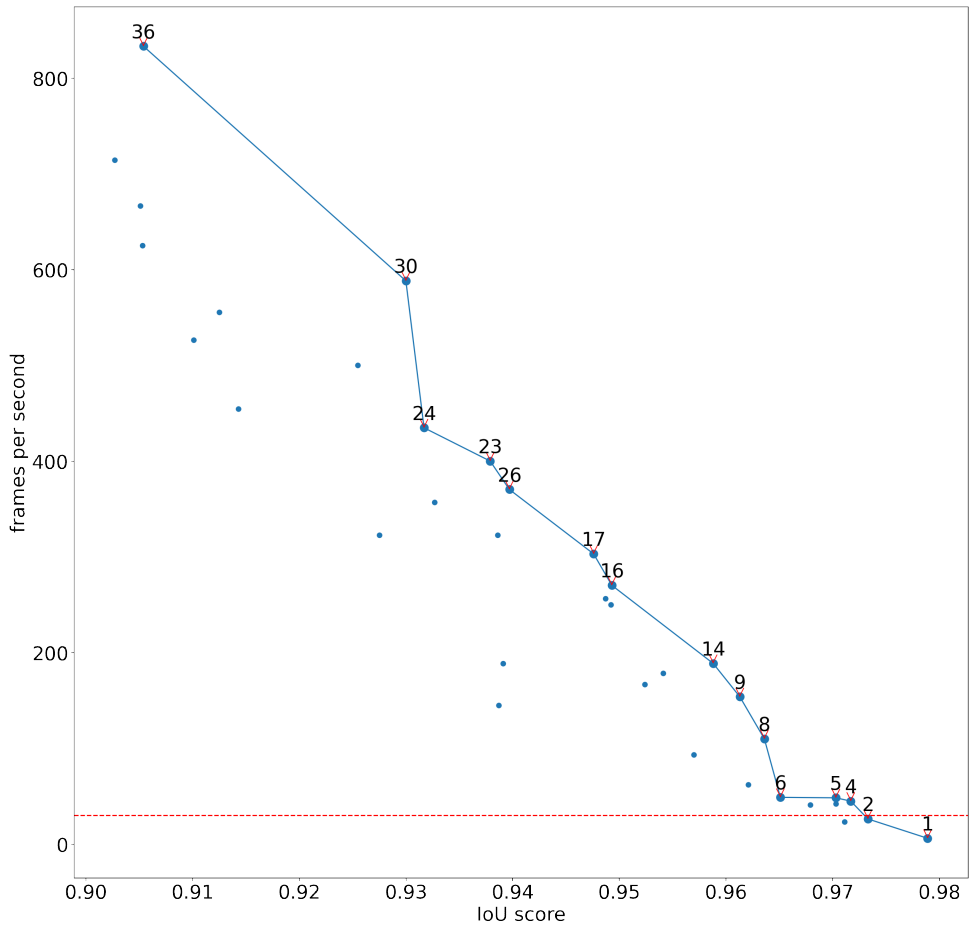

Figure 4.14. Pareto Frontier Plot - Stage 7

4.6 Important Considerations from the Experiments

Having analyzed the results of each experimental stage in detail, we can now gather the most important considerations that sum up what we learned so far. For the purpose of this Thesis we have experimented on a single baseline and on one particular dataset, but the following rules of thumb apply to most use-cases.

The first important observation is that while it is more time and resource consuming, re-training a model is necessary when changing input resolution. In the case of our baseline we re-trained the entire network, but with bigger models it is possible to freeze the parameters of the networks that make up the "feature extraction" portion of the model and re-train only the classification head, in order to save resources.

Next we discovered how reducing the number of input channels to 1 damages the intersection over union score more than it reduces inference latency. From Table 4.9 and

4.7 we can clearly see how reducing the input resolution keeping 3 input channels always provides both faster inference and higher intersection over union score. Therefore we concluded that the Input Channel Reduction stage is not a suitable optimization for semantic segmentation.

Moving to the Network Topology Optimization we understood that the optimizations belonging to this branch can have a varying degree of effectiveness according to the structure of the selected baseline. In our case we analyzed a smaller than usual network and we were able to optimize its performance by a good margin. It is straightforward to say that if we had started from a bigger baseline, the improvement due to these optimization would have been even bigger, although it is important to understand that a bigger baseline after applying the optimization would probably be slower than a smaller baseline with the same optimizations so the reasoning behind the model selection carried out in Chapter 3 still stands.

Speaking of Network Topology Optimization, a guideline to identify possible tuning knobs in a general semantic segmentation model is that Width Multiplication is the most universally applicable simplification, since it relies on the output channels of the convolutional layers and most state-of-the-art semantic segmentation models are indeed Fully Convolutional Networks. On the other hand the expansion rate t relies of the presence of bottleneck blocks, which are still very present throughout the literature because of the popularity of the ResNet backbone for image classification. Operator Repetition should also be widely applicable, but it requires a deeper model understanding to identify which functional blocks are repeated or removable.

When comparing the effect of the two optimization branches on the performance of the baseline model, we clearly visualized 4.12 how reducing the resolution of the input yielded the fastest inference, but also resulted in bigger drops in intersection over union score. On the other hand operating on the structure of the network did not bring the latency as far down, but allowed us to keep a much higher IoU score. It was also interesting to see how models with different input resolutions were more spaced out on the Pareto frontier plot, while the models belonging to the second branch were much closer together, meaning that changing the values of the network topology tuning knobs provides a finer control over the performance of the model, which can be very useful when accuracy and latency constraints create a smaller window of acceptable solutions and simply changing the input resolution could cause to miss such window.

Finally we combined the two macro branches and found out that it is the way to obtain the largest amount of candidate models. We noticed that the plot 4.14 can be divided into regions according to the requirements of the task at hand and we can choose models belonging to the region that better fits such requirements. Faster but less accurate models will appear in the top left region of the plot, while slower but more accurate models will be on the bottom right. More balanced results will more likely lay in the middle of the plot.

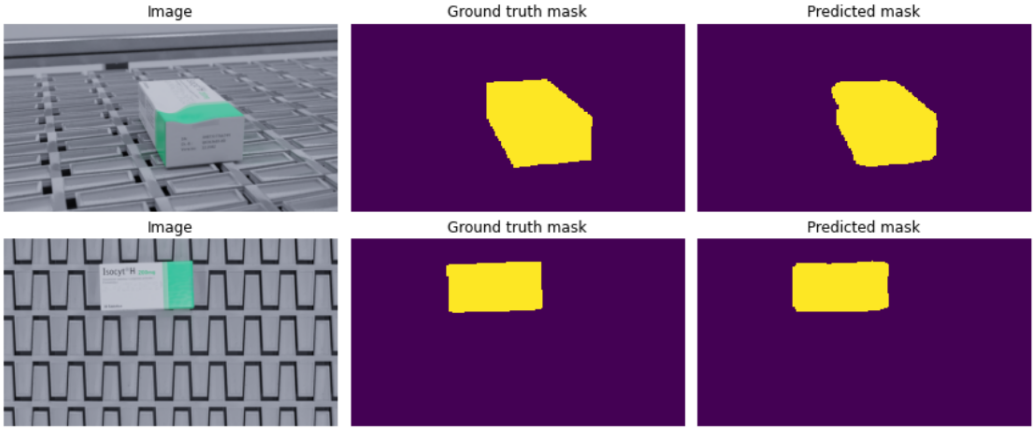

Figure 4.15. Output of model n. 17 capable of 3.3ms latency

Chapter 5

Conclusion

When we introduced the experiments, we stated that the purpose of these was to explore several optimization techniques in order to define a pipeline of optimizations that can be implemented as an automatic system. After reasoning on the results of the experiments we can put together said pipeline, structured based on the effort level of each stage.

The Input Data Optimization branch, in the definitive pipeline, consists of only the Input Resolution Reduction with retraining, because we observed that changing the resolution without retraining degrades the output quality and also because reducing the number of input channels to 1 proved to be much less effective than changing resolution.
This optimization stage will be the only stage available to Entry-Level users, because it does not require any understanding of the underlying network structure. It is also the best options for users that do not have much time available to them, or who want to save training time.

Contrarily to the Input Data Optimization branch, after the experiments, the Model Topology Optimization branch keeps all of its stages. This branch requires the users to be able to identify tuning knobs based on the structure on the network. In particular, the Width Multiplication stage requires the user to know how to change the number of output channels of a convolutional network, but it is still less complicated than the Operation Repetition stage, which requires a deeper understanding of the model to find the functional blocks or the layers that are repeated and can be removed without impacting the output of the network. Therefore, an intermediate user will be able to use the Input Resolution Reduction Stage and the Width Multiplication stage, while an expert user will be able to take advantage of the entire pipeline.
However, we have observed how the Width Multiplication stage is by far the most effective, therefore, if a user with pressing time constraints is dealing with a segmentation task that needs further optimization than the Input Resolution Reduction, Width Multiplication is the next stage to try.

This thesis also lends itself for many possible future developments. In fact, the experimental framework that we built can be used to build a novel model architecture. If we think about the fact that many models in the literature share building blocks if not entire

backbones, by evaluating the effect of various optimizations on an existing model, we can gain insights that can prove useful when designing a network.
Another interesting future development could be to expand the scope of the optimization pipeline. In fact, Semantic Segmentation shares many important traits with other computer vision tasks and it could be very interesting to try and build an optimization pipeline that can be used across several computer vision tasks.

Bibliography

[1] Stefan Ainetter and Friedrich Fraundorfer. End-to-end trainable deep neural network for robotic grasp detection and semantic segmentation from rgb, 2021.

[2] Amazon. Amazon prime air webpage, 2016.

[3] Gabriel J. Brostow, Julien Fauqueur, and Roberto Cipolla. Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, 30(2):88–97, 2009. Video-based Object and Event Analysis.

[4] Alexander V. Buslaev, Alex Parinov, Eugene Khvedchenya, Vladimir I. Iglovikov, and Alexandr A. Kalinin. Albumentations: fast and flexible image augmentations. *CoRR*, abs/1809.06839, 2018.

[5] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2018.

[6] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation, 2017.

[7] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. *CoRR*, abs/1802.02611, 2018.

[8] Francois Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[9] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding, 2016.

[10] Francesco Daghero, Daniele Jahier Pagliari, and Massimo Poncino. Chapter eight - energy-efficient deep learning inference on edge devices. In Shiho Kim and Ganesh Chandra Deka, editors, *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, volume 122 of *Advances in Computers*, pages 247–301. Elsevier, 2021.

[11] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks, 2015.

[12] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John M. Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88:303–338, 2009.

[13] Rahul Ghosh, Praveen Ravirathinam, Xiaowei Jia, Ankush Khandelwal, David Mulla, and Vipin Kumar. Calcrop21: A georeferenced multi-spectral dataset of satellite imagery and crop labels, 2021.

[14] Ke Gong, Xiaodan Liang, Dongyu Zhang, Xiaohui Shen, and Liang Lin. Look into person: Self-supervised structure-sensitive learning and a new benchmark for human parsing, 2017.

[15] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(9):1904–1916, 2015.

[18] Christopher J. Holder and Muhammad Shafique. On efficient real-time semantic segmentation: A survey, 2022.

[19] Yuanduo Hong, Huihui Pan, Weichao Sun, and Yisong Jia. Deep dual-resolution networks for real-time and accurate semantic segmentation of road scenes, 2021.

[20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[21] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016.

[22] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5mb model size, 2016.

[23] Md Jahidul Islam, Chelsey Edge, Yuyang Xiao, Peigen Luo, Muntaqim Mehtaz, Christopher Morse, Sadman Sakib Enan, and Junaed Sattar. Semantic segmentation of underwater imagery: Dataset and benchmark, 2020.

[24] Jianshu Li, Jian Zhao, Yunchao Wei, Congyan Lang, Yidong Li, Terence Sim, Shuicheng Yan, and Jiashi Feng. Multiple-human parsing in the wild, 2017.

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

[26] Si Liu, Zitian Wang, Yulu Gao, Lejian Ren, Yue Liao, Guanghui Ren, Bo Li, and Shuicheng Yan. Human-centric relation segmentation: Dataset and solution, 2021.

[27] Wei Liu, Andrew Rabinovich, and Alexander C. Berg. Parsenet: Looking wider to see better. *CoRR*, abs/1506.04579, 2015.

[28] Yan Liu, Qirui Ren, Jiahui Geng, Meng Ding, and Jiangyun Li. Efficient patch-wise semantic segmentation for large-scale remote sensing images. *Sensors (Basel, Switzerland)*, 18, 2018.

[29] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[30] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[31] Bjoern H. Menze, Andras Jakab, Stefan Bauer, Jayashree Kalpathy-Cramer, Keyvan Farahani, Justin Kirby, Yuliya Burren, Nicole Porz, Johannes Slotboom, Roland Wiest, Levente Lanczi, Elizabeth Gerstner, Marc-André Weber, Tal Arbel, Brian B. Avants, Nicholas Ayache, Patricia Buendia, D. Louis Collins, Nicolas Cordier, Jason J. Corso, Antonio Criminisi, Tilak Das, Hervé Delingette, Çağatay Demiralp, Christopher R. Durst, Michel Dojat, Senan Doyle, Joana Festa, Florence Forbes, Ezequiel Geremia, Ben Glocker, Polina Golland, Xiaotao Guo, Andac Hamamci, Khan M. Iftekharuddin, Raj Jena, Nigel M. John, Ender Konukoglu, Danial Lashkari, José António Mariz, Raphael Meier, Sérgio Pereira, Doina Precup, Stephen J. Price, Tammy Riklin Raviv, Syed M. S. Reza, Michael Ryan, Duygu Sarikaya, Lawrence Schwartz, Hoo-Chang Shin, Jamie Shotton, Carlos A. Silva, Nuno Sousa, Nagesh K. Subbanna, Gabor Szekely, Thomas J. Taylor, Owen M. Thomas, Nicholas J. Tustison, Gozde Unal, Flor Vasseur, Max Wintermark, Dong Hye Ye, Liang Zhao, Binsheng Zhao, Darko Zikic, Marcel Prastawa, Mauricio Reyes, and Koen Van Leemput. The multimodal brain tumor image segmentation benchmark (brats). *IEEE Transactions on Medical Imaging*, 34(10):1993–2024, 2015.

[32] Ilias Papadeas, Lazaros Tsochatzidis, Angelos Amanatiadis, and Ioannis Pratikakis. Real-time semantic image segmentation with deep learning for autonomous driving: A survey. *Applied Sciences*, 11(19), 2021.

[33] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation, 2016.

[34] Vasilis Pollatos, Loukas Kouvaras, and Eleni Charou. Land cover semantic segmentation using resunet, 2020.

[35] Rudra P K Poudel, Stephan Liwicki, and Roberto Cipolla. Fast-scnn: Fast semantic segmentation network, 2019.

[36] Balakrishnan Ramalingam, Abdullah Aamir Hayat, Mohan Rajesh Elara, Braulio Félix Gómez, Lim Yi, Thejus Pathmakumar, Madan Mohan Rayguru, and Selvasundari Subramanian. Deep learning based pavement inspection using self-reconfigurable robot. *Sensors*, 21(8), 2021.

[37] O. Ronneberger, P.Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]).

[38] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M. Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[39] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. 2018.

[40] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning, 2016.

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

[42] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[43] WikiChip. Fsd chip - tesla, 2021.

[44] Xiongwei Wu, Xin Fu, Ying Liu, Ee-Peng Lim, Steven C. H. Hoi, and Qianru Sun. A large-scale benchmark for food image segmentation. *CoRR*, abs/2105.05409, 2021.

[45] Changqian Yu, Changxin Gao, Jingbo Wang, Gang Yu, Chunhua Shen, and Nong Sang. Bisenet v2: Bilateral network with guided aggregation for real-time semantic segmentation, 2020.

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

[47] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[48] Zhendong Zhang, Xinran Wang, and Cheolkon Jung. Dcsr: Dilated convolutions for single image super-resolution. *IEEE Transactions on Image Processing*, 28:1625–1635, 2019.

[49] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[50] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.