Real-Time Instance and Semantic Segmentation Using Deep Learning

by

Dhanvin Kolhatkar

Thesis submitted to the
University of Ottawa
In partial fulfillment of the requirements
For the M.A.Sc. degree in
Electrical and Computer Engineering

Under the supervision of Professor Robert Laganière

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

© Dhanvin Kolhatkar, Ottawa, Canada, 2020

Abstract

In this thesis, we explore the use of Convolutional Neural Networks for semantic and instance segmentation, with a focus on studying the application of existing methods with cheaper neural networks. We modify a fast object detection architecture for the instance segmentation task, and study the concepts behind these modifications both in the simpler context of semantic segmentation and the more difficult context of instance segmentation. Various instance segmentation branch architectures are implemented in parallel with a box prediction branch, using its results to crop each instance's features. We negate the imprecision of the final box predictions and eliminate the need for bounding box alignment by using an enlarged bounding box for cropping. We report and study the performance, advantages, and disadvantages of each. We achieve fast speeds with all of our methods.

Dans cette thèse, nous explorons l'utilisation des réseaux neuronaux convolutifs pour la segmentation des classes et la segmentation des objets, avec emphase sur l'étude de l'application de méthodes existantes avec des réseaux neuronaux moins coêteux. Nous modifions une architecture de détection rapide d'objets pour la segmentation des objets et étudions les concepts qui justifient ces modifications dans le contexte plus simple de la segmentation des classes, et dans le concept plus complexe de la segmentation des objets. Diverses architectures de segmentation sont implémentées en parallèle avec la branche de détection d'objets, utilisant les résultats de celle-ci pour découper les traits qui correspondent à chaque objet. Nous réduisons les problèmes de précision et éliminons l'alignement des boîtes en utilisant une boîte agrandie pour découper les traits. Nous présentons et étudions la précision, les avantages, et les désavantages de chaque architecture. Nous atteignons des vitesses de prédiction rapides pour chacune de nos méthodes.

Acknowledgements

I would like to thank my supervisor, Professor Robert Laganière, for his guidance throughout this project. I am grateful to Rytis Verbickas, Jonathan Blanchette, Prince Kapoor, Erlik Nowruzi, Wassim El Ahmar, and all the other members of the VIVA lab for helping me complete this thesis.

I would like to acknowledge the Fonds de Recherche du Québec (FRQ) for their financial support.

I want to thank my parents and my siblings for their support, encouragement, and advice, not only throughout the last three years, but for many years before that.

I want to thank Gildas Vinet and Fredéric Lavoie for our endless discussions about various scientific endeavors, technological innovations, and many other topics. I also want to thank Simon Bourbonnais and Mathieu Lebeau-Bérubé for their priceless friendship over more than a decade.

Finally, I want to thank Éric Carrière, John Bain and Geoff Robinson for teaching me to think for myself.

”Ignorance is a wonderful thing - it’s the state you have to be in before you can really learn anything.”

- Terry Pratchett

Table of Contents

List of Figures ix

List of Tables xii

Nomenclature xiv

1 Introduction 1

1.1 Semantic Segmentation 2

1.2 Instance Segmentation 2

1.3 Thesis Objective 3

1.4 Thesis Contribution 3

1.5 Thesis Structure 4

2 Literature Review 6

2.1 Neural Networks 6

2.1.1 Neurons 7

2.1.2 Activation Function 8

2.1.3 Loss Function 10

2.1.4 Backpropagation 11

2.1.5 Regularization 11

2.2 Convolutional Neural Networks 12

2.2.1 Convolutional Layers . . . 13
2.2.2 Fully Connected Layers . . . 19
2.2.3 Pooling Layers . . . 20
2.2.4 Transposed Convolutional Layers . . . 21
2.2.5 Transfer Learning . . . 21
2.3 Classification Networks . . . 23
2.3.1 AlexNet . . . 24
2.3.2 ResNet . . . 25
2.3.3 SqueezeNet . . . 27
2.3.4 MobileNet . . . 29
2.4 Object Detection Networks . . . 33
2.4.1 Region proposal with CNNs (R-CNN) . . . 34
2.5 Semantic Segmentation Networks . . . 37
2.5.1 Fully Convolutional Networks for Semantic Segmentation . . . 39
2.5.2 DeepLab . . . 42
2.6 Instance Segmentation Networks . . . 44
2.6.1 Mask R-CNN . . . 45
3 Single Shot Detector for Instance Segmentation **48**
3.1 SSD-ISB Architecture . . . 49
3.1.1 Single Shot Detector (SSD) . . . 49
3.1.2 Instance Segmentation Branch (ISB) . . . 51
3.1.3 Encompassing Bounding Box . . . 54
3.1.4 Single Branch Instance Segmentation . . . 58
3.1.5 Multiple Branch Instance Segmentation . . . 60
3.2 Implementation Details . . . 62

3.2.1 Network Details and Parameters 63
3.3 Conclusion 65

4 Semantic Segmentation with Low-Cost Networks 67
4.1 Dataset 67
4.1.1 Cityscapes 68
4.2 Network Comparison with Cityscapes Dataset 69
4.2.1 Comparison Details 70
4.2.2 Implementation Details 71
4.2.3 Experimentation Results 72
4.3 Cityscapes Person Only Segmentation 78
4.3.1 Comparison Details 78
4.3.2 Implementation Details 81
4.3.3 Experimentation Results 82
4.4 Conclusion 85

5 Instance Segmentation with SSD-ISB Architectures 87
5.1 Dataset 87
5.1.1 MS-COCO 88
5.1.2 Training and Validation subsets 90
5.2 Experimentation Results 90
5.2.1 COCO Subset from ISB Shapes 91
5.2.2 COCO Person 94
5.2.3 Speed and Memory Requirements 96
5.2.4 Additional Tests 98
5.3 Conclusion 103

6 Conclusions and Future Work **104**

6.1 Conclusions 104

6.2 Future Work 105

References **107**

List of Figures

2.1 A neural network with two hidden layers. 7

2.2 An example of a convolutional neural network. 13

2.3 Example of two subsequent steps in one 3x3 two-dimensional convolution; the depth dimension is omitted for clarity, as the input filter would simply cover the entire depth and there is a single output feature map for each convolutional filter. 14

2.4 Example of two subsequent steps in one 3x3 two-dimensional convolution with a stride parameter of 2; note the decrease in the feature map height and width by a factor equal to the stride parameter. 16

2.5 Example of two subsequent steps in one 3x3 two-dimensional convolution with an atrous rate of 1; note that the output is unchanged when compared to a regular two-dimensional convolution. 18

2.6 Example of two subsequent steps in one 3x3 two-dimensional transposed convolution. The opaque squares in the input feature map show the field of view of the 3x3 filter, while the coloured square indicates the center of the filter. 22

2.7 Shortcut connection as implemented in ResNet 26

2.8 Structure of a regular convolution compared to a depthwise separable convolution. 30

2.9 R-CNN region proposal architecture. Note that the cropping step and the steps that follow are repeated for each region proposal. 35

2.10 Fast R-CNN region proposal architecture. Note that the RoIPool step and the steps that follow are repeated for each region proposal. 36

2.11 Faster R-CNN region proposal architecture. Note that the RoIPool step and the steps that follow are repeated for each region proposal. 37

2.12 Semantic Segmentation example from the Cityscapes dataset. 38

2.13 FCN skip architecture, showing variants FCN-32s, FCN-16s and FCN-8s. . 40

2.14 Instance segmentation annotation example from the MS-COCO dataset. . 45

2.15 Mask R-CNN region proposal architecture. Note that the RoIAlign step and the steps that follow are repeated for each region proposal. 45

2.16 A comparison of RoIPool - used in Faster R-CNN - and RoIAlign - used in Mask R-CNN - for the generation of a 2x2 feature map; note that the region proposal with RoIAlign is perfectly aligned with the region proposal in the input feature map. 46

3.1 SSD MobileNet architecture. 50

3.2 SSD default anchor examples for two grid sizes. Anchors are shown for one cell only. 51

3.3 Full architecture of the Single Shot Detector adapted for Instance Segmentation. 52

3.4 Instance segmentation branch architecture. 53

3.5 The encompassing bounding box (bold) generated from the accurate bounding box (dashed). 54

3.6 Image examples when using the *encompassing bounding box*. The black box shows the ground truth and the green box shows the encompassing box. . . 55

3.7 Example from the COCO val dataset of two person being segmented as the same person. 57

3.8 Single branch instance segmentation with SSD box predictions. 58

3.9 Full architecture of the Single Shot Detector adapted for Instance Segmentation with multiple instance segmentation branches. 61

4.1 Image examples from the Cityscapes dataset. 69

4.2 Sample output with all 19 Cityscapes classes, for the FCN8 architectures and the default architectures of the MobileNet and SqueezeNet encoders. . 74

4.3 Sample output with all 19 Cityscapes classes, for the FCN8 architecture and the output stride 8 architecture of the MobileNet encoder. 75

4.4 A comparison of the different additional background options compared in this section. 80

4.5 A comparison of the output for different additional background options with no skip architecture, an output stride of 32, and an atrous rate of 3 in the last layer. 83

4.6 A comparison of the output for the different segmentation methods: output stride 8, FCN-16 and FCN8 respectively, all with an atrous rate of 3 in the last layer. 83

5.1 Image examples from the COCO dataset. 89

5.2 An example of an image from the COCO dataset with *Single-Shape* predictions; black boxes indicate the groundtruth, green boxes indicate the predictions. 93

5.3 Sample predictions of the *Multi-Shape* architecture on the COCO person dataset; black boxes indicate the groundtruth, green boxes indicate the predictions. 96

List of Tables

2.1 Detailed Architecture of the AlexNet network [1]. 25

2.2 Shortcut block depth for 50, 101 and 152 layer versions of ResNet [2]. . . . 27

2.3 Detailed Architecture of the ResNet-v1 network [2]. 27

2.4 Detailed Architecture of the SqueezeNet network [3]. 29

2.5 Detailed Architecture of the MobileNet-v1 network [4]. 32

2.6 Results of the MobileNet-v1 network on the ImageNet classification task [4] for an input of size 224x224. 33

2.7 Results of other networks on the ImageNet classification task [4]. 33

2.8 Results of variants of the FCN-VGG16 architecture on a subset of the PASCAL VOC 2011 val set 42

2.9 Results of varying the output stride of ResNet-50 with 8 blocks 44

4.1 Mean IoU results for the SqueezeNet/MobileNet comparison study of semantic segmentation on Cityscapes. 73

4.2 Speed and memory usage for SqueezeNet/MobileNet comparison study of semantic segmentation on Cityscapes. 77

4.3 IoU results of MobileNet for person-only study; baseline 82

4.4 IoU results of MobileNet for person-only study; FCN8 configuration. 84

4.5 IoU results of MobileNet for person-only study; FCN16 configuration. . . . 84

4.6 IoU results of MobileNet for person-only study; atrous convolutions with stride 8. 85

5.1 The distribution of person bounding boxes in the COCO dataset (height by width) for a 20x20 feature map, with encompassing bounding boxes, for the training and validation subsets. 91

5.2 Precision and Recall results on the COCO-subset from ISB shapes for an IoU threshold of 0.50. 92

5.3 Precision and Recall results on the COCO-subset from ISB shapes for an IoU threshold of 0.50, using only the annotations with a matching ISB shape for both training and validation. 94

5.4 Precision and Recall results on the COCO person annotations for an IoU threshold of 0.50. 95

5.5 Speed and memory usage for the various SSD-IS architectures. 97

5.6 Precision and Recall results on the COCO person annotations for an IoU threshold of 0.50; comparison between various Single-Branch parameters. . 99

5.7 Speed and memory usage for the Single-Branch and Multi-Shape SSD-IS architectures' comparative tests. 100

5.8 Precision and Recall results on the COCO person annotations for an IoU threshold of 0.50; comparison between various Multi-Shape parameters. . . 101

5.9 Speed and memory usage for the Single-Branch and Multi-Shape SSD-IS architectures' comparative tests. 101

Nomenclature

Abbreviations

AP	Average Precision
AR	Average Recall
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CRF	Conditional Random Field
DCNN	Deep Convolutional Neural Network
DSC	Depthwise Separable Convolution
FC	Fully Connected
FCN	Fully Convolutional Network
FN	False Negatives
FP	False Positives
FPS	Frames Per Second
GPU	Graphics Processing Unit
IoU	Intersection over Union
ISB	Instance Segmentation Branch
m	meters
mIoU	mean Intersection over Union
NMS	Non-Maximum Suppression
R-CNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
RoI	Region of Interest
RPN	Region Proposal Network

TP True Positives

Chapter 1

Introduction

Over the last decade, many fields related to artificial intelligence have benefited from significant developments in *deep learning*: the *machine learning* methods built from a succession of feature extraction layers. An important field among these is *Computer Vision*, which includes tasks such as object classification, object detection, segmentation, keypoint detection, and depth estimation. Of particular interest to this work are the semantic segmentation and instance segmentation tasks: respectively, identifying the object class of each pixel in an image and identifying the object instance of each pixel in an image.

The majority of deep learning approaches focus on the accuracy of the results often lacking in emphasis on speed or computational requirements. While this is an inherent issue of deep learning as a concept, some methods still succeed in balancing accuracy and speed. For the particular case of segmentation, we see a tendency to use costly feature extractors, or to expand on slow, multi-stage, architectures. This severely limits the usefulness of most state-of-the-art work for common, sometimes specialized, applications, which are implemented in devices with limited resources, such as embedded systems and mobile phones.

This leaves an open question as to the usability of current state-of-the-art approaches for the semantic and instance segmentation tasks, inspired by speed-focused methods for the image classification and the object detection tasks.

1.1 Semantic Segmentation

Semantic segmentation is the task of recognizing objects in an image at a *pixel* level. In other words, the task is to assign a label to each pixel in an image, indicating the object class of that pixel.

The first major type of method for semantic segmentation with deep learning uses an *encoder-decoder* architecture: the *encoder*, generally based on state-of-the-art image classifiers, extracts features from an image, while the *decoder*, which is often much shallower than the encoder, extracts a segmentation mask[1] from said features. The second type of method increases the memory cost and the computational cost of the feature extractor by increasing the size of the feature maps, thereby generating more accurate segmentation masks without using a decoder.

Semantic segmentation work that uses deep neural networks is often based on the category of the more expensive image classifiers, thereby making both types of methods expensive to implement. However, both types should be usable with less costly image classifiers. In particular, the use of shallow decoder with a cheap image classifier should result in a fairly light semantic segmentation network.

Additionally, the task of instance segmentation can be seen as a more stringent variation of semantic segmentation, due to the added challenge of separating individual instances within each object class. As such, work on reducing the computational cost of semantic segmentation networks proved to be very useful in devising more efficient approaches for instance segmentation.

1.2 Instance Segmentation

Instance segmentation is a task heavily related to that of semantic segmentation. The goal is not only the recognition of an object's class, but also the differentiation of instances within said class. In other words, the task is to label each object in an image at a pixel

[1]For the task of semantic segmentation, the segmentation mask labels the type of object in each pixel of the input image.

level, differentiating each person, each car, each bicycle, etc. at the most precise level possible.

In this work, we focus on instance segmentation methods that are based on state-of-the-art object detection methods. These can separate object instances using the existing bounding box prediction architecture and modify it to generate an instance segmentation mask[2] for each individual object in an image.

Specifically, we are interested in coupling fast, single-stage, object detection methods with a simple segmentation architecture, thereby creating a low-cost approach for instance segmentation. The resulting instance segmentation architecture would be interesting to compare to more costly architectures based on slower object detection methods, which generally include a dedicated region proposal stage.

This also leaves other open questions regarding the kind of speed and accuracy compromises that can be achieved for the task of instance segmentation. As such, architectures of varying depth, complexity, and computational cost are tested and compared to each other.

1.3 Thesis Objective

The objective of this work is to study how current approaches to segmentation tasks can benefit from the recent advancements in deep learning without becoming as computationally costly as most recent work in the field. This is a particularly interesting objective to tackle because generating a pixel-precise mask prediction is by definition a memory-intensive task.

1.4 Thesis Contribution

This thesis brings the following contributions of this field:

- Applying methods used to improve semantic segmentation architectures to less computationally expensive image classification neural networks, and studying how these

[2]For the task of instance segmentation, the segmentation mask labels each individual object separately with pixel-precision.

can apply to an object detection-based instance segmentation framework;

- Proposing an instance segmentation architecture usable in low computational cost use-cases by adapting a single-stage object detection method;
- Studying the speed and accuracy compromises that can be attained with our proposed instance segmentation architecture, while maintaining significantly higher speeds and lower model size than state-of-the-art approaches;
- Presenting an alternative solution to the alignment issue within object detection networks used for instance segmentation by enlarging predicted bounding boxes;
- Studying the benefits and the cost of using parallel branches of different sizes for instance segmentation instead of resizing all objects to a single size.

To our knowledge, we propose the first instance segmentation deep learning model that maintains high speed and low model size. Both of these are necessary for implementation in low-power embedded devices.

1.5 Thesis Structure

The structure of this thesis is as follows:

Chapter 1 is this *Introduction*, which quickly explains the context and problem that motivated this work and the solutions it studied.

Chapter 2 is a *Literature Review*, covering the theory and some recent important work in the field of deep learning. We start by explaining some important concepts related to neural networks and convolutional neural networks, with some emphasis on the parts important to segmentation. We then cover some of the recent literature, relevant either in its inspiration of other work or in its direct application to the work presented in this thesis. We cover research in the tasks of image classification, object detection and instance segmentation.

Chapter 3 details our network proposal, based on using a single-stage object detection architecture for *Instance Segmentation*, and covers various approaches for instance segmentation without a dedicated region proposal stage for box prediction. Here, we study various

ways of using box predictions for mask prediction: using default anchors, using resizing predicted boxes, and using parallel segmentation branches. This last approach bypasses the need to crudely resize the instances and thus lose their aspect ratio by using multiple Instance Segmentation Branches (ISB's) of different sizes in parallel. We also present ways of mixing the latter two approaches.

Chapter 4 details our research related to the task of *Semantic Segmentation*. It covers the seminal work in this field and our efforts in applying state-of-the-art methods to smaller neural networks, studying their accuracy, their speed, and their memory cost. Most importantly, we study the effects of using semantic segmentation methods in situations that approximate the task of instance segmentation by cropping pedestrian instances from the Cityscapes dataset. This also serves to inform us on how well the network can deal under specific circumstances.

Chapter 5 details our experiments on the COCO person instance annotations, resulting from the work presented in chapters 3. We run various person instance segmentation experiments on the model proposed in Chapter 3 and all of its proposed variations. We show that our model achieves fast speed and low model size while achieving reasonable results, making it an ideal candidate for use in embedded systems.

Chapter 6 is the *Conclusion*, restating the main conclusions and contributions of this work.

Chapter 2

Literature Review

Major developments in computer vision have resulted from the increased use of neural networks. This chapter covers important notions for their design and details the most relevant work in the computer vision fields most directly related to ours.

We first cover Neural Networks (NN) in Section 2.1 followed by Convolutional Neural Networks in Section 2.2, introducing the general ideas used to build these networks. In Section 2.3, we detail the important image classification convolutional neural networks which are commonly retasked for use in semantic segmentation, object detection and instance segmentation. Recent advances using CNN for these three tasks are then covered in sections 2.4, 2.5 and 2.6 respectively.

2.1 Neural Networks

The architecture of neural networks was originally based on current theories on how human neurons transmit information [5, 6]. The basic idea is that electrical signals are sent as inputs to a neuron, with these signals being more or less attenuated during transmission. These signals are then all sent out to other neurons.

By analogy, the attenuation is a multiplication by a certain *trainable* parameter, which we call a *weight*. We sum all these products and then perform an addition with another trainable parameter, which we call a *bias*. Putting this sum through an activation function gives us the output of the neuron. A sequence of these neurons would then be a neural

network. The value of the weights and biases can be trained to produce the output that we want for a certain input. An example of a neural network with two layers of neurons can be seen in Figure 2.1.

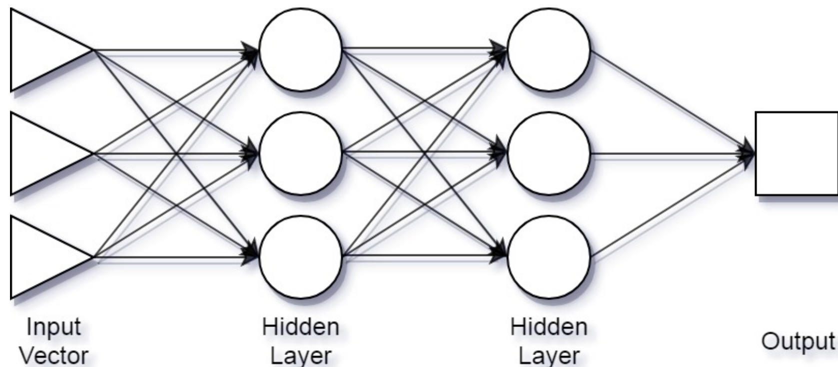


Figure 2.1: A neural network with two hidden layers.

The following subsections will detail these elements: Section 2.1.1 covers the neuron's equations, Section 2.1.2 covers the activation functions mentioned in the previous paragraph, Section 2.1.3 covers details on how we calculate the performance of the network, Section 2.1.4 covers the optimization of the parameters in the network - *i.e.* the weights and the biases - and Section 2.1.5 covers some ways of controlling the changes in those parameters.

2.1.1 Neurons

The operation implemented by each neuron in a neural network was described above by analogy, but it is also written more clearly in Equation 2.1 [6]. Function f is the activation function, which will be detailed in Section 2.1.2, b is the bias, w_i and x_i are one of the weights and its corresponding input, respectively.

$$o = f\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

When we're building a neural network, we'll place multiple such neurons in parallel. We refer to all of these neurons as a layer. Each of them will be connected to all the x_i inputs, but the weights and biases for each neuron are independent and will be optimized as such. Through this training, each neuron should learn its own type of relevant information.

We can put Equation 2.1 in matrix form, as can be seen in Equation 2.2. Since we have multiple neurons in parallel which all depend on the same x inputs, this form can also be used to express the outputs of all the neurons of a layer. In this case, W is a $[m \times n]$ matrix of weights, x is a $[n \times 1]$ vector of input values, b is a $[m \times 1]$ vector of biases and o is a $[m \times 1]$ vector of output values. Function f refers to the same activation function. Note that there are no restrictions on the values of n and m: defining them is an important part of designing a neural network.

$$o = f\left(Wx + b\right) \tag{2.2}$$

We then place multiple layers in a row to create a neural network, where each layer is connected to the previous layer's outputs, except the first layer which is connected to the network's inputs. The final layer's outputs are also the output of the network. Note that this type of layer, which is commonly called a *hidden layer*, is used in convolutional neural networks as well, where it is called a *fully connected* layer, as each neuron is connected to each value in the input vector. This is mentioned again in Section 2.2.2.

2.1.2 Activation Function

The other important building block for neural networks is the activation functions that we place after each neuron. An interesting property of the matrix multiplication used in neural networks is that, due the linear nature of the operation, any number of successive neurons could be reduces to a single neuron. Since this would nullify the network's effectiveness, we introduce non-linear functions after each layer. In recent literature [1, 2, 4, 7, 8], the most common of these is the Rectified Linear Unit - or ReLU - which are explained in this section. Two other important activation functions for this work are the Softmax activation and the Sigmoid activations, which are also detailed in this section. Note that these functions are also used in convolutional neural networks which are the main focus of this work.

ReLU

The ReLU function is shown in Equation 2.3 [1].

$$f(x) = max(0, x) \tag{2.3}$$

Effectively, this means that any negative input will be increased to 0 while any positive or zero input will stay the same. This type of activation function has shown great results since being first used in AlexNet [1] which reported training to be six times faster than when using tanh as the activation function. The main drawback here is that the ReLU operation can cause some weights to never activate, as they will consistently be set to 0, but it is possible to monitor this issue.

Softmax

Equation 2.4 is the Softmax function [6].

$$f(x) = \frac{e^x}{\Sigma_i e^{x_i}} \tag{2.4}$$

A softmax activation layer is commonly used in the last layer of a network for classification. As this operation squashes its input to values between zero and one that sum to one, it generating outputs that can be interpreted as normalized probabilities. It is used at the end of every classification network detailed in Section 2.3 and every segmentation network detailed in Section 2.5.

Sigmoid

Equation 2.5 is the Sigmoid function [6].

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

The sigmoid function has similar properties to the softmax function: its outputs are between zero and one. The sigmoid activation function has one significant drawback: very large inputs, be they positive or negative, will generate a very small gradient during backpropagation. This can severely limit the effectiveness of the network's training. Nevertheless, they are used as a binary classification operation in *Mask R-CNN* [9], making it very relevant to this work.

2.1.3 Loss Function

We often refer to the *training* of the parameters in a neural network for a specific task such as image classification: we give an image and want the network to output the class of the predominant object in the image. This is typically implemented through a loss function, which expresses the error between the network's prediction and the expected result - the *groundtruth*. The main loss function used throughout this work is the *cross-entropy* loss for image and pixel classification, while Smooth L1 loss is used for bounding box regression in the object detection task.

Cross-Entropy

The general form of cross-entropy [6] is shown in Equation 2.6, where $g(x)$ stands for the groundtruth distribution and $p(x)$ stands for the measured distribution.

$$f(x, y) = -\Sigma g(x) \log p(x) \tag{2.6}$$

Since cross-entropy loss is commonly used as a classification loss, $g(x)$ often has a single expected 1 - the class to be predicted - and zeroes for the rest of the distribution. If a softmax operation is applied to the output of the network's final layer, cross-entropy loss can be formulated as follows:

$$f(x, y) = -\log\left(\frac{e^{x}}{\Sigma_i e^{x_i}}\right) \tag{2.7}$$

This results in a high loss when a low probability is associated with the expected class.

Smooth L1 Loss

While cross-entropy loss works quite well for classification tasks, another type of loss is required for regression tasks, which require prediction of real values as opposed to the prediction of a distribution. Smooth L1 loss [10] is commonly used [11, 12] and is defined in Equation 2.8, where p and g stand for the predicted value and the groundtruth value respectively.

$$\text{smooth}_{L1}(p,g) = \begin{cases} 0.5(p-g)^2, & \text{if } |x| < 1. \\ |p-g| - 0.5, & \text{otherwise.} \end{cases} \tag{2.8}$$

This smooth L1 loss is less impacted by outliers than typical L2 loss - the square of the difference - which can result in large loss and, consequently, in a large gradient which will harm the training of the network.

2.1.4 Backpropagation

The main concept behind the training of neural networks is that of *backpropagation*. The idea is that, within a single neuron, we want to go backwards from the outputs and calculate how each input contributed to their respective output. Effectively, this is done through a recursive use of the chain rule: each layer can be represented as a sequence of operations for which we can calculate the gradient, letting us backpropagate from the output of a layer to its input.

This in turn allows us to calculate how we should change the inputs - or, more specifically, the weights we multiply with, and the biases we add to, the inputs - to get a different output. The input of a certain neuron being the output of another, with the exception of the first layer, we can thus backpropagate the network's loss through the entire neural network and optimize every single parameter.

Backpropagation is tied to two hyper-parameters of the network: the learning rate and the optimizer. The learning rate is multiplied by the gradient to control how much weights change between two successive training steps: a learning rate too low will train slower than ideal, or even fail to train properly, while one too high will fail to converge to the best solution. The optimizer controls the way that the learning rate is applied to the weights.

2.1.5 Regularization

One final component of designing a neural network is that of regularizing the weights, which solves one main issue with deep learning: that of overfitting to the training subset.

This is implemented either by adding a regularization element to the loss function or by using methods such as *Dropout*.

L2 Regularization

L2 regularization is commonly used to increase loss more significantly when weights are large, as shown in Equation 2.9.

$$f(x) = 0.5 \times \lambda_R \times w^2 \tag{2.9}$$

The term $f(x)$ in Equation 2.9 is added to the loss function for each weight in the network. By adding the square of each weight to the loss, the network will optimize more evenly, rather than relying on a few high parameters throughout the network.

Dropout

As opposed to L2 Regularization, *Dropout* [13] is a method which controls the ability to fully use the layers during training. Each neurons in the network has a certain probability p of being used normally and a complementary probability $1 - p$ of being deactivated for the current training step. If deactivated, the neuron output is set to zero and the neuron itself is not used for prediction. This prevents the network from relying too heavily on any single neuron which can be randomly disabled during training. Note that this process is not used during evaluation.

2.2 Convolutional Neural Networks

The neural networks described in Section 2.1 take 1-dimensional inputs and have 1-dimensional neuron outputs within the network - including the neural network's final output. The convolutional neural networks (CNNs) detailed in this section will instead take a 3-dimensional input and use 3-dimensional feature maps within most, if not all, of the network, as seen in Figure 2.2.

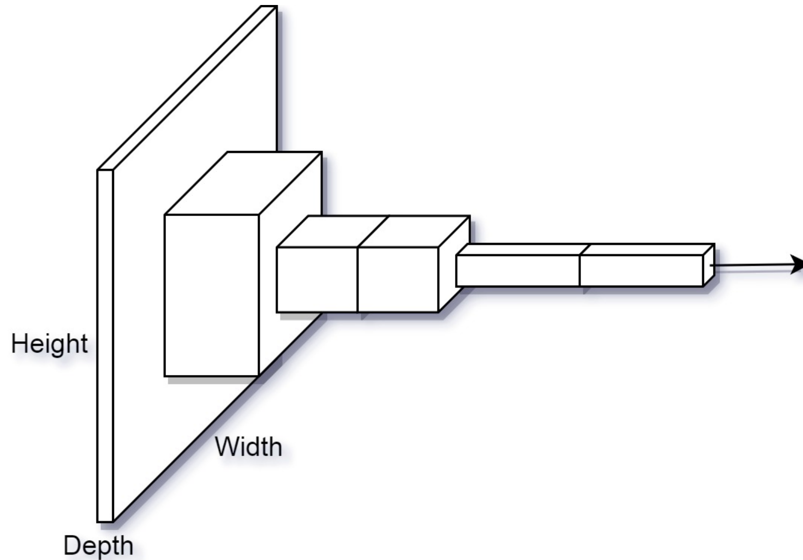


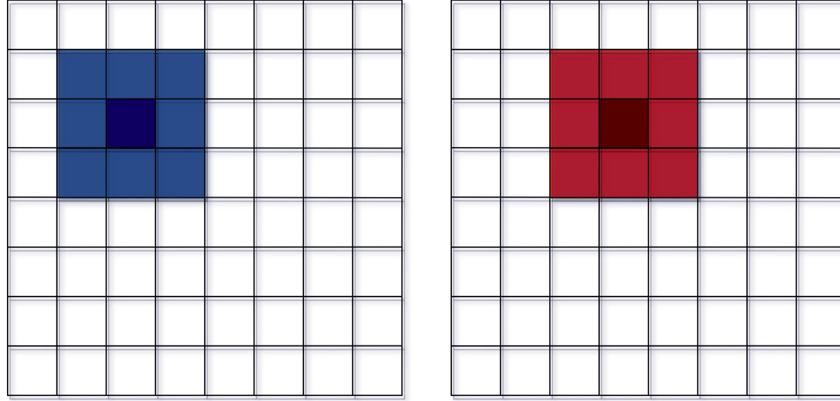
Figure 2.2: An example of a convolutional neural network.

We will discuss the advantages of using convolutional layers, fully connected layers and pooling layers in sections 2.2.1, 2.2.2 and 2.2.3. Section 2.2.4 will cover the uses of transposed convolutions and Section 2.2.5 will explain how we can reuse a trained CNN for a different task.

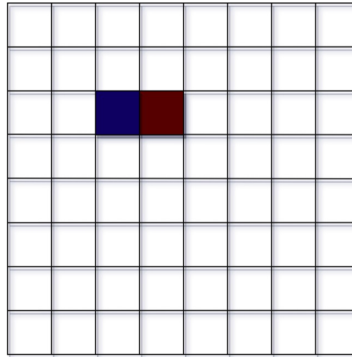
2.2.1 Convolutional Layers

The main building block of any CNN is the convolutional layer, in which each value of the output feature map in based on a limited number of values from the input feature map - see Figure 2.3. In this way, the number of parameters does not scale with the size of the input, as the weights in the kernel are reused through a convolution with the input feature map. It is worth noting here that the convolutional kernel is connected along the entire depth of the input feature map. It is only along the first and second dimensions that we restrict the kernel's size.

In other words, the number of parameters in the convolutional layer for a certain $c \times c$ kernel with input depth d_i and output depth d_o can be calculated as per Equation 2.10 [4]. This accounts for $c \times c \times d_i \times d_o$ weights and the d_o biases.



(a) Input feature maps with the 3x3 filter overlayed.



(b) Output feature map, with colours that match the 3x3 filters in the input feature map.

Figure 2.3: Example of two subsequent steps in one 3x3 two-dimensional convolution; the depth dimension is omitted for clarity, as the input filter would simply cover the entire depth and there is a single output feature map for each convolutional filter.

$$c \times c \times d_i \times (d_o + 1) \tag{2.10}$$

From this, we can calculate the computational cost of a convolutional layer as shown in Equation 2.11 [4] where h_o and w_o stand for the height and width respectively of the output feature map.

$$c \times c \times d_i \times (d_o + 1) \times h_o \times w_o \tag{2.11}$$

Effectively, the computational cost is equal to the number of parameters multiplied by the number of calculations in the convolution.

The output of a 3x3 convolutional filter operation can be calculated as per Equation 2.12 [4] with output feature map G, convolutional kernel K and input feature map F. Indices i and j take values between 0 and c, while indices m and n take values between 0 and d_i, and 0 and d_o respectively.

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \times F_{k+i-1,l+j-1,m} \tag{2.12}$$

Spatial Awareness

A major advantage of convolutional layers is in looking only at pixels adjacent to the one that the convolutional kernel is centred on.

When trying to find patterns in an image, we can assume that the pixels adjacent to a pixel of interest will generally reveal some interesting information. While a neural network would discard the spatial information through its fully connected layers, a CNN will not. Instead, it will generate a 3D feature map where values along the height and width dimensions have a correlation with the pixels in the original image. In other words, we could crop an area from a feature map late in the network and know that it contains information pertaining to its respective area in the input image [14].

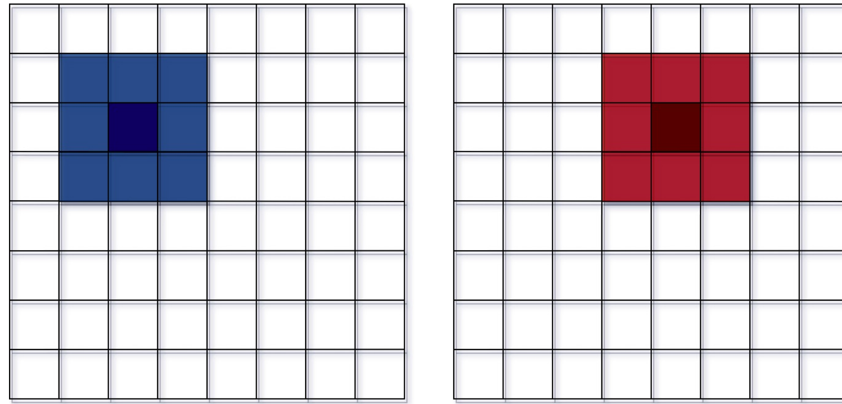
Padding

An important concept to discuss with regards to convolutional layers is that of padding. When we run a convolution for a pixel at the border of the feature map, we're trying to use values that are not in the feature map. This is either dealt with by zero-padding or by not doing any padding. In the former case, we add a border of zeroes around the feature map which will be used for the convolution. In the later case, we do not run the convolutions along the border of the feature map and will thus output a smaller feature map.

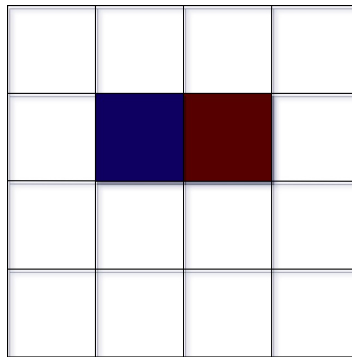
Stride

The second important concept in CNNs, particularly for segmentation tasks, is that of a layer's stride [15]. While convolutional filters are generally applied to adjacent pixel

locations of a feature map (i.e. with a stride of 1), we can also skip some of the pixels in this feature map. We will effectively downsample the input height and width by a factor equal to the striding parameter. Figure 2.4 shows this concept for a two-dimensional convolution.



(a) Input feature maps with the 3x3 filter overlayed.



(b) Output feature map, with colours that match the 3x3 filters in the input feature map.

Figure 2.4: Example of two subsequent steps in one 3x3 two-dimensional convolution with a stride parameter of 2; note the decrease in the feature map height and width by a factor equal to the stride parameter.

There are two main advantages to striding [16]. First, the use of striding throughout the network will reduce the network's memory requirements by reducing the number of values in each feature map. Second, it will effectively increase any layer's field of view by reducing the size of the feature maps. 32 is a very typical striding factor, though some networks will reduce this to 16 or 8. Reducing the striding factor further is possible, but the increase in the network's accuracy on these tasks is generally not worth the significant

increase in memory requirement.

Striding is often implemented in either convolutional layers or pooling layers - see Section 2.2.3. Section 2.3 details the architecture of a few image classifiers and their use of striding.

Feature Map Arithmetic

When designing a CNN, it is important to study the way that the feature map dimensions will be affected by the layout of the layers. Equation 2.13 [17] shows the output size of a convolutional layer, where i and o indicate either the input and output width respectively or the input and output height respectively. The output depth d_o will simply be equal to the number of convolutional kernels, represented by parameter k. Note that c is the kernel's size - we have a $c \times c$ filter - p is the number of amount of padding used in the layer and s is the layer's stride.

$$\begin{aligned} o &= \left\lfloor \frac{i - c + 2p}{s} \right\rfloor + 1 \\ d_o &= k \end{aligned} \tag{2.13}$$

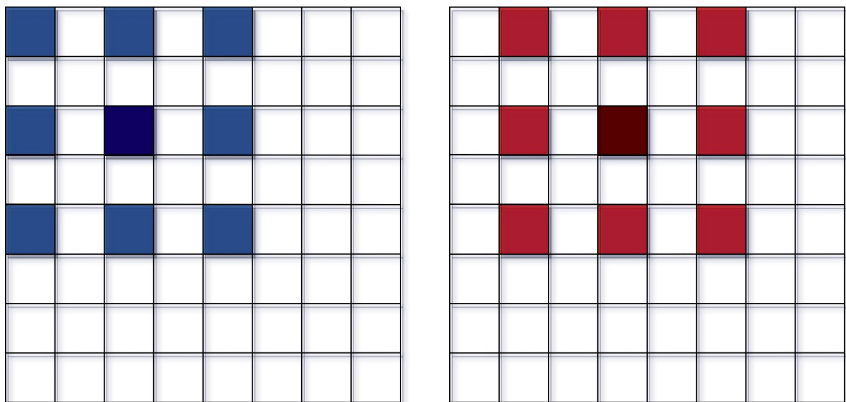
Atrous Convolution

Atrous convolution [16], sometimes called dilated convolution, is a subtype of convolution often used for computer vision. Of particular interest for this work are its uses for semantic segmentation, which we will define in Section 2.5.

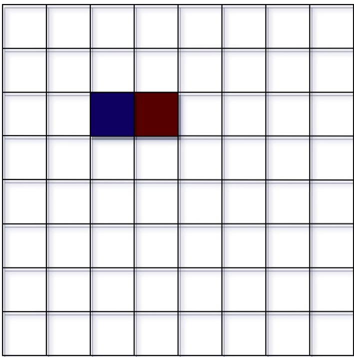
As mentioned above, the typical convolution filter will cover pixels adjacent to the pixel of interest. This effectively limits the convolution's field of view to the size of the kernel, though we can still cover a large part of an input image by downsampling the image throughout the network. However, because we are lowering the resolution, using downsampling also decimates the amount of detail contained in each feature map.

Using atrous convolution attempts to rectify this by looking at pixels that are farther away from the target pixel while using the same kernel and architecture as a convolutional layer. We introduce a dilation factor, or dilation rate, which has to be an integer larger or

equal to 1. This factor indicates how many adjacent values we will skip in each direction from the pixel of interest when calculating the convolution between the input feature map and the filter. This effectively increases the layer's field of view by the dilation factor without reducing the input's resolution, enabling us to generate much finer results for some of these tasks - see Figure 2.5 for an example of a 3x3 atrous convolution with dilation factor 2. More details can be found in Section 2.5.2, which presents details on the Deeplab network [16].



(a) Input feature maps with the 3x3 filter overlayed.



(b) Output feature map, with colours that match the 3x3 filters in the input feature map.

Figure 2.5: Example of two subsequent steps in one 3x3 two-dimensional convolution with an atrous rate of 1; note that the output is unchanged when compared to a regular two-dimensional convolution.

Similarly to a regular convolution, the output of a 3x3 atrous convolution can be calculated as per Equation 2.14 [4, 16] with output feature map G, convolutional kernel K and input feature map F. Indices i, j, m and n take values between 0 and c, 0 and c, 0

and d_i, and 0 and d_o respectively. The new parameter r is the chosen dilation factor for the atrous convolution. Note that for a dilation factor of 1, Equation 2.14 is equivalent to Equation 2.12.

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \times F_{k+(i-1)\times r, l+(j-1)\times r, m} \tag{2.14}$$

Atrous convolution has almost no drawbacks on its own, with a computational cost comparable to a regular convolution [16] and no increase in the number of parameters. However, we often use it in combination with a reduction of the network's stride because we want a higher resolution output: this will result in a higher memory requirement throughout the network.

2.2.2 Fully Connected Layers

It was stated in Section 2.2.1 that convolutional layers solve a lot of issues when using *fully connected* layers - which are identical to the *hidden layers* defined in Section 2.1.1 - with a large image. However, they are still commonly used towards the end of a CNN, taking advantage of the downsampling which significantly reducing the number of parameters we require to connecting to every single value in the feature map. It is here worth noting that, even when the feature map are downsampled to this extent, the fully connected layers will often contain a large amount, or even the majority, of the parameters in the entire network [1, 4]. Equation 2.15 shows the number of parameters in a fully connected layer where h_i, w_i and d_i are the input height, width and depth respectively and d_o is the output depth.

$$h_i \times w_i \times d_i \times (d_o + 1) \tag{2.15}$$

It is important to note that we can transform a fully connected layer into a convolutional layer. As mentioned previously, a convolutional layer is very similar to a fully connected layer except that it takes features from a subset of the feature map. However this subset can have the same size as the input feature map, in which case it operates exactly like a fully connected layer. As such, we can take the weights from a fully connected layer and

use them to define a convolutional layer, which will be usable with a larger feature map and will output a feature map of its own.

2.2.3 Pooling Layers

Pooling layers are another key type of layer in CNNs [6, 17]. They are used to to aggregate values over a certain field of view (e.g. a 2x2 grid). This is often combined with a stride equal to the grid size, effectively replacing multiple values with a single one. Various types of pooling can be used, though maximum pooling is the most common, replacing all values in the pooling grid with the largest one. Unlike the convolutional layers which calculate their output over the entire depth of the feature maps, the pooling layers are only calculated over the height and width of a feature map, and are applied separately along the depth. As such, the output's depth is equal to the input depth. Note that a pooling layer does not have any trainable parameters.

As we will discuss in Section 2.3.3, a pooling layer can be used over the entire feature map as a way to replace fully connected layers: this is called *global pooling*. This was used in SqueezeNet [3]; the network's focus on efficiency was incompatible with the high memory requirements of a fully connected layer, whereas a global pooling layer does not have this drawback.

Feature Map Arithmetic

Similar to the convolutional layers, we can study the feature map arithmetic of the pooling layers to easily calculate the shape of the layer's output. Equation 2.16 [17] shows the size of a pooling layer's output; as in Section 2.2.1, i and o indicate either the input and output width respectively or the input and output height respectively, as height and width will be calculated in the same way. Unlike the case of convolutional layers, the output depth d_o of a pooling layer is simply equal to the input depth d_i. Note that p is the size of the pooling grid in the same dimension as the input and output we are calculating and s is the stride factor for the pooling layer.

$$
\begin{aligned}
o &= \left\lfloor \frac{i-p}{s} \right\rfloor + 1 \\
d_o &= d_i
\end{aligned}
\tag{2.16}
$$

2.2.4 Transposed Convolutional Layers

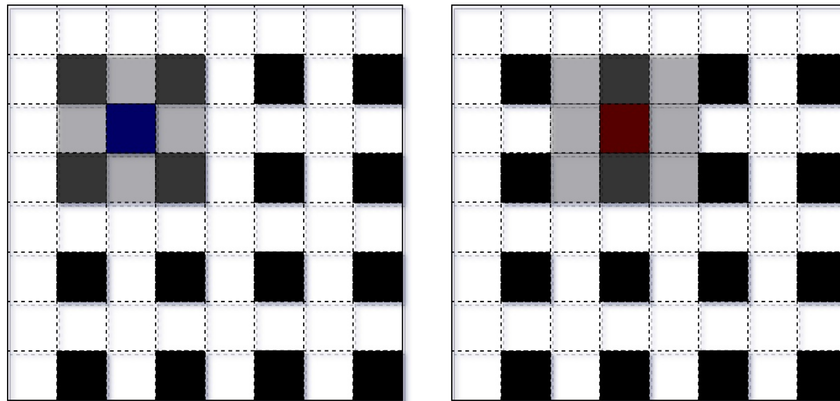
Some computer vision tasks require outputs with large height and width [15]. Because of this requirement, we want to increase the size of the feature maps towards the end of the network while taking advantages of the smaller feature maps throughout the network. This is a situation where *transposed convolutions* can be used [15, 18].

A transposed convolution can be described most simply as a backwards convolution [17]. That is, where a regular convolution with striding bigger than one would take an $H \times W \times D$ feature map and generate a smaller $h \times w \times d$ feature map, a transposed convolution will generate the larger feature map from the smaller feature map. Effectively, we are running the backward pass of a convolutional layer. This in turn is calculated by multiplying the smaller feature map with the *transpose* of the weights matrix. For example, for a regular convolution, a certain value in the input feature map might be used as a part of the calculation for nine output values. If we want to go the other way, we can use a filter that covers these nine output values and thus calculates the input value. This is shown in Figure 2.6.

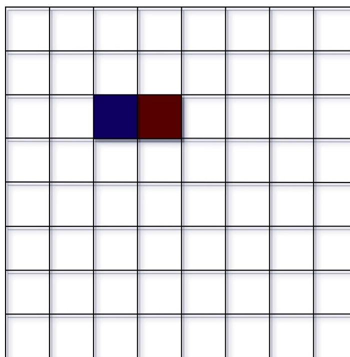
Section 2.5.1 will detail some of the first implementation details of transposed convolution for the task of semantic segmentation.

2.2.5 Transfer Learning

While CNNs are often first trained and designed for the task of image classification - see Section 2.3 - we often want to use pre-trained networks for other tasks, such as semantic segmentation or object detection - see sections 2.5 and 2.4 respectively. In this situation, we want to start from the weights trained for the first task - as mentioned, this is usually image classification - modify the network architecture as necessary for a new task, and continue training. The modifications include removing the final classification layer of the



(a) Input feature maps with the 3x3 filter overlayed. The black squares indicate the values from the input feature map, while the white ones indicate the zero-padding which simulates the 2x2 stride; applied to the input, unlike forward convolutions.



(b) Output feature map, with colours that match the 3x3 filters in the input feature map.

Figure 2.6: Example of two subsequent steps in one 3x3 two-dimensional transposed convolution. The opaque squares in the input feature map show the field of view of the 3x3 filter, while the coloured square indicates the center of the filter.

network, whose output is specific to the task at hand but can be more extensive. This is referred to as *fine-tuning*.

Of note, this can also be used to train on the same task but on a different dataset faster or with better accuracy [19, 20]. The idea is that the image classifier trained on, for example, the massive ImageNet dataset [21], will provide us with robust feature maps that can then be put through a few more layers to generate the output we are interested in. This can also limit the issues from using a small dataset, which might not contain enough data to train a network from scratch.

Specific uses of fine-tuning can be found in sections 2.5 and 2.4 on the tasks of semantic segmentation and object detection respectively. In section 2.5, we will also cover how to deal with fine-tuning a network that contains fully connected layers: where convolutional and pooling layers can be applied to feature maps of any size, fully connected layers need a fixed size input which prevents changes in the network's input size.

2.3 Classification Networks

While this work focuses on the task of segmentation, a lot of important developments in the field of CNNs were shown on the task of image classification. As such, this section will cover various classification networks, most of which will be mentioned again later in this chapter.

Image classification is the task of identifying the main object within an image from a list of possible object classes. A neural network used for this task will generate a vector of probabilities with one value for each of those class. The highest value in this vector indicates the predicted class.

The accuracy in this task is calculated from those predicted classes. We look at two main metrics: top-1 error rate and top-5 error rate. The former is based on how many times the expected class is not the predicted class. The latter is based on how many times the expected class is not in the five classes with the highest prediction value; it is a more forgiving metric than top-1 error rate. This metric is defined in Equation 2.17 for a number n of groundtruths per image - equal to 1 for ImageNet - where j iterates over the number of predicted labels - 1 or 5 for top-1 and top-5 error rate respectively - and where k iterates over n.

$$
\begin{aligned}
e_r &= \frac{1}{n} \times \Sigma_k \min_j d(l_j, g_k) \\
\text{where } d(x,y) &= \begin{cases} 0, & \text{if } x = y. \\ 1, & \text{otherwise.} \end{cases}
\end{aligned}
\tag{2.17}
$$

The networks covered in this section are AlexNet, ResNet, SqueezeNet and MobileNet. AlexNet is a seminal work in the field of computer vision and ResNet is a very important

network to study as it is commonly used for state-of-the-art results on various tasks. On the other hand, SqueezeNet and MobileNet are networks that sacrifice accuracy for faster computation time and lower memory requirements.

2.3.1 AlexNet

A seminal work in the field of deep learning, *AlexNet* [1] used GPUs to deal with the prohibitively large memory requirement of training a CNN. Using their network they achieved impressive results on the ImageNet dataset, motivating further research in the use of DCNNs for computer vision tasks. Specifically, their paper mentions a top-5 error rate of 15.3% for AlexNet compared to the 26.2% of its runner-up on the ILSVRC-2012 competition (using the test set of ImageNet dataset).

The network introduced multiple concepts of interest:

- Using Rectified Linear Units (ReLU) as a nonlinearity operation in the neural network. We discussed this in detail in Section 2.1.2 because it has become a mainstay of CNN architectures.
- Using multiple GPUs for training a network, especially with the 3GB of memory available on a GPU around AlexNet's time. More recently, the 12GB of memory available with a single GPU can still hamper the training of a deeper CNN, especially when working on a memory-intensive task such as segmentation (see Section 2.5).
- Using pooling operations with overlap (as opposed to using pooling operations on adjacent areas).

AlexNet is made up of five convolution layers with kernel sizes 11x11, 5x5, 3x3, 3x3 and 3x3, followed by three fully connected layers, as shown in detail in Table 2.1. Note that, for use with the ImageNet dataset, the last layer - *FC8* - has an output size of 1000.

This architecture is referred to in older papers, but its accuracy has since been matched by more efficient networks such as SqueezeNet [3] and MobileNet [4]. On the other hand, networks like VGG [7] and ResNet [22] have significantly improved upon its accuracy.

Layer Name	Input Dimensions ($W \times H \times D$)	Kernel ($w \times h \times d$/stride)
Conv1	$224 \times 224 \times 3$	$11 \times 11 \times 96/4$
Pool1	$55 \times 55 \times 96$	$3 \times 3/2$
Conv2	$27 \times 27 \times 96$	$5 \times 5 \times 256$
Pool2	$27 \times 27 \times 256$	$3 \times 3/2$
Conv3	$13 \times 13 \times 256$	$3 \times 3 \times 384$
Conv4	$13 \times 13 \times 384$	$3 \times 3 \times 384$
Conv5	$13 \times 13 \times 384$	$3 \times 3 \times 256$
Pool5	$13 \times 13 \times 256$	$3 \times 3/2$
FC6	$6 \times 6 \times 256$	$6 \times 6 \times 4096$
FC7	4096	$1 \times 1 \times 4096$
FC8	4096	$1 \times 1 \times 1000$

Table 2.1: Detailed Architecture of the AlexNet network [1].

2.3.2 ResNet

One of the more important modern DCNNs, ResNet [2] uses its *shortcut connections* to solve the problem of *vanishing gradients* [23, 24] when the very deep CNNs. Adding more and more layers will eventually result in poor training as the gradients become so small that the backpropagation step does not change values significantly enough to train the network. ResNet does not solve this problem, but does avoid it through the use of its shortcut connections. Variations of the ResNet network are presented with 18, 34, 50, 101 and 152 layers. This 152 layer version achieved 3.57% top-5 error rate on the ImageNet test set.

Shortcut Connections

ResNet implements residual learning in its design through its *shortcut connections*, which bypass any number of layers by adding a simple operation in parallel with these layers. An identity operation is used in the shortcut by the authors, and an addition is used to connect

the shortcut's output to the main path. In other words, if we represent the sequence of layers to be bypassed as $F(x)$, the output with the shortcut connection is $F(x) + x$. This is shown in Figure 2.7.

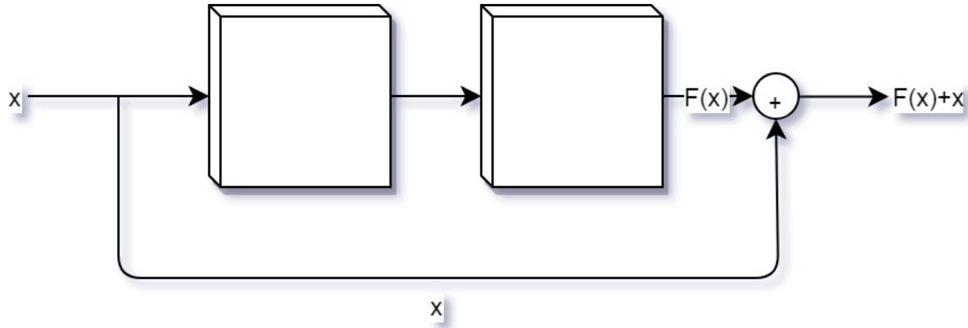


Figure 2.7: Shortcut connection as implemented in ResNet

Architecture

ResNet's architecture can be divided in five blocks. The first one is made up of a single 7x7 convolutional filter while all the others use 3x3 and 1x1 convolutional filters with shortcut connections. The number of these shortcut connections in each block changes between the versions of ResNet: from 2 of them in each block for the 18-layer version to 3, 8, 36 and 3 in blocks 2, 3, 4 and 5 respectively for the 152-layer version. In the following descriptions, we mainly focus on the versions with 50 layers and more.

In the 50, 101 and 152 layer versions of ResNet, each shortcut block is made up of a 1x1 convolution followed by a 3x3 and another 1x1 convolution, in parallel with the identity shortcut connection outlined above. The depth of each type of convolution changes per block, but is outlined in Table 2.2 for the 50, 101 and 152 layer versions of ResNet.

The amount of shortcut blocks in each main block of ResNet is outlined in Table 2.3 for the three versions with 50 layers and more. Note that each block starts with stride of two: this is done by a max pooling layer of kernel 3x3 in the case of block 2, and by the first layer in the case of blocks 3 through 5.

ResNet uses the same CNN design concepts as VGG [7]: double the depth of the feature maps whenever using a stride of 2 such that the complexity of the feature maps remains

Shortcut Block Layer	Conv2_x	Conv3_x	Conv4_x	Conv5_x
1x1 depth	64	128	256	512
3x3 depth	64	128	256	512
1x1 depth	256	512	256	2048

Table 2.2: Shortcut block depth for 50, 101 and 152 layer versions of ResNet [2].

Layer Name	Input Dimensions ($W \times H \times D$)	Kernel ($w \times h \times d$/stride)	50-layer Depth	101-layer Depth	152-layer Depth
Conv1	$224 \times 224 \times 3$	$7 \times 7 \times 64/2$	-	-	-
MaxPool2	$112 \times 112 \times 64$	$3 \times 3/2$	-	-	-
Conv2_x	$56 \times 56 \times 64$	-	3	3	3
Conv3_x	$56 \times 56 \times 256$	-	4	4	8
Conv4_x	$28 \times 28 \times 512$	-	6	23	36
Conv5_x	$14 \times 14 \times 1024$	-	3	3	3
Avgpool	$7 \times 7 \times 2048$	7×7	-	-	-
FC	$1 \times 1 \times 2048$	$1 \times 1 \times 1000$	-	-	-

Table 2.3: Detailed Architecture of the ResNet-v1 network [2].

the same [2]. Consequently, each block has depth equal to twice that of the previous one, with the exception of the first block's depth of 64.

Comparisons between the *plain* networks - without shortcut connections - and the *residual* networks - with shortcut connections - reveal a significant reduction in training error.

2.3.3 SqueezeNet

While networks like VGG and ResNet focus on accuracy, there are also DCNNs that instead try to balance accuracy and resource efficiency for specific real world applications. A good

example is the SqueezeNet network [3], which achieves the same accuracy as AlexNet with a much smaller model. Specifically, SqueezeNet has about 50 times fewer parameters than AlexNet for a model size of about 4.8 megabytes. This can further be reduced to 0.47 megabytes through the use of Deep Compression [25] and quantization, while still achieving an ImageNet top-1 and top-5 accuracy of 57.5% and 80.3% respectively; the same accuracy as the non-compressed version of the model.

The authors attribute the network's efficiency to three main strategies:

- 3x3 convolutional filters require nine times as many parameters as 1x1 convolutional filters;
- Reducing the feature map depth before using a 3x3 filter within the network;
- Downsampling later in the network later in the network to preserve larger feature maps through most of the network.

Practically, this is implemented through the *fire* module which are made up of two layers:

1. The squeeze layer: 1x1 filters to reduce the depth of the feature maps (by having a smaller output depth than input depth);
2. The expand layer: 1x1 and 3x3 filters in parallel whose outputs are concatenated to generate the module's outputs.

The amount of filters of each type (i.e. 1x1 squeeze filters, 1x1 expand filters and 3x3 expand filters) can be tuned and can differ between the different fire modules in the same network. Table 2.4 shows the sequence of layers and the number of filters used throughout the official SqueezeNet model. The fire module configuration is expressed as the number of 1x1 filters in the squeeze layer ($s_{1\times1}$) followed by the number of 1x1 and 3x3 filters in the expand layer ($e_{1\times1}$ and $e_{3\times3}$ respectively).

Note that SqueezeNet does not use any fully connected layers for its final classification, but instead relies on a 1x1 filter followed by a global average pooling later - an average pooling layer whose kernel covers the entire feature map. This is significantly more parameter efficient, as even a single fully connected layer has a significant number of parameters [4].

Layer Name	Input Dimensions ($W \times H \times D$)	Kernel ($w \times h \times d$/stride)	Fire Module Configuration ($s_{1\times1}- > e_{1\times1} + e_{3\times3}$)
Conv1	$224 \times 224 \times 3$	$7 \times 7 \times 96/2$	-
Maxpool1	$111 \times 111 \times 96$	$3 \times 3/2$	-
Fire2	$55 \times 55 \times 96$	-	$16 - > 64 + 64$
Fire3	$55 \times 55 \times 128$	-	$16 - > 64 + 64$
Fire4	$55 \times 55 \times 128$	-	$32 - > 128 + 128$
Maxpool4	$55 \times 55 \times 25$	$3 \times 3/2$	-
Fire5	$27 \times 27 \times 256$	-	$32 - > 128 + 128$
Fire6	$27 \times 27 \times 256$	-	$48 - > 192 + 192$
Fire7	$27 \times 27 \times 384$	-	$48 - > 192 + 192$
Fire8	$27 \times 27 \times 384$	-	$64 - > 256 + 256$
Maxpool8	$27 \times 27 \times 512$	$3 \times 3/2$	-
Fire9	$13 \times 13 \times 512$	-	$64 - > 256 + 256$
Conv10	$13 \times 13 \times 512$	$1 \times 1 \times 1000$	-
Avgpool10	$13 \times 13 \times 1000$	13×13	-

Table 2.4: Detailed Architecture of the SqueezeNet network [3].

2.3.4 MobileNet

Another important DCNN focused on efficiency is MobileNet [4]. As the name implies, MobileNet is designed for implementation on various mobile devices through its *depthwise separable convolution (DSC)* and the addition of two hyperparameters to control the depth of the feature maps throughout the network.

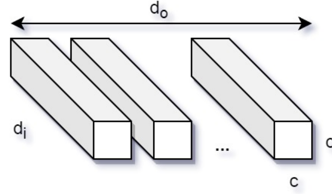
Depthwise Separable Convolution

As mentioned in Section 2.3.3, using 3x3 filters instead of a 1x1 filter increases the number of parameters by a factor of nine, making them quite computationally expensive. The design of MobileNet compromises in this regard by using the depthwise separable convolutions mentioned above, separating a regular 3x3 convolution - as shown in Figure 2.8a - into a

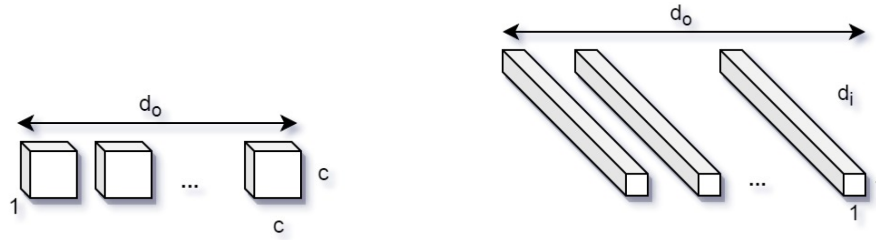
series of two convolutions:

1. A 3x3 *depthwise* convolution, which multiplies a single 3x3 filter over each single feature map along the depth - shown in Figure 2.8b;

2. A *pointwise* convolution, which is a regular 1x1 convolution - shown in Figure 2.8c.



(a) Convolutional filters in a $c \times c$ convolution



(b) Depthwise filters in a $c \times c$ depthwise separable convolution

(c) Pointwise filters in a $c \times c$ depthwise separable convolution

Figure 2.8: Structure of a regular convolution compared to a depthwise separable convolution.

It is worth noting that both the depthwise and the pointwise convolutions are directly followed by batch normalization and ReLU operations.

The number of parameters in a separable convolution can be expressed as the sum of the number of parameters in each part, as shown in Equation 2.18.

$$(c \times c \times d_o) + (d_i \times d_o) \tag{2.18}$$

Subsequently, we can calculate Equation 2.19 for the cost of the separable convolution.

$$(c \times c \times d_o + d_i \times d_o) \times (h_o \times w_o) \tag{2.19}$$

With the 3x3 convolutions used in MobileNet, the computational cost of a regular convolution is reduced by a factor between eight and nine, obtained by dividing the computational cost of a separable convolution by that of a regular convolution. The calculation for this factor is shown in Equation 2.20; the cost reduction depends on the size of the convolutional kernel and depth of the input feature map.

$$\frac{(c \times c \times d_o + d_i \times d_o) \times (h_o \times w_o)}{(c \times c \times d_i \times d_o) \times (h_o \times w_o)} = \frac{1}{d_i} + \frac{1}{c^2} \tag{2.20}$$

This increased efficiency does cause a reduction in accuracy; this will be detailed further on, as the depthwise separable convolution is used extensively in MobileNet.

MobileNet Hyperparameters

The other important aspect of MobileNet's design is the addition of two hyperparameters:

- A depth multiplier, which reduces (or increases) the depth of each set of feature maps in the network by a certain factor.
- A resolution multiplier, which reduces (or increases) the height and width of the input image and each feature map in the network by a certain factor.

Note that the depth multiplier is the more interesting hyperparameter when using MobileNet for segmentation. The limitations on the input resolution are already removed when generating 2D outputs: this will be detailed further in Section 2.5.1.

MobileNet Architecture

MobileNet's architecture is mostly made up of depthwise convolutions, with the exception of the regular convolution in the first layer, and the average pooling and fully connected layers at the end of the network. The others thirteen layers are depthwise convolutions. Table 2.5 shows the detailed architecture of MobileNetv1 for an input image of size $224 \times 224 \times 3$ and a width multiplier of 1.0. Note that *conv dw* indicates a depthwise convolution as defined above - a $3 \times 3 \times d_i$ convolution followed by a $1 \times 1 \times d_o$ convolution, where d_i is the input depth and d_o is the output depth defined in the last column.

Layer Name	Input Dimensions ($W \times H \times D$)	Kernel ($w \times h \times d$/stride)	Pointwise Convolution Output Depth
Conv	$224 \times 224 \times 3$	$3 \times 3 \times 32/2$	-
Conv dw	$112 \times 112 \times 32$	-	64
Conv dw	$112 \times 112 \times 64$	-/2	128
Conv dw	$56 \times 56 \times 128$	-	128
Conv dw	$56 \times 56 \times 128$	-/2	256
Conv dw	$28 \times 28 \times 256$	-	256
Conv dw	$28 \times 28 \times 256$	-/2	512
Conv dw	$14 \times 14 \times 512$	-	512
Conv dw	$14 \times 14 \times 512$	-	512
Conv dw	$14 \times 14 \times 512$	-	512
Conv dw	$14 \times 14 \times 512$	-	512
Conv dw	$14 \times 14 \times 512$	-	512
Conv dw	$14 \times 14 \times 512$	-/2	1024
Conv dw	$7 \times 7 \times 1024$	-	1024
Avgpool	$7 \times 7 \times 1024$	7×7	-
FC	$1 \times 1 \times 1024$	$1 \times 1 \times 1000$	-

Table 2.5: Detailed Architecture of the MobileNet-v1 network [4].

Results

Howard et. al. [4] show extensive results of the MobileNet architecture. Table 2.6 outlines the change in accuracy, computational cost, and number of parameters that results from changing the depth of the network and from using depthwise or regular convolutions. Note that the shallow configurations has a depth multiplier of 1.0 but does not have five of the six $14 \times 14 \times 512$ convolutional layers shown in table 2.5.

Finally, Table 2.7 shows the same information for networks comparable to MobileNet, either in accuracy or in computational and memory cost.

Model	ImageNet Accuracy	Mult-Adds (10^6)	Parameters (10^6)
Conv full, depth 1.0	71.7%	4866	29.3
Conv dw, depth 1.0	70.6 %	569	4.2
Conv dw, depth 0.75	68.4 %	325	2.6
Conv dw, depth 0.50	63.7 %	149	1.3
Conv dw, depth 0.25	50.6 %	41	0.5
Conv dw, *shallow*	65.3 %	307	2.9

Table 2.6: Results of the MobileNet-v1 network on the ImageNet classification task [4] for an input of size 224x224.

Model	ImageNet Accuracy	Mult-Adds (10^6)	Parameters (10^6)
GoogleNet (Inception-v1)	69.8 %	1550	6.8
VGG 16	71.5 %	15300	138
SqueezeNet	57.5 %	1700	1.25
AlexNet	57.2 %	720	60

Table 2.7: Results of other networks on the ImageNet classification task [4].

2.4 Object Detection Networks

The object detection task is the attempt to detect various objects within an image. The ideal result for a certain object is a rectangle that contains the entire object and as little extraneous background as possible. This can be a more specific detection task, such as face detection [26, 27, 28], or very generic, with datasets like Pascal VOC [29] and Microsoft COCO (Common Objects in Context) [30] - the former is also used for semantic segmentation (see Section 2.5). MS-COCO, which is used for object detection and instance segmentation, among others, has a 118 287/5 000/40 670 train/val/test split over 80 object classes.

Accuracy is measured in terms of Average Precision (AP) and Average Recall (AR), calculated as per Equation 2.4:

- Average Precision: percentage of results that are accurate;
- Average Recall: percentage of expected results that have been detected.

$$
\begin{aligned}
AP &= \frac{TP}{TP + FP} \\
AR &= \frac{TP}{TP + FN}
\end{aligned}
\tag{2.21}
$$

Both of these metrics are measured for a specific IoU between the ground truth and the detection output: detections falling below that IoU threshold are considered *negative* results or *misses* while detections falling above that threshold are considered *positive* results. On MS-COCO, the final results for both AP and AR are the average over multiple IoUs, ranging between 0.50 and 0.95 with a step of 0.05.

2.4.1 Region proposal with CNNs (R-CNN)

One of the major way to implement object detection is through the use of region proposals. The goal here is to first propose a region that should contain objects we wish to detect, followed by a classification step that identifies the object within the region. In the case of Fast R-CNN [10] and Faster R-CNN [12], the output of the region proposal is also regressed to a proper bounding box.

The R-CNN framework has three major versions, each building on the concepts of the previous one. These are *R-CNN* [31], *Fast R-CNN* [10] and *Faster R-CNN* [12].

R-CNN

The original R-CNN framework [31] is a fairly straightforward implementation. It is made up of two successive, independent steps:

1. A region proposal step. Any region proposal method can be used here. The authors use selective search [32] for comparison with other detection methods.

2. Separate classification of each proposed regions using AlexNet [1]. This requires resizing of the region to 227x227.

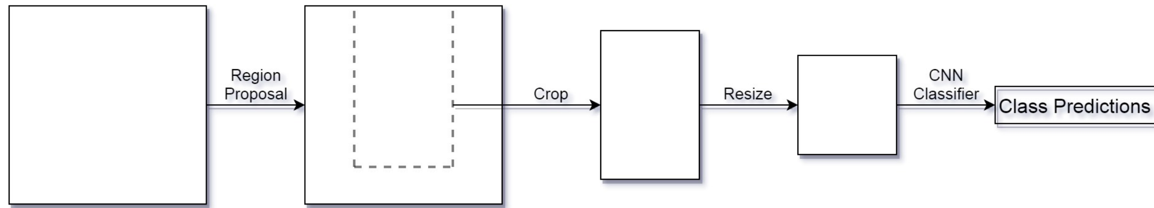


Figure 2.9: R-CNN region proposal architecture. Note that the cropping step and the steps that follow are repeated for each region proposal.

While this approach shows a significant improvement over methods that do not use CNN, there is much room for improvement, as can be seen in the Fast R-CNN framework.

Fast R-CNN

While the original R-CNN paper [31] performs the refinement step separately for each proposal, its follow-up Fast R-CNN [10] aims to generate feature maps that can then be reused for all the region proposals. Much like R-CNN, the region proposals are used as an input for the CNN, in conjunction with the image. However, unlike R-CNN, a CNN is then used to generate features for the entire image and not just for the region proposal.

The features relevant to each region proposal are then taken from the CNN's output and put through an RoI pooling layer and a chosen number of layers to generate a class output and a box regression output for that specific proposal. This is illustrated in Figure 2.10, which can be compared to Figure 2.9. In the Fast R-CNN paper, the branch is made up of fully connected layers, with two final fully connected layers in parallel: one to generate the class predictions and the other to generate the box regression outputs.

The RoI pooling layer is used to take the proposal region from the feature map and create a new fixed-size feature map (7x7 is used by the authors) through the use of max pooling. The RoI is aligned with the feature map values by dividing its coordinates by the downsampling factor and rounding the result. This quantized RoI is then divided into a grid of the same size as the new feature maps where each cell is also aligned with the

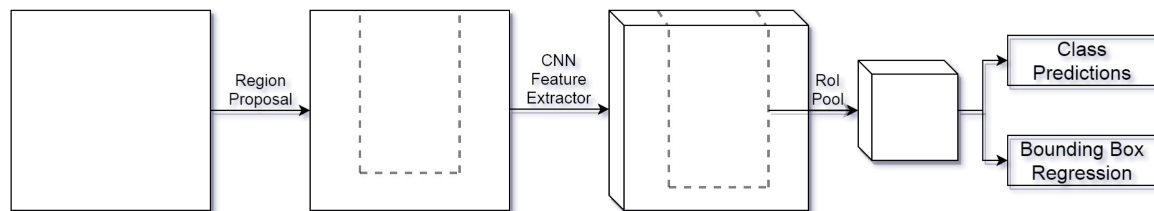


Figure 2.10: Fast R-CNN region proposal architecture. Note that the RoIPool step and the steps that follow are repeated for each region proposal.

feature map values. A maximum pooling operation is then applied to each cell, generating an output of the correct size [9].

Girshick et. al. use use cross-entropy as the classification loss and smooth-L1 loss as the box regression loss. The latter is not applied to background RoI's, as those to not have a regression groundtruth from which to calculate loss.

Note that this approach introduces non-negligible misalignment between the original RoI and the RoI pooling layer's output. This proved to be an issue when adapting the R-CNN architecture for instance segmentation (see Section 2.6.1).

Faster R-CNN

Faster R-CNN [12] further expands on the work from Fast R-CNN [10] by merging the region proposal step and the refinement step. Similarly to Fast R-CNN, a set of feature maps are generated from any CNN (e.g. VGG-16, ResNet). These feature maps are used for two sequential tasks, as can be seen in Figure 2.11:

1. Generating region proposals: an $n \times n$ convolutional layer (3×3 is used by the authors) followed by two 1x1 convolutional layers in parallel - one for box regression, the other for classification with classes *object* or *not object*.

2. Classification and box regression: using RoI pooling and fully connected layers as in Fast R-CNN.

As such, Faster R-CNN is an architecture that takes any image as its input, generates its own region proposals, and refines and classifies them.

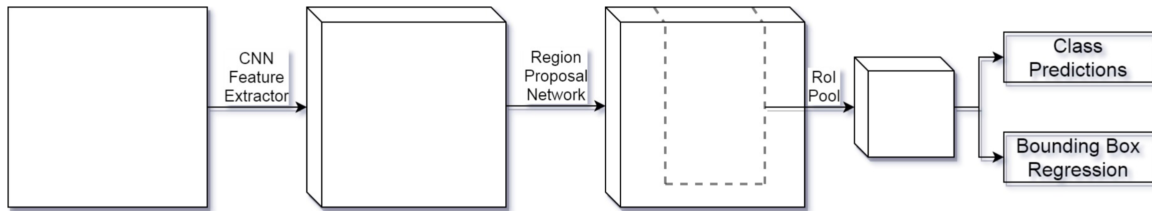


Figure 2.11: Faster R-CNN region proposal architecture. Note that the RoIPool step and the steps that follow are repeated for each region proposal.

Faster R-CNN uses the same loss as Fast R-CNN for the class prediction and bounding box regression branches. On the other hand, the Region Proposal Network (RPN) is trained with a binary class label which, for a certain groundtruth box, is considered positive for anchors with an IoU of at least 0.7 or with the largest IoU. The training ran by the authors alternates training of the RPN and training of Fast R-CNN.

2.5 Semantic Segmentation Networks

The goal of semantic segmentation [15, 16, 33, 34, 35, 36, 37, 38] is to label every pixel within an input image with a class label. Depending on the dataset, this can be every single pixel (e.g. COCO's *stuff* categories) or simply every pixel that belongs to the classes of interest (e.g. Pascal VOC only has 20 foreground classes and everything else is considered background). This task introduces some challenges regarding memory requirements and the need to produce pixel-precise outputs. An example of an image from the Cityscapes dataset, annotated for semantic segmentation, can be seen in Figure 2.12.

Note that instance segmentation is a task closely related to semantic segmentation, but which also requires the separation of each instance within a class.

This task is evaluated on the Intersection over Union (IoU) metric between the predicted segmentation and the ground truth segmentation. For each class, we take the area of the intersection between the two and divide it by the area of the union between the two. This is equivalent to dividing the number of true positives (TP) by the sum of the true positives, false positives (FP) and false negative (FN), as shown in equation 2.22. The final evaluation metric is simply the average IoU over all the classes, referred to as the mean IoU (mIoU).

Figure 2.12: Semantic Segmentation example from the Cityscapes dataset.

$$IoU = \frac{TP}{TP + FP + FN} \tag{2.22}$$

The datasets commonly used for this task are:

- Pascal VOC (Visual Object Classes) [29]: 1465 train annotations and 1450 val images over 20 classes (additional images are available with object detection annotations only).

- Cityscapes [39]: 5000 fine segmentation annotations with a 2 975/500/1 525 train/val/test split over 30 classes, with an extra 20 000 *coarse* annotations also available for training.

Pascal VOC's small size enables faster training, making it ideal for faster testing of experimental ideas, while Cityscapes is the more commonly used dataset. Specifically, Cityscapes' image resolution of 2048x1024 increases memory requirements and the training time significantly.

Section 2.5.1 covers the seminal work which first used existing image classifiers for semantic segmentation by converting them into fully convolutional networks. Section 2.5.2 covers the DeepLab network, which uses atrous convolution to great effect.

2.5.1 Fully Convolutional Networks for Semantic Segmentation

Following the success of AlexNet with image classification, CNNs were used to achieve similar success on other computer vision tasks. Fully convolutional networks [15] have become the main way of doing this in the case of semantic segmentation.

The main concept in this paper was to use transfer learning (see Section 2.2.5) on the accurate image classifiers at the time (AlexNet [1], VGG-16 [7] and GoogLeNet [40]) and retask them for semantic segmentation. This required some changes to the original networks' architecture because they were designed to produce a probability for each class for the entire image and not for each pixel. This is accomplished by converting an image classifier into a fully convolutional network and by adding transposed convolutions to increase the size of the feature maps.

This is an example of an *encoder-decoder* architecture: the truncated network serves as the *encoder* by extracting complex features from an image, while the transposed convolutions serve as the *decoder* by interpreting those features for the task at hand.

Fully Convolutional Network (FCN)

The fully connected layers found in most image classifiers have two main issues for segmentation, as mentioned in Section 2.2.1:

- The loss of spatial awareness;
- The layer input size must always be the same (which means the network's input size must always be the same as well).

As such, convolutional layers are used to replace the fully connected layers in three networks: AlexNet, VGG-16 and GoogLeNet. The resulting type of network is called a *fully convolutional network* (FCN) and can take input images of any size; if we use zero-padding to prevent size changes without striding, the output feature map will have a height and width 32 times smaller than the input's because of the stride values throughout those three networks. This introduces the other main problem for semantic segmentation: the need to generate an output the same size as the original image.

Skip Architecture

Introduced in Section 2.2.4, a transposed convolution is equivalent to the backward pass of a convolution. As such, it is possible to have trainable transposed convolutions at the end of a network to upsample feature maps. This results in significantly better upsampled results than using simple methods such as bilinear interpolation [36].

While this can produce outputs of the correct size, they still do not capture the fine details required for precise segmentation. To this end, the *skip architecture* is used with the goal of refining the segmentation predictions with feature maps taken from earlier in the network, as can be seen in Figure 2.13.

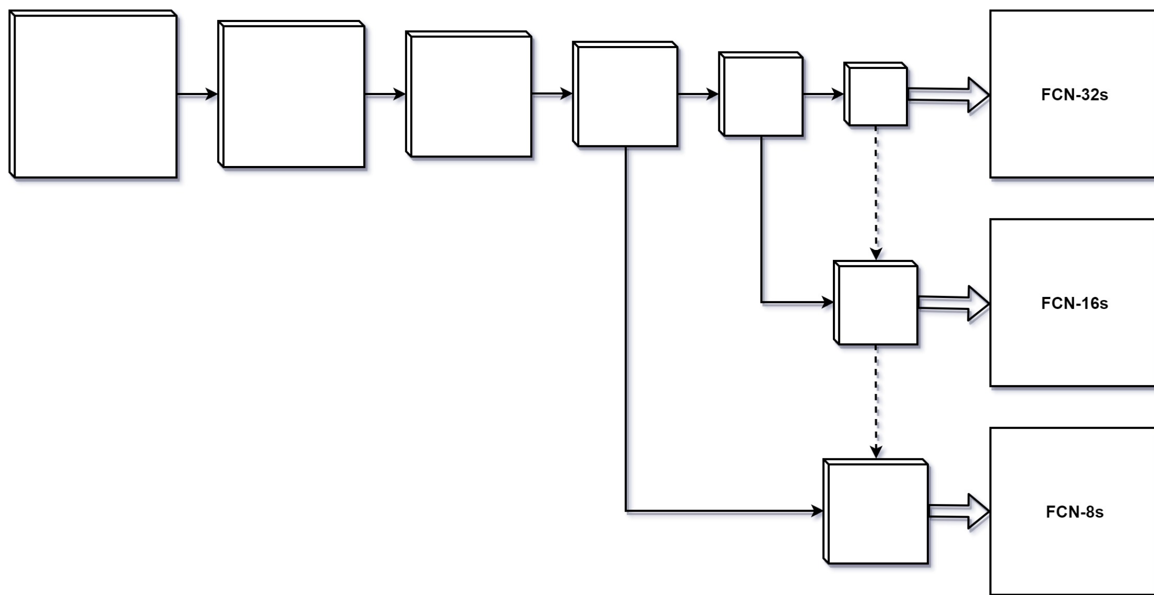


Figure 2.13: FCN skip architecture, showing variants FCN-32s, FCN-16s and FCN-8s.

In Figure 2.13, single arrows represent convolutions, dashed arrows represent transposed convolutions that upsample the input feature map by a factor of two, and hollow arrows represent upsampling back to the size of the input image to generate the final prediction. The skip architecture itself is built as follows:

1. A transposed convolution is applied to the smaller feature map to increase its size to that of the larger feature map;

2. The upsampled feature map is concatenated with a feature map of the same size from earlier in the network;

3. The two are fused together through a 1x1 convolution.

These steps can be applied once for each downsampling step in the network, using the output of the previous feature map fusion as the smaller of the two input feature maps. In practice, only two of the feature maps from within the network (i.e. downsampled by 16 times and 8 times) are used, as eventually, the cost of using more feature maps outweighs the diminishing increase in accuracy. These configurations are shown in Figure 2.13, under the names *FCN-16s* when using one additional set of feature maps and *FCN-8s* when using two additional sets of feature maps.

Regardless of the complexity of the skip architecture, its output is then upsampled to the size of the input image. The final prediction has depth equal to the number of classes in the dataset.

Loss Function

The loss function used for training is fairly straightforward. For each pixel in the input image, the FCN networks generate one prediction for each class. Class probabilities are calculated using a softmax operation, to which a cross-entropy loss is applied, much as we would for an image classifier. We then take the loss for the entire image as the sum of the loss over all the pixels.

Similarly to top-1 accuracy for image classification, the final prediction for a pixel is chosen as the most likely prediction for each pixel, predicting one class for each pixel in the image.

Results

Shelhamer et. al. present their results on multiple segmentation datasets: PASCAL VOC, NYUDv2, SIFT Flow and PASCAL-Context. They report state-of-the-art results on all of them, in addition to a significant improvement in the inference time compared to the runner-up on PASCAL VOC - approximately 100 milliseconds for FCN-8s compared to approximately 50 seconds for *Simultaneous detection and segmentation* (SDS) [41].

As mentioned above, three classification networks are adapted for the task of semantic segmentation. AlexNet, VGG16 and GoogLeNet achieve results of 39.8, 56.0 and 42.5 mIoU

on the PASCAL VOC validation set, with inference times of 16, 100 and 20 milliseconds. As such, the authors choose VGG16 for all their other tests.

Table 2.8 recapitulates results of the main FCN-VGG16 variants on the PASCAL VOC validation set. The *pool* variants are truncated at said pool layer. The FCN-8s-staged architecture is first trained with no skip architecture, before adding a single-step skip architecture and, finally, a second step to the skip architecture. These results show the increase in mIoU resulting from each additional step in the skip architecture, and the significant decrease in mIoU resulting from removing parts of the network entirely.

Architecture	Mean IoU
FCN-32s	63.6
FCN-16s	65.0
FCN-8s-at-once	65.4
FCN-8s-staged	65.5
FCN-pool5	50.0
FCN-pool4	22.4
FCN-pool3	9.2

Table 2.8: Results of variants of the FCN-VGG16 architecture on a subset of the PASCAL VOC 2011 val set

2.5.2 DeepLab

The DeepLab network [16, 33, 34] takes another approach for semantic segmentation through its use of atrous convolutions, both within the network and in the atrous spatial pyramid pooling (ASPP).

Atrous Convolution for Semantic Segmentation

Atrous convolution was introduced in Section 2.2.1 as a type of convolution that looks at pixels farther away than a default convolution, thereby achieving a larger field of view.

This is particularly useful for a task like segmentation where predictions should use both local and global information, but downsampling severely hurts their accuracy.

DeepLab uses atrous convolution in two ways:

- Throughout the network, any number of stride settings can be set to 1 and the dilation of any subsequent convolutions is increased by a factor of 2: this reduces the loss in precision from downsampling without reducing the field of view.
- At the end of the network, the ASPP module which is made up of parallel atrous convolutions with increasing dilation rates (four branches with rates 6, 12, 18 and 24 showing the best results): this enables the final predictions to be based on information farther away in the feature map without increasing the filter sizes.

A drawback for both of these is the increased memory requirements. Particularly, reducing the downsampling in any one layer will increase the size of the successive feature map by a factor of 2 in both dimension. As such, training on large images - such as those in the Cityscapes dataset - can require very small batch sizes, even with the latest available GPUs. It is also important to factor in the increase in computational cost. DeepLab only reduces downsampling to a factor of 8 (meaning the last few convolutional layers have a dilation rate of 4).

Variations

DeepLabv3 built on the concepts introduced by DeepLab and DeepLabv2 in two main ways:

1. Building a network with output stride 256 and reducing that stride through atrous convolution;
2. Using image level features by adding a global average pooling layer in parallel with the ASPP's parallel branches, in addition to testing variants of the ASPP dilation rates.

The deeper network is by using ResNet as a backbone and adding additional blocks 6, 7 and 8 - replicas of block Conv5_x in Table 2.3 - for a total output stride of 256.

Subsequently, atrous rates 2, 4, 8 and 16 are applied in blocks Conv4_x through Conv7_x to maintain an output stride of 16. Decreasing the output stride between 256 and 8 shows an improvement in results [33], as shown in Table 2.9.

Output Stride	Mean IoU
256	20.29
128	42.34
64	59.99
32	70.06
16	73.88
8	75.18

Table 2.9: Results of varying the output stride of ResNet-50 with 8 blocks

2.6 Instance Segmentation Networks

Instance Segmentation [9, 42, 43, 44, 45] can be described as a mix of object detection (Section 2.4) and semantic segmentation (Section 2.5). Whereas semantic segmentation consists of separating the classes within an image at a pixel level, instance segmentation also requires that we separate each instance for each class at a pixel level - each individual person in the person class, each individual car in the car class, etc. We show an example of instance segmentation annotations from the MS-COCO dataset in Figure 2.14.

The metric used to measure performance on instance segmentation are typically the ones used for object detection, evaluated on masks instead of bounding boxes. That is, average precision (AP) and average recall (AR) evaluated at various IoU thresholds.

The main datasets for this task are Cityscapes and MS-COCO, which are also commonly used for semantic segmentation and object detection respectively. The primary evaluation metric for both of these is the average of the ten AP values for IoU thresholds between 0.50 and 0.95 with a step of 0.05. Cityscapes also provides AP for objects at ranges of 50 m and 100 m.

Figure 2.14: Instance segmentation annotation example from the MS-COCO dataset.

2.6.1 Mask R-CNN

Mask R-CNN [9] extends the Faster R-CNN framework [12] - see Section 2.4.1 - for instance segmentation by implementing a mask prediction branch in parallel with the classification and bounding box regression branches, as can be seen in Figure 2.15. The architecture of the mask prediction branch itself is rather simple but it does require a major change to the *RoIPool* operation.

Since its publication, Mask R-CNN's architecture has been used in application-driven research in the field of instance segmentation [46, 47, 48, 49].

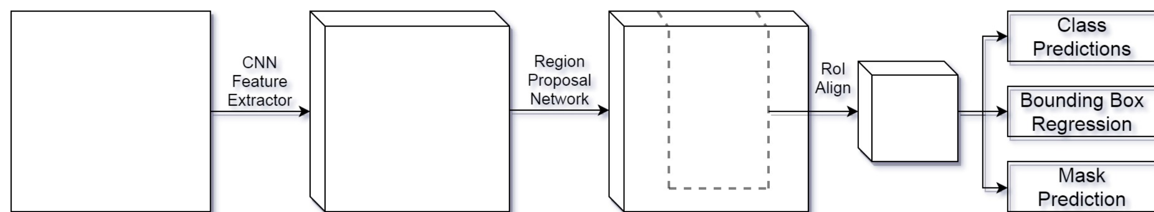
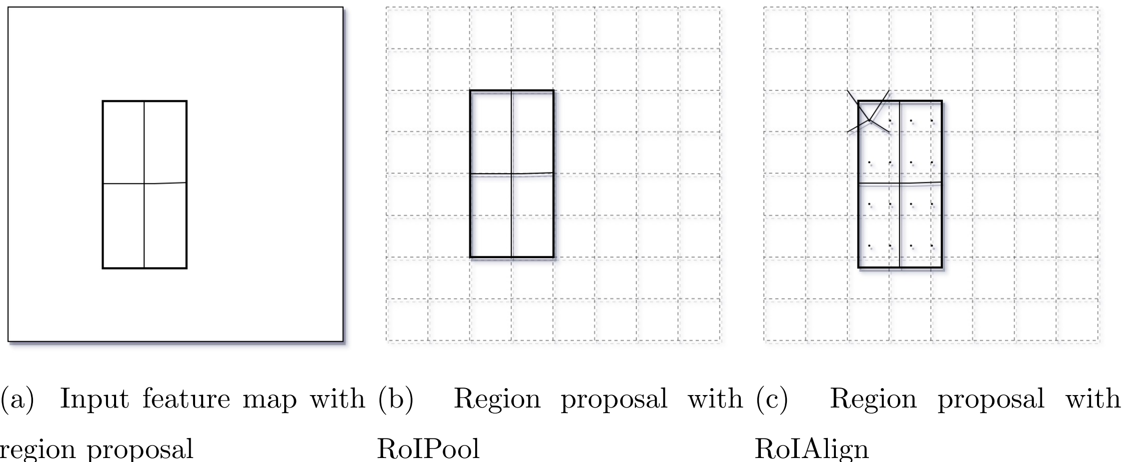


Figure 2.15: Mask R-CNN region proposal architecture. Note that the RoIAlign step and the steps that follow are repeated for each region proposal.

RoIPool and RoIAlign The only change made to the Faster R-CNN architecture is in the *RoIPool* operation used to generate the feature maps for bounding box regression.

If we want to generate a 7x7 feature map from an RoI with *RoIPool*, we divide the RoI into bins placed in a 7x7 grid. Then, we aggregate the values in each bin - using max pooling in the case of Faster R-CNN [12]. This results in poor alignment between the resulting 7x7 feature map and the original RoI: when aligning the RoI with the values in the larger feature map and when creating the bins. While object detection is resilient to this, pixel-level segmentation is not [9], which led to the use of RoIAlign instead of RoIPool in Mask R-CNN. It is worth noting that using RoIAlign also increases the results on object detection [9].

RoIAlign's main goal is to remove these approximations. Instead of aligning the RoI and the bins with the feature map, their coordinates are kept as continuous values. Bilinear interpolation is then used to take the values at four exact points within each bin which are then aggregated through pooling (either max or average). These two pooling operations are compared in Figure 2.16. This results in a significant improvement of the mask's accuracy, from 26.9 AP with RoIPool to 30.2 AP or 30.3 AP with maximum and average RoIAlign respectively.



(a) Input feature map with region proposal (b) Region proposal with RoIPool (c) Region proposal with RoIAlign

Figure 2.16: A comparison of RoIPool - used in Faster R-CNN - and RoIAlign - used in Mask R-CNN - for the generation of a 2x2 feature map; note that the region proposal with RoIAlign is perfectly aligned with the region proposal in the input feature map.

Mask Predictions The mask prediction branch is based on the fully convolutional network [15] concept: where a fully connected layer has significant disadvantages for mask prediction, simple convolutional layers do not. In Mask R-CNN, the mask branch is made

up of one transposed convolution, upsampling the feature maps by a factor of two, followed by a convolution, resulting in a 14x14 mask output.

Mask Loss Function As is common for semantic segmentation, the mask branch's output has a depth equal to the number of classes. Unlike semantic segmentation, the only mask prediction to be kept are those associated with the class prediction of Fast R-CNN. As such, the mask loss is only based on the class being predicted for this specific bounding box, and is calculated using a sigmoid operation followed by binary cross-entropy loss [9]. During training, mask loss is only calculated for RoI's that have a IoU superior to 0.50 with a ground truth box. During evaluation, however, the mask is predicted after the bounding box regression and the non-maximum supression (NMS): the bounding boxes are better aligned and fewer masks are predicted.

Chapter 3

Single Shot Detector for Instance Segmentation

Instance segmentation can be described as a mix of object detection and semantic segmentation. As such, some frameworks [9] approach this task by adapting an object detection network to also generate instance segmentation predictions - specifically, Mask R-CNN takes Faster R-CNN and adds an segmentation branch in parallel with the classification and regression branches. The work presented in this chapter focuses on adapting the Single Shot Detector (SSD) for instance segmentation. The approaches that were considered can be divided into two main categories:

1. Single branch instance segmentation: adding a single branch which generates a mask prediction for each detection box.

2. Multiple branch instance segmentation: adding multiple branches each of which generates mask predictions for a subset of the detection boxes.

We propose a new way of eliminating alignment issues when cropping features for mask prediction. We also propose an approach for greatly limited the scale of resizing operations before instance segmentation.

Basing our architecture on the SSD object detection framework enables us to minimize the cost of predicting bounding boxes, thus designing a fast, small, instance segmentation

model. This model would be ideal for use cases such as autonomous driving, where low model size, low memory footprint, and high speed are a requirement.

Section 3.1 covers the instance segmentation architecture that we propose, while section 3.2 covers some of the important or unusual aspects in the implementation of our networks.

3.1 SSD-ISB Architecture

The goal of this work is to take the feature maps used in SSD to generate object detection, and to add some simple architecture that can reuse them for instance segmentation. While other networks could be equally interesting to test, MobileNet-v1 is the main network used for this purpose, as it provides a few advantages:

- Relatively fast training and low memory cost, when compared to expensive networks such as ResNet;
- MobileNet SSD's low computational cost shows synergy with likely applications;
- Availability of pre-trained SSD COCO models.

SSD's architecture, designed by Liu et. al., is detailed in Section 3.1.1, our general segmentation branch architecture used in combination with SSD is detailed in Section 3.1.2 and the idea of cropping from encompassing bounding boxes is detailed in Section 3.1.3. The way that we use the segmentation branches to build our single-branch and multi-branch architectures are detailed in sections 3.1.4 and 3.1.5.

3.1.1 Single Shot Detector (SSD)

The *Single Shot Detector* (SSD) [11] takes an alternate approach to object detection by bypassing the region proposal step. Instead, it overlays multiple grids over the input image and uses the grid cells as default anchors which are then refined. The grid sizes can be customized, but the original paper uses the VGG-16 image classifier network as a base and uses grids of size 38x38, 19x19, 10x10, 5x5, 3x3 and 1x1. The same concept is illustrated

for the MobileNet image classifier in Figure 3.1, with feature maps of sizes 19x19, 10x10, 5x5, 3x3, 2x2 and 1x1.

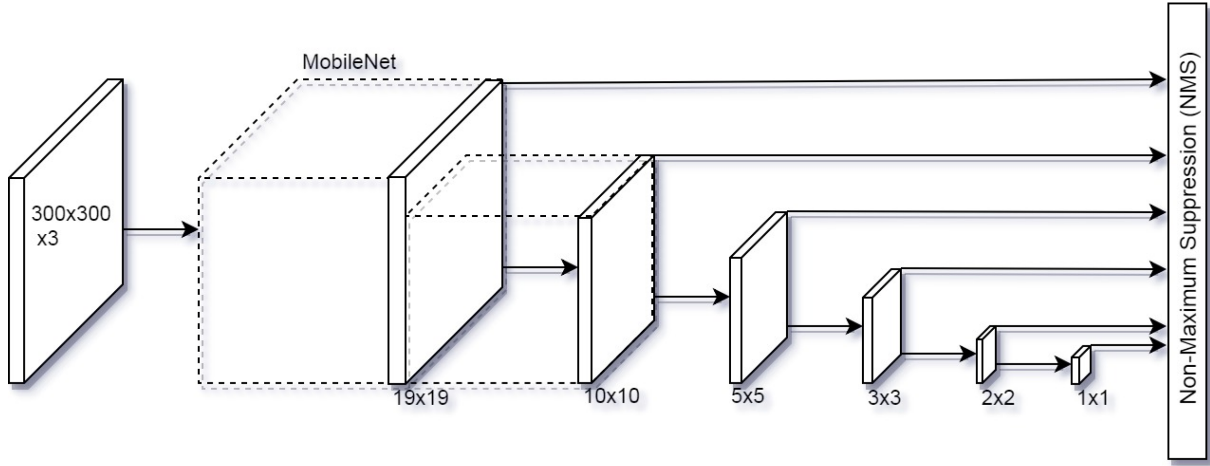


Figure 3.1: SSD MobileNet architecture.

These additional feature maps are implemented by taking the 19x19 feature maps from VGG-16 and adding additional convolutional layers to generate the feature maps of size 10x10, 5x5, 3x3 and 1x1. Each of these feature maps has its own 3x3 convolutional layer to generate the box regression and classification outputs.

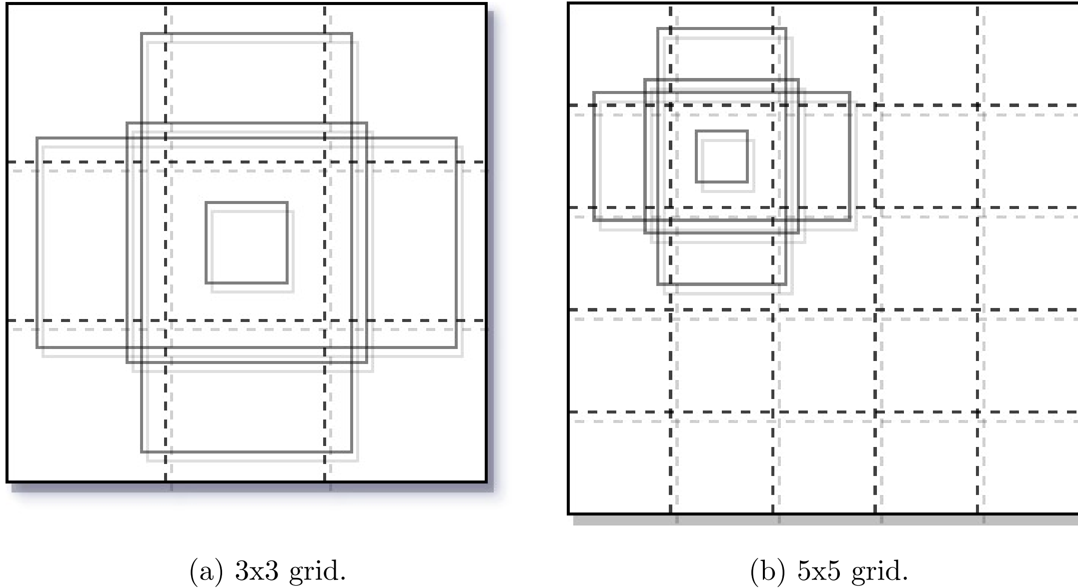
For every single cell in each grid, multiple anchors are used with varying scale and aspect ratio, thereby accounting for various possible shapes. Figure 3.2 shows some possible anchor boxes for one grid cell in a 3x3 grid and a 5x5 grid - note that the same default anchor boxes would be used for every single cell in the grid. All default boxes are centered on the center of their respective cell in the grid.

For each default box, a class score is generated for each class in addition to four offset values which realign and rescale the default box.

The aspect ratios typically used are $1, 2, \frac{1}{2}, 3$ and $\frac{1}{3}$. The scales typically used are between s_{min} and s_{max} inclusively, with the other scales spaced evenly between them, as shown in Equation 3.1, where k iterates from 1 to the number of scales m.

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1) \tag{3.1}$$

The default for s_{min} and s_{max} are 0.2 and 0.9 respectively.



(a) 3x3 grid.

(b) 5x5 grid.

Figure 3.2: SSD default anchor examples for two grid sizes. Anchors are shown for one cell only.

Loss Function

Much like Faster R-CNN, SSD's loss function is calculated as the sum of the localization loss - smooth-L1 loss - and the classification loss - cross-entropy loss applied to the output of a softmax layer. The localization loss regresses offsets relative to the center of the default boxes in addition to their width and height.

SSD's particular approach requires matching the default boxes to the various groundtruth bounding boxes in the input image. This is accomplished in two steps, which can result in multiple default boxes matching to the same groundtruth box:

- Each groundtruth is matched to the default box with the best IoU overlap;
- Default boxes are matched to groundtruth boxes with an IoU overlap higher than a 0.5 threshold.

3.1.2 Instance Segmentation Branch (ISB)

For an input of size 300x300, as typically used with SSD, MobileNet's final feature map has a shape of 10x10, for a downsampling factor of around 32. We modify the input size to be a

multiple of 32, thus removing the need for padding and simplifying the upsampling process. Similar to the approach of Mask R-CNN, we can extract features from this 10x10 feature map that correspond to the bounding box prediction, thereby generating a segmentation mask for each object instance in an image.

We illustrate the resulting architecture in Figure 3.3 for an input of size 320x320 and a segmentation branch size of 5x5.

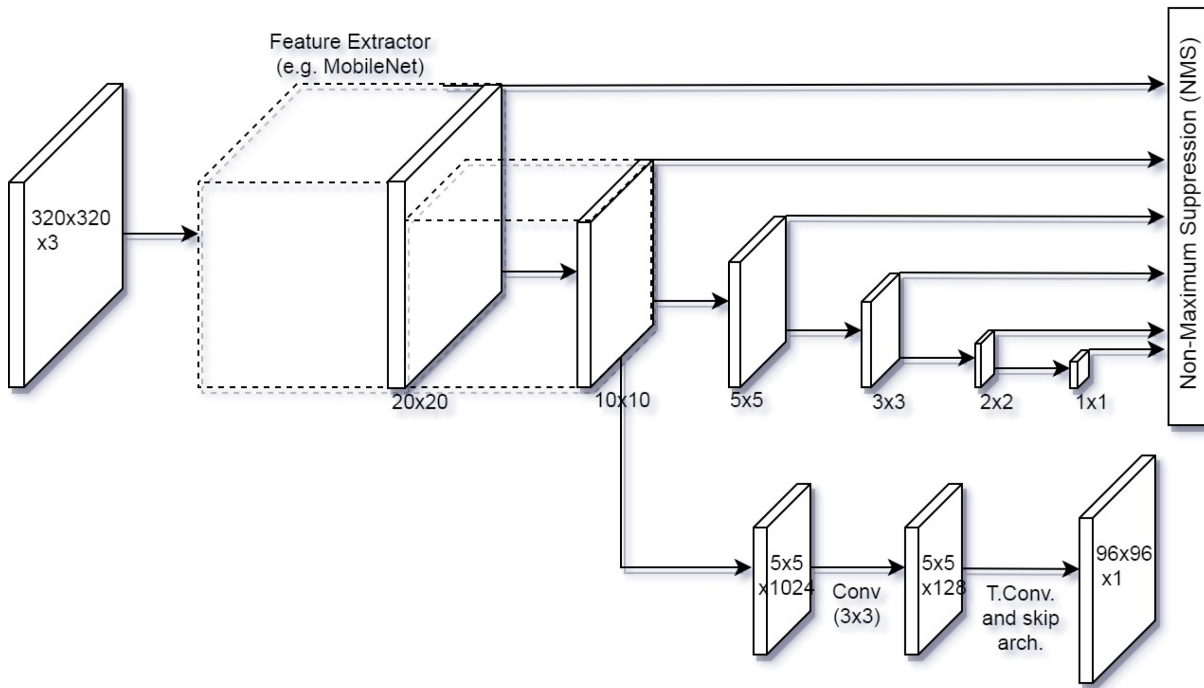


Figure 3.3: Full architecture of the Single Shot Detector adapted for Instance Segmentation.

Segmentation Architecture

The ISB's architecture is fairly simple. It is made up of 3x3 convolutional filters, inspired from Mask R-CNN's architecture, followed by a mix of transposed convolutions and skip architectures, inspired by FCN's simple but effective approach. The main variations of interest include either zero, one or two skip architecture steps, each of which is preceded by a transposed convolution to upsample the feature maps by a factor of two. The last piece is a bilinear upsampling operation to return to the correct size of the mask in the input image followed by a 1x1 scoring convolution to generate per-class mask predictions. This is

illustrated in Figure 3.4 in which the sizes of the crops and feature maps are written for a 5x5 input cropped from SSD MobileNet's 10x10 feature map - indicating a downsampling factor of 32.

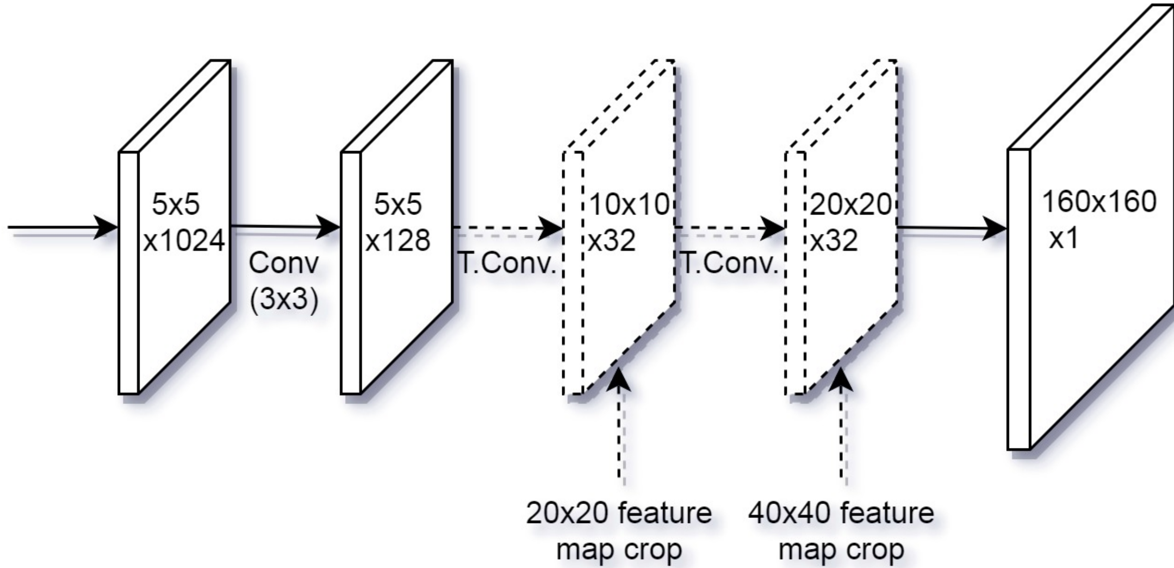


Figure 3.4: Instance segmentation branch architecture.

Note that the strides of the transposed convolutions (*T. Conv.*) can be inferred from the size of their outputs and that the dashed lines indicate the optional skip architecture. Also note that the output depth is only 1, as person-only segmentation was a focus of this research.

Loss Function

We also follow Mask R-CNN's approach of generating a single mask prediction per class and using the box classification result to choose the correct mask. As such, we also use a sigmoid binary cross-entropy loss for the mask prediction. The regression loss and the classification loss remain unchanged from SSD's: smooth-L1 and softmax cross-entropy, respectively.

3.1.3 Encompassing Bounding Box

To remove the need for any interpolation after cropping a bounding box, we study the possibility of aligning the box with the surrounding values in the feature maps, as illustrated in Figure 3.5. Using this approach, the possible bounding box sizes are reduced to a more manageable number of possible sizes, depending on the size of the feature maps from which we crop the boxes.

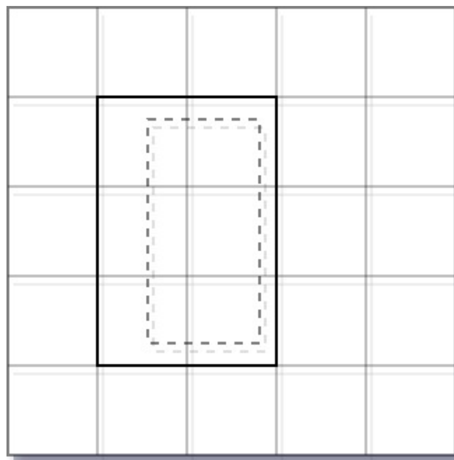
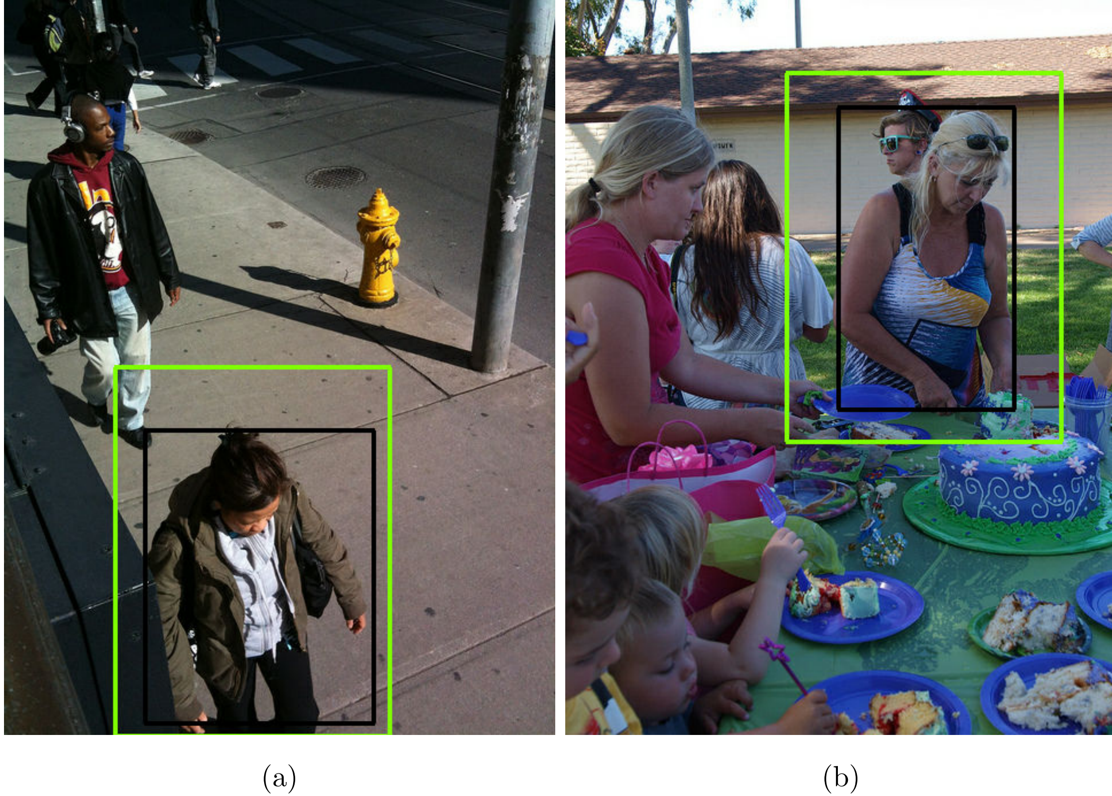


Figure 3.5: The encompassing bounding box (bold) generated from the accurate bounding box (dashed).

Figure 3.6 shows examples of this approach, where the black box corresponds to the ground truth bounding box while the green box corresponds to the encompassing bounding box. These images are first resized to a fixed input size and the bounding boxes are extended such that they are aligned with the 10x10 feature map generated late in the network. The two bounding boxes shown here have a size of 5x5 in this 10x10 feature map.

Box Regression Error Compensation

When using the network's object detection prediction for instance segmentation, this approach will avoid issues that would result from a sufficiently small offset between the bounding box prediction and the bounding box groundtruth. That is, a bounding box doesn't have to be perfectly accurate to contain an entire instance for mask prediction.



(a) (b)

Figure 3.6: Image examples when using the *encompassing bounding box*. The black box shows the ground truth and the green box shows the encompassing box.

Removing Alignment Operations and Limiting Resizing Operations

Since the bounding boxes are perfectly aligned with the feature map values, this approach does not require any alignment operation. This is different from Mask R-CNN [9] which uses its RoIAlign operation to align the bounding boxes with the feature maps. This operation requires interpolation of the feature map values to generate a 14x14 feature map.

Additionally, Mask R-CNN's single-branch approach uses this single feature map size for all the detected objects. Resizing both the feature map crop and the mask prediction result is expected to negatively affect the accuracy of the mask. The encompassing bounding box approach, on the other hand, lends itself well to avoiding, or reducing the severity of, this kind of resizing operation.

Separate Branch Weights

Not using a resizing operation has one major drawback: each possible crop shape will require its own branch, as tensors of different shapes cannot be stacked or concatenated together. This leads to one main complication: each shape will require its own set of weights and will generate its own set of feature maps, making this approach potentially very memory intensive. The expected upside is that each branch's weights will adapt to the characteristics inherent to the size of its inputs. Additionally, the architecture of the branch could be manually customized. For example, using atrous convolutions for larger boxes, where the filter needs to look further than the adjacent pixels, or fully connected layers for smaller boxes, where they do not require as many parameters.

Crop Overlap

The other disadvantage of this method is that adjacent people are more likely to be found within the same box than if the boxes encompasses the person more accurately. For instance, a 3x3 box in the 10x10 feature map can contain a relatively large amount of background pixels which, in a crowd, could be other persons standing close to the person of interest. An example, using a moderately sized bounding box, can be seen in Figure 3.7, where two persons are segmented as the same person due to their proximity.

Multiple persons standing close to each other and/or far from the camera can result in the same exact bounding boxes being cropped from the feature map, particularly when they correspond to some of the smaller possible box sizes. In this case, the network would generate the exact same segmentation mask for two different person detections.

Implementation

The segmentation branch is implemented as follows:

1. The encompassing bounding boxes' coordinates are calculated from the four bounding box coordinates (stored as maximum and minimum along both the x and the y axis) and the network's downsampling factor.

Figure 3.7: Example from the COCO val dataset of two person being segmented as the same person.

2. The available bounding boxes are pruned - depending on the branch shapes defined for the network - and zero-padded - to a set maximum of boxes for that shape.

3. For each shape, the encompassing boxes are cropped from the SSD feature map of interest and stacked together along the batch dimension, such that all boxes are treated individually.

4. When using any amount of skip architectures in the segmentation branch, the same boxes are cropped from the larger feature map(s).

5. These cropped boxes are then used as the input to the segmentation branch, whose outputs are cropped along the batch dimension to account for the earlier padding.

Training

This method can complicate the training process. Most importantly, each branch's training is dependent on the presence of appropriately shaped crops within the image. As such, an

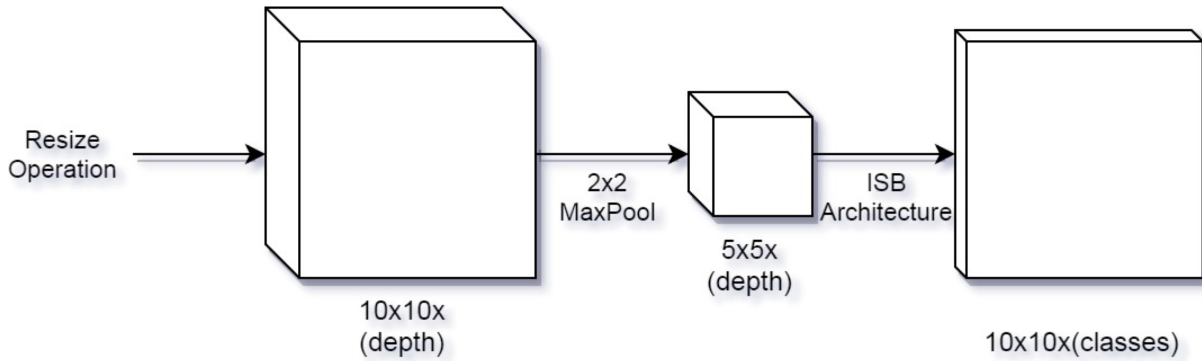


Figure 3.8: Single branch instance segmentation with SSD box predictions.

entire batch might contain relatively little training data for some branches, creating some additional training challenges.

Since the inputs for the segmentation branch are cropped using the detection bounding boxes, each branch's training will depend on the accuracy of said bounding boxes. This raises the possibility of using the ground truth boxes for training instead of the boxes generated by SSD. This avoids a problem that comes up when the segmentation training is heavily dependent on SSD's incomplete training; early training steps would get mostly incorrect crops. This will be further discussed in Chapter 5.

3.1.4 Single Branch Instance Segmentation

A direct adaptation of the Mask R-CNN framework for SSD uses a single branch for mask prediction. All areas of interest - whether a default region of interest or a predicted bounding box - are cropped and resized to a certain fixed shape. A single branch of convolutions and transposed convolutions is then used to generate the mask predictions, which are finally resized to the box's size in the original image. This is shown in Figure 3.8, with the ISB architecture shown in Figure 3.4, for a branch input size of 5x5.

While multiple resizing approaches could be considered, we follow the segmentation branch architecture of Mask R-CNN: cropping twice the input size followed by a 2x2 max pooling layer to generate the instance segmentation inputs.

Two main approaches were considered for single branch segmentation:

- Using the SSD default anchors which are also used for box regression: this enables mask prediction in parallel with box regression;
- Using the SSD final outputs: box regression and mask predictions are generated sequentially.

These approaches are detailed in sections 3.1.4 and 3.1.4 respectively.

Cropping from Default Anchor

The direct analogue to Mask R-CNN's approach for training is to use SSD's default anchors as the input for the segmentation branch. There are some important drawbacks to this approach:

- The high number of boxes will make the segmentation branch very costly, with the computing cost increasing rapidly with the batch size.
- If the objects in the dataset do not match the default anchors sufficiently well, the segmentation branch will not train properly from its inputs.
- As with Mask R-CNN, inference will still require a sequential approach to generate a proper mask: the default anchors will not be properly aligned with the bounding boxes.

While we present this approach as it would be the closest approximation of Mask R-CNN's approach, its drawbacks were too significant for a detailed comparison with the other methods. Indeed, the high memory cost of the approach severely limited the possibilities for training. The training that was undertaken did not converge in any meaningful way, resulting in an AP of 0.00 and an AR of 0.00. As such, we will not include detailed testing of this approach.

Cropping from SSD Output

The drawbacks observed when cropping from the default anchors renders it uninteresting for the purposes of this research. An alternative approach is to predict the instance seg-

mentation mask after finalizing the SSD bounding boxes, including the NMS step. This solves the issues mentioned in Section 3.1.4 as follows:

- The number of masks to predict will be reduced significantly by the NMS operation;
- The bounding boxes will be better aligned with the objects in the dataset than when using the default anchors, during both training and inference.

Apart from the difference in choosing the areas to crop, this method and the one presented above are identical: the cropped feature maps are resized to a fixed size which are passed through a series of convolutions and transposed convolutions to generate a mask prediction, as seen in Figure 3.8.

3.1.5 Multiple Branch Instance Segmentation

Single branch instance segmentation uses a crop and resize operation which we expect to hurt the instance segmentation accuracy: small regions are crudely upsampled, odd aspect rations are lost, the mask prediction must be resized again to evaluate the metrics, etc.

An alternative approach would be to use multiple instance segmentation branches in parallel, where each branch can be tailored for a specific input shape. With this architecture, shown in Figure 3.9, the layers in each branch are trained individually and can even be customized. This is implemented by looking at a slightly larger bounding box than what would only encompass the object, as described in Section 3.1.3.

We compare three ways of generating the input to these parallel branches:

- Removing all resizing after cropping: each ISB only takes the inputs of its own size in the feature map;
- Defining a range of sizes for each ISB: we implement this by setting a list thresholds for box size, and resizing all boxes to the next smallest threshold in the last;
- Mixing both methods above to segment some shapes with their own ISB and resizing the remaining boxes into an additional ISB.

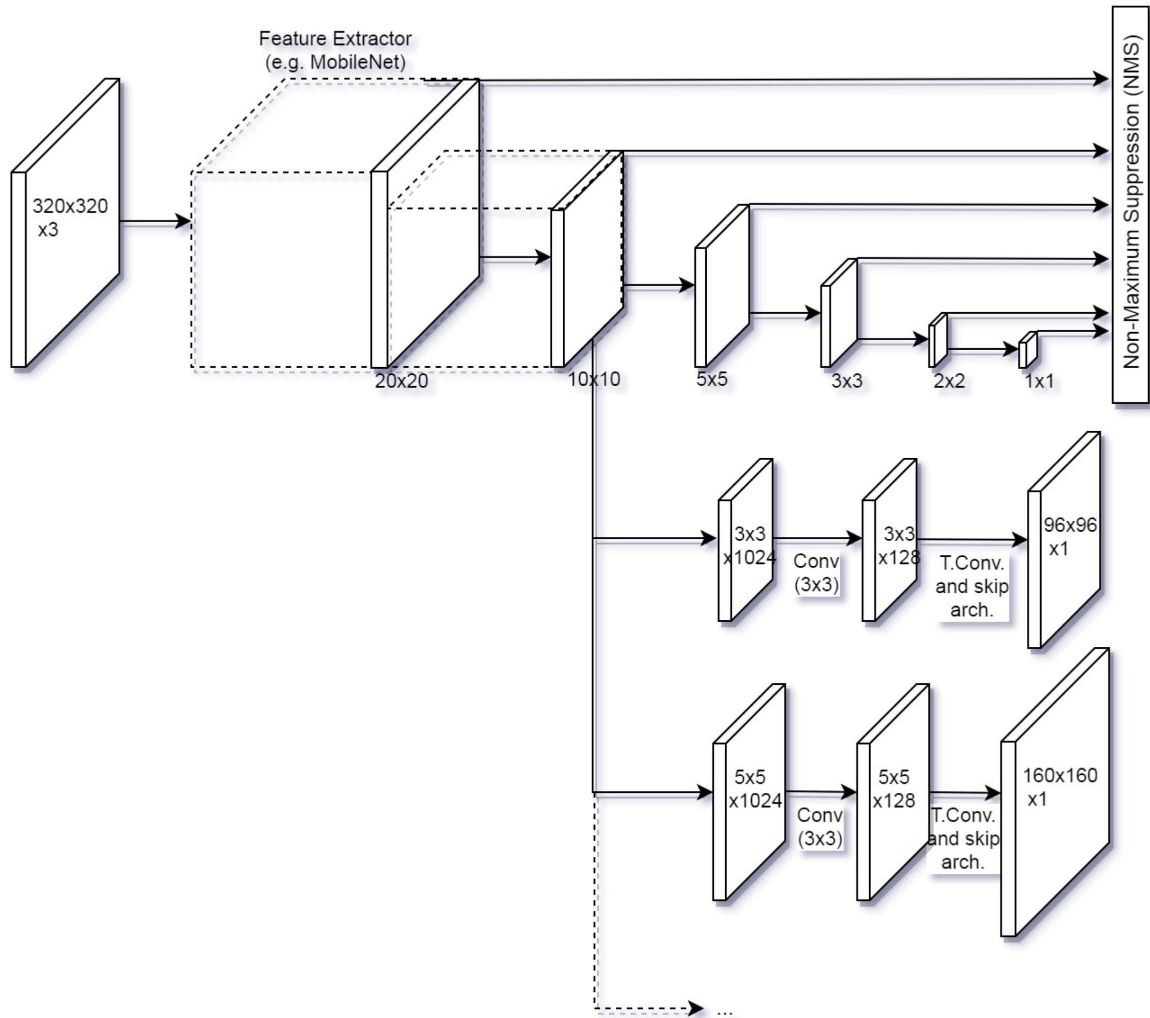


Figure 3.9: Full architecture of the Single Shot Detector adapted for Instance Segmentation with multiple instance segmentation branches.

Single-shape ISB

The single-shape has the advantage of completely removing the need for a resizing operation in the network. However, it will need to be much more specialized than the size range approach due to the increase in memory requirement and model size with every additional branch: we will not be able to cover every single possible size without significantly increasing the size and computational cost of the model.

The separation of instance sizes also open up the possibility of customizing the branch's architecture on a per-size basis. For example, using atrous convolutions for large objects,

reducing the size of the convolutional filters for very small objects (i.e. smaller than 3x3 in either dimension), using additional convolutional layers for numerous objects sizes, etc.

Size Range ISB

The size range approach, on the other hand, offers a compromise between the single-branch approach and the single-shape approach. It still requires some amount of resizing, but without completely decimating the aspect ratio, and it can cover a wider range of possible shapes without requiring a set of parameters and feature maps for each of them. Much like the single-shape approach, this creates the possibility of customizing the size thresholds and the architecture of the branches based on the requirements. The leaves a question as to the effect of the compromise on the quality of the results when compared to the other two approaches.

We choose to implement this approach by defining a set of size thresholds and resizing all boxes smaller than one threshold to the size of the next threshold. While it would be intuitive to resize to the middle of the threshold, we choose this approach as it reduces the memory cost of each ISB by reducing the size of the feature maps after resizing.

Single-shape ISB with Remaining Boxes ISB

In addition to the two approaches detailed above, we define a third approach which mixes them together. We define a list of single-shape ISB's for the most common smaller instances in the dataset and a single ISB with resizing for the rest of the possible shapes. It would be costly to cover the range of possible large instances with parallel ISB's, due to number of possible shapes with, for example, at least one side larger than 10 in a 20x20 feature map. As such, we simply resize all of them, in addition to the remaining boxes of smaller sizes, and use a single branch for their mask prediction.

3.2 Implementation Details

The only feature extractor we test with our instance segmentation architecture is MobileNet-v1, using the TensowFlow Object Detection API [50]'s implementation of SSD. Modifica-

tions were made to SSD for our instance segmentation experiments.

Due to the specialized nature of the multi-branch architecture, we focus on the person class for our instance segmentation tests. This should enable us to cover most of the annotation's possible shapes with fewer branches than if considering a wide range of classes, or a single class with a wider range of shapes.

3.2.1 Network Details and Parameters

To provide a fair comparison between the various methods, we use the same network parameters whenever possible. This includes using a consistent input size, weight initialization, learning rate, batch size, etc.

Initial Weights We initialize the network with the SSD-MobileNetv1 weights pre-trained on the COCO dataset. These weights are available for the versions of SSD-MobileNetv1 with a depth multipler of 1.0 and 0.75.

Input Size We report the architecture's accuracy for images resized to an input size of 640x640. While the COCO images have a range of image sizes, a 640x640 input upsamples most of the images in the dataset without increasing the memory cost past the GPU's available VRAM. Note that the 640x640 input downsamples by 32 without requiring and padding, thereby simplifying the task of upsampling with transposed convolutions.

Loss The final loss function used here is the sum of three functions: the smooth-L1 box regression loss, the softmax cross-entropy box classification loss, and the sigmoid cross-entropy mask loss. These three are simply weighted as per their respective hyper-parameters and summed together. The mask loss is calculated after the skip steps, but without upsampling to the size of the bounding box in the input image; instead, the groundtruth mask is resized to the size of the predicted mask. This reduces the memory cost during training, enabling the use of a larger batch size.

As mentioned in Section 3.1.3, we use the groundtruth boxes to crop the instances to train the ISB's. This removes the need for SSD to predict accurate bounding boxes before

improving the quality of the mask predictions. It also alleviates the issue resulting from the lack of an equivalent to Mask R-CNN's region proposals in the SSD architecture.

Loss Weights The use of pre-trained weights for the box classification and box regression parts of the network raises an important issue since we have to train mask prediction from scratch. We wish to modify the mask prediction parameters at a large scale, but only to fine-tune the box prediction parameters to fit our dataset with a single class. As such, we multiply the classification and regression loss by 1.0 and multiply the mask loss by a larger number, thereby training the weights of the latter much faster.

We use a factor of 150 for all the multi-branch approaches and a factor of 30 for the single-branch approach; the difference is due to the need to train each branch separately for the multi-branch approaches.

Learning Rate We use an exponential decay learning rate with an initial learning rate of 4×10^{-5}, decay steps of 10000 and a decay factor of 0.75, with the RMSProp optimizer. We increase the decay steps to 15000 when training on all COCO person annotations, to account for the larger number of images. Equation 3.2 shows the resulting learning rate function, for learning rate lr, initial learning rate lr_0, decay rate k, current step number t and decay steps t_d.

$$lr = lr_0 k^{\lfloor t/t_d \rfloor} \tag{3.2}$$

Due to the fact that we are using pre-trained weights for initialization, we start with a small learning rate.

Batch Size To counteract the high memory cost of training a network for instance segmentation, we use *gradient accumulation*: instead of backpropagating after every single batch, we accumulate the gradients for a set number of batches and average them before backpropagation. This increases the effective batch size of the network during training. We then use a batch size of 6 with a gradient accumulation of 4 batches, resulting in an effective batch size of 24.

Stop Gradient Due to the difficult nature of training many branches in parallel, we opt to prevent the gradient from backpropagating past each branch's cropping operation. Effectively, this means we train each ISB individually and that the feature map from which we crop is trained from box prediction task only. The main reason behind this is to prevent the disparity in the number of boxes of each size from affecting the training of the branches.

Mask Binarization Threshold Since the masks are calculated using a sigmoid activation function, we need to pick a threshold above which pixels are considered as positive detections for the class of interest. We use a threshold of 0.0 for these tests.

Instance Segmentation Branch Architecture The ISB architecture was previously outlined in Figure 3.4. We use a depth of 128 for the 3x3 convolutional layers, and a depth of 32 for the transposed convolutions. Depthwise convolutions are used instead of regular convolutions for the 3x3 convolutional layers to reduce the cost of each branch. The final output has depth equal to the number of classes.

Number of Boxes per Branch While the single-branch approach can simply generate a mask for all 100 boxes predicted by SSD, we need to limit the number of boxes used with any of the multi-branch approaches. Padding to 100 boxes in every parallel branch would quickly increase the memory cost of the network. As such, we use 100 boxes in each branch for the single-branch approach and 30 boxes in each branch for the multi-branch approaches.

3.3 Conclusion

In this chapter, we proposed an instance segmentation model architecture, based on the SSD object detector and the MobileNet image classifier, which should achieve low model size and fast speeds. We also propose various methods for minimizing resizing operations within a network, which are expected to be both computationally costly and to harm the accuracy of the segmentation masks.

All variations of the SSD-ISB architecture presented in this chapter are tested thoroughly in Chapter 5.

Chapter 4

Semantic Segmentation with Low-Cost Networks

As mentioned before, there are two main area of interest for this work regarding the task of semantic segmentation:

- Applying simple methods for semantic segmentation to networks that are more computationally efficient, thereby studying the possibility of fast, low-cost semantic segmentation networks for use in wider applications;
- Studying the segmentation of cropped pedestrians instances with small changes to the network's hyperparameters for application in a fast, end-to-end, instance segmentation network for application in low-power use-cases.

Section 2.5 covers relevant work in the field of semantic segmentation, while section 4.1 covers the dataset used for these experiments, which are detailed in sections 4.2 and 4.3 respectively.

4.1 Dataset

The only dataset used for the tests in this chapter is the *Cityscapes* dataset. The other commonly used semantic segmentation dataset is PASCAL VOC, which is not used in these experiments due to its very limited number of images.

4.1.1 Cityscapes

The Cityscapes dataset [39] is built from images taken from a specific vantage point: the dashboard of a car as it drives around 50 European cities. All the pictures are taken in the daytime. The annotations for this dataset are available for both semantic and instance segmentation; evaluation servers and corresponding online leaderboards are available for both tasks. 30 classes are used for semantic segmentation annotations, but 11 of those are not included for evaluation. Only 10 of those 30 classes are used for instance segmentation, and 2 of those 10 are not included for evaluation. When the adjacent objects cannot be clearly distinguished, they are labeled as a group of the appropriate class.

5000 images are provided with *fine* annotations - that is, annotations for which the segmentation ground truths closely follow the objects' borders - and 20 000 images with *coarse* annotations - for which the segmentation ground truths do not follow the object's borders and instance annotations are not consistently provided. These images are taken from the 20$^{\text{th}}$ image of many 30 frame video sequences, which are also available but not annotated. The *fine* images create the main dataset in a 2975/500/1525 train/val/test split, while the *coarse* images are provided for additional training. Figure 4.1 shows some annotated images from the Cityscapes dataset[1]: see Figure 4.1a for an example of the fine annotations and Figure 4.1b for an example of the coarse annotations.

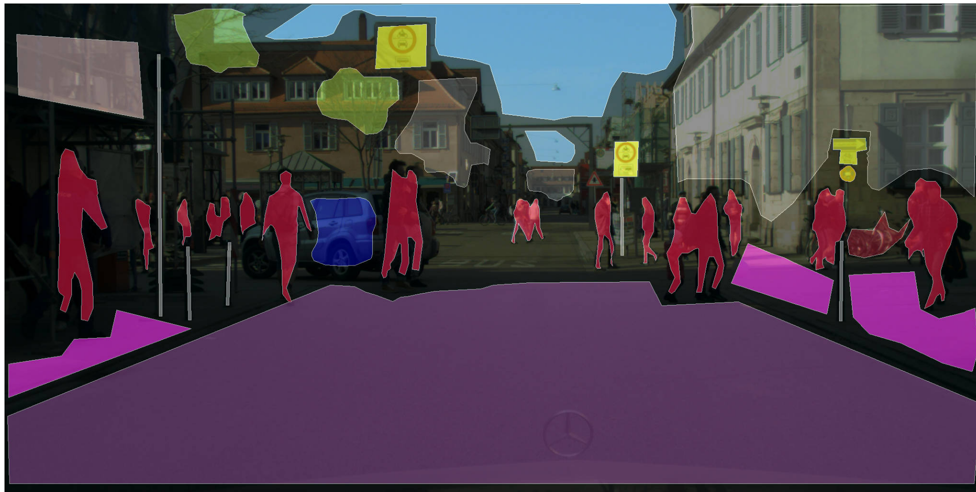
The persons typically seen in this dataset have more regular poses than those seen in other datasets such as MS-COCO [30] or PASCAL VOC [29], where they might be playing sports or where the camera might be at an unusual angle. Instead, they either fall in the *person* (or pedestrian) class or in the *rider* class for persons on motorcycles or bicycles. This is of particular interest for the work shown in section 3.1.5 on instance segmentation.

Metrics

The main metric used by Cityscapes for semantic segmentation is the mean IoU presented in Section 2.5: we calculate the intersection-over-union between the predicted pixels and the groundtruth pixels for each class, and average over all the classes in the dataset.

[1] We use the default Cityscapes colour map: light blue for sky, dark blue for car, light red for pedestrian, light pink for road, bright pink for sidewalk, green for vegetation, grey for building, yellow for traffic signs, light orange for traffic lights, dark orange for *dynamic* objects, etc.

(a) Fine annotations.



(b) Coarse annotations

Figure 4.1: Image examples from the Cityscapes dataset.

For instance segmentation, Cityscapes uses the same metrics as MS-COCO, which are defined in Section 5.1. As secondary metrics, they also use the average precision only for objects within 100 meters and 50 meters: AP_{100m} and AP_{50m}, respectively.

4.2 Network Comparison with Cityscapes Dataset

The first test we ran on semantic segmentation was a rather straightforward comparison of two low-cost CNNs adapted for segmentation following the common approaches described

in sections 2.5.1 and 2.5.2. The goal was to compare the two CNNs to each other, using simple but effective methods for segmentation mask prediction.

4.2.1 Comparison Details

The implementation of the CNNs used in these tests is composed of two main parts:

1. The main network used as an encoder, called the *backbone* or *feature extractor*;
2. The segmentation method applied to the encoder - note that we always follow the base concept of removing parts of the network that decimate spatial awareness, such as fully connected layers and global pooling layers.

Encoders

The two networks compared in this section are SqueezeNet and MobileNet, described in section 2.3.3 and 2.3.4 respectively. Since MobileNet achieves better results for image classification with a lower number of operations, we intuitively expect it to beat the performance of SqueezeNet on semantic segmentation. However, we believed both were worth exploring, as their designs have different solutions to deep learning's computational cost problem.

SqueezeNet SqueezeNet's implementation follows the default architecture outlined in Table 2.4. This includes not only the number of layers, but also the number of squeeze and expand filters used in each fire module.

MobileNet Similarly, MobileNet's implementation follows the architecture presented by Howard et. al. in their paper, outlined in Table 2.5. For this experiment, we use a depth multiplier of 0.5. As shown in Table 2.6, this provides results slightly higher than SqueezeNet at a cost much lower than that of MobileNet with a depth multiplier of 1.0. Due to the modification of the network for semantic segmentation, we do not include the input resolution as a hyperparameter, though we mention that all tests are undertaken with the same input size.

Segmentation Methods

We compare two main segmentation methods: FCN's skip architecture and DeepLab's use of atrous convolution. Both generate decent results for semantic segmentation when applied to deep networks such as VGG-16, as outlined in sections 2.5.1 and 2.5.2. In this section, we apply both of them to the SqueezeNet and MobileNet networks.

Skip Architecture The skip architecture introduced by Long et. al. [15] increases the number of layers in the network by adding a short decoder applied to the output of the encoder. Because SqueezeNet downsamples the input by a factor of 16, instead of the more common factor of 32, we use a different naming scheme for the three SqueezeNet FCN architectures: FCN-16s, FCN-8s and FCN-4s, using zero, one and two upsampling steps. The skip architectures applied to MobileNet follow the same naming scheme as that used by Long et. al.: FCN-32, FCN-16 and FCN-8 for zero, one and two upsampling steps.

Atrous Convolution Based on the applications of atrous convolution introduced by Chen et. al. [16], we apply atrous convolution in two ways in this section: reducing the output stride of the network to 8 and 4 for SqueezeNet and to 16 and 8 for Mobilenet, compared to the typical 32, and increasing the atrous rate of the last convolutional layer in the network. The latter of these does not cause any increase in the cost of the network, while the former significantly increases the size of the feature maps in the network without increasing the number of parameters. For both applications of atrous convolution, we use a softmax operation followed by bilinear interpolation to generate class predictions for each pixel in the input image, rather than using a decoder.

4.2.2 Implementation Details

All architectures studied in this work were implemented using the TensorFlow framework [51]. This choice was made due to its common usage in deep learning research and to the easy availability of various pre-trained models.

Loss We use the exact same softmax cross-entropy as Long et. al. for FCN. The output of the network has the height and width of the input image and depth equal to the number

of classes in the dataset. The softmax operation generates probabilities for each class on a per-pixel basis which we compare with the per-pixel class labels of the groundtruth. A cross-entropy loss is calculated for each pixel and then summed over the entire image, generating the loss for one image.

Learning Rate Unlike many deep learning applications, we found that our architectures fine-tunes well with a constant learning rate of 10^{-5}, which we used this for every network trained in this chapter. The optimizer used to train with this learning rate is RMSProp. We train the networks for 300 000 steps and report the best mIoU on the Cityscapes val subset.

Initial Weights For both SqueezeNet and MobileNet, we initialize the weights from those provided by after training on ImageNet for image classification and fine-tune them for semantic segmentation.

Input Size The Cityscapes images are not resized for segmentation. We use an input size of 2048×1024. This does come with a heavy memory cost due to the large size of all the feature maps in the network.

Batch Size Due to the large memory footprint of the Cityscapes images, we train with a batch size of 1.

4.2.3 Experimentation Results

We train each network for 300 000 steps with a batch size of 1, evaluating on the val subset every 10 minutes of training. In Table 4.1, we report the best accuracy for each model using the mIoU metric defined in Section 2.5, used in the Cityscapes leaderboard for semantic segmentation. In Table 4.2, we report the speed and memory usage of each configuration.

Note that the two *SqueezeNet* configurations with stride 16 and the *MobileNet* configuration with stride 32 are equivalent to using bilinear upsampling on the output of the respective encoder with no semantic segmentation methods being applied.

Final Feature Map Stride	SqueezeNet FCN	SqueezeNet Atrous	MobileNet FCN	MobileNet Atrous
32	-	-	41.07	41.07
16	35.61	35.61	43.78	42.75
8	36.22	36.37	43.46	44.25

Table 4.1: Mean IoU results for the SqueezeNet/MobileNet comparison study of semantic segmentation on Cityscapes.

Looking at Table 4.1, we see that the most promising architectures all have MobileNet as an encoder. The output stride of 8 provides the best accuracy, but the increase in feature map resolution - by a factor of 8 in the last feature map of Mobilenet, and a factor of 4 in the preceding 13 feature maps - will result in a significant increase in memory cost for this network, as seen in Table 4.2. Comparatively, the FCN16 and FCN8 architectures achieve slightly worse accuracy, at the cost of about 10000 and 20000 additional parameters respectively, and an extra two feature maps with a depth of 19 in the decoder.

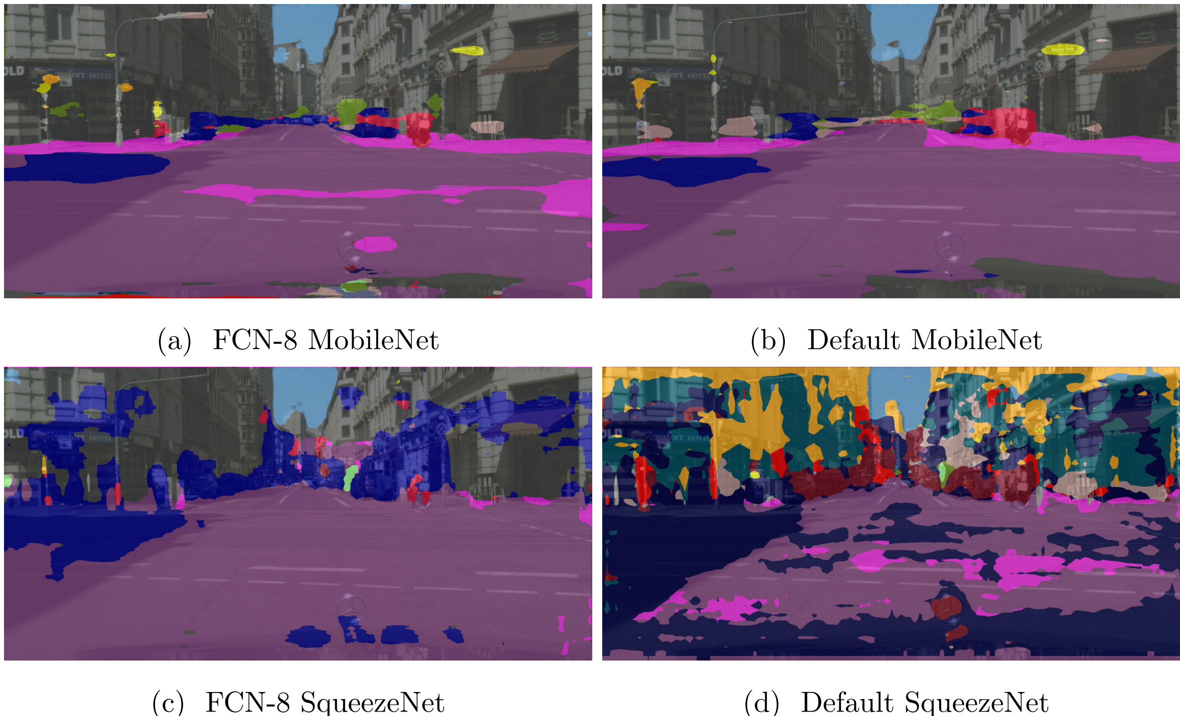
Unexpectedly, we see FCN16 performing better than FCN8 when implemented on the MobileNet encoder. However, we do expect to see diminishing returns with each added step in the decoder, and MobileNet's large number of layers between the last stride and the one preceding it would reduce the usefulness of the feature map with a stride of 8.

Encoder Comparison

With MobileNetv1 with a depth multiplier of 0.5 performing better than SqueezeNet on ImageNet by 6.2%, we would expect to see MobileNet achieving better results on Cityscapes by a similar margin. This is indeed what we see in Table 4.1, comparing SqueezeNet's accuracy for stride 16 and MobileNet's accuracy for stride 32. The difference is of about 5.46 mIoU.

Figure 4.2 shows a comparison of the *FCN-8* architecture and the default architecture for both the MobileNet encoder and the SqueezeNet encoder. We clearly see the poor quality of the SqueezeNet segmentation prediction, particularly when compared to the

MobileNet predictions.



(a) FCN-8 MobileNet (b) Default MobileNet

(c) FCN-8 SqueezeNet (d) Default SqueezeNet

Figure 4.2: Sample output with all 19 Cityscapes classes, for the FCN8 architectures and the default architectures of the MobileNet and SqueezeNet encoders.

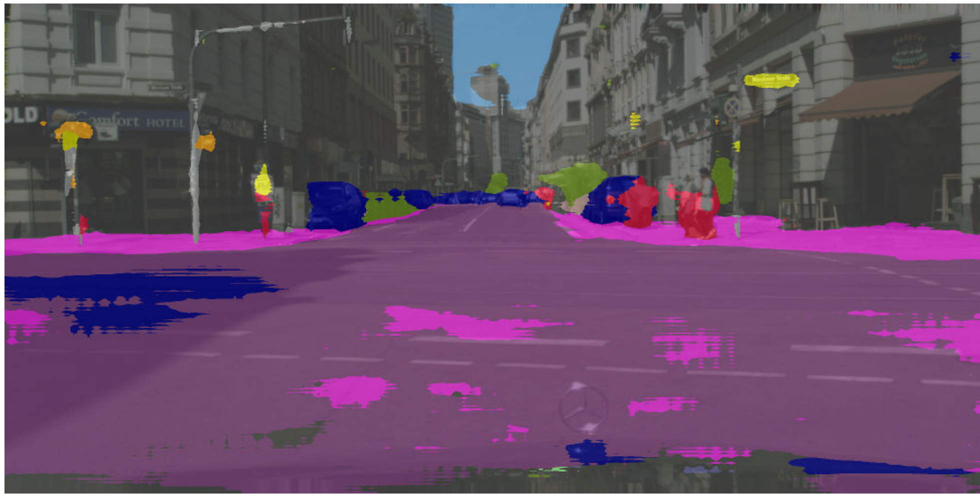
The same trend can be observed when comparing SqueezeNet with stride 8 and MobileNet with stride 16 - both of which are implemented by using a single decoding step or by reducing the last stride to 1. In this case, we see a difference in the mIoU metric of about 7.56 and 6.38 for the FCN architecture and the atrous architecture respectively.

Segmentation Method Comparison

Comparing the FCN and atrous segmentation methods doesn't show a pattern as clearly. Using a single skip step at the end of the encoder before the bilinear upsampling shows an improvement of 0.61 mIoU and 2.71 mIoU for SqueezeNet and MobileNet respectively at a fairly low cost.

However, this pattern does not hold for MobileNet's FCN8 and output stride 8. More importantly, FCN16 achieves slightly better results than FCN8, indicating that the additional feature map does not contribute useful features to the results. On the other hand,

(a) FCN-8 MobileNet



(b) Output Stride 8 MobileNet

Figure 4.3: Sample output with all 19 Cityscapes classes, for the FCN8 architecture and the output stride 8 architecture of the MobileNet encoder.

both reductions of the encoder's output stride show a significant improvement in the quality of the network's predictions: MobileNet-stride8 shows an improvement of 1.5 mIoU over MobileNet-stride16.

Figure 4.3 shows a visual comparison of the *FCN-8* architecture and the *Stride-8* architecture for the MobileNet encoder. Most importantly, we see that the stride 8 approach generates some artifacts in the smaller patches of incorrect predictions. Other research in the semantic segmentation field [14] posits that this is due to the way that a sequence of atrous convolutions with the same rate. This will incur a *gridding* effect by preventing the

networks from looking at all the surrounding values in the feature map.

Speed and Memory Requirements

We evaluate and compare the number of frames per second (FPS) of each architecture on both GPU and CPU, in addition to the memory usage for inference using TensorFlow [51] and the number of parameters in the network. These numbers are collected using TensorFlow's *Profiler*, which provides various useful information about a TensorFlow graph. Note that there are multiple memory usage metrics, none of which are not perfectly accurate. We use the "output bytes" metric - the memory that is output from the operation - as the memory usage reported by the other metrics did not correctly represent the amount of memory used by the FCN architectures. Most importantly, the *Memory Usage* metric represents the difference in memory requirement between the different architectures, even if the absolute number cannot be directly applied to a system.

All results were measured on the same system, with a GeForce RTX 2080 Ti GPU and an Intel Core i9-9900K CPU, and are reported in Table 4.2. The inputs used are Cityscapes images of size 2048x1024x3; the analysis will generalize to other images, though it will be much easier to reach high framerate and low memory cost with smaller input sizes.

While most of the FPS measurements are consistent with what we would expect: the time and memory cost of an architecture increases with either its complexity or the size of the feature maps. We see three main exceptions:

1. SqueezeNet-Base is slower than both its more complex counterparts;
2. MobileNet-Base is slower than MobileNet-FCN16 and MobileNet-Stride16, but faster than MobileNet-FCN8 and MobileNet-Stride8;
3. MobileNet-FCN8 is faster on CPU, but slower on GPU, than MobileNet-Stride16.

The low speed of the base networks might be caused by inefficiencies in the bilinear resizing operation, in which case resizing the 64x32 feature map to 2048x1024 would be much more costly than resizing from a 128x64 feature map. Additionally, the low depth within the FCN architecture makes it cheap to implement while the stride16 architecture

Architecture	FPS (GPU)	FPS (CPU)	Memory Usage (MB)	Number of Parameters
SqzNet-Base	43.29	3.69	2089.03	732243
SqzNet-FCN8	74.40	3.74	2122.40	740470
SqzNet-Stride8	48.45	2.43	3744.08	732243
MN-Base	75.47	5.59	1609.49	839283
MN-FCN16	80.19	5.62	1612.21	849942
MN-FCN8	46.92	5.33	1628.20	858169
MN-Stride16	67.84	5.09	1874.31	839283
MN-Stride8	33.61	1.91	5088.76	839283

Table 4.2: Speed and memory usage for SqueezeNet/MobileNet comparison study of semantic segmentation on Cityscapes.

only changes the size of the last feature map when applied to MobileNet. This concept also applies to SqueezeNet, where the last two feature maps change size with the stride8 architecture.

The difference in speed between MobileNet-FCN8 and MobileNet-Stride16 probably results from a more efficient implementation of the convolution operation on GPU. This explains the much smaller impact on GPU of increasing the feature map size, while the operations on CPU don't suffer as much from adding the two skip architectures.

On the other hand, we see that MobileNet-FCN8 and MobileNet-Stride8 are slower than MobileNet-Base and, very importantly, that the striding architectures are constantly slower than the FCN architectures. Particularly, the MobileNet-Stride8 architecture has 14 feature maps of increased size, which explains it is the slowest architecture on both GPU and CPU - though it is still fast enough to be considered *real time*.

Memory Usage The memory usage of the striding architectures is constantly much higher than the base architectures, while the FCN architecture are only slightly more costly than the base architectures. Particularly, the FCN-16 architecture only increases the memory usage by about 3MB, most likely due to the small size of the feature map

required in the first skip step: only 128x64x19.

Number of Parameters The number of parameters in constant in all the baseline networks and their corresponding striding architectures; this is to be expected as atrous convolution does not change the number of parameters. On the other hand, the FCN architectures require a slightly higher number of parameters than their base counterparts. Each skip step adds about 10000 parameters, which will only cause a small change in the size of the network.

4.3 Cityscapes Person Only Segmentation

While studying semantic segmentation with a large number of classes is useful from a research standpoint, there are few studies that compare segmentation methods for fewer classes. This is of particular interest for specific applications such as assisted driving, where we can focus on the classes, or even the categories, that offer relevant information for the driver, such as pedestrians and cars.

Additionally, the task of instance segmentation can benefit from studies of semantic segmentation. The instance segmentation method studied in chapter 5 predicts a segmentation mask for a single cropped instance, without regard for the instance's class. This type of approach can be approximated by first cropping objects from an image, and then predicting a segmentation mask. This enables us to study the segmentation of instances without the complexity of training a CNN on both segmentation and object detection.

As such the results observed in this section informed latter tests with instance segmentation. The effect on segmentation accuracy of the tighter or wider cropping of a person partially motivated the generous cropping method used in chapter 5.

4.3.1 Comparison Details

Instead of calculating a segmentation mask for an entire image, we generate a dataset from the Cityscapes dataset. Using the person instance annotations, we crop each person instance into a new image which we then use to train a network on person segmentation.

We only keep the segmentation annotation for the person in the center of the instance, effectively measuring the accuracy of the instance segmentation task instead of the semantic segmentation task.

The goal of the following comparisons is to study the effect of details related to the cropping of instances on the prediction of accurate segmentation masks. To reduce the number of tests to be undertaken, we used MobileNet as the base architecture, with either *FCN-8s* or simple bilinear upsampling. Three aspects are compared in this section:

1. Input size of the cropped image: two image sizes for the cropped objects are tested and compared - 320x192 and 160x96;
2. Additional background information: while bounding boxes are generally expected to tightly encompass an object, segmentation masks can benefit from additional context - the height and width of the crop before resizing are increased by 0, 5 and 10 percent of the crop size on every side of the instance;
3. Atrous and striding: keeping in line with the need for an object's context, we compare all possible combinations of atrous rates 1, 3 and 5, and of output strides 8 and 32.

Of note is that the aspect ratio of the cropped instances is not modified during resizing. Instead, the instance's bounding box is modified to a certain aspect ratio, while keeping the instance in the middle of the box, before cropping - see Figure 4.4a for an example.

Image Size and Effect of Downsampling

The two input sizes tested here are 320x192 and 160x96 (height by width). Both share the same aspect ratio of approximately 1.67, which is a good approximation of a pedestrian's size (i.e. a human doing regular activities). After downsampling by a factor of 32, the smallest feature maps will have sizes of 10x6 and 5x3 respectively.

Additional Background Information

It was also important to study the effect of additional background on the quality of the segmentation mask. From the ground truth information, we can crop an instance with a

minimal amount of extraneous data. This kind of background information might might consistently be misclassified or it might help in the classification of the instance's pixels - for example, a red car that stands out from the gray road.

Additionally, if we obtain bounding boxes from an object detection network, we do not expect it to be perfectly accurate. Any missing pixels belonging to the instance cannot be recovered by the segmentation mask, but any additional background information can be misclassified. The results of this test are particularly relevant for the multi-branch instance segmentation network studied in section 3.1.5 of chapter 5, as it purposefully adds background information during mask prediction in an effort to reduce the variance in crop sizes.

Figure 4.4 compares the size after cropping when using no additional background, using 5% additional background on every side, and using 10% additional background on every side.

(a) Sample instance with no additional background.

(b) Sample instance with 5% extra background on every side.

(c) Sample instance with 10% extra background on every side.

Figure 4.4: A comparison of the different additional background options compared in this section.

Atrous and Striding

Finally, we compare two parameters which take advantage of atrous convolution:

- Values of 1, 3, and 5 for the atrous rate of the last layer in the network;
- An output stride of 32 and 8 for the network, based on the concepts introduced for the DeepLab network.

4.3.2 Implementation Details

Loss As in Section 4.2, we use the same softmax cross-entropy as Long et. al. for FCN: the final mask prediction after softmax has the same size as the input image, and we apply cross-entropy loss on a per-pixel basis.

Learning Rate We use the same learning rate as in Section 4.2: a constant learning rate of 10^{-5}, optimized with RMSProp, for 400 000 steps. We report the best mIoU on the validation subset.

Initial Weights We initialize the network with the weights trained on ImageNet for image classification.

Training and Validation subsets We generate the training subset by using the instance annotations provided for the regular Cityscapes training subset. We take the bounding box for those annotations, increase the height or width as necessary to maintain a constant aspect ratio of approximately 1.67, and crop the result from the original image. The same procedure is applied to the Cityscapes validation images, from which we create our validation subset. This provides us with 16 949 images for training and 3 201 images for validation.

Batch Size With the significantly smaller input size, as compared to the Cityscapes images, we can train with a batch size of 4. To provide a fair comparison, we use the same batch size for both input sizes.

4.3.3 Experimentation Results

We train each architecture for 400 000 steps with a batch size of 4. Instead of reporting the mean IoU as in Section 4.2, we report the IoU on the person class only. As we focus on a single class for these tests, the mIoU would only average two metrics: the background IoU and the person IoU.

Table 4.3 shows the results for our baseline network, using no segmentation architecture, while Table 4.4 shows the results when using the FCN8 decoder.

Additional Background	Last Atrous Rate 1	Last Atrous Rate 3	Last Atrous Rate 5
0.00	54.24	54.82	54.70
0.05	50.78	50.47	50.56
0.10	46.88	49.62	48.61

Table 4.3: IoU results of MobileNet for person-only study; baseline

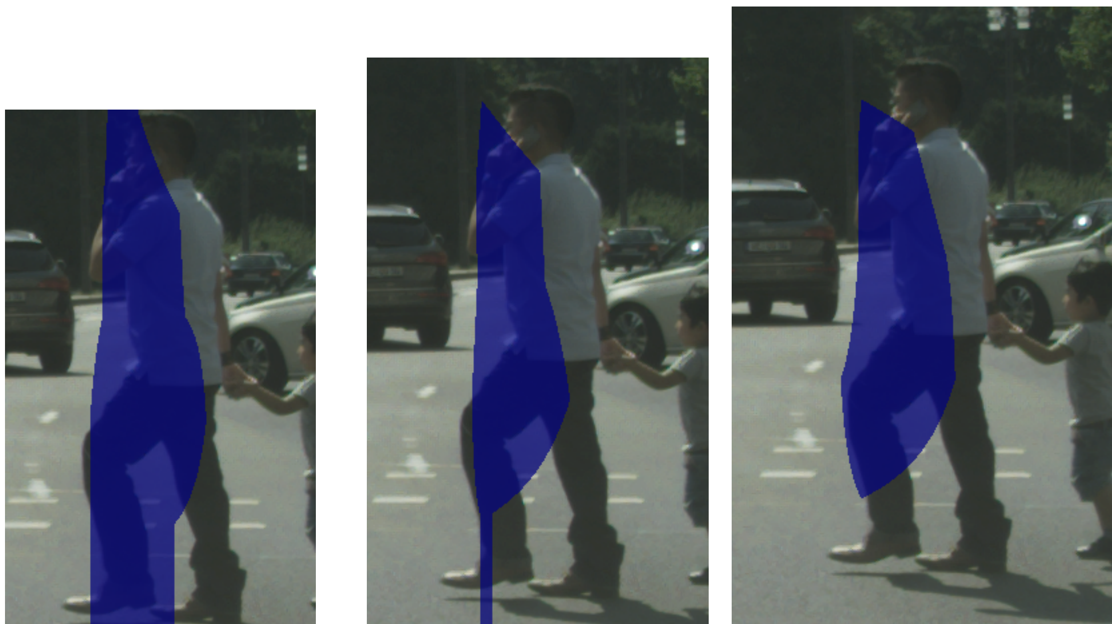
Figure 4.5 shows a comparison of the output of the MobileNet network without skip architecture and an output stride of 32. We can notice that the network does not produce a particularly useful output in any of the cases.

FCN Architecture

Based on the results of Section 4.2, we are mainly interested in studying the effect of the FCN architecture. Tables 4.5 and 4.4 show the results for the FCN-16 architecture and the FCN-8 architecture respectively. Sample outputs for FCN-16 and FCN-8 can be seen in figures 4.6b and 4.6c respectively.

Comparing tables 4.3 and 4.4, we notice that the addition of the FCN8 decoder shows a significant improvement in the accuracy of the person segmentation masks.

Most importantly, we notice a significant drop in IoU when increasing the size of the instance crop in the baseline: from 54.24 to 46.88 for an atrous rate of 1. This drop is particularly interesting because its amplitude is reduced significantly when using FCN8:



(a) No additional background. (b) 5% extra background on every side. (c) 10% extra background on every side.

Figure 4.5: A comparison of the output for different additional background options with no skip architecture, an output stride of 32, and an atrous rate of 3 in the last layer.

(a) Output stride 8. (b) FCN-16. (c) FCN-8.

Figure 4.6: A comparison of the output for the different segmentation methods: output stride 8, FCN-16 and FCN8 respectively, all with an atrous rate of 3 in the last layer.

Additional Background	Last Atrous Rate 1	Last Atrous Rate 3	Last Atrous Rate 5
0.00	68.44	68.80	68.30
0.05	68.60	68.07	67.88
0.10	67.57	68.60	67.36

Table 4.4: IoU results of MobileNet for person-only study; FCN8 configuration.

Additional Background	Last Atrous Rate 1	Last Atrous Rate 3	Last Atrous Rate 5
0.00	67.78	67.84	67.48
0.05	66.58	66.52	66.04
0.10	66.08	66.23	66.38

Table 4.5: IoU results of MobileNet for person-only study; FCN16 configuration.

from 68.44 to 67.57 for an atrous rate of 1. This difference is fairly consistent between the three atrous rates.

We notice that the FCN-16 architecture achieves comparable accuracy to the FCN-8 architecture, with a difference between 0.5 and 2 IoU. Since we're looking at a similar cost differences as reported in Table 4.2, the FCN-16 architecture looks very promising for future tests.

While the increased crop size does hinder the segmentation task, it does not prevent the prediction of an accurate mask. Since a low-cost instance segmentation method is unlikely to generate very accurate bounding boxes, this is a very important result for the experiments of Chapter 5.

Atrous and Striding

In Table 4.6, we report the accuracy of MobileNet with an output stride of 8. A sample output can be seen in Figure 4.6a.

Additional Background	Last Atrous Rate 1	Last Atrous Rate 3	Last Atrous Rate 5
0.00	70.05	69.98	69.93
0.05	69.50	69.63	69.53
0.10	69.12	69.66	69.74

Table 4.6: IoU results of MobileNet for person-only study; atrous convolutions with stride 8.

We can then compare the use of atrous convolution both within the network to reduce its output stride, and in the last layer of the the encoder to provide a larger field of view.

As observed in Section 4.2, an output stride of 8 coupled with bilinear upsampling provides better results than the FCN8 architecture. In this case, we observe a difference between 1% and 2%, once again depending on the other parameters. This comes with the cost outlined in Table 4.2.

The only tests where the change of the last convolution's atrous rate has a significant effect is the baseline tests of Table 4.3: the 10% results show a difference of 2.74% between atrous rate of 1 and 3. In the other cases, the last atrous rate seems to have a very minor effect on the quality of the network's predictions.

4.4 Conclusion

In this chapter, we studied some aspects of common semantic segmentation methods when applied either to less costly neural networks or to much more specific applications. We drew some important conclusions as to the potential usability of MobileNet in particular, especially with the FCN16 and FCN8 architectures. We also noticed some important improvements for person-only segmentation when using any of the basic segmentation architectures. This only grows in importance as the percentage of background pixels increases, revealing some important details about the potential issues of imprecise bounding boxes for instance segmentation.

This sets the ground for future work in the field of instance segmentation with the networks and methods studied here, as we will cover in Chapter 5.

Chapter 5

Instance Segmentation with SSD-ISB Architectures

We evaluate the instance segmentation architecture presented in Chapter 3 through in-depth experiments and comparison of its variations, presenting some interesting advantages and drawbacks of the various approaches. Through our computational cost analysis, we show that our proposed architecture achieves low model sizes, fast speeds on a GPU, and a reasonable memory footprint for the task.

To simplify the task at hand, we take a similar approach as in Section 4.3 and consider person annotations only. Note that the ideas here should generalize to a higher number of classes, but that the networks are easier to train and test extensively with a more limited number of relevant annotations in a particular dataset.

The MS-COCO dataset used in this chapter is covered in Section 5.1. The results of the various architectures and an analysis thereof can be found in Section 5.2.

5.1 Dataset

As mentioned in Section 2.6, the two main datasets for instance segmentation research are MS-COCO - commonly used for object detection - and Cityscapes - commonly used for semantic segmentation.

Due to MS-COCO more manageable image size, and to its more even distribution of instance sizes, we do most of our tests on MS-COCO instead of Cityscapes.

The metrics used to evaluate the quality of the predictions are Average Precision (AP) and Average Recall (AR), shown in Equation 5.1:

- Average Precision: percentage of instance predictions that are accurate;
- Average Recall: percentage of groundtruth instances that have been predicted.

$$\begin{aligned} AP &= \frac{TP}{TP + FP} \\ AR &= \frac{TP}{TP + FN} \end{aligned} \tag{5.1}$$

The same metric is used for both MS-COCO and Cityscapes: AP is measured at threshold IoU values varying between 0.50 and 0.95 with a step of 0.05, where an IoU between the ground truth and the prediction superior to the threshold counts as a *positive* prediction.

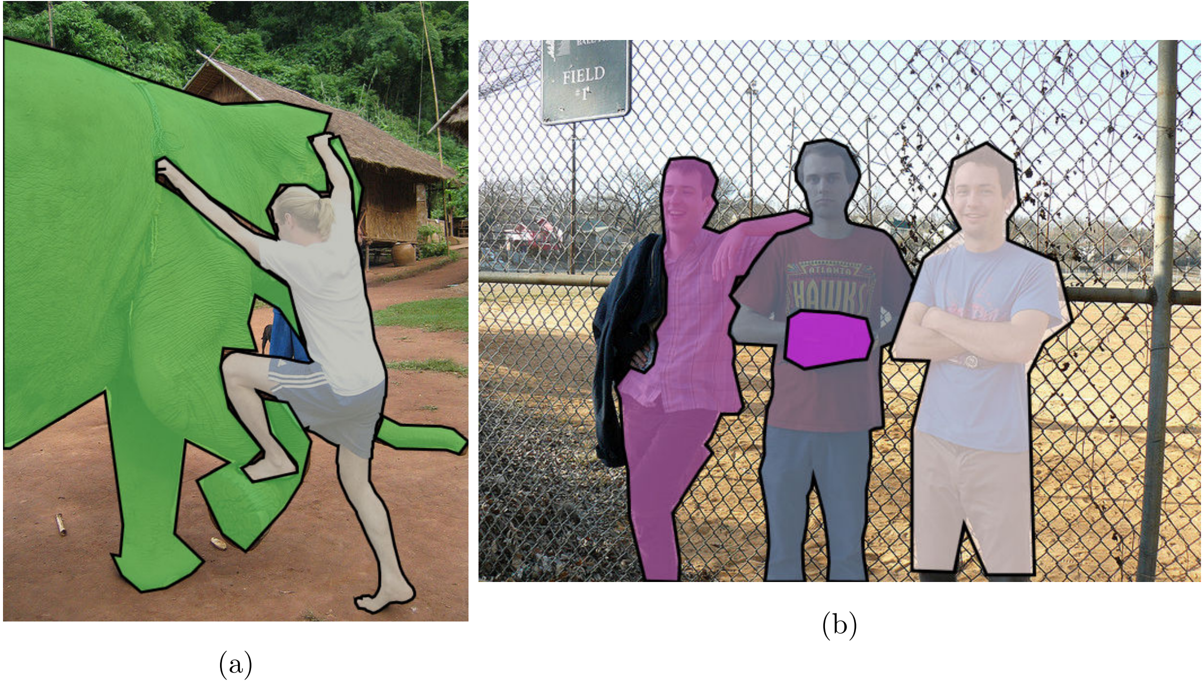
Of note is that Pascal VOC, which has annotations for semantic segmentation and instance segmentation, is not used for this chapter's tests, mostly due to its very low number of image; this problem is only accentuated when looking at a subset of instances. However, we do use Pascal VOC's metric - AP with a threshold IoU of 0.50 - to report our results, as it is more useful to compare the lower accuracy of our low-cost networks.

5.1.1 MS-COCO

MS-COCO (or *Microsoft Common Objects in Context*) [30] contains 80 easily recognizable object classes, each containing a large number of instances. The instances within this dataset are taken from various typical scenes in which the object types might be seen - for example, an instance in the *person* class might be standing, sitting or doing a variety of activities. The dataset contains a total of 328 thousand images and 2.5 million labeled instances.

Instance segmentation - sometimes referred to as *object segmentation* in the context of the COCO dataset - annotations are not available for all of these instances, but only

for approximately 123 thousand images with a 118K/5K train/val split and another 41 thousand annotations for testing, which are not publicly available. Note that this split is used since the 2017 challenge, replacing the earlier split of 83K/41K for train/val [30]. Some images from the COCO dataset can be seen in Figure 5.1.



(a) (b)

Figure 5.1: Image examples from the COCO dataset.

Secondary Metrics

The COCO reports 11 secondary metrics in its leaderboard, all of which are based on either AP or AR:

- AP only with an IoU threshold of 0.50 (the Pascal VOC metric) and only with an IoU threshold of 0.75;
- AP for small, medium, and large objects;
- AR with 1, 10, and 100 detections per image;
- AR for small, medium, and large objects.

Small, medium, and large objects are defined based on the are of their segmentation mask. Small objects have an area smaller than 32^2 pixels, large objects have an area larger than 96^2 pixels, and medium objects have an area between those for the small objects and the large objects.

5.1.2 Training and Validation subsets

Since we're only looking at person instance segmentation for this experiment, we create a subset of the COCO dataset, only keeping images with any number of person annotations. The training dataset is created from the COCO training images, and the validation dataset is created from the COCO validation images.

Additionally, we run some tests on a more limited subset of images to study the accuracy of the multiple branch approaches more carefully. In this case, we create two different datasets with the same images: one with all the person annotations, the other only with the person annotations that have a matching shape in the ISB's, after calculating the encompassing bounding box. More details can be found in Section 5.2, including the shapes used to pick the images and annotations, and the statistics of all three datasets created from COCO.

5.2 Experimentation Results

We report AP and AR metrics typically used for instance segmentation, using an IoU threshold of 0.50 like the PASCAL VOC dataset. We decide against COCO's approach of averaging the AP and AR for multiple IoU thresholds between 0.50 and 0.95 because the more difficult nature of that metric renders comparisons between methods more difficult. However, we do report AP and AR for the small, medium and large objects, as well as AR for 1, 10 and 100 detections, for an IoU threshold of 0.50.

We first compare our proposed methods on a subset of the COCO person annotations matching our chosen ISB shapes in Section 5.2.1, and then compare them on all the person annotations in the COCO dataset in Section 5.2.2. We study the speed and memory

requirements of these methods in Section 5.2.3 and run additional tests on some of these methods in Section 5.2.4.

Note that we use the shorthands *Single-Branch* for the single-branch resizing architecture, *Single-Shape* for the multi-branch architecture with no resizing, *Single-Shape+* for the multi-branch architecture with one additional resizing branch and *Multi-Shape* for the multi-branch architecture that uses resizing in every single branch.

5.2.1 COCO Subset from ISB Shapes

For studying the multi-branch approach, particularly with no resize, we first run tests on a subset of images from COCO. This subset is generated by choosing a list of possible ISB shapes, and then keeping only the images with person class annotations of that shape. To present results on the generalization of the multi-branch approaches, we create two sets of annotations: one with only the annotations of shape matching an ISB shape, and one with all the person annotations kept previously. Note that we always use encompassing bounding boxes to determine the shape of the bounding boxes. Table 5.1 shows the number of annotations for the ISB's chosen shapes.

	7x2	6x3	6x2	5x3	5x2	4x3	4x2	3x3	3x2	3x1	2x2	2x1	Other
Train	2654	4812	4793	5739	7899	6207	13632	5126	21666	9794	22249	24019	133876
Val	129	214	217	225	315	279	536	236	882	379	945	929	5718

Table 5.1: The distribution of person bounding boxes in the COCO dataset (height by width) for a 20x20 feature map, with encompassing bounding boxes, for the training and validation subsets.

Note that the *Multi-Shape* approach uses the following size thresholds: 15x10, 9x4, 7x3, 6x3, 5x2, 4x2, 3x2, 2x2, 2x1 and 1x1. Most of these were chosen as a subset of the ISB shapes, with two branches added to account for larger objects. Additionally, the *Single-Branch* approach resizes the bounding boxes to 10x10 after cropping and the branch added to create *Single-Shape+* is of size 14x14.

The resulting subset of COCO is made up of 30545 images in its training dataset and 1284 images in its evaluation dataset.

All Annotations

Table 5.2 shows the results of all four methods for this dataset with all the person annotations.

	AP(0.50)	AP(small)	AP(medium)	AP(large)	AR(1)	AR(10)	AR(100)	AR(small)	AR(medium)	AR(large)
Single-Branch	26.81	1.475	28.29	65.71	10.28	35.18	46.27	13.97	56.56	84.00
Single-Shape	11.80	1.102	29.58	3.960	7.095	20.25	25.84	3.107	49.66	11.69
Single-Shape+	26.53	1.407	27.59	67.33	10.30	34.31	44.96	12.36	57.36	78.94
Multi-Shape	**30.59**	1.870	35.22	73.48	10.74	37.89	**51.59**	16.25	63.95	90.62

Table 5.2: Precision and Recall results on the COCO-subset from ISB shapes for an IoU threshold of 0.50.

Small Objects We first notice that the average precision on small objects is very low. Liu et al.'s analysis of SSD's performance on objects of varying area revealed that the SSD architecture itself deals poorly with small objects [11]. This will affect all our instance segmentation methods based on SSD, as there is no way to create a mask for a missed object. Additionally, many small objects will have a feature map of size 1x1, and would most likely not be segmented as precisely as we would like.

Single-Shape The precision and recall on medium and large objects, on the other hand, is quite satisfactory for the low cost classifier, and high speed object detection architecture, that we are studying here. The *Single-Shape* architecture prunes too many box predictions to achieve good accuracy, particularly due to its abysmally low precision on large objects. Figure 5.2 shows this issue: while many medium size persons are annotated in the crowd,

the largest, and thus easiest, annotations are missed completely. However, this method does show the second best accuracy on medium objects.

Figure 5.2: An example of an image from the COCO dataset with *Single-Shape* predictions; black boxes indicate the groundtruth, green boxes indicate the predictions.

Single-Shape+ Resizing the remaining object predictions in an additional ISB significantly increases the average precision on large objects, from 3.96 to 67.33, bypassing that of the *Single-Branch* approach. Oddly, the AP on medium objects decreases below that of the *Single-Branch* approach, implying that there is little performance to be gained by adding parallel branches in a dataset with such a wide distribution of objects.

Multi-Shape Unlike the single-shape approaches, the *Multi-Shape* approach improves upon the *Single-Branch* approach for every metric. The AP for an IoU threshold of 0.50 increases from 26.81 to 30.59 overall, from 28.29 to 35.22 for medium objects, and from 65.71 to 73.48 for large objects.

ISB Annotations

We report the results for the annotations matching an ISB shape in Table 5.3. We only evaluate the single-branch architecture and the single-shape architecture for this test.

	AP(0.50)	AP(small)	AP(medium)	AP(large)	AR(1)	AR(10)	AR(100)	AR(small)	AR(medium)	AR(large)
Single-Branch	13.62	1.836	28.74	8.052	9.578	8.71	40.33	18.91	59.80	68.75
Single-Shape	**15.19**	0.4020	37.08	71.55	10.63	31.52	**42.64**	17.81	65.25	81.82

Table 5.3: Precision and Recall results on the COCO-subset from ISB shapes for an IoU threshold of 0.50, using only the annotations with a matching ISB shape for both training and validation.

Based on the ISB shapes used in the *Single-Shape* architecture, presented in Table 5.1, we know that the *large* objects in this set of annotations do not include most of the large annotations in the full COCO dataset: the largest annotations considered have size 6x3 in the 20x20 feature map.

While the *Single-Shape* architecture does not achieve particularly good results on this subset of annotations, it does achieve fairly high results on both medium and large annotations. In fact, we see significantly higher results than that of the *Single-Branch* architecture. This seems to indicate that the *Single-Shape* approach could be useful for very specific applications, where one might be interested in only a specific range of potential sizes, or where one might have the available memory to consider a very wide range of potential sizes. Note that the latter case would require a large dataset to properly train every single branch.

5.2.2 COCO Person

We also compare our methods on all the COCO person annotations. This subset of COCO includes 64116 training images and 2693 validation images. While we expect the multi-branch without resize to fail to generalize to the entire dataset, due to its inability to predict masks for the 5718 annotations that do not match any ISB shape, we can still study its results and compare them to the other three methods. We should also note that

the SSD prediction could generate a bounding box that matches an ISB's shape even if the groundtruth bounding box does not.

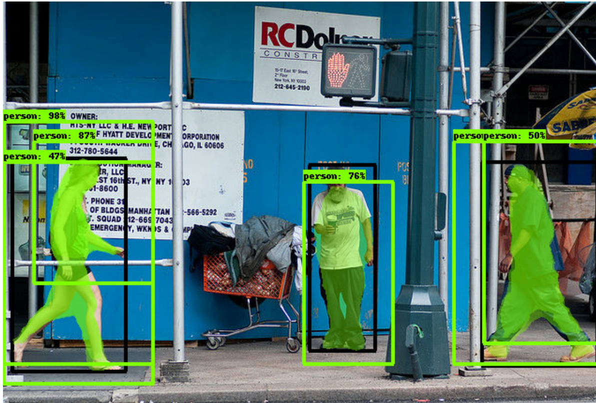
	AP(0.50)	AP(small)	AP(medium)	AP(large)	AR(1)	AR(10)	AR(100)	AR(small)	AR(medium)	AR(large)
Single-Branch	32.58	0.4717	21.90	66.26	18.17	42.80	51.06	10.70	49.87	58.84
Single-Shape	8.824	0.9345	23.43	2.961	5.846	15.18	18.72	10.82	41.94	2.832
Single-Shape+	34.86	0.6803	23.56	69.95	18.99	42.04	51.96	10.95	54.11	84.05
Multi-Shape	**38.60**	0.8881	30.46	74.18	19.36	46.87	**57.19**	12.26	61.18	90.75
Mask R-CNN	73.61	36.41	76.97	93.33	23.12	72.86	85.74	68.56	89.6	96.31

Table 5.4: Precision and Recall results on the COCO person annotations for an IoU threshold of 0.50.

The experiments on the COCO person dataset lead to similar conclusions as the COCO subset experiments. The AP on small objects is very low for all architectures, the *Single-Shape* approach fails to generalize to a wider distribution of annotations, and the *Multi-Shape* approach shows the overall precision. All methods except for the *Single-Shape* method show a higher precision when compared to the COCO subset experiments, most likely due to a higher number of large annotations which all methods seem to segment precisely.

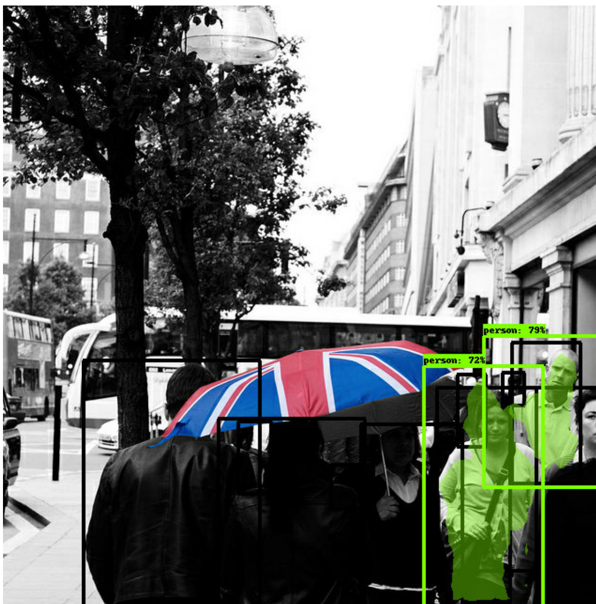
Both *Single-Shape* approaches achieve better precision on small and medium objects than the *Single-Branch* approach. This is what we expect from both these experiments and the previous COCO subset, as the chosen ISB shapes cover most of the possible sizes for medium objects. The *Multi-Shape* approach still achieves significantly higher precision than the other approaches; this is also true for the large objects and for the recall metrics.

Figure 5.3 shows some typical predictions for the *Multi-Shape* architecture. Figures 5.3a and 5.3b show good predictions on medium to large objects, while figures 5.3c and 5.3d show failure cases: the former from false negatives in box prediction and the latter from false positives in box prediction.

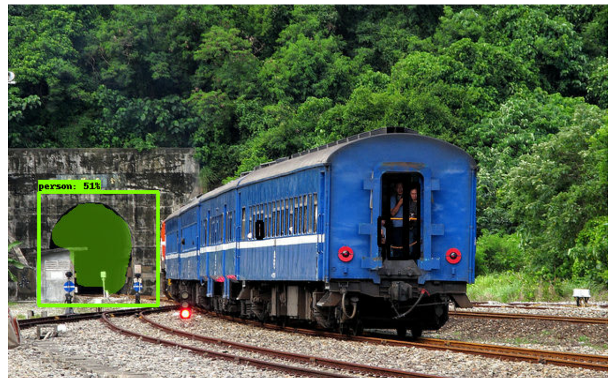


(a) Successful mask predictions.

(b) Mostly successful mask predictions.



(c) False negatives.



(d) False positives.

Figure 5.3: Sample predictions of the *Multi-Shape* architecture on the COCO person dataset; black boxes indicate the groundtruth, green boxes indicate the predictions.

5.2.3 Speed and Memory Requirements

We study the performance of the architectures as in Section 4.2.3. We report the frames per second on CPU and on GPU from averaging the time taken 200 inference tests, the memory usage based on the TensorFlow [51] *Profiler*'s *output bytes* metric, and the number of parameters in the network. All tests reported in Table 5.5 were undertaken on the same system with a GeForce RTX 2080 Ti GPU and an Intel Core i9-9900K CPU.

The *Single-Shape* is the fastest and smallest of the architectures, no doubt due to the

Architecture	FPS (GPU)	FPS (CPU)	Memory Usage (MB)	Number of Parameters
Single-Branch	51.95	8.566	852.76	**5790631**
Single-Shape	**56.11**	**12.66**	**405.89**	8862738
Single-Shape+	43.68	6.259	2179.24	9002580
Multi-Shape	43.68	7.983	1909.90	8014288
Mask R-CNN	3.87	0.38	11927.95	47673551

Table 5.5: Speed and memory usage for the various SSD-IS architectures.

complete absence of resizing operations in the network. It shows a significant memory usage advantage, and a noticeable inference speed advantage, over the *Single-Branch* architecture, the second fastest architecture in our experiments. The *Single-Branch* architecture is however the smallest in parameters count, because it only has one set of branch parameters, as opposed to the many sets of parameters in all the other architectures. This results in the *Single-Shape*'s model size being larger than the *Multi-Shape*'s as well, since the latter has fewer parallel ISB's.

The *Single-Shape+* and the *Multi-Shape* architectures achieve the same number of frames per second on GPU, the lowest speed in these baseline experiments. However, they both stay above comfortably above the 30 FPS threshold required to be considered *real-time*. Their memory usage is also worse than the other two methods, at approximately 2179 MB and 1909MB respectively, more than doubling the *Single-Branch*'s memory usage.

Importantly, the main factor impacting the memory usage of each architecture seems to be the size of the ISB's rather than the number of the ISB's. The 14x14 resizing branch added to the *Single-Shape* architecture to create the *Single-Shape+* architecture significantly increases its precision on large objects, but also increases its memory usage past all the other methods. Considering the relative ease of predicting masks for large objects, it seems that the emphasis should be put on using smaller shapes for the ISB's, which come with a significantly smaller memory cost, even if this requires downsampling the feature maps corresponding to the larger objects.

Mask R-CNN Comparison

We include a comparison to Mask R-CNN with ResNet-101 backbone trained on the COCO person dataset in tables 5.4 and 5.5.

We notice that Mask R-CNN, which uses a very large network as its backbone and the slow Faster R-CNN for box prediction, achieves much more precise results than our proposed architectures. In particular, we notice that there is a major difference in the precision and recall of small objects: 36.41 and 68.56 for Mask R-CNN to 0.8881 and 12.26 for our most accurate approach. This comparison confirms the idea that our methods' predictions on larger objects are acceptably precise, while the precision on smaller and smaller objects falls behind the state-of-the-art.

However, we also need to look at Mask R-CNN's speed, memory usage, and model size. Our proposed architectures dwarf Mask R-CNN in all these aspects. The Single-Branch approach achieves speeds more than 13 times higher on GPU, memory usage almost 14 times smaller, and a model size approximately 8 times smaller. We notice a fairly similar difference when comparing Mask R-CNN to the Multi-Shape approach, but with a smaller memory usage factor of about 6.

As expected, we notice that Mask R-CNN generates more precise mask predictions, but does not come close to achieving a real-time speed of 30fps, let alone being implemented on an embedded device for low-power applications.

A very interesting comparison can be made between the Single-Branch approach and Mask R-CNN, as they use a similar single-branch segmentation architecture. This shows that our Multi-Shape approach could be combined with a more robust object detection architecture and with a deeper feature extractor to improve upon Mask R-CNN's results, albeit with a noticeable memory cost.

5.2.4 Additional Tests

We run additional tests on the *Single-Branch* and the *Multi-Shape* architectures. We choose the former due to its competitive results relative to its complexity, and we choose the latter due to its promising results.

Single Branch Tests

We run two additional tests to measure the drop in accuracy when reducing the computational cost of the network, reported in Table 5.6. There are:

- SB-MN0.75: using a depth multiplier of 0.75 instead of 1.0 in the MobileNet feature extractor;
- SB-FCN16: using the *FCN-16* ISB architecture instead of the *FCN-8* ISB architecture;

For easy reference, we report the previous *Single-Branch* results as *SB-Default*.

	AP(0.50)	AP(small)	AP(medium)	AP(large)	AR(1)	AR(10)	AR(100)	AR(small)	AR(medium)	AR(large)
SB-Default	**32.58**	0.4717	21.90	66.26	18.17	42.80	**51.06**	10.70	49.87	58.84
SB-MN0.75	30.77	0.3217	17.67	65.92	17.76	40.43	48.56	8.484	45.75	84.65
SB-FCN16	29.80	0.1578	16.41	64.54	16.92	39.50	47.41	5.178	45.86	84.07

Table 5.6: Precision and Recall results on the COCO person annotations for an IoU threshold of 0.50; comparison between various Single-Branch parameters.

We report the speed, memory usage, and number of parameters of the *Single-Branch* variations in Table 5.7.

While our *Single-Branch* baseline already achieved a fast inference speed, we notice here that we get a noticeable speed improvement by reducing either the depth of the network or the size of the upsampling architecture. Both of these changes result in a drop of accuracy: approximately 1.8 AP and 2.78 AP respectively. Most importantly, using the *FCN16* architecture offers a significant improvement in terms of memory requirements: from 852.76MB for the *FCN8* architecture to 365.63MB, for an input image of size 640x640. On the other hand, reducing the depth of the feature maps in the encoder reduces the

Architecture	FPS (GPU)	FPS (CPU)	Memory Usage (MB)	Number of Parameters
SB-Default	51.95	8.566	852.76	5790631
SB-MN0.75	62.80	11.57	523.76	**3358647**
SB-FCN16	**65.91**	**11.93**	**365.63**	5765991

Table 5.7: Speed and memory usage for the Single-Branch and Multi-Shape SSD-IS architectures' comparative tests.

number of parameters by approximately 42%, while using the *FCN16* decoder has no noticeable impact of the size of the model.

While the default parameters for this architecture achieve good speeds and memory requirements for the task, these additional tests show some possible ways of further reducing the network's requirements. The best choice among these would depend on the application at hand.

Multi-Shape Tests

We run four additional tests on the *Multi-Shape* architecture, with results reported in Table 5.8:

- MS-MN0.75: using a depth multiplier of 0.75 instead of 1.0 in the MobileNet feature extractor;
- MS-FCN16: using the *FCN-16* ISB architecture instead of the *FCN-8* ISB architecture;
- MS-NewShapes: using shapes 15x15, 10x10, 8x8, 6x6, 4x4, 2x2 and 1x1 for a total of seven parallel ISB's;
- MS-SmallShapes: using shapes 10x10, 8x6, 6x4, 5x3, 4x2, 2x2, 2x1, 1x1 for a total of eight parallel ISB's.

	AP(0.50)	AP(small)	AP(medium)	AP(large)	AR(1)	AR(10)	AR(100)	AR(small)	AR(medium)	AR(large)
MS-Default	**38.60**	0.8881	30.46	74.18	19.36	46.87	**57.19**	12.26	61.18	90.75
MS-MN0.75	35.97	**2.141**	26.81	70.29	18.53	45.34	55.91	14.72	58.44	87.76
MS-FCN16	32.71	0.184	16.9	73.12	17.92	39.70	46.94	2.183	43.34	87.71
MS-NewShapes	36.73	1.012	27.59	71.36	19.11	45.33	54.89	13.07	57.87	86.83
MS-SmallShapes	37.88	1.120	29.28	72.99	19.43	46.61	56.61	12.45	61.98	90.05

Table 5.8: Precision and Recall results on the COCO person annotations for an IoU threshold of 0.50; comparison between various Multi-Shape parameters.

Here as well, we report the previous *Multi-Shape* results as *MS-Default*.

We report the speed, memory, and model requirements of the *Multi-Shape* variations in Table 5.9.

Architecture	FPS (GPU)	FPS (CPU)	Memory Usage (MB)	Number of Parameters
MS-Default	43.68	7.983	1909.90	8014288
MS-MN0.75	**50.27**	10.14	652.33	**5211360**
MS-FCN16	49.90	**10.85**	**571.23**	7767888
MS-NewShapes	39.30	5.940	2314.30	7273069
MS-SmallShapes	48.28	9.203	825.22	7520142

Table 5.9: Speed and memory usage for the Single-Branch and Multi-Shape SSD-IS architectures' comparative tests.

The 0.75 depth multipler and *FCN-16* variations on the *Multi-Shape* architecture show similar results to the same variations on the *Single-Shape*: an major improvement in inference speed, and either model size or memory requirements at the cost of precision and

recall. The decrease in AP is more noticeable in this case, reaching approximately 2.6 and 5.9 respectively. Thus, we see similar possibilities of modifying this architecture to better fit a certain applications, which might have particular memory or power consumption limitations. It is worth considering that the *MS-MN0.75* variation achieves better accuracy than the *Single-Branch* and *Single-Shape+* variations while achieving lower memory requirements and model size, and inference speeds comparable to the default *Single-Branch* approach.

MS-NewShapes The *MS-NewShapes* and *MS-SmallShapes* are tests that can only be tested with the multi-branch approaches. The shapes for the former were chosen to give a better coverage of the possible branch shapes, thereby requiring less resizing for each of them. This actually resulted in lower accuracy and recall than the original shape choices, in addition to decreasing the speed of the network, and increasing its memory usage. These impact on performance result from increased size of the feature maps after cropping and resizing, particularly the 15x15 ISB. The impact on accuracy is more surprising, but seems to indicate that there is more to be gained from covering common smaller shapes than having more general coverage of all the shapes. Larger shapes in particular seem to be predicted fairly well with a 15x10 ISB.

MS-SmallShapes This analysis is confirmed by the results of the *MS-SmallShapes* test. The branch shapes in this case give a good coverage of the smaller objects in the dataset, and predict all objects larger than 10x10 in the 10x10 ISB. It achieves an AP of 72.99 on large objects, compared to the 74.18 AP of the original ISB shapes. This method only shows a slight decrease when compared to those original shapes of approximately 0.72 AP for a threshold of 0.50. It also improves upon the speed, memory usage, and parameter count of the default *Multi-Shape* approach, decreasing its memory requirement by more than half.

Once again, we believe that these tests show that a lot of small modifications could be made to these architecture to adapt them for various low-cost applications. Additionally, these results show an interesting relationship between the resizing that we apply to smaller objects and the quality of the mask prediction. This is of particular interest when considering that many instance segmentation branches dedicated to mask prediction of small and

medium objects show a memory usage comparable to that of a single large segmentation branch.

5.3 Conclusion

In this chapter, we studied potential modifications to the *Single-Shot Detector (SSD)* for instance segmentation. We proposed multiple architectures, all building on the core concept of using SSD's box predictions to crop features belonging to an instance from the last feature map in SSD's feature extractor. To properly align the box predictions with this feature map, we calculate the smallest box that completely encompasses the predicted box, while still being aligned with the final extractor's feature map.

We studied four main instance segmentation architectures: single-branch with resizing, multi-branch with no resizing, multi-branch with an additional branch which uses resizing, and multi-branch with resizing on every single branch. While the last of these architectures achieved the best results on the COCO dataset, we also presented some important aspects of each architecture, specifically regarding which parts hurt the speed, memory usage, model size, and accuracy of each approach.

In particular, the Multi-Shape approach achieves the best overall results and shows a significant improvement over its Single-Branch counterpart with more dramatic resizing operations, while achieving a high speed and a range of possible computational costs. We believe this architecture in particular shows promise as a fast instance segmentation solution.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we studied some aspects of existing semantic segmentation methods with an emphasis on speed, memory cost, and the potential application for instance segmentation. We then devised multiple ways of adapting the SSD object detection architecture for instance segmentation. These approaches create an idea of the possible compromises between accuracy and computational cost, achieving some impressive segmentation results for such a small network as MobileNet. Most importantly, the multi-branch and multi-shape architecture shows a significant improvement over its straightforward single-branch counterpart, achieving an average precision of 38.60 at an IoU threshold of 0.50 with a speed of approximately 44 frames-per-second.

The model sizes of the various instance segmentation methods increase with the number of instance segmentation branches, while the memory footprint and the speed of the architecture varies with the way that we define the inputs to the branches. This creates a very interesting way to optimize the precision and the efficiency of the network for a certain task, or a certain dataset.

All of the methods studied in this thesis, for both semantic segmentation and instance segmentation, achieve very fast speeds for this task. In the case of instance segmentation, we use the MobileNet feature extractor and the Single-Shot Detector object detection architecture to ensure fast speeds and low model size. This incurs interesting issues related

to SSD's poorer and poorer performance as the size of objects decrease. However, our method is easily capable of generating very precise segmentation mask predictions for larger objects.

To our knowledge, neither the idea of using parallel branches for instance segmentation, nor the idea of aligning boxes by enlarging them have been proposed before, but they show very promising results throughout this work. This outcome leaves room for more research on these methods.

6.2 Future Work

While our work studied some very interesting facets of instance segmentation architectures, it opens some more questions about the potential of those well-known methods. These could be the basis for future work on the way that we adapt object detection networks for instance segmentation.

We focused on adapting SSD for instance segmentation, but did not consider how our new ideas might affect the accuracy of a more costly architecture such as Faster R-CNN. In and of itself, there is no reason that we could not use parallel segmentation branches in Faster R-CNN to take advantage of its lower miss rate; our method has no way of recovering an object missed by SSD's box classification branch. As a matter of fact, the cost of a high number of ISB's, even shallow ones, would better mesh with the typically slow speed of a two-stage solution and the deeper feature extractors often used with them.

Additionally, we only considered the COCO and Cityscapes datasets within this thesis, for the instance segmentation experiments and the semantic segmentation experiments respectively. The former is a very generic dataset, and the latter does not apply itself very well to the SSD architecture due to its very large input size. This leaves an unanswered question as to the usefulness of the parallel branches method for a very specific application. An application with a sufficiently small deviation in the size of the instances would no doubt greatly benefit from an approach that completely discards resizing.

While we tested some variations of the architectures proposed in this work, there remain many potential areas for further experiments. Changing the depth of feature maps in the ISB's, the number of layers used in each ISB's, and the use of many state-of-the-art

methods in semantic segmentation, particularly in encoder-decoder architectures, could all significantly improve the quality of the network's predictions. In addition, the parallel branch approach opens the interesting possibility of customizing the architecture of each branch depending on its size: using atrous convolution for large objects, using more up-sampling steps for very small ones, etc. There is also the possibility of testing the same network on a larger number of classes, for example by using the entire COCO class list, though this might be more realistic if building on a different object detection method.

The models tested in this work all have one major drawback: they will predict the same masks for objects which result in the same encompassing bounding box. This opens the possibility of perhaps predicting multiple masks for each branch, though there would be a need to train the network, and choose the masks, appropriately.

Finally, while the number of frames processed per second for an input size of 640x640 stays quite high on a GeForce 2080 Ti, there remains some space for optimization of the network, and for in-depth testing of our architectures on embedded devices.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *Arxiv.Org*, 7(3):171–180, 2015.

[3] 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. pages 1–13, 2016.

[4] 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.

[5] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[6] Andrej Karpathy. CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/.

[7] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. pages 1–14, 2014.

[8] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. 2016.

[9] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. 2017.

[10] Ross Girshick. Fast R-CNN. *Proceedings of the IEEE International Conference on Computer Vision*, 2015 Inter:1440–1448, 2015.

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9905 LNCS:21–37, 2016.

[12] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, 2017.

[13] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[14] Panqu Wang, Pengfei Chen, Ye Yuan, Ding Liu, Zehua Huang, Xiaodi Hou, and Garrison Cottrell. Understanding Convolution for Semantic Segmentation. 2017.

[15] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. pages 1–12, 2016.

[16] 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. pages 1–14, 2016.

[17] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. pages 1–28, 2016.

[18] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. pages 1–8, 2015.

[19] Judy Hoffman, Dequan Wang, Fisher Yu, and Trevor Darrell. FCNs in the Wild: Pixel-level Adversarial and Constraint-based Adaptation. 2016.

[20] Farzan Erlik, Nowruzi Prince, Kapoor Dhanvin, Kolhatkar Fahed, Al Hassanat, and Robert Laganiere. How much real data do we actually need : Analyzing object detection performance using synthetic and real data. 2019.

[21] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9908 LNCS:630–645, 2016.

[23] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 1994.

[24] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of the 13th International Conference On Artificial Intelligence and Statistics*, 2010.

[25] Song Han, Huizi Mao, and William (Bill) J. Dally. Deep Compression. *ICLR*, 2016.

[26] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014.

[27] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2015.

[28] Iacopo Masi, Yue Wu, Tal Hassner, and Prem Natarajan. Deep Face Recognition: A Survey. In *Proceedings - 31st Conference on Graphics, Patterns and Images, SIBGRAPI 2018*, 2019.

[29] Mark Everingham, Luc Van Gool, Christopher K.I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 2010.

[30] Tsung-yi Lin, C Lawrence Zitnick, and Piotr Doll. Microsoft COCO : Common Objects in Context. pages 1–15.

[31] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[32] J. R.R. Uijlings, K. E.A. Van De Sande, T. Gevers, and A. W.M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 2013.

[33] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking Atrous Convolution for Semantic Image Segmentation. 2017.

[34] Liang Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11211 LNCS:833–851, 2018.

[35] Mennatullah Siam, Mostafa Gamal, Moemen Abdel-Razek, Senthil Yogamani, and Martin Jagersand. Rtseg: Real-time semantic segmentation comparative study, 2018.

[36] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[37] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation, 2018.

[38] Farzan Erlik Nowruzi, Dhanvin Kolhatkar, Prince Kapoor, Fahed Al Hassanat, Elnaz Jahani Heravi, Robert Laganiere, Julien Rebut, and Waqas Malik. Deep open space segmentation using automotive radar, 2020.

[39] Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, and Stefan Roth. The Cityscapes Dataset for Semantic Urban Scene Understanding.

[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2015.

[41] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Simultaneous detection and segmentation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014.

[42] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation, 2018.

[43] Xinlei Chen, Ross Girshick, Kaiming He, and Piotr Dollár. Tensormask: A foundation for dense object segmentation, 2019.

[44] Cheng-Yang Fu, Mykhailo Shvets, and Alexander C. Berg. Retinamask: Learning to predict masks improves state-of-the-art single-shot detection for free, 2019.

[45] Weicheng Kuo, Anelia Angelova, Jitendra Malik, and Tsung-Yi Lin. Shapemask: Learning to segment novel objects by refining shape priors, 2019.

[46] Weixing Zhang, Chandi Witharana, Anna K. Liljedahl, and Mikhail Kanevskiy. Deep convolutional neural networks for automated characterization of arctic ice-wedge polygons in very high spatial resolution aerial imagery. *Remote Sensing*, 10(9), 2018.

[47] Sharada Prasanna Mohanty. Crowdai mapping challenge 2018 : Baseline with mask rcnn. https://github.com/crowdai/crowdai-mapping-challenge-mask-rcnn, 2018.

[48] Hsieh-Fu Tsai, Joanna Gajda, Tyler F.W. Sloan, Andrei Rares, and Amy Q. Shen. Usiigaci: Instance-aware cell tracking in stain-free phase contrast microscopy enabled by machine learning. *SoftwareX*, 9:230 – 237, 2019.

[49] Colin J Burke, Patrick D Aleo, Yu-Ching Chen, Xin Liu, John R Peterson, Glenn H Sembroski, and Joshua Yao-Yu Lin. Deblending and classifying astronomical sources

with Mask R-CNN deep learning. *Monthly Notices of the Royal Astronomical Society*, 490(3):3952–3965, 10 2019.

[50] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. 2016.

[51] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.