NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS
DEPARTMENT OF PHYSICS
SECTION OF ELECTRONIC PHYSICS AND SYSTEMS
MASTER'S DEGREE ON CONTROL AND COMPUTING
ESTABLISHED IN 1837

Master Thesis

A comparative analysis of fully convolutional neural networks for cloud image segmentation

Tziolos Philippos

2019511

ATHENS

November 2021

Supervisors

Dionysios Reisis, Professor

Evaluation Committee

Dionysios Reisis, Professor

Dr. Nikolaos Vlassopoulos, Research Associate

Dr. Konstantinos Nakos, Research Associate

Abstract

This thesis investigates at a multilateral level the performance of different fully convolutional neural networks on the task of cloud semantic segmentation for ground-based sky images. Specifically, the networks are evaluated on the Singapore Whole Sky Image Segmentation dataset via the metrics: F1 score, Intersection over Union, Precision, Recall, Specificity and Accuracy. Initially, five novel variations of the Unet architecture are proposed and benchmarked on five disparate training/validation/test set ratios to determine both the networks' competence and the finest ratio. Subsequently, further research is conducted to define the optimal optimization algorithm and loss function for relatively small networks like Unets. Finally, the technique of transfer learning is examined on cloud segmentation through networks pretrained on the ImageNet dataset.

Keywords: Deep Learning, Cloud Segmentation, SWIMSEG Dataset, Transfer Learning, Fully Convolutional Neural Networks

Περίληψη

Η παρούσα διπλωματική εργασία έχει ως στόχο τη μελέτη και σύγκριση διαφόρων πλήρως συνελικτικών νευρωνικών δικτύων που προορίζονται για αναγνώριση νεφών από εικόνες ουρανού σε επίπεδο εικονοκυττάρου. Συγκεκριμένα, τα δίκτυα αυτά αξιολογούνται σε εικόνες νεφών από τη βάση δεδομένων SWIMSEG της Σιγκαπούρης μέσω των μετρικών: F1 score, Intersection over Union, Precision, Recall, Specificity και Accuracy. Αρχικά, παρουσιάζονται πέντε νέες παραλλαγές της αρχιτεκτονικής Unet, οι οποίες συγκρίνονται σε πέντε σύνολα εικόνων προπόνησης/επιβεβαίωσης/τεστ διαφορετικής αναλογίας, με σκοπό την εξέταση της επίδοσης τους και τον καθορισμό της βέλτιστης αναλογίας εικόνων. Ακολούθως, η έρευνα επεκτείνεται στην εύρεση του καταλληλότερου αλγορίθμου βελτιστοποίησης και της ευνοϊκότερης συνάρτησης κόστους. Τέλος, διερευνάται η τεχνική της μεταφοράς γνώσης για αναγνώριση νεφών από δίκτυα ήδη προπονημένα στο σύνολο δεδομένων ImageNet.

Λέξεις Κλειδιά: Βαθιά Μάθηση, Αναγνώριση Νεφών, SWIMSEG Δεδομένα, Μεταφορά Γνώσης, Πλήρως Συνελικτικά Νευρωνικά Δίκτυα

Table of Contents

Index of Tables

Table of Figures

Chapter 1

1 Introduction

Chapter 1 outlines the motivation and background for the study of cloud segmentation as part of cloud analysis on photovoltaic systems. In addition, the pros and cons of ground-based and satellite images in cloud analysis are presented. Moreover, an overview of the different types of cloud segmentation is delivered. Furthermore, a brief analysis of the main categories of cloud segmentation methodologies is provided. Finally, thesis objectives and the written outline are stated.

1.1 Motivation and Background

Concerned by the thought of climate change[1][2], nowadays more and more countries are shifting away their dependence on conventional power plants[3][4][5][6], in order to reduce the environmental impact of fossil fuel. However, green energy from renewable sources is not currently enough to meet the global demand, albeit rivaling coal generated electricity and even outstripping it sometimes[7][8]. Particularly, about 26 percent of the global electricity produced comes from renewables[9].

Although solar photovoltaic systems generate only 2.4 percent of the global electricity, they account for more than 50 percent of the renewable power capacity installed worldwide. Consequently, knowledge of the cloud coverage over areas with photovoltaic power systems plays a crucial role both in their proper operation and their maintenance. In more detail, by estimating the irradiance that reaches the surface of the solar panels it is possible to predict the power generated by the system ahead of time. Thus, grid operators can utilize this information to balance the energy load, monitor the system's efficiency and make decisions based on the energy market. This is quite

1https://www.epa.gov/climate-indicators
2https://ec.europa.eu/clima/climate-change/climate-change-consequences_en
3https://ec.europa.eu/clima/eu-action/european-green-deal/2030-climate-target-plan_en
4https://www.gov.uk/government/news/uk-enshrines-new-target-in-law-to-slash-emissions-by-78 -by-2035
5https://www.nature.com/articles/d41586-021-03044-x
6https://www.nature.com/articles/d41586-020-02927-9
7https://www.eia.gov/todayinenergy/detail.php?id=39992
8https://europeansting.com/2019/08/28/5-charts-that-show-renewable-energys-latest-milestone/
9https://www.ren21.net/gsr-2019/chapters/chapter_01/chapter_01/

important, because overpowering or underpowering the grid system can be catastrophic for the devices connected to it.

Miscalculation of the predicted irradiance can occur on cloudy days, particularly on systems with high temporal resolution [1], due to cloud shadows and sunlight reflected by clouds. As a result, large ramp rates and high peaks increase power fluctuations which require ancillary services, like battery systems and fuel based power generators. To face these challenges short temporal resolutions and cloud analysis methods have been introduced aiming to minimize the error assesment.

1.2 Pros and Cons of Ground-Based & Satellite Images

Cloud analysis studies have been done either via geo-stationary satellite images or ground-based ones. While satellite images suffer from low spatial and temporal resolution and added complexity due to the background scenery, ground-based images can provide higher spatial and temporal resolution with lower complexity. On the other hand ground based images are limited to local areas whereas those from satellites provide a global coverage. Even though both types of images provide enough information for cloud-free and cloud-covered skies to algorithmic cloud analysis methods, in case of broken clouds ground-based images yield better results than the ones from geo-stationary satellites [2].

These ground-based images are captured at regular intervals by special cameras with a fish-eye lens, defined as Whole Sky Imagers, providing a wide field of view. One drawback of those cameras is the fact that the images get geometrically distorted. As a result, all depicted objects appear deformed. To alleviate this problem rectification methods that depend on lens design specification, calibration patterns or machine learning algorithms must be applied, though they induce noise. To make matters worse, the combination of the sky's dynamic luminance range with the fact that clouds cannot be defined by their structure or contour, neither their shape nor size, renders their detection a quite challenging task.

1.3 Overview of the Different Types of Cloud Segmentation

Typically, cloud detection methodologies refer to cloud image classification. The classification process can be applied either on the whole image or on each pixel of the image separately. The first method aims to identify and label the picture, based on its characteristics and according to specific criteria, with a single category from a well defined set. The second method is usually called image segmentation and intends on assigning each pixel to a specific class from an established set, thus generating a pixel-wise map of classifications.

There are three types of image segmentation, based on the way depicted objects are grouped. The first and most popular technique is defined as semantic segmentation, because objects of the same class are grouped together as one entity. Conversely, the second one is called instance segmentation for every distinguishable object of interest is treated as a discrete instance of the general class. The third method is specified as panoptic segmentation and combines both semantic and instance segmentation processes.

1.4 Cloud Segmentation Methodologies

In the literature various methods of cloud semantic segmentation have been studied, addressing different classification problems [3]. These include both binary prediction of cloud/no cloud images and categorical detection of thin cloud/thick cloud/no cloud [4], cloud/cloud shadow/neither cloud nor shadow [5] and cloud/snow/neither cloud nor snow [6].

As far as the techniques of cloud segmentation are concerned, the most prevalent ones can fall into three main categories: threshold, clustering and deep learning. Threshold based methods utilize fixed or adaptive thresholds on handcrafted formulas to generate ratios which intensify the differences between the classes of interest. Consequently, their performance depends on the dexterity of the data analysts to fabricate formulas that can differentiate efficiently cloud pixels from non cloud ones. Clustering techniques also use formulas to highlight the discrepancies of the classes, but the segmentation of the image is conducted by clustering algorithms. Deep learning approaches employ self-supervised sophisticated architectures of fully convolutional neural networks. The term self-supervised refers to the learning method that these networks are subjected to. Specifically, they are trained on colored images and get evaluated on their respective ground truth maps. Although, neural networks don't require handcrafted formulas to operate, their performance depends on the quality and quantity of the training images.

1.5 Thesis Objective

The present thesis is concerned with comparing different architectures of fully convolutional neural networks on ground-based sky images for cloud semantic segmentation of cloud/no cloud ones. The objectives of this thesis are multiple. First, the optimal ratio for the training, validation and test set on a small benchmark dataset called SWIMSEG will be explored. Second, the finest optimization algorithm as well as the most appropriate loss function, out of a collection available on the Keras API, the tensorflow_addons library and custom implemented, will be determined. Third, investigation of tranfer learning for cloud semantic segmentation utilizing pretrained networks on the ImageNet dataset will be carried on. Last, comparison between all implemented architectures based on their performance on the SWIMSEG dataset will be performed.

1.6 Thesis Outline

The rest of this thesis is structured into four parts, with each part constituting a discrete chapter:

Chapter 2 provides a literature review, where relevant approaches will be presented and elaborated. All of the presented studies have been conducted with ground-based images utilizing different methodologies.

Chapter 3 elaborates on the fully convolutional neural networks. It describes in great detail the architectures of the networks that have been implemented. Furthermore, it delves into the collection of the optimization algorithms and the set of loss functions destined for investigation. Moreover, it introduces the commonly used, in image segmentation, evaluation metrics for the comparison of the results.

Chapter 4 presents the conducted experiments as well as the training, validation, and test results on the SWIMSEG dataset of the implemented networks.

Chapter 5 concludes the study discussing the contributions of this work as well as suggesting future research.

Chapter 2

2 Approaches to cloud segmentation

2.1 Traditional Threasholding Algorithms

Over the years several techniques have been developed for semantic segmentation of ground-based sky images to cloud/no cloud ones. The first notable attempts were carried out utilizing sophisticated thresholding algorithms.

The hybrid thresholding algorithm proposed by Q. Li et al [7] is one of them and employs a combination of fixed and adaptive thresholding methods. Specifically, by transforming the color images into normalized red/blue channel ratio ones, they can be distinguished more easily by their standard deviations as unimodal or bimodal, containing practically either sky or clouds or exhibiting both clouds and sky respectively. Unimodal images are segmented with a fixed value because cloud and sky ratios are totally different, whereas minimum cross entropy is applied between the original and segmented bimodal ones so as to search for the best threshold value.

A similar algorithm based on traditional threshold analysis, called hybrid entropy threshold method, has been developed recently by R. Shen et al [8]. Images are again transformed into normalized red/blue channel ratio ones and are segmented via a combination of a fixed threshold method, a maximum information entropy threshold and a minimum cross entropy threshold. Based on the variance of the normalized red/blue ratio image, it is classified either as sunny/overcast or cloudy. A fixed value is applied on the sunny/overcast images for segmentation, while cloudy ones are segmented by the quarter point closer to the minimum cross entropy threshold, in the interval formed by the values of the maximum information entropy threshold and the minimum cross entropy threshold.

Another threshold algorithm designed recently by X. Li et al [9], which performs better than the hybrid thresholding algorithm, aims at reducing the sunlight interference in the image by introducing an adjustable red green difference. The image is divided into four circumsolar regions and based on the absence or presence of solar interference different cloud detection criteria are applied to the regions. Solar interference is determined by two factors, the solar intensity calculated as the average intensity of the pixel block taken from the center of the solar disk and the saturation difference calculated by subtracting the average saturation of the first layer from that of the entire image. The threshold for the four regions is calculated by multiplying the red channel values with a weight k and subtracting from it the green channel values. The k weight is set to the same constant value for all regions in the presense of sunlight interference, while different fixed values are assigned to them in its absence.

2.2 Clustering Algorithms

Cloud segmentation techniques were simplified, as far as the repeated testing for fine tuning the hyperparameters is concerned, with the incorporation of machine learning algorithms. Great examples are the works of S. Dev et al [10] [11], which don't rely on manually determined thresholds. In their first work they use manually defined ground truth maps and apply the partial least squares regression method to provide a probabilistic indication of each pixel's identity. The important color components for the images are determined via a Principal Component Analysis and the Receiver Operating Characteristics curve by checking the degree of correlation on 16 color channels. In order to generate binary maps from the probabilistic ones, a fixed threshold is applied to them. In their second work they measure Pearson's Bimodality Index to determine quantitatively the color channels which exhibit the most bimodal distributions, based on the assumption that they yield better results in binary segmentation. Furthermore, they extend their research by conducting a Principal Component Analysis to determine the most significant individual and pairs of color channels for cloud detection. Finally, by applying fuzzy clustering on the best individual and pair of color channels, they achieve similar performance with the hybrid thresholding algorithm.

Another remarkable work on the field of machine learning is that of G. Terren-Serrano et al [12] which compares several techniques, in infrared ground-based images, based on the J-Statistic metric. These techniques involve the supervised methods of Gaussian Discriminant Analysis and Naive Bayes Classifier, as well as the unsupervised ones of Gaussian Mixture Model, k-means and Markov Random Fields. Additionally, they included some discriminative algorithms such as Ridge Regression, Primal solution for Support Vector Machines and Primal solution for Gaussian Processes. The results of the study show that the Markov Random Fields is the best performing technique among both the unsupervised and the supervised implemented techniques.

2.3 Deep Learning Algorithms

Deep learning algorithms has brought forth a revolution in machine vision, and particularly in the image classification field, by automating the process of data analysis. Furthermore, their superior performance against conventional methods, combined with their ability to adapt efficiently on different classification problems, has rendered them the most preferable choice for both scientific and industrial applications. In image segmentation deep learning has prevailed through fully convolutional networks which have demonstrated remarkable success, outperforming alternative network architectures.

A recent study on cloud semantic segmentation provided by M. Hasenbalg et al [13] has shown that fully convolutional networks deliver the best overall accuracy against conventional cloud segmentation techniques, including the hybrid thresholding algorithm and an improved version of it, defined as hybrid thresholding algorithm plus, a Clear Sky Library based approach, a region growing algorithm and a color-channel fixed threshold based algorithm. The network is based on the work of J. Long et al [14] and contains two parts, an encoder and a decoder. The encoder integrates the VGG16 [15] architecture without the final layers of the classifier, containing only convolutions and max pooling operations. Furthermore, the decoder is comprised by three upsampling operations whose outputs are added together with outputs of the same size from the encoder.

More recent approaches, however, utilize different variations of an improved fully convolutional network, called U-Net [16], which employs a decoder symmetric to the encoder as far as the filters and input dimensions are concerned. Specifically, S. Dev et al [17] proposed a light-weight convolutional network, called CloudSegNet, which is shown to outperform the fully convolutional network described above. CloudSegNet is a symmetric encoder-decoder architecture without skip connections, producing a probability map of pixel-wise cloud predictions. This probability map is transformed into a binary one by applying a fixed thresholding process, the value of which is determined by a Receiver Operating Curve.

In a similar manner Q. Song et al [18] has designed another convolutional network, where again there are no skip connections in the architecture. In more detail, the encoder integrates the ResNet [19] architecture without the final layers of the classifier while the decoder uses several special networks to generate a probability map of pixel-wise cloud predictions.

Finally, in the work of W. Xie et al [20], a deep convolutional network named SegCloud is proposed. This architecture follows likewise the symmetric encoder-decoder design pattern described above, albeit producing a three channel probability map. Furthermore, it introduces some skip connections, aimed for the concatenation of the encoder outputs with the ones of the same size produced by the decoder, in order to have the features of the output in each upsampling stage accurately restored.

Chapter 3

3 Fully convolutional neural networks

Fully convolutional neural networks intended for semantic segmentation can be generally considered as an encoder-decoder architecture. As its name implies, the encoder captures and stores context information from the input image, relative to the classes of interest via feature extraction, producing a high dimensional feature vector often referred as code. Conversely, the decoder utilizes the features provided by the code to build a pixel-wise map of the classes of interest with the same size as the input image. In addition, some architectures called Unets introduce skip connections from the encoder outputs to the decoder inputs so as to achieve a more precise localization. Furthermore, transfer learning has been a common practice in fully convolutional neural networks by incorporating in the encoder part various architectures pretrained on other datasets. However, the fully connected layers of their classifier are omitted. As a result, their operation purpose is redefined but the knowledge gained from the previous task, through the learned features, is intact and is employed to improve generalization on other tasks.

In this study five modified Unet architectures along with other 50, which use as backbone popular architectures from ImageNet Large Scale Visual Recognition Competition, have been implemented in order to be conducted a comparative analysis on their use for cloud image segmentation. Furthermore, the study extends to comparing different optimization algorithms and loss functions in order to determine the most suitable for the cloud segmentation task from the implemented collection.

The five Unets have the codename Unet, A_Unet, D_Unet, W_Unet and R_Unet. In all Unet implementations downsampling is handled exclusively by convolutions and upsampling by transposed ones. Additionally, instead of employing the common ReLU activation function, an improved version called GELU [21] is utilized. Another difference of these variants is that all convolutions except those dedicated for resampling are atrous [22] ones. A_Unet, additionally, utilizes an attention mechanism guiding it to focus more on the regions of interest. D_Unet employs parallel atrous convolutions of different dilation rates to capture different features. In a similar way, W_Unet utilizes parallel atrous convolutions of different kernel size. Finally, R_Unet integrates a modified inverted residual block with skip connections to improve the information flow.

As far as the networks exploiting architectures from ImageNet Large Scale Visual Recognition Competition are concerned, two variations have been implemented for each available architecture, except NasNetLarge, on the Keras API[10]. The first one is a simple encoder-decoder architecture while the second introduces skip connections. The decoder part of the networks albeit the simplest possible, it is adjusted to the downsampling stages of the encoder rendering the expansion process symmetric to the contraction one.

3.1 Unets

Unet [16]: This architecture was named after its symmetric u-shaped encoder-decoder structure. There are two building blocks that constitute this encoder-decoder structure. The first block utilized in the encoder consists of two convolutions of kernel size 3×3, which are followed by a convolution of kernel size 2×2 with stride 2. The second block is employed in the decoder and is comprised by a transposed convolution of kernel size 2×2 with stride 2, followed by two convolutions of kernel size 3×3. These blocks are repeated four times in the encoder and the decoder respectively. In each stage the number of filters in the convolutions is doubled in the encoder while in the decoder it is halved. Additionally, all decoder blocks take as input the output of the previous stage concatenated with the output of the encoder block that has the same number of filters as the decoder block. Furthermore, two convolutions of kernel size 3×3 are placed between the encoder and the decoder while another one of kernel size 1×1 is placed at the end. Finally, all convolutions are followed by batch-normalization and a GeLU activation function except for the last which is followed by a sigmoid.

Unet
Input
conv $[3\times 3, 16, \text{dilations}=3]$ conv $[3\times 3, 16, \text{dilations}=3]$ conv $[2\times 2, 16, \text{strides}=2]$
conv $[3\times 3, 32, \text{dilations}=3]$ conv $[3\times 3, 32, \text{dilations}=3]$ conv $[2\times 2, 32, \text{strides}=2]$
conv $[3\times 3, 64, \text{dilations}=3]$ conv $[3\times 3, 64, \text{dilations}=3]$ conv $[2\times 2, 64, \text{strides}=2]$

10 https://keras.io/api/applications/

conv $[3\times 3,128, \text{dilations}=3]$ conv $[3\times 3,128, \text{dilations}=3]$ conv $[2\times 2,128, \text{strides}=2]$
conv $[3\times 3,256, \text{dilations}=3]$ conv $[3\times 3,256, \text{dilations}=3]$
concatenation of encoder output $[128]$ & conv $[256]$ transposed conv $[2\times 2,128, \text{strides}=2]$ conv $[3\times 3,128, \text{dilations}=3]$ conv $[3\times 3,128, \text{dilations}=3]$
concatenation of encoder output $[64]$ & decoder output $[128]$ transposed conv $[2\times 2,64, \text{strides}=2]$ conv $[3\times 3,64, \text{dilations}=3]$ conv $[3\times 3,64, \text{dilations}=3]$
concatenation of encoder output $[32]$ & decoder output $[64]$ transposed conv $[2\times 2,32, \text{strides}=2]$ conv $[3\times 3,32, \text{dilations}=3]$ conv $[3\times 3,32, \text{dilations}=3]$
transposed conv $[2\times 2,16, \text{strides}=2]$ conv $[3\times 3,16, \text{dilations}=3]$ conv $[3\times 3,16, \text{dilations}=3]$
conv $[1\times 1,1]$

Table 3.1: Unet architecture

A_Unet : Heavily inspired by the work of [23] and [24] this architecture integrates a spatial-channel attention mechanism to the Unet model described above. This mechanism contains two gates, a spatial attention gate and a channel attention gate. Both gates take as input an output from the encoder and an output from the decoder. The output of the decoder is half the size (width,height) of the encoder output but it has the double number of filters. The spatial attention gate consists of three convolutions, one addition and one multiplication operation. In more detail, the encoder output passes through a convolution of kernel size 3×3 with stride 2 and 1 filter while the decoder output is filtered by a convolution of kernel size 3×3 with stride 2, dilation 3 and 1 filter. Afterwards an addition operation takes place and is followed by a convolution of kernel size 1×1 and 1 filter and a transposed convolution of kernel size 3×3, strides 2 and 1 filter. All convolutions in the spatial gate are followed by batch-normalization. Additionally the first two convolutions are followed by a GeLU activation function and the last one by a sigmoid. The channel attention gate is comprised by 2 global max pooling operations, 2 global average pooling operations, 4 reshape operations, 2 concatenations and 2 convolutions. More specifically, a global max pooling and a global average pooling operation is applied to each input. Afterwards, the outputs have their dimensions reshaped and the global average pooling output gets concatenated with the global max pooling one. Each output is then filtered by a convolution of kernel size 2×1 with the same number of filters as the input of the encoder. Finally, their outputs are concatenated and then filtered by a convolution of kernel size 1 with the same number of filters as the previous one followed by a sigmoid activation function. The outputs of the two gates are multiplied and a convolution of kernel size 1×1 with the same number of filters as the other two takes place. This last convolution is followed by batch-normalization and a GeLU activation function.

Decoder Input	Encoder Input
conv $[3 \times 3, \text{dilations}=3, \text{filters}=1]$	conv $[3 \times 3, \text{strides}=2, \text{filters}=1]$
Addition	
conv $[1 \times 1, \text{filters}=1]$	
transposed conv $[3 \times 3, \text{strides}=2, \text{filters}=1]$	
Sigmoid Activation Function	

Table 3.2: Spatial attention gate

Decoder Input	Decoder Input	Encoder Input	Encoder Input
global average pooling	global max pooling	global average pooling	global max pooling
Reshape	Reshape	Reshape	Reshape
Concatenation		Concatenation	
conv $[2 \times 1, \text{strides}=2]$		conv $[2 \times 1, \text{strides}=2]$	
Concatenation			
conv $[1 \times 1]$			
Sigmoid Activation Function			

Table 3.3: Channel attention gate

Decoder Input	Encoder Input	Decoder Input	Encoder Input
Spatial Attention Gate		Channel Attention Gate	
Multiplication			
conv $[1 \times 1]$			

Table 3.4: Attention mechanism

A_Unet
Input
conv$[3\times 3, 16, \text{dilations}=3]$ conv$[3\times 3, 16, \text{dilations}=3]$ conv$[2\times 2, 16, \text{strides}=2]$
conv$[3\times 3, 32, \text{dilations}=3]$ conv$[3\times 3, 32, \text{dilations}=3]$ conv$[2\times 2, 32, \text{strides}=2]$
conv$[3\times 3, 64, \text{dilations}=3]$ conv$[3\times 3, 64, \text{dilations}=3]$ conv$[2\times 2, 64, \text{strides}=2]$
conv$[3\times 3, 128, \text{dilations}=3]$ conv$[3\times 3, 128, \text{dilations}=3]$ conv$[2\times 2, 128, \text{strides}=2]$
conv$[3\times 3, 256, \text{dilations}=3]$ conv$[3\times 3, 256, \text{dilations}=3]$
spatial-channel attention gate concatenation of spatial-channel attention gate & conv$[256]$ transposed conv$[2\times 2, 128, \text{strides}=2]$ conv$[3\times 3, 128, \text{dilations}=3]$ conv$[3\times 3, 128, \text{dilations}=3]$
spatial-channel attention gate concatenation of spatial-channel attention gate & decoder output$[128]$ transposed conv$[2\times 2, 64, \text{strides}=2]$ conv$[3\times 3, 64, \text{dilations}=3]$ conv$[3\times 3, 64, \text{dilations}=3]$
spatial-channel attention gate concatenation of spatial-channel attention gate & decoder output$[64]$ transposed conv$[2\times 2, 32, \text{strides}=2]$ conv$[3\times 3, 32, \text{dilations}=3]$ conv$[3\times 3, 32, \text{dilations}=3]$
spatial-channel attention gate concatenation of spatial-channel attention gate & decoder output$[32]$ transposed conv$[2\times 2, 16, \text{strides}=2]$ conv$[3\times 3, 16, \text{dilations}=3]$ conv$[3\times 3, 16, \text{dilations}=3]$
conv$[1\times 1, 1]$

Table 3.5: A_Unet architecture

D_Unet : Inspired by the work of [25], this architecture utilizes a block of dilated convolutions in the encoder. The block consists of three branches, each one containing two consecutive convolutions of the same kernel size and dilation rate. The first group of two convolutions adopts a kernel size 3×3 with dilation rate 1, the second group a kernel size 3×3 with dilation rate 3 and the third a kernel size 3×3 with dilation rate 5. Additionally, these groups of convolutions operate in parallel with each other and their outputs are then concatenated. The different dilation rates used by these convolutions aim at increasing the variety of features extracted from the input. Contrary to using convolutions of kernel sizes 1×1, 3×3 and 5×5, dilated convolutions can be executed in less time, as no bottleneck occurs, and with less computations than bigger kernels. Furthermore, all three convolutions are followed by batch-normalization and a GeLU activation function. Finally, as far as the decoder is concerned, it utilizes only one convolution with a dilation rate of 1.

Input		
conv $[3 \times 3, \text{dilations}=1]$ conv $[3 \times 3, \text{dilations}=1]$	conv $[3 \times 3, \text{dilations}=3]$ conv $[3 \times 3, \text{dilations}=3]$	conv $[3 \times 3, \text{dilations}=5]$ conv $[3 \times 3, \text{dilations}=5]$
Concatenation		

Table 3.6: D_Unet module

D_Unet
Input
block of dilated convolutions $[\text{filters}=16]$ conv $[2 \times 2{,}16, \text{strides}=2]$
block of dilated convolutions $[\text{filters}=32]$ conv $[2 \times 2{,}16, \text{strides}=2]$
block of dilated convolutions $[\text{filters}=64]$ conv $[2 \times 2{,}16, \text{strides}=2]$
block of dilated convolutions $[\text{filters}=128]$ conv $[2 \times 2{,}16, \text{strides}=2]$
block of dilated convolutions $[\text{filters}=256]$
concatenation of encoder output $[128]$ & conv $[256]$ transposed conv $[2 \times 2{,}128, \text{strides}=2]$ conv $[3 \times 3{,}128, \text{dilations}=3]$ conv $[3 \times 3{,}128, \text{dilations}=3]$

concatenation of encoder output [64] & decoder output [128] transposed conv [2 × 2,64 , strides= 2] conv [3 × 3,64 , dilations= 3] conv [3 × 3,64 , dilations= 3]
concatenation of encoder output [32] & decoder output [64] transposed conv [2 × 2,32 , strides= 2] conv [3 × 3,32 , dilations=3] conv [3 × 3,32 , dilations=3]
transposed conv [2 × 2,16 , strides= 2] conv [3 × 3,16 , dilations= 3] conv [3 × 3,16 , dilations= 3]
conv [1 × 1,1]

Table 3.7: D_Unet architecture

W_Unet : Inspired by the Inception modules this architecture employs a block of convolutions with different kernel sizes in the encoder. In greater detail, the block starts with two convolutions of kernel size 1×1 and is separated to two branches. The first branch is comprised by two convolutions of kernel size 3×3 and dilation rate of 3. The second branch consists of four convolutions, with the first two having a kernel size of 1×3 and a dilation rate of 3 and the last two a kernel size of 3×1 and a dilation rate of 3. The outputs of the two branches are then concatenated. All convolutions in the block are followed by batch-normalization and a GeLU activation function. Finally, everything else remains the same as in the Unet model described above.

Input	
conv [1 × 1 , dilations= 1] conv [1 × 1 , dilations= 1]	
conv [3 × 3 , dilations=3] conv [3 × 3 , dilations=3]	conv [1 × 3 , dilations= 3] conv [1 × 3 , dilations= 3] conv [3 × 1 , dilations= 3] conv [3 × 1 , dilations= 3]
Concatenation	

Table 3.8: W_Unet module

W_Unet
Input
block of convolutions $[\text{filters}=16]$ conv $[2\times 2{,}16\,,\text{strides}=2]$
block of convolutions $[\text{filters}=32]$ conv $[2\times 2{,}32\,,\text{strides}=2]$
block of convolutions $[\text{filters}=64]$ conv $[2\times 2{,}64\,,\text{strides}=2]$
block of convolutions $[\text{filters}=128]$ conv $[2\times 2{,}128\,,\text{strides}=2]$
block of convolutions $[\text{filters}=256]$
concatenation of encoder output $[128]$ & conv $[256]$ transposed conv $[2\times 2{,}128\,,\text{strides}=2]$ conv $[3\times 3{,}128\,,\text{dilations}=3]$ conv $[3\times 3{,}128\,,\text{dilations}=3]$
concatenation of encoder output $[64]$ & decoder output $[128]$ transposed conv $[2\times 2{,}64\,,\text{strides}=2]$ conv $[3\times 3{,}64\,,\text{dilations}=3]$ conv $[3\times 3{,}64\,,\text{dilations}=3]$
concatenation of encoder output $[32]$ & decoder output $[64]$ transposed conv $[2\times 2{,}32\,,\text{strides}=2]$ conv $[3\times 3{,}32\,,\text{dilations}=3]$ conv $[3\times 3{,}32\,,\text{dilations}=3]$
transposed conv $[2\times 2{,}16\,,\text{strides}=2]$ conv $[3\times 3{,}16\,,\text{dilations}=3]$ conv $[3\times 3{,}16\,,\text{dilations}=3]$
conv $[1\times 1{,}1]$

Table 3.9: W_Unet architecture

R_Unet : This architecture integrates techniques from the ResNet [19], the MobileNetV2 [33], the DenseNet [34] model and the work of [26] into a more complicated bottleneck inverted residual block. This block starts and ends with a convolution of kernel size 1×1, thus creating a bottleneck block like ResNet. Between them are deployed three convolutions of kernel size 3×3 whose outputs are concatenated sequentially with their input like in the DenseNet architecture. Additionally, the outputs of the first convolution and the last are added together. Furthermore, the number of filters in the convolutions of kernel size 3×3 are double than those of kernel size 1×1 in a similar fashion to the MobileNetV2 inverted block structure. All convolutions in the block are followed by batch-normalization and a GeLU activation function. Finally, this block is utilized only by the encoder while everything else remains the same as in the Unet model described above.

Input
conv $[1 \times 1, \text{filters}= f]$
conv $[3 \times 3, \text{dilations}=3, \text{filters}= 2f]$
Concatenation
conv $[3 \times 3, \text{dilations}=3, \text{filters}= 2f]$
Concatenation
conv $[3 \times 3, \text{dilations}=3, \text{filters}= 2f]$
Concatenation
conv $[1 \times 1, \text{filters}= f]$
Addition

Table 3.10: R_Unet module

R_Unet
Input
complicated block $[\text{filters}=16]$ conv $[2 \times 2{,}16, \text{strides}=2]$
complicated block $[\text{filters}=32]$ conv $[2 \times 2{,}32, \text{strides}=2]$
complicated block $[\text{filters}=64]$ conv $[2 \times 2{,}64, \text{strides}=2]$
complicated block $[\text{filters}=128]$ conv $[2 \times 2{,}128, \text{strides}=2]$
block of convolutions $[\text{filters}=256]$
concatenation of encoder output $[128]$ & conv $[256]$ transposed conv $[2 \times 2{,}128, \text{strides}=2]$ conv $[3 \times 3{,}128, \text{dilations}=3]$ conv $[3 \times 3{,}128, \text{dilations}=3]$
concatenation of encoder output $[64]$ & decoder output $[128]$ transposed conv $[2 \times 2{,}64, \text{strides}=2]$ conv $[3 \times 3{,}64, \text{dilations}=3]$ conv $[3 \times 3{,}64, \text{dilations}=3]$

concatenation of encoder output $[32]$ & decoder output $[64]$ transposed conv $[2\times2{,}32,\text{strides}=2]$ conv $[3\times3{,}32,\text{dilations}=3]$ conv $[3\times3{,}32,\text{dilations}=3]$
transposed conv $[2\times2{,}16,\text{strides}=2]$ conv $[3\times3{,}16,\text{dilations}=3]$ conv $[3\times3{,}16,\text{dilations}=3]$
conv $[1\times1{,}1]$

Table 3.11: R_Unet architecture

3.2 ImageNets

VGG [15]: Convolutional layers with filters of small receptive field and max-pooling layers are the key components of this architecture. Particularly, VGG-16 and VGG-19 consist of 13 and 16 convolutional layers respectively. Additionally, four max-pooling layers are interconnected with those, forming batches of two and three and of two and four respectively. All the convolutions are conducted using a 3×3 kernel size, preserving the spatial resolution of the input and are also followed by a ReLU activation function. The number of their filters is initialized at 64 and is doubled after each batch until it reaches the limit of 512. All the max-pooling operations are computed using a 2×2 kernel size and a 2×2 stride.

VGG-16	**VGG-19**
Input	Input
conv $[3\times3{,}64]$ conv $[3\times3{,}64]$	conv $[3\times3{,}64]$ conv $[3\times3{,}64]$
maxpool $[2\times2{,}\text{stride}=2]$	maxpool $[2\times2{,}\text{stride}=2]$
conv $[3\times3{,}128]$ conv $[3\times3{,}128]$	conv $[3\times3{,}128]$ conv $[3\times3{,}128]$
maxpool $[2\times2{,}\text{stride}=2]$	maxpool $[2\times2{,}\text{stride}=2]$
conv $[3\times3{,}256]$ conv $[3\times3{,}256]$ conv $[3\times3{,}256]$	conv $[3\times3{,}256]$ conv $[3\times3{,}256]$ conv $[3\times3{,}256]$ conv $[3\times3{,}256]$
maxpool $[2\times2{,}\text{stride}=2]$	maxpool $[2\times2{,}\text{stride}=2]$

conv $[3\times3,512]$ conv $[3\times3,512]$ conv $[3\times3,512]$	conv $[3\times3,512]$ conv $[3\times3,512]$ conv $[3\times3,512]$ conv $[3\times3,512]$
maxpool $[2\times2,\text{stride}=2]$	maxpool $[2\times2,\text{stride}=2]$
conv $[3\times3,512]$ conv $[3\times3,512]$ conv $[3\times3,512]$	conv $[3\times3,512]$ conv $[3\times3,512]$ conv $[3\times3,512]$ conv $[3\times3,512]$

Table 3.12: VGG architectures

ResNet [19]: The novelty of this architecture are the shortcut connections which add the input of the first convolution to the output of the third convolution after every three convolutions creating pairs which are called residual blocks. This method tackles the degradation problem, thus making it possible to create models like ResNet50, ResNet101 and ResNet152 which consist of 49, 100 and 151 convolutional layers respectively. In all three variations what really differs is not the composition of the main building blocks but the number of times each one is applied. The first block consists of a convolution of 64 kernels of size 7×7 with a 2×2 stride, followed by a max-pooling layer of kernel size 3×3 with stride 2×2 and a batch of three convolutions which is applied three times in all variations. The batch consists of a convolution of 64 kernels of size 1×1, followed by a second of 64 kernels of size 3×3 and a third of 256 kernels of size 1×1. The second block consists of the same convolutional batch as before but doubled the kernels and is applied four times in the first two variations and eight times in the last one. The third block consists of the same convolutional batch as before but doubled the kernels again and is applied six times in ResNet50, twenty three times in ResNet101 and thirty six times in ResNet152. The last block consists of the same convolutional batch as before but doubled the kernels once more and is applied three times in all variations. In the second, third and fourth block only the first convolution uses a stride of size 2×2 in order to reduce the dimension of the feature maps to half (marked with an asterisk on the following table) while all the others use a stride of size 1×1, preserving at the same time the spatial resolution of the input. Furthermore all convolutions are followed by batch-normalization and a ReLU activation function. Interestingly, there is also a second version of these models where the addition of the inputs in each residual block takes place after they pass through the activation function [27].

ResNet50	**ResNet101**	**ResNet152**
Input	Input	Input
$\text{conv}\begin{bmatrix}7\times7,64,\\ \text{stride}=2\end{bmatrix}$ $\text{maxpool}\begin{bmatrix}3\times3,\\ \text{stride}=2\end{bmatrix}$ $\begin{bmatrix}\text{conv}[1\times1,64]\\ \text{conv}[3\times3,64]\\ \text{conv}[1\times1,256]\end{bmatrix}\times3$	$\text{conv}\begin{bmatrix}7\times7,64,\\ \text{stride}=2\end{bmatrix}$ $\text{maxpool}\begin{bmatrix}3\times3,\\ \text{stride}=2\end{bmatrix}$ $\begin{bmatrix}\text{conv}[1\times1,64]\\ \text{conv}[3\times3,64]\\ \text{conv}[1\times1,256]\end{bmatrix}\times3$	$\text{conv}\begin{bmatrix}7\times7,64,\\ \text{stride}=2\end{bmatrix}$ $\text{maxpool}\begin{bmatrix}3\times3,\\ \text{stride}=2\end{bmatrix}$ $\begin{bmatrix}\text{conv}[1\times1,64]\\ \text{conv}[3\times3,64]\\ \text{conv}[1\times1,256]\end{bmatrix}\times3$
$\begin{bmatrix}\text{conv}[1\times1,128]*\\ \text{conv}[3\times3,128]\\ \text{conv}[1\times1,512]\end{bmatrix}\times4$	$\begin{bmatrix}\text{conv}[1\times1,128]*\\ \text{conv}[3\times3,128]\\ \text{conv}[1\times1,512]\end{bmatrix}\times4$	$\begin{bmatrix}\text{conv}[1\times1,128]*\\ \text{conv}[3\times3,128]\\ \text{conv}[1\times1,512]\end{bmatrix}\times8$
$\begin{bmatrix}\text{conv}[1\times1,256]*\\ \text{conv}[3\times3,256]\\ \text{conv}[1\times1,1024]\end{bmatrix}\times6$	$\begin{bmatrix}\text{conv}[1\times1,256]*\\ \text{conv}[3\times3,256]\\ \text{conv}[1\times1,1024]\end{bmatrix}\times23$	$\begin{bmatrix}\text{conv}[1\times1,256]*\\ \text{conv}[3\times3,256]\\ \text{conv}[1\times1,1024]\end{bmatrix}\times36$
$\begin{bmatrix}\text{conv}[1\times1,512]*\\ \text{conv}[3\times3,512]\\ \text{conv}[1\times1,2048]\end{bmatrix}\times3$	$\begin{bmatrix}\text{conv}[1\times1,512]*\\ \text{conv}[3\times3,512]\\ \text{conv}[1\times1,2048]\end{bmatrix}\times3$	$\begin{bmatrix}\text{conv}[1\times1,512]*\\ \text{conv}[3\times3,512]\\ \text{conv}[1\times1,2048]\end{bmatrix}\times3$

Table 3.13: ResNet architectures

Inception [28]: The most distinctive features in this architecture are the inception modules, which are comprised of a max-pooling operation and three convolutions of kernel size 1×1, 3×3 and 5×5. All of these operations are conducted on the same input and their outputs are then concatenated to be passed on to the next module. Furthermore, because the convolutions of kernel size 3×3 and 5×5 are computationally expensive, two more convolutions of kernel size 1×1 are employed before them so as to reduce the number of feature maps. As a result these modules have the potential to capture information that is distributed both locally and globally on the input.

Input			
$\text{conv}[1\times1]$	$\text{maxpool}[3\times3]$ $\text{conv}[1\times1]$	$\text{conv}[1\times1]$ $\text{conv}[3\times3]$	$\text{conv}[1\times1]$ $\text{conv}[5\times5]$
Concatenation			

Table 3.14: Inception module

Several improvements on this module have led to the creation of InceptionV3 [29] model. Specifically, three new blocks were introduced in the InceptionV3 architecture which substituted the original inception module in order to reduce both the representational bottleneck and the computational complexity. The first block had the same structure as the original module except that the convolution of kernel size 5×5 has been substituted with two others of kernel size 3×3, so as to reduce the number of parameters and thus the chances of overfitting and accelerate the learning process at the same time. In the second block the 3×3 convolutions have been factorized to two consecutive convolutions of kernel size 1×3 and 3×1, in order to reduce even further the computational cost. At last, in the third block the consecutive convolutions of kernel size 1×3 and 3×1 have been separated and put in parallel with the same input, thus expanding the number of its filters but at the same time speeding up the training process. InceptionV3 consists of 94 convolutional layers and 10 pooling layers. In all blocks the max-pooling operation has been substituted by an average-pooling one. Furthermore all convolutions are followed by batch-normalization and a ReLU activation function.

Input			
conv [1 × 1,64]	avgpool [3 × 3, 192] conv [1 × 1,32]	conv [1 × 1,48] conv [3 × 3,64]	conv [1 × 1,64] conv [3 × 3,96] conv [3 × 3,96]
Concatenation			

Table 3.15: Inception module 1

Input			
conv [1 × 1,192]	avgpool [3 × 3, 768] conv [1 × 1,192]	conv [1 × 1,160] conv [1 × 3,160] conv [3 × 1,192]	conv [1 × 1,160] conv [1 × 3,160] conv [3 × 1,160] conv [1 × 3,160] conv [3 × 1,192]
Concatenation			

Table 3.16: Inception module 2

Input					
conv [1 × 1,320]	avgpool [3 × 3, 2048] conv [1 × 1,192]	conv [1 × 1,384]		conv [1 × 1,448] conv [3 × 3,384]	
		conv [1 × 3,384]	conv [3 × 1,384]	conv [1 × 3,384]	conv [3 × 1,384]

Concatenation

Table 3.17: Inception module 3

Further improvements on these modules combined with the introduction of residual connections have resulted in the creation of InceptionResNetV2 [30] model. The three blocks which were utilized in the InceptionV3 architecture have been modified and being given the code names of Inception-A block, Inception-B block and Inception-C block. Although, the structure of Inception-A block has been kept intact, the Inception-B block and Inception-C block have been mildly modified. In more detail, kernel size of 1×7 and 7×1 have been employed in Inception-B block instead of 1×3 and 3×1. Additionally, in Inception-C block the 3×3 convolution has been factorized to two consecutive convolutions of kernel sizes 1×3 and 3×1. Furthermore, two reduction modules have been introduced in this architecture with code names Reduction-A block and Reduction-B block. The Reduction-A block is comprised of three branches. The first branch is a max-pooling operation of kernel size 3×3 and stride 2. The second branch is a convolution of kernel size 3×3 and stride 2. The last branch contains a convolution of kernel size 1×1, followed by a convolution of kernel size 3×3 and another one of the same kernel size but with stride 2. The Reduction-B block is comprised of four branches. The first branch is a max-pooling operation of kernel size 3×3 and stride 2. The second and third branch contain a convolution of kernel size 1×1, followed by a convolution of kernel size 3×3 and stride 2 but with different number of filters. The fourth branch contains a convolution of kernel size 1×1, followed by a convolution of kernel size 3×3 and another one of the same kernel size but with stride 2. InceptionResNetV2 consists of 196 convolutional layers and 24 pooling layers. Finally, all convolutions are followed by batch-normalization and a ReLU function.

Input			
conv$[1\times 1,96]$	avgpool$[3\times 3]$ conv$[1\times 1,96]$	conv$[1\times 1,64]$ conv$[3\times 3,96]$	conv$[1\times 1,64]$ conv$[3\times 3,96]$ conv$[3\times 3,96]$
Concatenation			

Table 3.18: Inception A block

Input			
conv $[1 \times 1, 384]$	avgpool $[3 \times 3]$ conv $[1 \times 1, 128]$	conv $[1 \times 1, 192]$ conv $[1 \times 3, 224]$ conv $[3 \times 1, 256]$	conv $[1 \times 1, 192]$ conv $[1 \times 3, 192]$ conv $[3 \times 1, 224]$ conv $[1 \times 3, 224]$ conv $[3 \times 1, 256]$
Concatenation			

Table 3.19: Inception B block

Input					
conv $[1 \times 1, 256]$	avgpool $[3 \times 3]$ conv $[1 \times 1, 256]$	conv $[1 \times 1, 384]$		conv $[1 \times 1, 384]$ conv $[1 \times 3, 448]$ conv $[3 \times 1, 512]$	
		conv $[1 \times 3, 256]$	conv $[3 \times 1, 256]$	conv $[3 \times 1, 256]$	conv $[3 \times 1, 256]$
Concatenation					

Table 3.20: Inception C block

Input		
maxpool $\begin{bmatrix} 3 \times 3, 384, \\ \text{stride}=2 \end{bmatrix}$	conv $\begin{bmatrix} 3 \times 3, 384, \\ \text{stride}=2 \end{bmatrix}$	conv $[1 \times 1, 256]$ conv $[3 \times 3, 256]$ conv $\begin{bmatrix} 3 \times 3, 384, \\ \text{stride}=2 \end{bmatrix}$
Concatenation		

Table 3.21: Reduction A block

Input			
maxpool $\begin{bmatrix} 3 \times 3, 384, \\ \text{stride}=2 \end{bmatrix}$	conv $[1 \times 1, 256]$ conv $\begin{bmatrix} 3 \times 3, 288, \\ \text{stride}=2 \end{bmatrix}$	conv $[1 \times 1, 256]$ conv $\begin{bmatrix} 3 \times 3, 384, \\ \text{stride}=2 \end{bmatrix}$	conv $[1 \times 1, 256]$ conv $[1 \times 3, 288]$ conv $\begin{bmatrix} 3 \times 1, 320, \\ \text{stride}=2 \end{bmatrix}$
Concatenation			

Table 3.22: Reduction B block

InceptionV3	**InceptionResNetV2**	
Input	Input	
conv $\begin{bmatrix} 3\times 3, 32, \\ \text{stride}=2 \end{bmatrix}$	conv $[3\times 3, 32, \text{stride}=2]$	
	conv $[3\times 3, 32]$	
conv $[3\times 3, 32]$	conv $[3\times 3, 64]$	
conv $[3\times 3, 64]$	maxpool $\begin{bmatrix} 3\times 3, 64, \\ \text{stride}=2 \end{bmatrix}$	conv $\begin{bmatrix} 3\times 3, 96, \\ \text{stride}=2 \end{bmatrix}$
	Concatenation	
maxpool $\begin{bmatrix} 3\times 3, 64, \\ \text{stride}=2 \end{bmatrix}$	conv $\begin{bmatrix} 1\times 1, \\ 64 \end{bmatrix}$	conv $[1\times 1, 64]$
conv $[3\times 3, 80]$		conv $[7\times 1, 64]$
	conv $\begin{bmatrix} 3\times 3, \\ 96 \end{bmatrix}$	conv $[1\times 7, 64]$
conv $[3\times 3, 192]$		conv $[3\times 3, 96]$
	Concatenation	
maxpool $\begin{bmatrix} 3\times 3, 192, \\ \text{stride}=2 \end{bmatrix}$	conv $[3\times 3, 192]$	maxpool $[\text{stride}=2]$
	Concatenation	
Inception block 1 × 3	Inception-A block × 5	
	Reduction block A	
Inception block 2 × 5	Inception-B block × 10	
	Reduction block B	
Inception block 3 × 2	Inception-C block × 5	

Table 3.23: Inception architectures

Xception [31]: This architecture integrates techniques from both the Inception and the ResNet models. Although its structure is similar to that of the InceptionV3, it utilizes different modules based on the assumption that cross-channel features can be extracted separately from spatial ones. The main components of these modules are depthwise separable convolutions and residual connections. The first module contains two branches whose outputs are added together at the end. The first branch consists of a convolution of kernel size 1×1 with stride 2, whereas the second branch is comprised of two depthwise separable convolutions of kernel size 3×3, followed by a max-pooling operation of kernel size 3×3 with stride 2. The second module is a residual block of three depthwise separable convolutions of kernel size 3×3. In these two modules only the depthwise separable convolutions are preceded by a ReLU activation function. Xception contains 40 convolutions and 4 max-pooling operations, which are all followed by batch-normalization.

Input	
conv $\begin{bmatrix} 3\times 3, \\ \text{stride}=2 \end{bmatrix}$	depthwise separable conv $[3\times 3]$
	depthwise separable conv $[3\times 3]$
	maxpool $[3\times 3, \text{stride}=2]$
Addition	

Table 3.24: Xception block 1

Input	
$\times 1$	depthwise separable conv $[3\times 3]$
	depthwise separable conv $[3\times 3]$
	depthwise separable conv $[3\times 3]$
Addition	

Table 3.25: Xception block 2

Xception
Input
conv $[3\times 3, 32, \text{stride}=2]$
conv $[3\times 3, 64]$
Xception-1 block $[\text{filters} = 128]$
Xception-1 block $[\text{filters} = 256]$
Xception-1 block $[\text{filters} = 728]$
Xception-2 block $[\text{filters} = 728] \times 8$
Xception-1 block $[\text{filters} = 728]$
depthwise separable conv $[3\times 3, 1536]$
depthwise separable conv $[3\times 3, 2048]$

Table 3.26: Xception architecture

MobileNet [32]: The most characteristic things in this architecture are its lightweight structure and its efficiency on computer vision applications. Apart from the first layer, which is a regular convolution, all other convolutions adopted by the network are separable depthwise convolutions. However, instead of utilizing the standard depthwise separable convolution, it employs separately a depthwise spatial convolution and a pointwise convolution which execute the same operation. The reason behind this is to have their outputs filtered by a batch-normalization operation and a ReLU activation function. Additionally, downsampling is managed exclusively by strided depthwise convolutions. MobileNet contains 27 convolutions which are all followed by batch-normalization and a ReLU activation function. Furthermore, there is a second version of this model which utilizes an inverted residual block with linear bottleneck, called mobile inverted bottleneck. In more detail this block uses more filters in the intermediate convolutions than the ones with residual connections. Additionally, the final pointwise convolution is followed by a linear activation (marked with an asterisk on the following table). Again, all downsampling is managed exclusively by the first depthwise convolution of the block (marked with a dagger on the following table) with stride 2. MobileNetV2 [33] contains 53 convolutions which are all followed by batch-normalization and a ReLU activation function, except for the final pointwise convolution in each block which is followed only by batch-normalization.

MobileNet	**MobileNetV2**
Input	Input
conv $[3\times3, 32, \text{stride}=2]$	conv $[3\times3, 32, \text{stride}=2]$
depthwise conv $[3\times3, 32]$ conv $[1\times1, 64]$	conv $[1\times1, 32]$ depthwise conv $[3\times3, 32]$ conv $[1\times1, 16]$
depthwise conv $\begin{bmatrix}3\times3, 64, \\ \text{stride}=2\end{bmatrix}$ conv $[1\times1, 128]$	$\begin{bmatrix}\text{conv } [1\times1, 96] \\ \text{depthwise conv } [3\times3, 96, \text{stride}=2]\dagger \\ \text{conv } [1\times1, 24]* \end{bmatrix} \times 2$
depthwise conv $[3\times3, 128]$ conv $[1\times1, 128]$	$\begin{bmatrix}\text{conv } [1\times1, 144] \\ \text{depthwise conv } [3\times3, 144, \text{stride}=2]\dagger \\ \text{conv } [1\times1, 32]* \end{bmatrix} \times 3$
depthwise conv $\begin{bmatrix}3\times3, 128, \\ \text{stride}=2\end{bmatrix}$ conv $[1\times1, 256]$	$\begin{bmatrix}\text{conv } [1\times1, 192] \\ \text{depthwise conv } [3\times3, 192, \text{stride}=2]\dagger \\ \text{conv } [1\times1, 64]* \end{bmatrix} \times 4$
depthwise conv $[3\times3, 256]$ conv $[1\times1, 256]$	$\begin{bmatrix}\text{conv } [1\times1, 384] \\ \text{depthwise conv } [3\times3, 384] \\ \text{conv } [1\times1, 96]* \end{bmatrix} \times 3$
depthwise conv $\begin{bmatrix}3\times3, 256, \\ \text{stride}=2\end{bmatrix}$ conv $[1\times1, 512]$	$\begin{bmatrix}\text{conv } [1\times1, 576] \\ \text{depthwise conv } [3\times3, 576, \text{stride}=2]\dagger \\ \text{conv } [1\times1, 160]* \end{bmatrix} \times 3$

$\begin{bmatrix} \text{depthwise conv}\left[3\times3,512\right] \\ \text{conv}\left[1\times1,512\right] \end{bmatrix} \times 5$	$\begin{bmatrix} \text{conv}\left[1\times1,960\right] \\ \text{depthwise conv}\left[3\times3,960\right] \\ \text{conv}\left[1\times1,320\right]* \end{bmatrix}$
depthwise conv $\begin{bmatrix} 3\times3,512, \\ \text{stride}=2 \end{bmatrix}$ conv $\left[1\times1,1024\right]$	conv $\left[1\times1,1280\right]$
depthwise conv $\begin{bmatrix} 3\times3,1024, \\ \text{stride}=2 \end{bmatrix}$ conv $\left[1\times1,1024\right]$	

Table 3.27: MobileNet architectures

DenseNet [34]: As its name suggests, this architecture consists of Dense blocks which have all their layers connected directly with each other. This results in surpassing the vanishing gradient problem and boosting both feature propagation and feature reuse. Each Dense block is comprised by a stack of Dense layers whose outputs are concatenated before passing on to the next dense layer. The Dense layer consists of a pointwise convolution followed by another convolution of kernel size 3×3. Additionally, a Transition layer is placed after each dense block in order to reduce both the number of feature maps and the dimensions of its output. The Transition layer contains a pointwise convolution followed by an average-pooling operation of kernel size 2×2. All convolutions in both the Transition and the Dense layer are preceded by batch-normalization and a ReLU activation function. DenseNet-121 contains 120 convolutions, DenseNet-169 contains 168 convolutions, DenseNet-201 contains 200 convolutions and all of them utilize 4 pooling operations.

Input
Dense Layer 1
Dense Layer 2
Concatenation of 1 & 2
Dense Layer 3
Concatenation of 1 & 2 & 3
…
Dense Layer N
Concatenation of 1 & 2 & 3 & ... & N

Table 3.28: DensNet module

DenseNet-121	**DenseNet-169**	**DenseNet-201**
Input	Input	Input
conv $[7\times7, 64, \text{stride}=2]$	conv $[7\times7, 64, \text{stride}=2]$	conv $[7\times7, 64, \text{stride}=2]$
maxpool $[3\times3, 64, \text{stride}=2]$	maxpool $[3\times3, 64, \text{stride}=2]$	maxpool $[3\times3, 64, \text{stride}=2]$
$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 6$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 6$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 6$
conv $[1\times1,128]$ avgpool $[3\times3,128, \text{stride}=2]$	conv $[1\times1,128]$ avgpool $[3\times3,128, \text{stride}=2]$	conv $[1\times1,128]$ avgpool $[3\times3,128, \text{stride}=2]$
$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 12$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 12$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 12$
conv $[1\times1,256]$ avgpool $[3\times3,256, \text{stride}=2]$	conv $[1\times1,256]$ avgpool $[3\times3,256, \text{stride}=2]$	conv $[1\times1,256]$ avgpool $[3\times3,256, \text{stride}=2]$
$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 24$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 32$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 48$
conv $[1\times1,896]$ avgpool $[3\times3,896, \text{stride}=2]$	conv $[1\times1,896]$ avgpool $[3\times3,896, \text{stride}=2]$	conv $[1\times1,896]$ avgpool $[3\times3,896, \text{stride}=2]$
$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 16$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 32$	$\begin{bmatrix}\text{conv}[1\times1,128]\\ \text{conv}[3\times3,32]\end{bmatrix}\times 32$

Table 3.29: DenseNet architectures

NASNet [35]: This novel architecture was generated by the Neural Architecture Search framework [36] using CIFAR-10 as the validation dataset. In more detail, the research was conducted on two modules, the Normal Cell which preserves the dimensions of the input and the Reduction Cell which reduces them by a factor of two. Additionally, each of those cells was determined to be comprised by five smaller blocks whose outputs are either added or concatenated. These blocks contain two operations that take an input either from the last or its previous layer and have their outputs added together. The candidate operations to be employed in those blocks were based on the prevalent operations utilized by the state of the art models. The best Normal Cell consists of 3 depthwise separable convolutions of kernel size 3×3, 2 depthwise separable convolutions of kernel size 5×5, 3 average-pooling operations of kernel size 3×3 and 2 identity operations. On the other hand the best Reduction Cell is comprised of 2 depthwise separable convolutions of kernel size 7×7, 2 depthwise separable convolutions of kernel size 5×5, 1 depthwise separable convolution of kernel size 3×3, 2 average-pooling operations of kernel size 3×3, 2 max-pooling operations of kernel size 3×3 and 1 identity operation. NASNetMobile contains 81 depthewise separable convolutions and 52 pooling operations while NASNetLarge contains 111 depthewise separable convolutions and 70 pooling operations. Finally, all convolutions are followed by batch-normalization and a ReLU activation function.

i	i	i-1	i	i	i-1	i-1	i-1	i-1	i-1
sep conv $[3\times 3]$	$\times$ 1	sep conv $[3\times 3]$	sep conv $[5\times 5]$	avgpool $[3\times 3]$	$\times$ 1	avgpool $[3\times 3]$	avgpool $[3\times 3]$	sep conv $[5\times 5]$	sep conv $[3\times 3]$
Addition		Addition		Addition		Addition		Addition	
Concatenation									

Table 3.30: NasNet Normal Cell

i	i-1	i	i	i-1	i	i-1
maxpool $[3\times 3]$	sep conv $[7\times 7]$	sep conv $[5\times 5]$	maxpool $[3\times 3]$	sep conv $[7\times 7]$	avgpool $[3\times 3]$	sep conv $[5\times 5]$
	Addition		Addition		Addition	
	sep conv $[3\times 3]$	avgpool $[3\times 3]$	$\times$ 1			
Addition		Addition				
Concatenation						

Table 3.31: NasNet Reduction Cell

NASNetMobile	**NASNetLarge**
Input	Input
conv $[3\times 3, 32, \text{stride}=2]$	conv $[3\times 3, 96, \text{stride}=2]$
Reduction Cell $[\text{filters} = 11]$	Reduction Cell $[\text{filters} = 42]$
Reduction Cell $[\text{filters} = 22]$	Reduction Cell $[\text{filters} = 84]$
Normal Cell $[\text{filters} = 44] \times 4$	Normal Cell $[\text{filters} = 168] \times 6$
Reduction Cell $[\text{filters} = 88]$	Reduction Cell $[\text{filters} = 336]$
Normal Cell $[\text{filters} = 88] \times 4$	Normal Cell $[\text{filters} = 336] \times 6$
Reduction Cell $[\text{filters} = 176]$	Reduction Cell $[\text{filters} = 672]$
Normal Cell $[\text{filters} = 176] \times 4$	Normal Cell $[\text{filters} = 672] \times 6$

Table 3.32: NasNet architectures

EfficientNet [37]: This is a state of the art architecture that was generated by the Neural Architecture Search framework [38], utilizing the same search space as [n]. Its key component is a convolutional block which was also used by MnasNet. Specifically, it integrates the mobile inverted bottleneck technique of MobileNetV2 and the squeeze and excitation technique of SENet [39]. The block starts with a regular convolution followed by a depthwise separable one and continues in two branches. The first branch contains a global average pooling followed by a reshape operation and two regular convolutions whose number of filters preserve a ratio of 24 for the squeeze and excitation technique. The second branch is an identity operation so that the two outputs can be multiplied together and preserve the initial dimensions. Finally, after the multiplication a regular convolution takes place. In this block only the first two convolutions are followed by batch-normalization and a ReLU activation function, while the last is followed by batch-normalization only. Depending on the variation of the model and its stage this block is repeated from 1 to 13 times in the stage. Additionally, the blocks are added together with residual connections. Furthermore, a dropout operation takes place after the last convolution of each block of the stage except for the first one. Finally, there are eight variations of the EfficientNet model and their number of convolutions vary from 80 (EfficientNetB0) to 275 (EfficientNetB7).

conv $[1\times1,6\cdot f]$	
depthwise separable conv $[n\times n,6\cdot f]$	
global average pooling	$\times 1$
reshape	
conv $[n\times n,(6\cdot f/24)]$	
conv $[n\times n,6\cdot f]$	
Multiplication	
conv $[n\times n]$	

Table 3.33: EfficientNet module

EfficientNet							
B0	B1	B2	B3	B4	B5	B6	B7
block $\times 1$ $[3\times3,32]$	block $\times 2$ $[3\times3,32]$	block $\times 2$ $[3\times3,32]$	block $\times 2$ $[3\times3,40]$	block $\times 2$ $[3\times3,48]$	block $\times 3$ $[3\times3,48]$	block $\times 3$ $[3\times3,56]$	block $\times 4$ $[3\times3,64]$
block $\times 2$ $[3\times3,16]$	block $\times 3$ $[3\times3,16]$	block $\times 3$ $[3\times3,16]$	block $\times 3$ $[3\times3,24]$	block $\times 4$ $[3\times3,24]$	block $\times 5$ $[3\times3,24]$	block $\times 6$ $[3\times3,32]$	block $\times 7$ $[3\times3,32]$
block $\times 2$ $[5\times5,24]$	block $\times 3$ $[5\times5,24]$	block $\times 3$ $[5\times5,24]$	block $\times 3$ $[5\times5,32]$	block $\times 4$ $[5\times5,32]$	block $\times 5$ $[5\times5,40]$	block $\times 6$ $[5\times5,40]$	block $\times 7$ $[5\times5,48]$

block $\times 3$ $[3 \times 3, 40]$	block $\times 4$ $[3 \times 3, 40]$	block $\times 4$ $[3 \times 3, 48]$	block $\times 5$ $[3 \times 3, 48]$	block $\times 6$ $[3 \times 3, 56]$	block $\times 7$ $[3 \times 3, 64]$	block $\times 8$ $[3 \times 3, 72]$	block $\times 10$ $[3 \times 3, 80]$
block $\times 3$ $[5 \times 5, 80]$	block $\times 4$ $[5 \times 5, 80]$	block $\times 4$ $[5 \times 5, 88]$	block $\times 5$ $[5 \times 5, 96]$	block $\times 6$ $[5 \times 5, 112]$	block $\times 7$ $[5 \times 5, 128]$	block $\times 8$ $[5 \times 5, 144]$	block $\times 10$ $[5 \times 5, 160]$
block $\times 4$ $[5 \times 5, 112]$	block $\times 5$ $[5 \times 5, 112]$	block $\times 5$ $[5 \times 5, 120]$	block $\times 6$ $[5 \times 5, 136]$	block $\times 8$ $[5 \times 5, 160]$	block $\times 9$ $[5 \times 5, 176]$	block $\times 11$ $[5 \times 5, 200]$	block $\times 13$ $[5 \times 5, 224]$
block $\times 1$ $[3 \times 3, 192]$	block $\times 2$ $[3 \times 3, 192]$	block $\times 2$ $[3 \times 3, 208]$	block $\times 2$ $[3 \times 3, 232]$	block $\times 2$ $[3 \times 3, 272]$	block $\times 3$ $[3 \times 3, 304]$	block $\times 3$ $[3 \times 3, 344]$	block $\times 4$ $[3 \times 3, 384]$

Table 3.34: EfficientNet architectures

3.3 Optimizers

SGD [40]: The Stochastic Gradient Descent algorithm aims at minimizing the loss function which is related to the error rate in the predictions of the network. In more detail, by tweaking the function's parameters a local minimum is pursuit in order to minimize the error rate.

RMSprop [40]: The Root Mean Squared Propagation method is an adaptive learning rate method that divides the gradient by the root of a moving average of the squared gradients. Furthermore, it utilizes only the sign of the gradient, adjusting the step size separately in order to determine a single global learning rate.

Adagrad [41]: This optimization method relies on the dynamic adaptation of the proximal function, regulating the gradient steps of the algorithm, over time in a data driven way. That means the learning rates depend on the frequency that parameters get updated, affecting them less with each update.

Adadelta [42]: This is a per-dimension learning rate method for stochastic gradient descent, derived from the Adagrad optimization algorithm. It introduces an adaptive learning rate so as to avoid its continual decay, resulting in becoming infinitesimally small after a certain number of epochs, and the need of manually selecting a global one. Specifically, the learning rate adapts dynamically over time based on a moving window of gradient updates, aiming at continuous learning throughout the training.

Adam [43]: This optimization algorithm was named after its adaptive moment estimation and integrates the ability of the Adagrad method to cope with sparse arrays along with the ability of the RMSprop method to handle non-stationary objectives. Based on the first and second orders of the gradient Adam adaptively computes individual learning rates for different parameters. Thus, succeeding in optimizing the stochastic objective functions.

Adamax [43]: This is a variant of the Adam optimization algorithm which differs in the update rule. In more detail, instead of scaling the gradients of individual weights inversely proportional to an L^2 norm it is generalized to an L^p norm where p is close to infinity.

Ftrl [44]: This optimization's algorithm full name is Follow The Regularized Leader and was developed to predict ad click-through rates for sponsored search advertising. Ftrl combines both the sparsity produced by the Regularized Dual Averaging method and the gradient-descent style of Online gradient Descent method which provides improved accuracy.

Nadam [45]: This optimization algorithm introduces a modified version of Nesterov's Accelerated Gradient technique to the Adam optimization method. By utilizing a decaying sum of the previous gradients into a momentum vector instead of the true gradient, it can regulate the learning rate more efficiently. Specifically, gradient descent learning is accelerated during training steps on stable dimensions and is decelerated on inconsistent ones avoiding oscillation.

AdamW [46]: This is a variant of the Adam optimization algorithm which improves its regularization by decoupling the weight decay from the gradient-based update. Thus, reducing the impact of the learning rate on the optimal choice of the weight decay factor.

SGDW [46]: This is a variant of the Stochastic Gradient Descent optimization algorithm which improves its regularization by decoupling the weight decay from the gradient-based update. Thus, reducing the impact of the learning rate on the optimal choice of the weight decay factor.

ConditionalGradient [47]: This optimization algorithm is based on the Frank-Wolfe optimization method. It is an iterative method which employs a linear approximation of the objective function in order to find a local minimum. Furthermore, several modifications have been introduced to the algorithm so as to handle computationally the various constraints incorporated.

LAMB [48]: Inspired by the LARS [49] method, LAMB is a large batch stochastic optimization method. Specifically, it utilizes an adaptive elementwise updating as well as a layerwise adaptive learning rate technique in order to accelerate the training of deep neural network on large mini-batches. Furthermore, it can also be used as a general purpose optimizer supporting both small and large batches.

LazyAdam [11]: This is another variant of the Adam optimizer that differs from the original in the handling of the sparse updates. Particularly, it only updates the moving-average accumulators that appear in the current batch, omitting the rest.

[11] https://www.tensorflow.org/addons/tutorials/optimizers_lazyadam

ProximalAdagrad [50]: This method is designed for optimization of convex problems. In more detail, after executing an unconstrained gradient descent step, it considers and solves an optimization problem which aims to minimize a regularization term while maintaining close proximity to the result produced from the first phase.

RectifiedAdam [51]: This is a variant of the Adam optimizer that introduces a new term to rectify the variance of the adaptive learning rate. This helps the variance to become more consistent avoiding convergence to bad local optima due to large variance of the learning rate in the early stage of the training caused by the limited amount of samples.

Yogi [52]: This is an additive adaptive stochastic optimization method addressing non convex problems. It employs a controlled increase of effective learning rate achieving convergence even with increasing minibatch size.

3.4 Loss Functions

Binary Cross-Entropy [53]: It calculates the difference between two probability distributions for a set of events. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{BCE}(y_{true}, y_{pred}) = -(y_{true} \times \log(y_{pred}) + (1 - y_{true}) \times \log(1 - y_{pred}))$$

Dice [54]: This loss is derived from the f1 score which is used as a segmentation evaluation metric measuring the difference between two images. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{DICE}(y_{true}, y_{pred}) = 1 - \frac{2 \times y_{true} \times y_{pred} + \varepsilon}{y_{true} + y_{pred} + \varepsilon}$$

Fbeta [54]: This loss is a generalization of the f1 score which is used as a segmentation evaluation metric measuring the difference between two images. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network with weight (β) is defined as:

$$L_{Fbeta}(y_{true}, y_{pred}) = 1 - \frac{(1+\beta^2) \times y_{true} \times y_{pred} + \varepsilon}{(1+\beta^2) \times y_{true} \times y_{pred} + \beta^2 \times y_{true} \times (1 - y_{pred}) + (1 - y_{true}) \times y_{pred} + \varepsilon}$$

Jaccard [54]: This loss is derived from the IoU which is used as a segmentation evaluation metric measuring the difference between two images. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{JACCARD}(y_{true}, y_{pred}) = 1 - \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true} + y_{pred} - y_{true} \times y_{pred} + \varepsilon}$$

PowerJaccard [54]: This loss is derived from the IoU which is used as a segmentation evaluation metric measuring the difference between two images. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network in the power (p) is defined as:

$$L_{powerJACCARD}(y_{true}, y_{pred}) = 1 - \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true}^{p} + y_{pred}^{p} - y_{true} \times y_{pred} + \varepsilon}$$

Tversky [53]: This loss emphasizes more on the tradeoff between false positive and false negative results produced from the comparison of two images. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network with weight (β) is defined as:

$$L_{TVERSKY}(y_{true}, y_{pred}) = 1 - \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true} \times y_{pred} + \beta \times (1 - y_{true}) \times y_{pred} + (1 - \beta) \times y_{true} \times (1 - y_{pred}) + \varepsilon}$$

Log Cosh Dice [53]: The Log Cosh is integrated in the Dice loss in order to make its curve smoother. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{LCD}(y_{true}, y_{pred}) = \log\left(\cosh\left(1 - \frac{2 \times y_{true} \times y_{pred} + \varepsilon}{y_{true} + y_{pred} + \varepsilon}\right)\right)$$

Log Cosh Jaccard: The Log Cosh is integrated in the Dice loss in order to make its curve smoother. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{LCJ}(y_{true}, y_{pred}) = \log\left(\cosh\left(1 - \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true} + y_{pred} - y_{true} \times y_{pred} + \varepsilon}\right)\right)$$

Binary Cross-Entropy Dice: This is a compound loss calculating the sum of binary Cross-Entropy and Dice loss functions. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{BCEDice}(y_{true}, y_{pred}) = L_{BCE} + L_{DICE}$$

Binary Cross-Entropy Log Cosh Jaccard: This is a compound loss calculating the sum of binary Cross-Entropy and Log Cosh Jaccard loss functions. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{BCELCJ}(y_{true}, y_{pred}) = L_{BCE} + L_{LCJ}$$

Fbeta powerJaccard: This is a compound loss calculating the sum of Fbeta and power Jaccard loss functions. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{FbpJ}(y_{true}, y_{pred}) = L_{Fbeta} + L_{powerJACCARD}$$

Tversky Fbeta: This is a compound loss calculating the sum of Fbeta and Tversky loss functions. The loss function for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$L_{TFb}(y_{true}, y_{pred}) = L_{Fbeta} + L_{TVERSKY}$$

3.5 Metrics

F1 score [55]: This is interpreted as the harmonic mean of the presicion and recall. The metric formula for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$F1score(y_{true}, y_{pred}) = \frac{2 \times y_{true} \times y_{pred} + \varepsilon}{y_{true} + y_{pred} + \varepsilon}$$

IoU [55]: This is a metric that measures the percentage of overlap between two targets. The metric formula for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$IoU(y_{true}, y_{pred}) = \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true} + y_{pred} - y_{true} \times y_{pred} + \varepsilon}$$

Presicion [55]: This is refered as measure of quality. The metric formula for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$Presicion(y_{true}, y_{pred}) = \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true} \times y_{pred} + (1 - y_{true}) \times y_{pred} + \varepsilon}$$

Recall [55]: This is refered as measure of quantity. The metric formula for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$Recall(y_{true}, y_{pred}) = \frac{y_{true} \times y_{pred} + \varepsilon}{y_{true} \times y_{pred} + y_{true} \times (1 - y_{pred}) + \varepsilon}$$

Specificity [55]: This is defined as the proportion of actual negative results. The metric formula for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$Specificity(y_{true}, y_{pred}) = \frac{(1 - y_{true}) \times (1 - y_{pred}) + \varepsilon}{(1 - y_{true}) \times (1 - y_{pred}) + y_{true} \times (1 - y_{pred}) + \varepsilon}$$

Accuracy [55]: This corresponds to the percentage of correct predictions. The metric formula for the true values (y_{true}) and the predicted ones (y_{pred}) by the network is defined as:

$$Accuracy(y_{true}, y_{pred}) = \frac{y_{true} \times y_{pred} + (1 - y_{true}) \times (1 - y_{pred}) + \varepsilon}{y_{true} \times y_{pred} + (1 - y_{true}) \times y_{pred} + y_{true} \times (1 - y_{pred}) + (1 - y_{true}) \times (1 - y_{pred}) + \varepsilon}$$

Chapter 4

4 Experiments & results

4.1 Dataset

For the evaluation of the fully convolutional neural networks the Singapore Whole Sky Image Segmentation or SWIMSEG dataset has been chosen. It was created by S. Dev et al [10] for binary cloud segmentation and is publicly available[12]. Furthermore, it consists of 1013 images of size 600×600 pixel, carefully chosen from a 2 year collection, whose ground truth maps has been produced with the help of cloud experts from the Singapore Meteorological Services. Additionally, the images are undistorted and the majority of them depict moderately cloud or overcast conditions. Finally, none of the images contain a sun trace.

Because the SWIMSEG dataset is small the first experiment was conducted to determine the optimal ratio for the training, validation and test set. The proposed ratios were 60/20/20, 65/15/20, 70/15/15/, 70/10/20 and 80/10/10 for the training, validation and test set respectively. Moreover, the images of the dataset were not divided in a sequential manner or randomly but uniformly so as to include as many as possible different conditions in the training set. Finally, the datasets were tested on the five Unets for 51 epochs utilizing the Adam optimizer on a scheduled learning rate and the Dice loss function. The implemented schedule had the value of learning rate decreased after the first 10 epochs from the initial value of 0.001 to 0.0005, after 20 epochs to 0.0001 and after 30 epochs to 0.00005. Finally the images were resized to 300×300 pixels.

The results of the experiment are presented below in three batches of six charts, corresponding to the six metrics for the evaluation of the training, validation and test set. First is the training set, which is followed by the validation set and the test set. In each chart representing only one metric are compared the different dataset ratios through the scores of the five Unet variations in the respective metric.

12 https://vintage.winklerbros.net/swimseg.html

4.1.1 Training Set

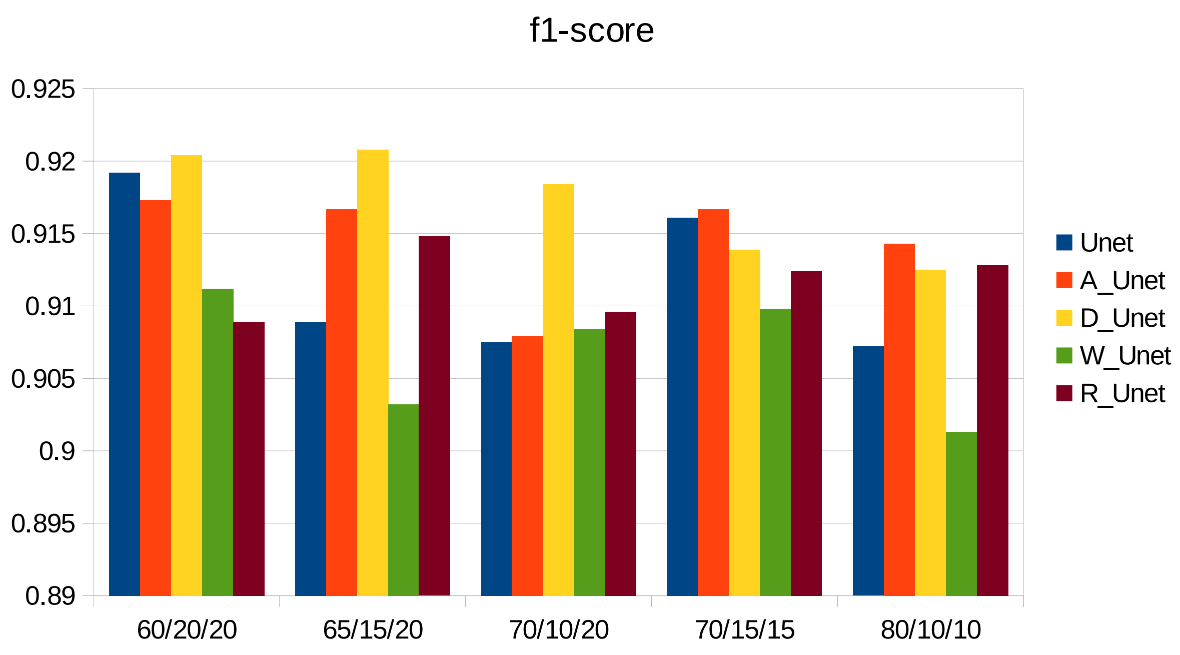

Figure 1: Comparison of the F1 score on the training set for the different dataset ratios

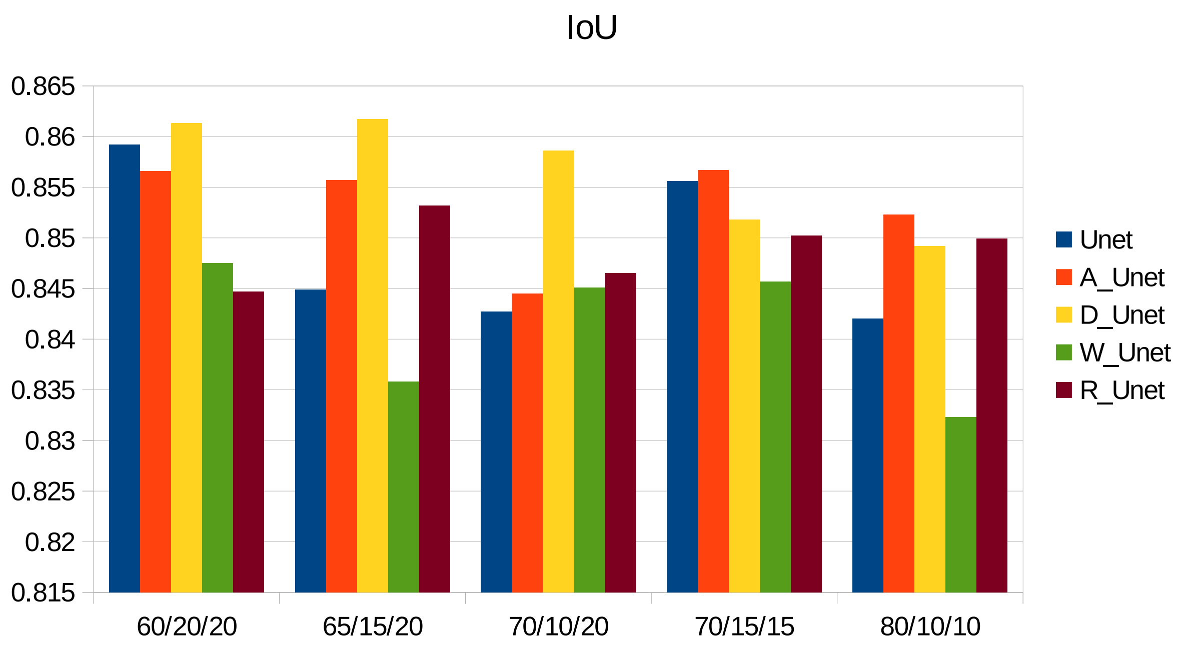

Figure 2: Comparison of the IoU on the training set for the different dataset ratios

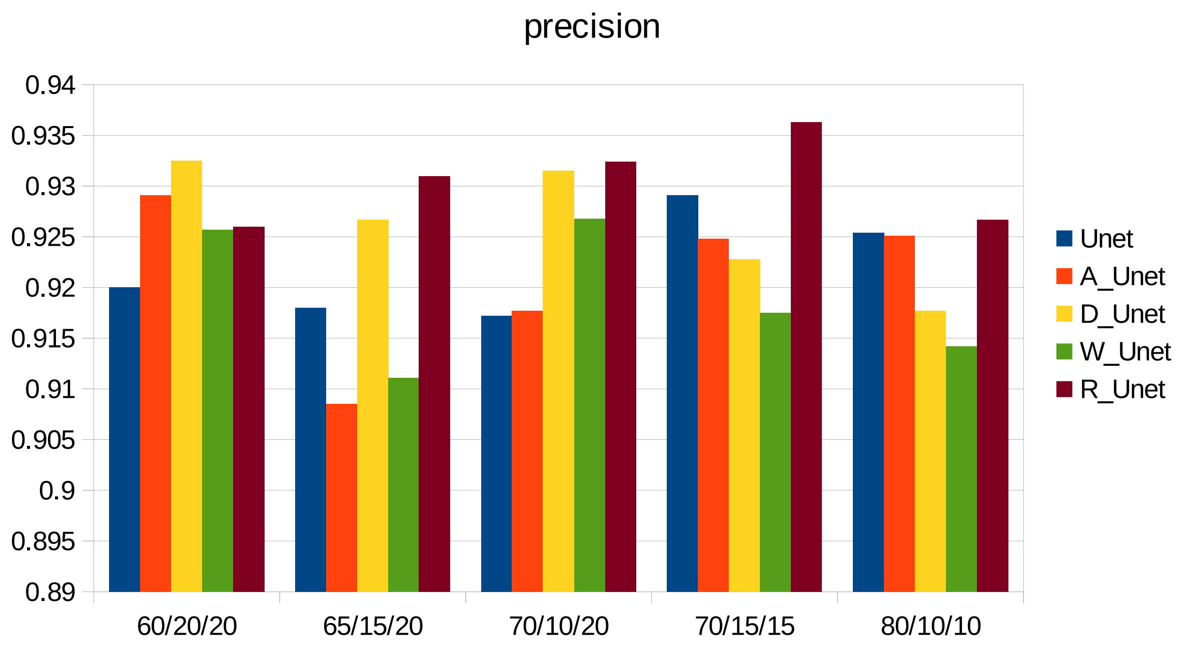

Figure 3: Comparison of the Precision on the training set for the different dataset ratios

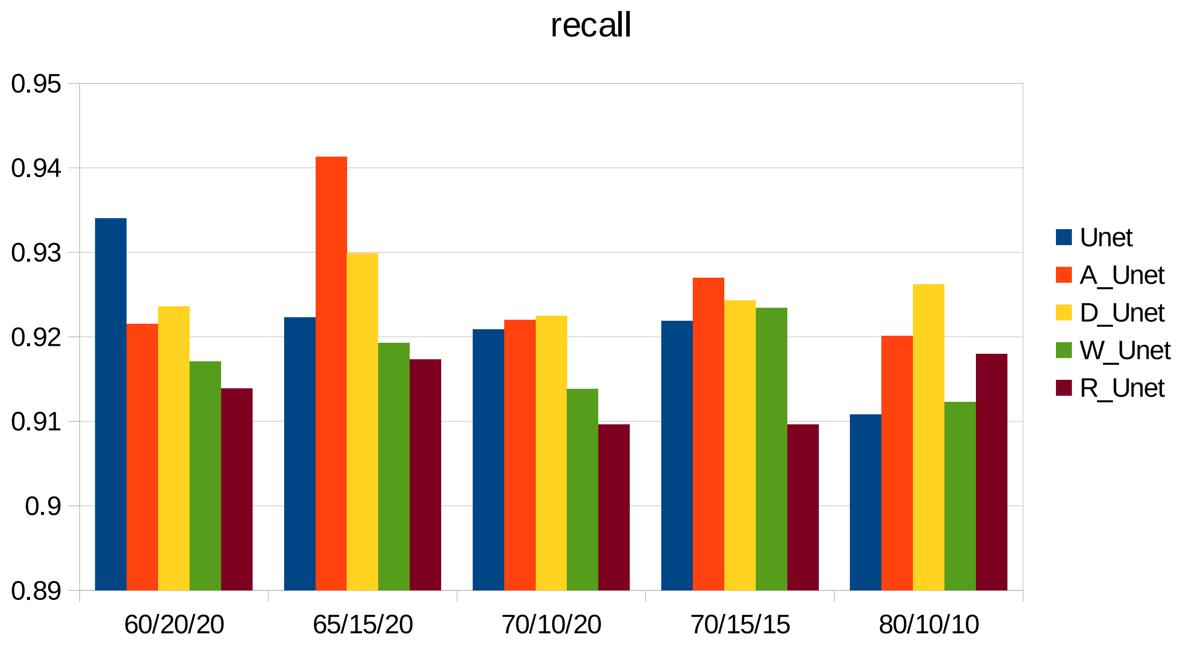

Figure 4: Comparison of the Recall on the training set for the different dataset ratios

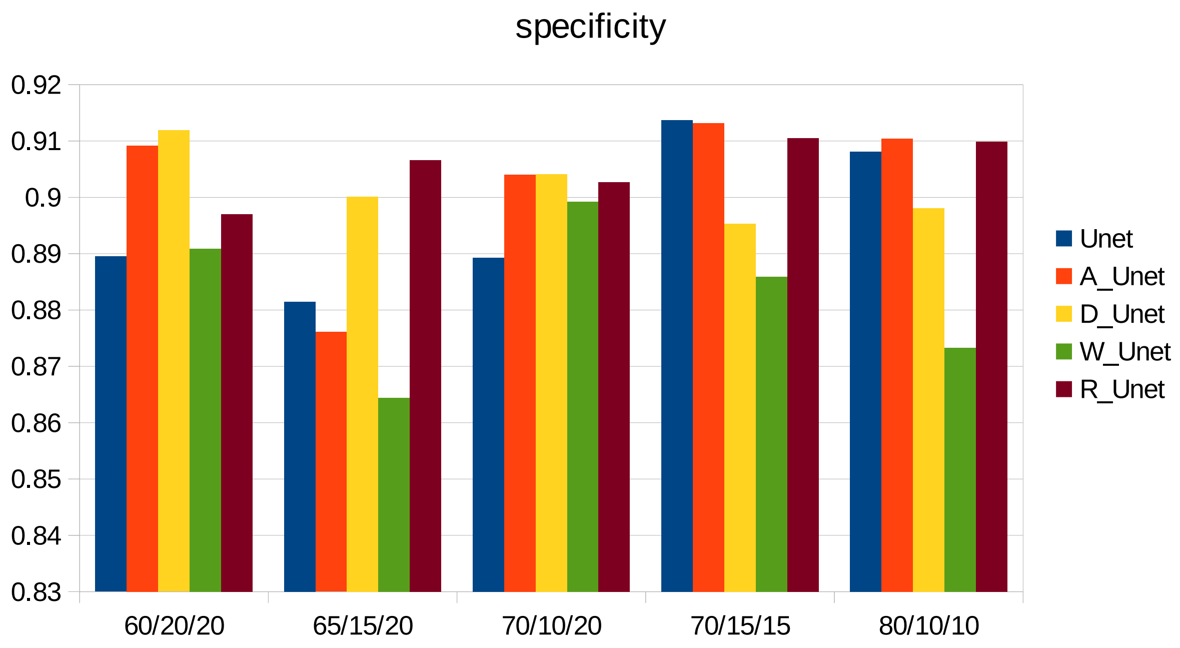

Figure 5: Comparison of the Specificity on the training set for the different dataset ratios

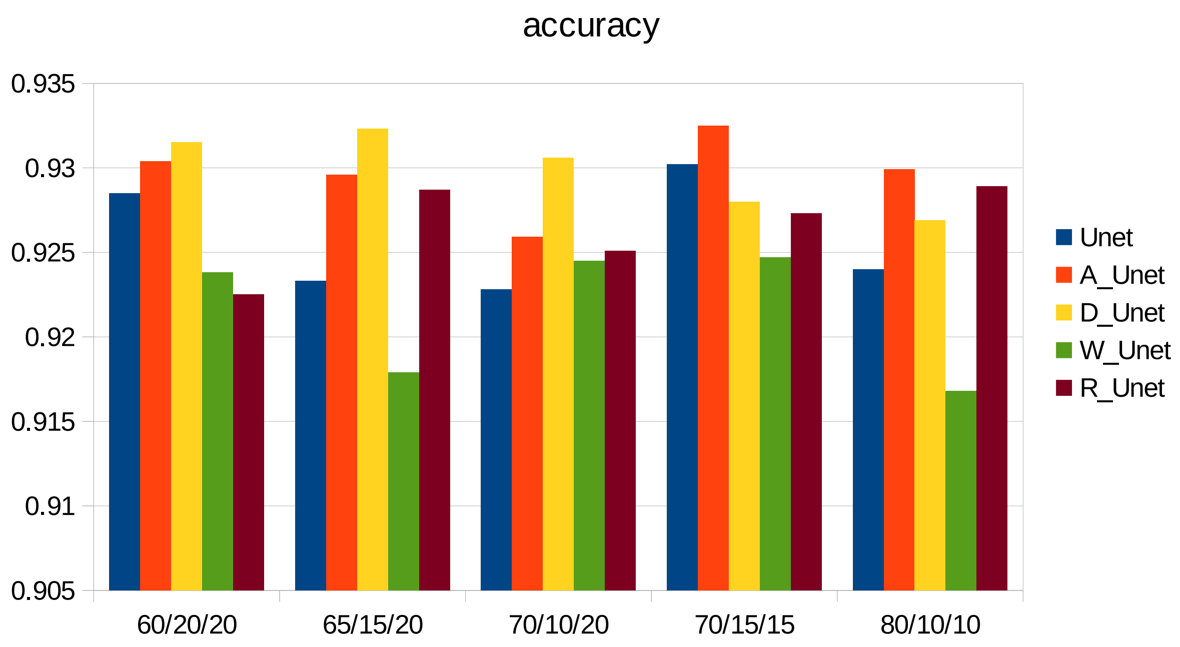

Figure 6: Comparison of the Accuracy on the training set for the different dataset ratios

4.1.2 Validation Set

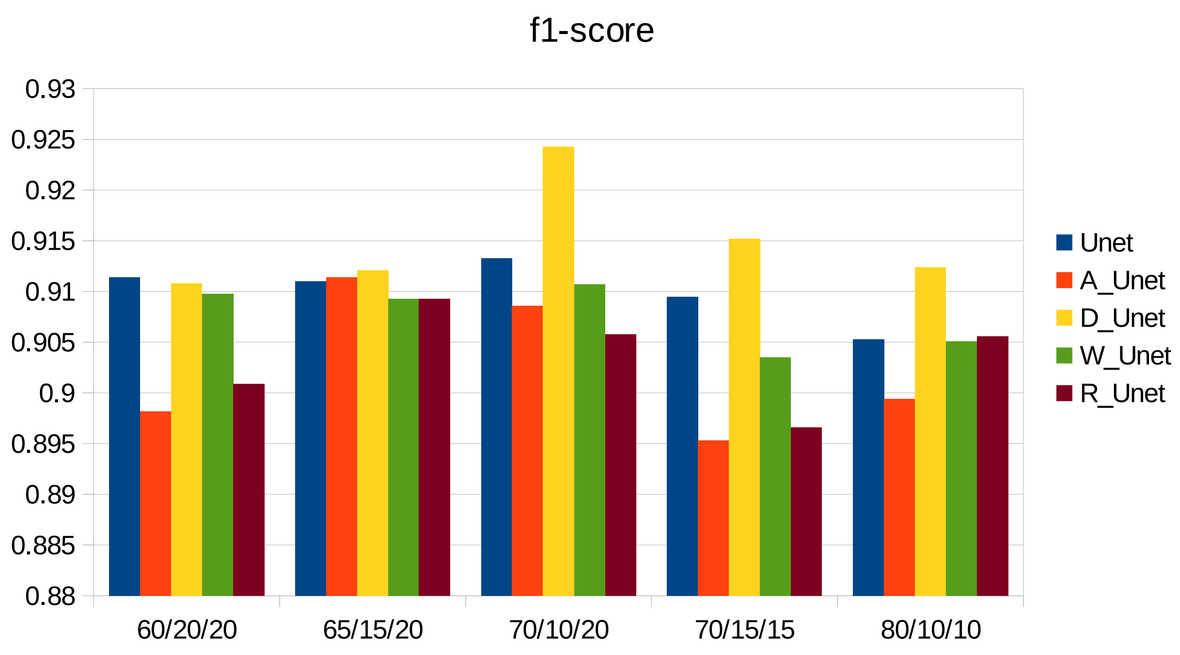

Figure 7: Comparison of the F1 score on the validation set for the different dataset ratios

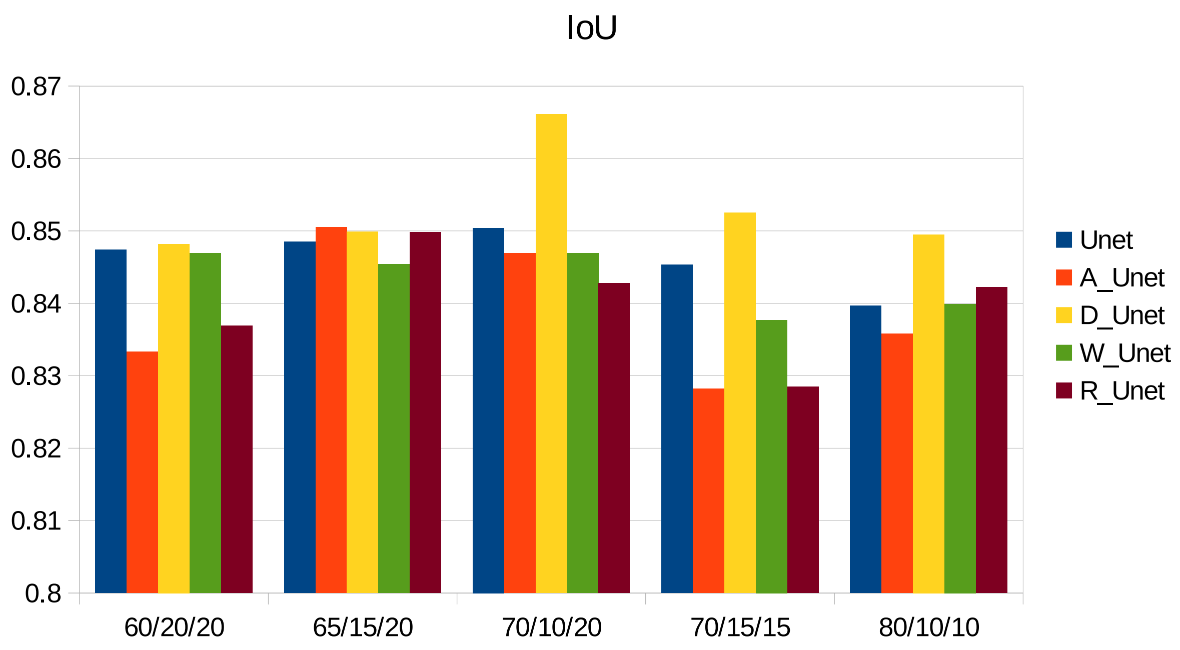

Figure 8: Comparison of the IoU on the validation set for the different dataset ratios

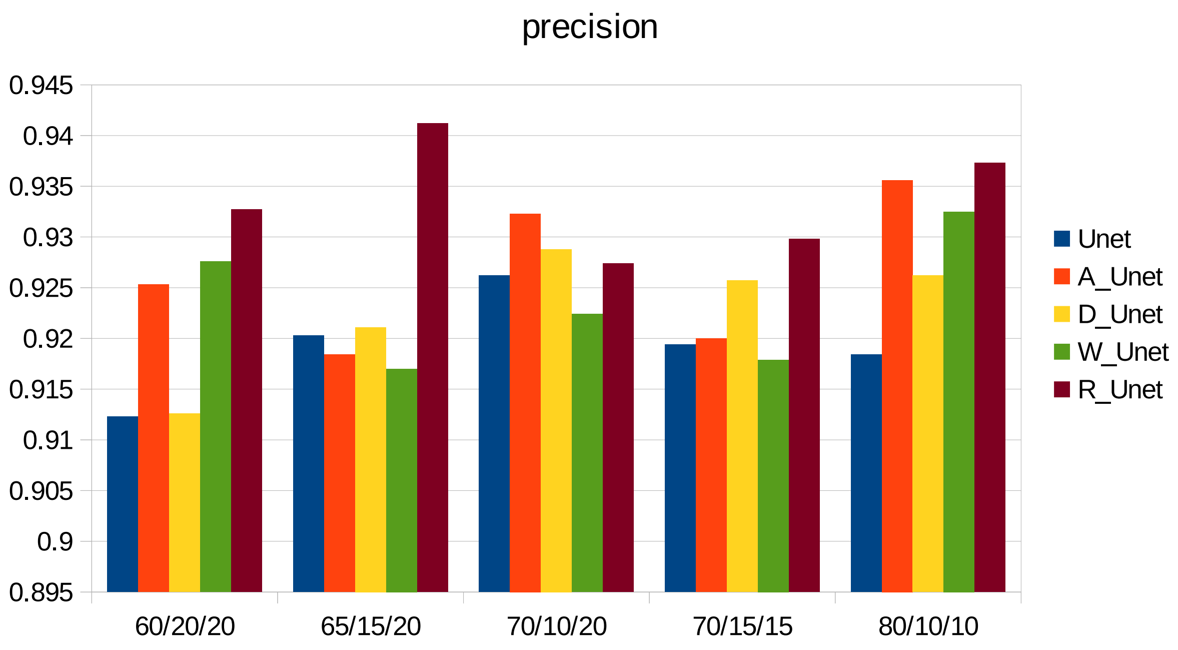

Figure 9: Comparison of the Precision on the validation set for the different dataset ratios

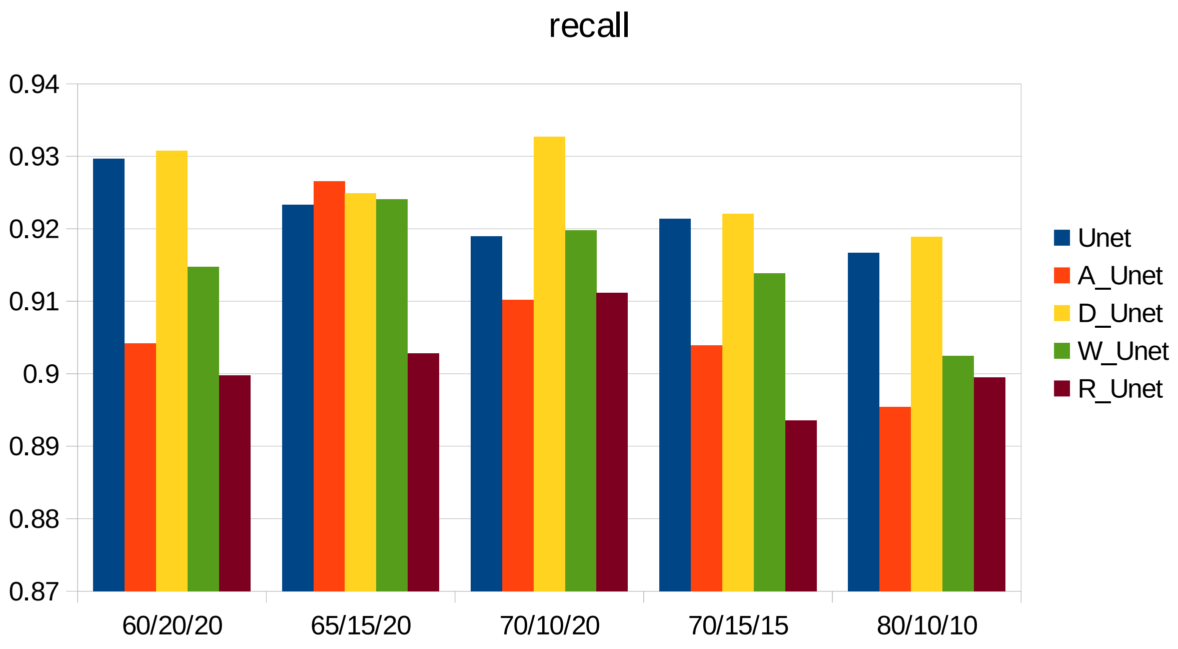

Figure 10: Comparison of the Recall on the validation set for the different dataset ratios

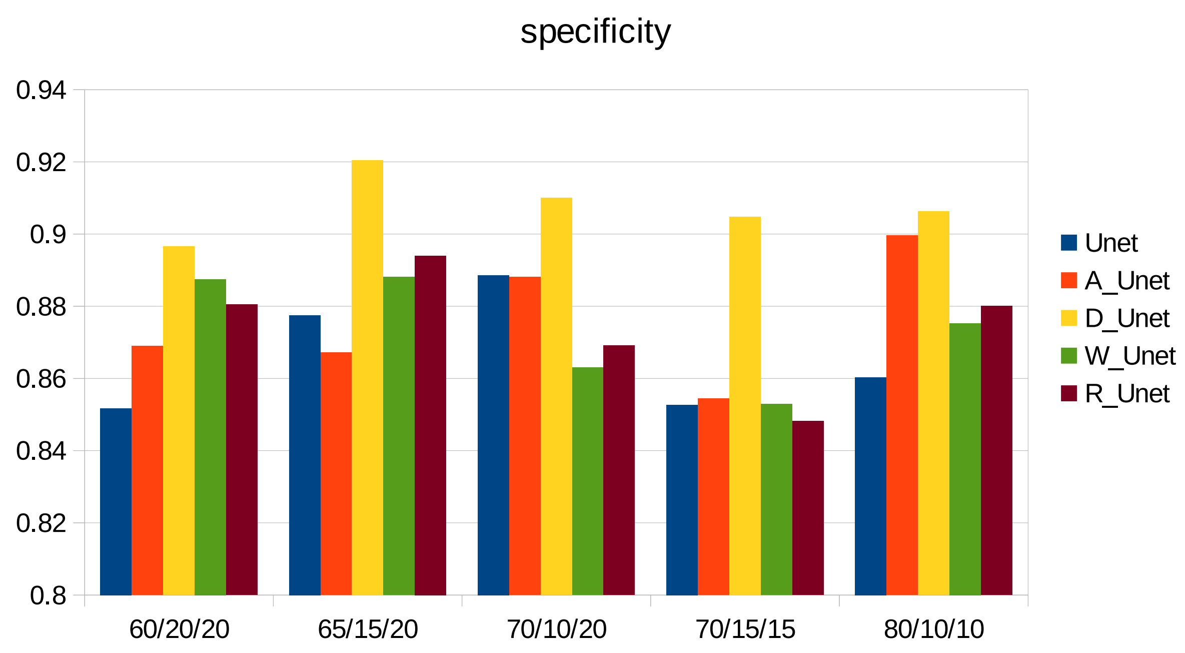

Figure 11: Comparison of the Specificity on the validation set for the different dataset ratios

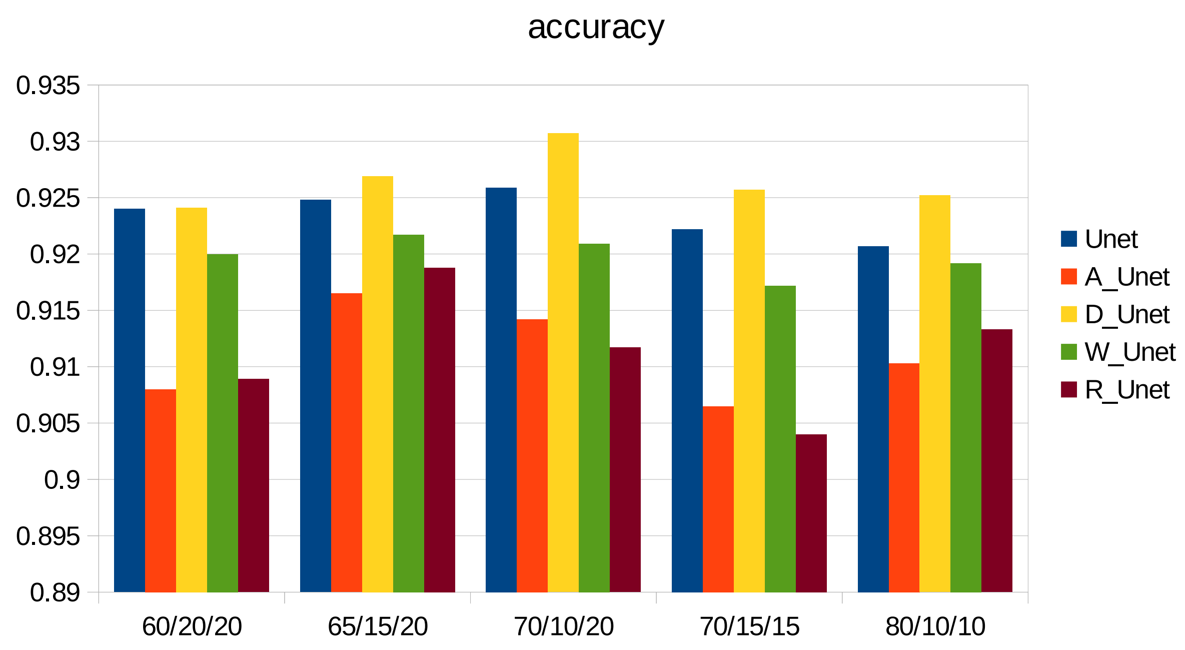

Figure 12: Comparison of the Accuracy on the validation set for the different dataset ratios

4.1.3 Test Set

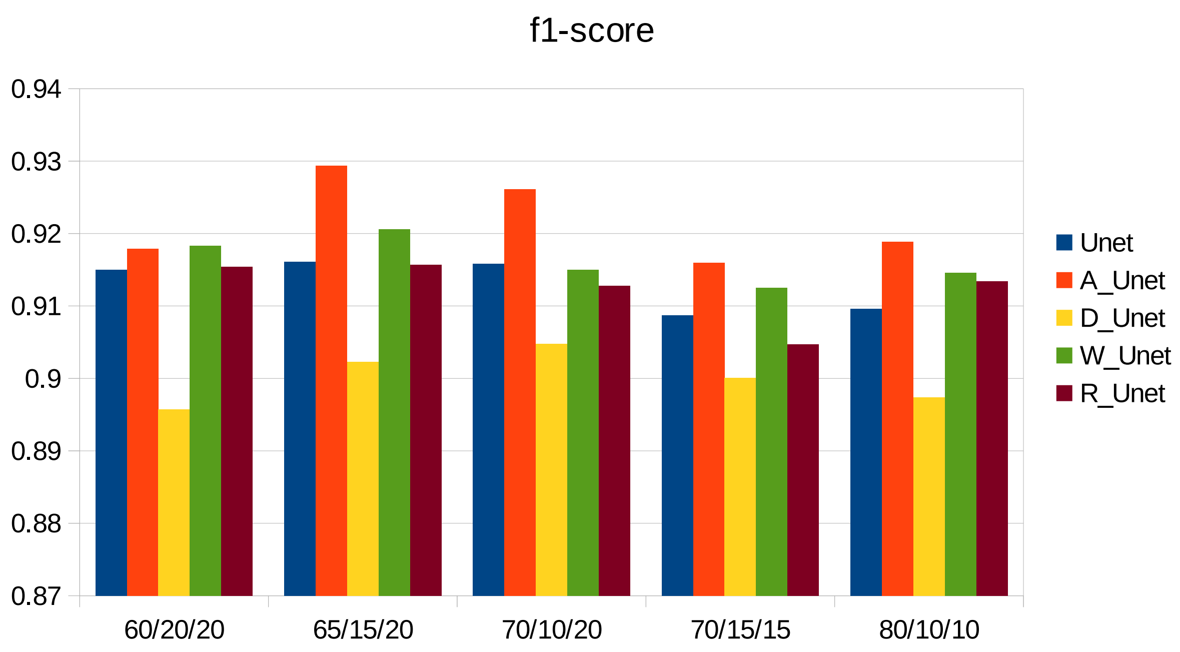

Figure 13: Comparison of the F1 score on the test set for the different dataset ratios

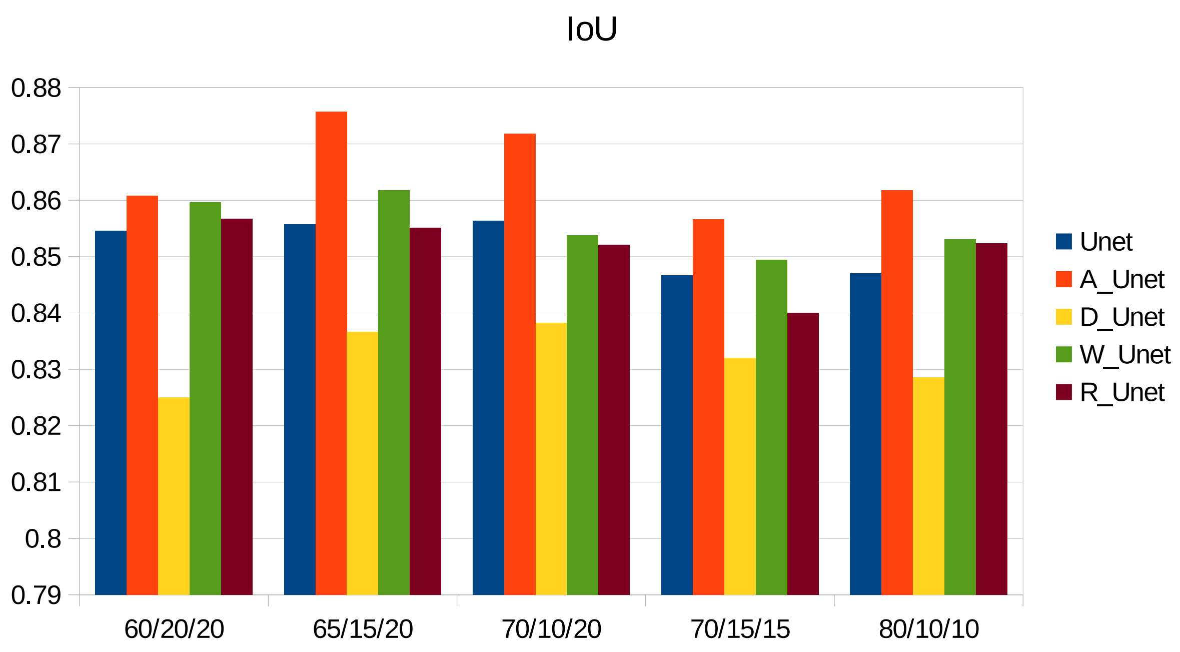

Figure 14: Comparison of the IoU on the test set for the different dataset ratios

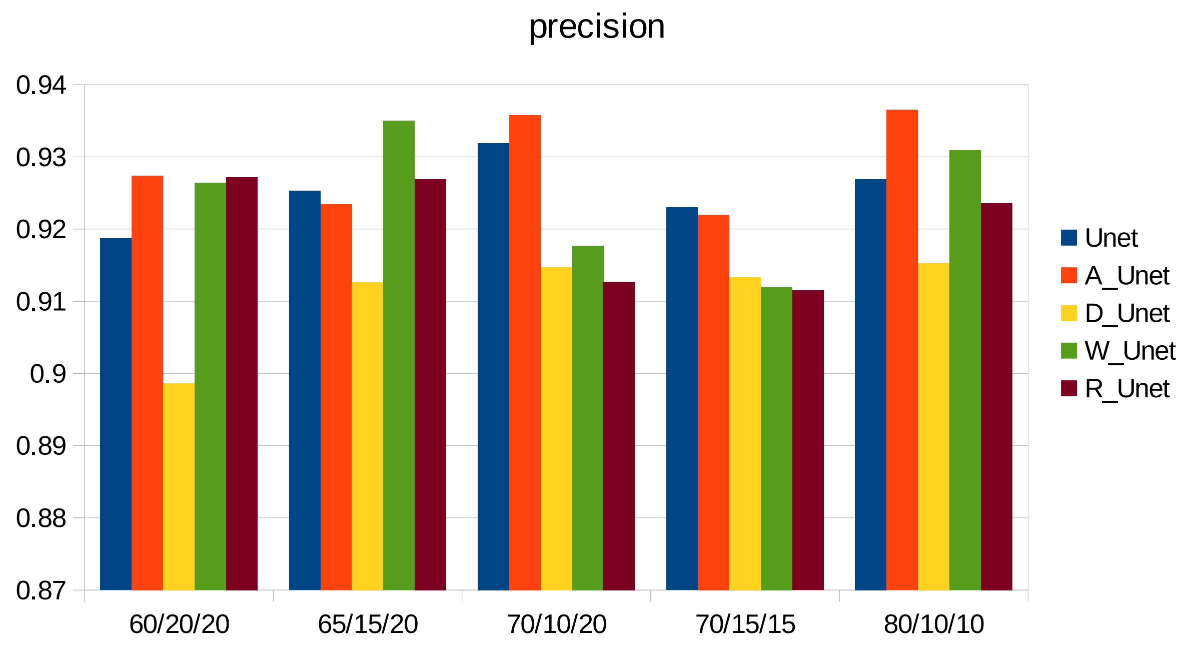

Figure 15: Comparison of the Precision on the test set for the different dataset ratios

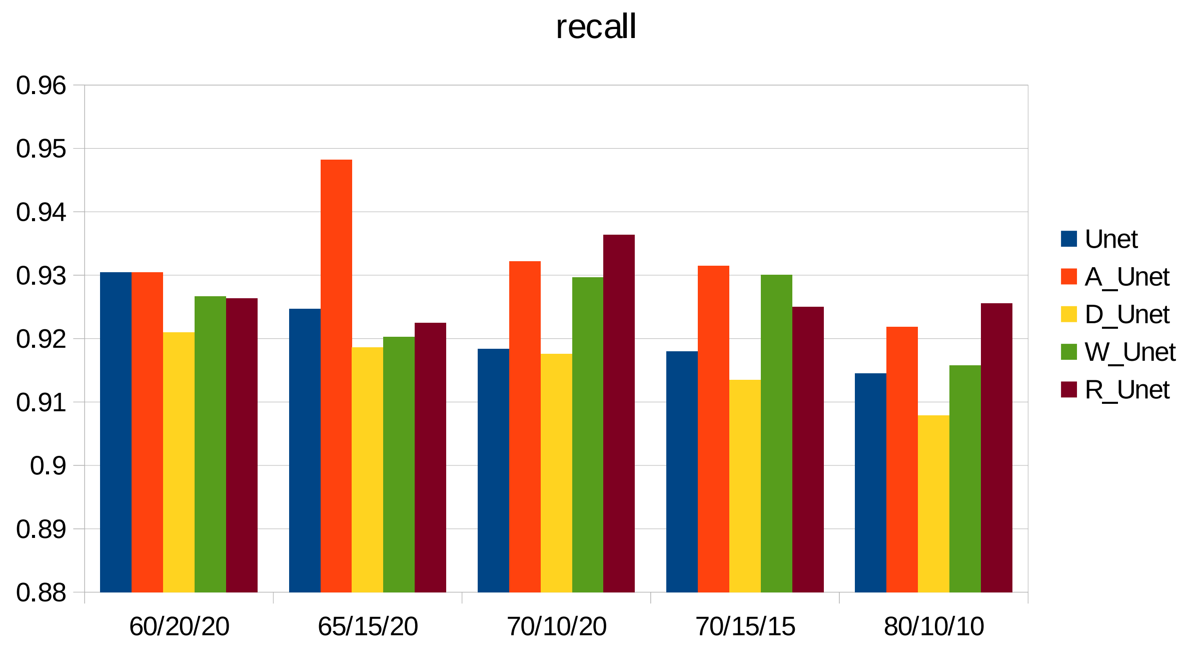

Figure 16: Comparison of the Recall on the test set for the different dataset ratios

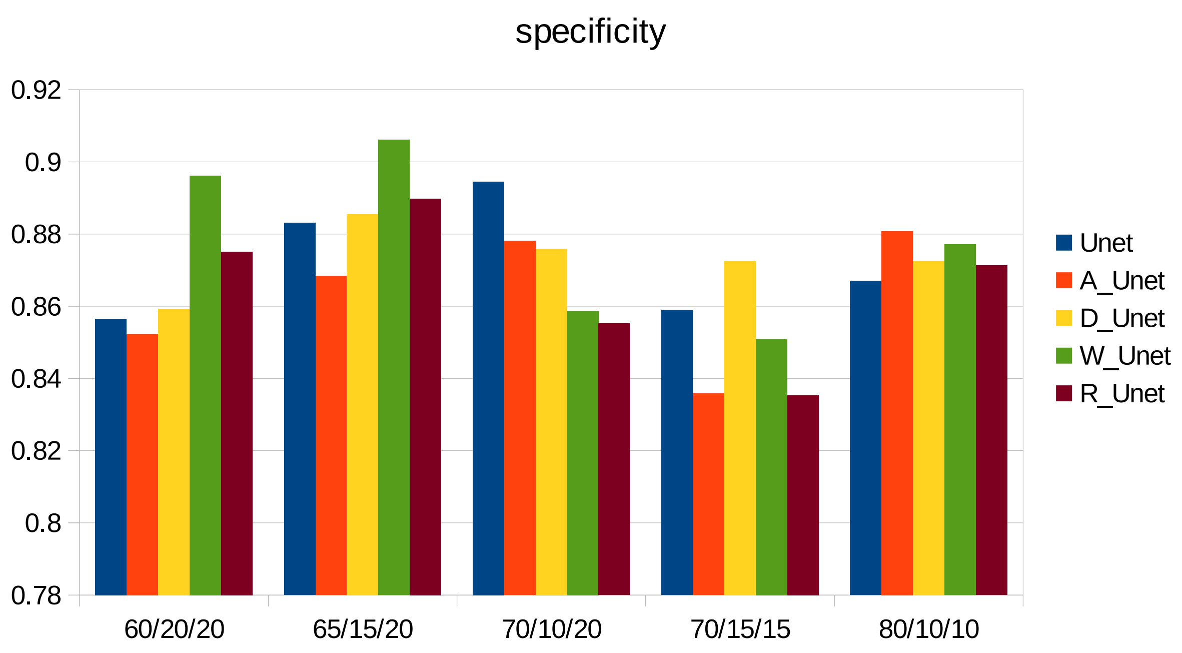

Figure 17: Comparison of the Specificity on the test set for the different dataset ratios

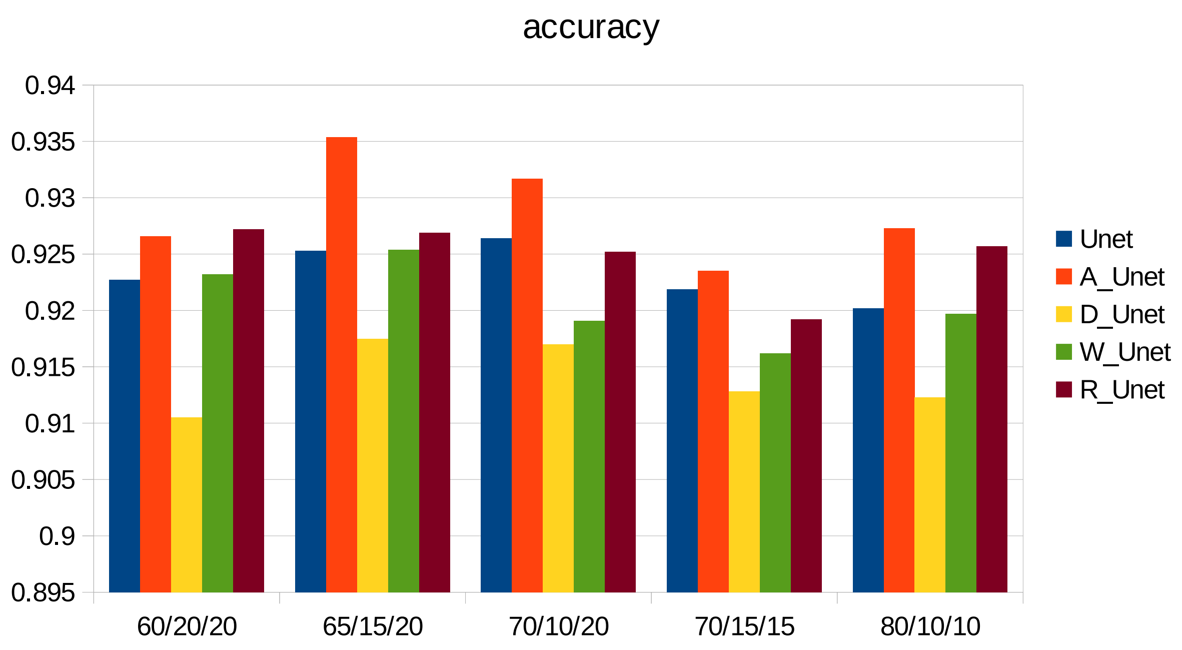

Figure 18: Comparison of the Accuracy on the test set for the different dataset ratios

From the above charts it can be vaguely concluded that the best ratios are 65/15/20 and 70/10/20, because they yield the best overall results both in the validation set and the test set. However, this is not absolute because of the fact that the compared ratios do not include the same images in their respective sets. As a result some images included only in certain training sets may contain information that is more rare than in other images. This can be observed in the 80/10/10 dataset which does not yield any significant performance in any set, although it was expected to have higher IoU and f1 scores on the validation set, due to the fact that the training set contains more images than all the other datasets.

4.2 Optimization algorithms

In this experiment 16 different optimization algorithms, which are all available on the tensorflow[13] and the tensorflow addons[14] libraries, have been tested. Specifically, for this test only the A_Unet was utilized because its overall accuracy was of the highest ones and its training time was of the shortest ones. The network was trained for 31 epochs on the 60/20/20 dataset with the dice loss function and a scheduled learning rate. Again, the value of the learning rate decreased from 0.001 to 0.0005 after 10 epochs, to 0.0001 after 20 epochs and to 0.00005 after 30 epochs. Finally the images were resized to 300×300 pixels.

The results of the experiment are presented below in only one batch of six charts, corresponding to the six metrics. The training, validation and test set scores of the same metric for the respective optimization algorithm are all included in the same chart.

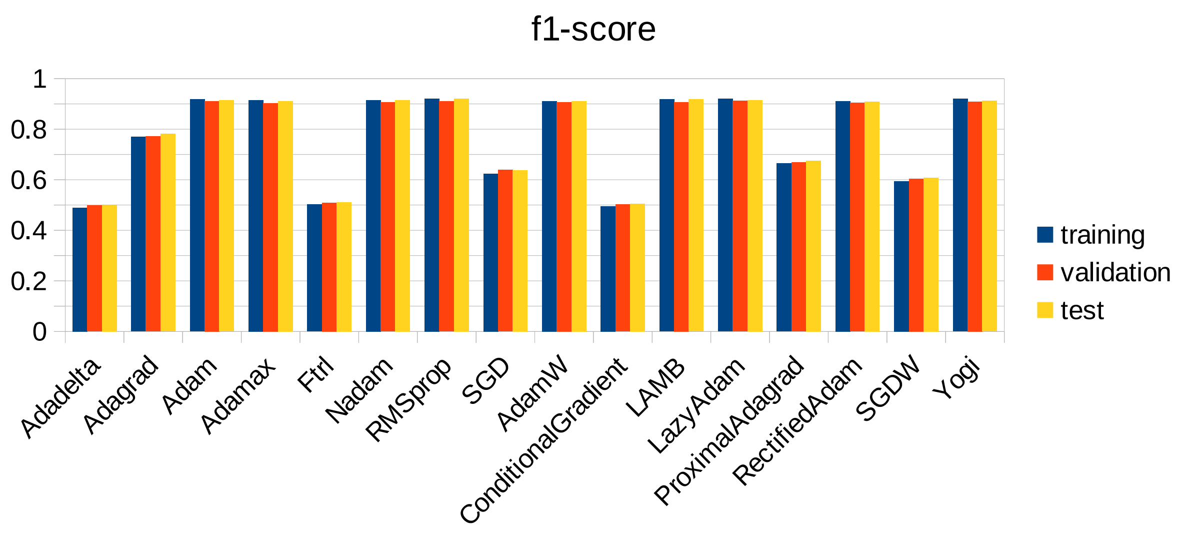

Figure 19: Comparison of the F1 score on the training/validation/test sets for the different optmizers

13 https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
14 https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers

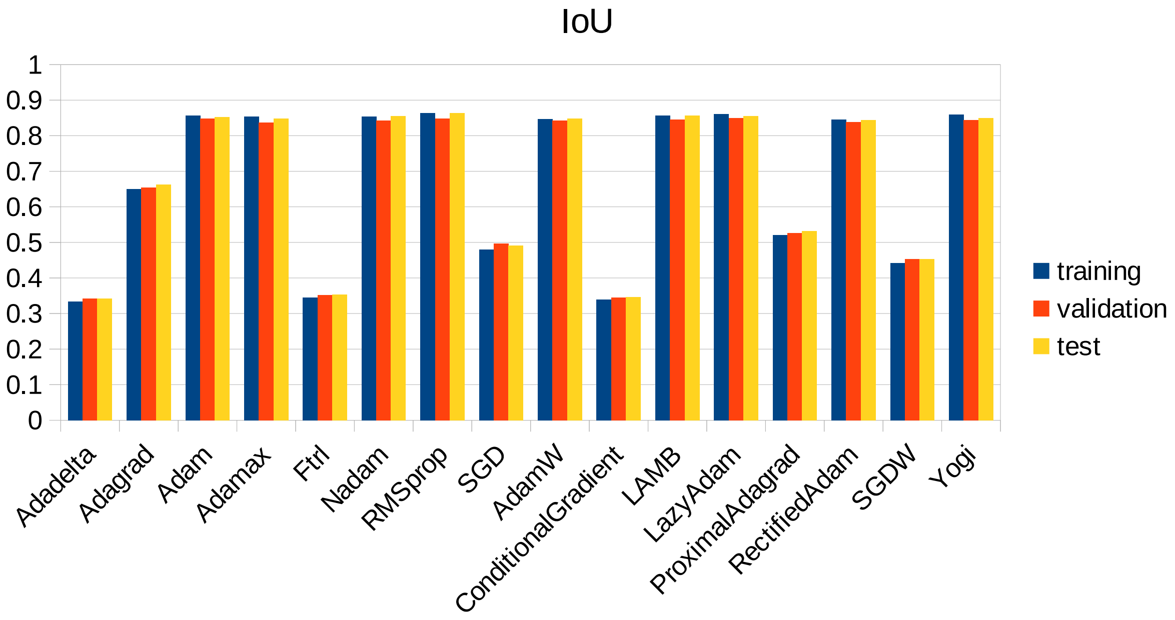

Figure 20: Comparison of the IoU on the training/validation/test sets for the different optmizers

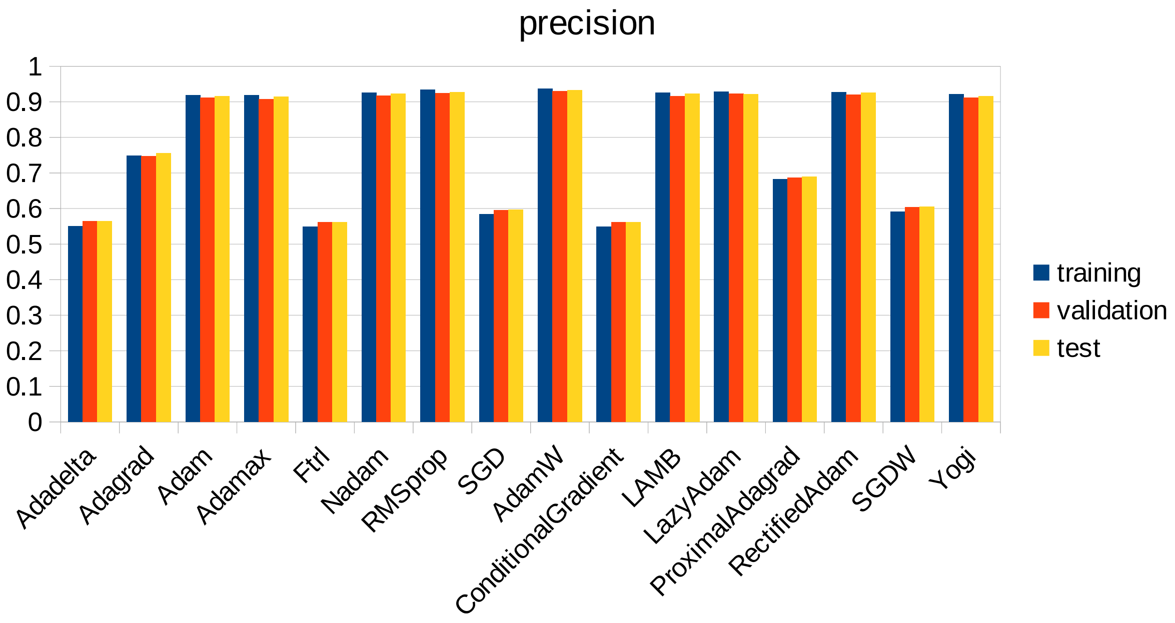

Figure 21: Comparison of the Precision on the training/validation/test sets for the different optmizers

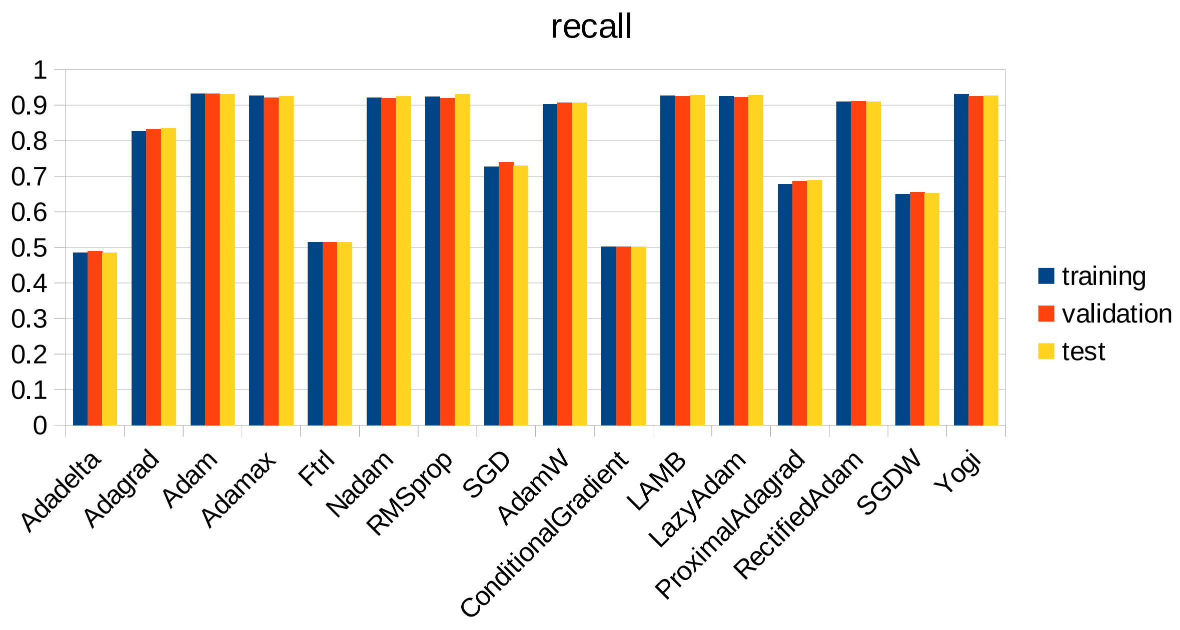

Figure 22: Comparison of the Recall on the training/validation/test sets for the different optmizers

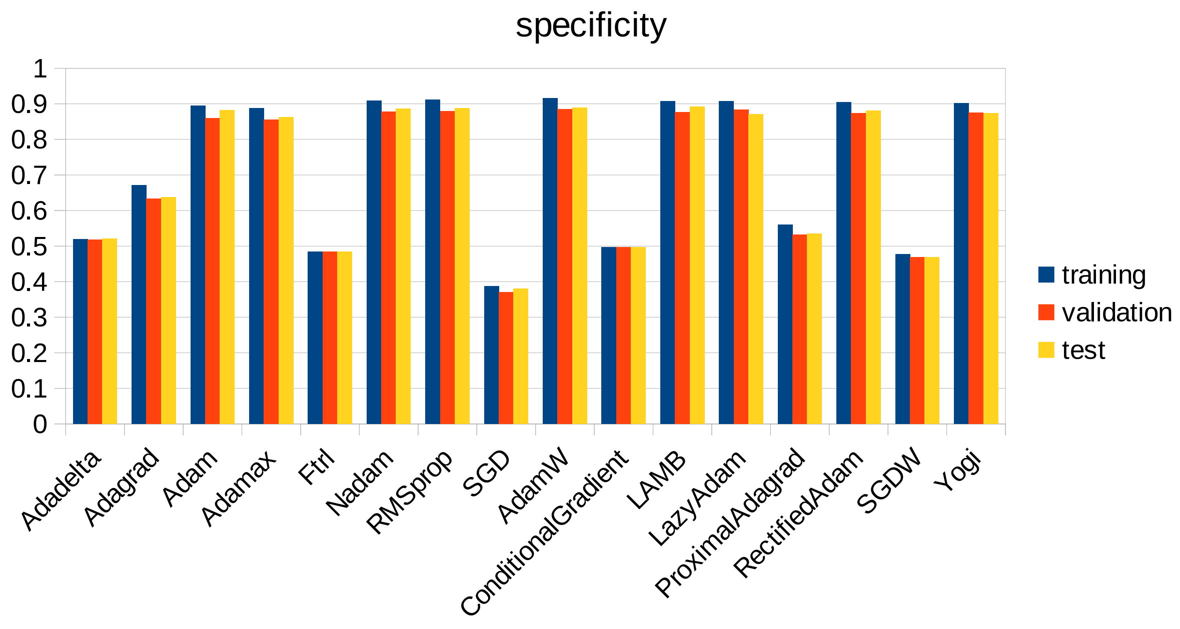

Figure 23: Comparison of the Specificity on the training/validation/test sets for the different optmizers

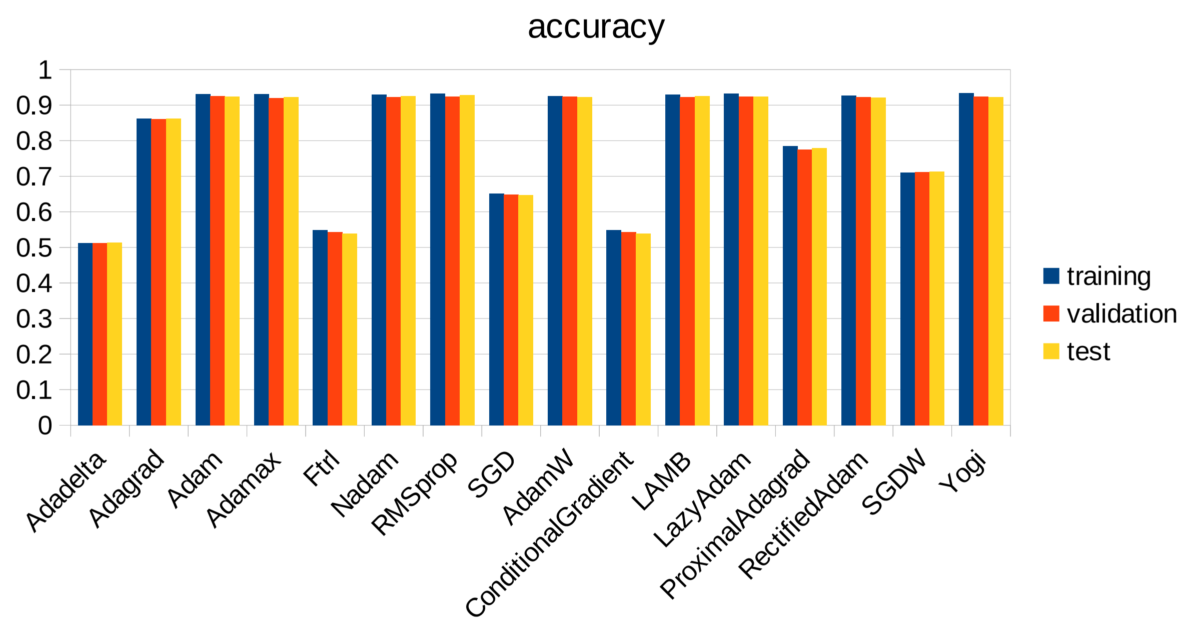

Figure 24: Comparison of the Accuracy on the training/validation/test sets for the different optmizers

From the above charts it can be concluded that the most fitted optimizers for the task of cloud segmentation on small networks like the Unet variations implemented in this thesis are: Adam, Adamax, Nadam, RMSprop, AdamW, LAMB, LazyAdam, RectifiedAdam and Yogi.

4.3 Loss Functions

In this experiment 12 different loss functions have been compared regarding their efficiency for cloud segmentation tasks. In more detail, for this test the A_Unet was chosen again to be trained for 31 epochs, on the 60/20/20 dataset, with the LAMB optimizer and a scheduled learning rate for each loss function. The implemented schedule had the value of learning rate decreased after the first 10 epochs from the initial value of 0.001 to 0.0005, after 20 epochs to 0.0001 and after 30 epochs to 0.00005. Finally the images were resized to 300×300 pixels.

The results of the experiment are presented below in only one batch of six charts, corresponding to the six metrics. The training, validation and test set scores of the same metric for the respective loss function are all included in the same chart.

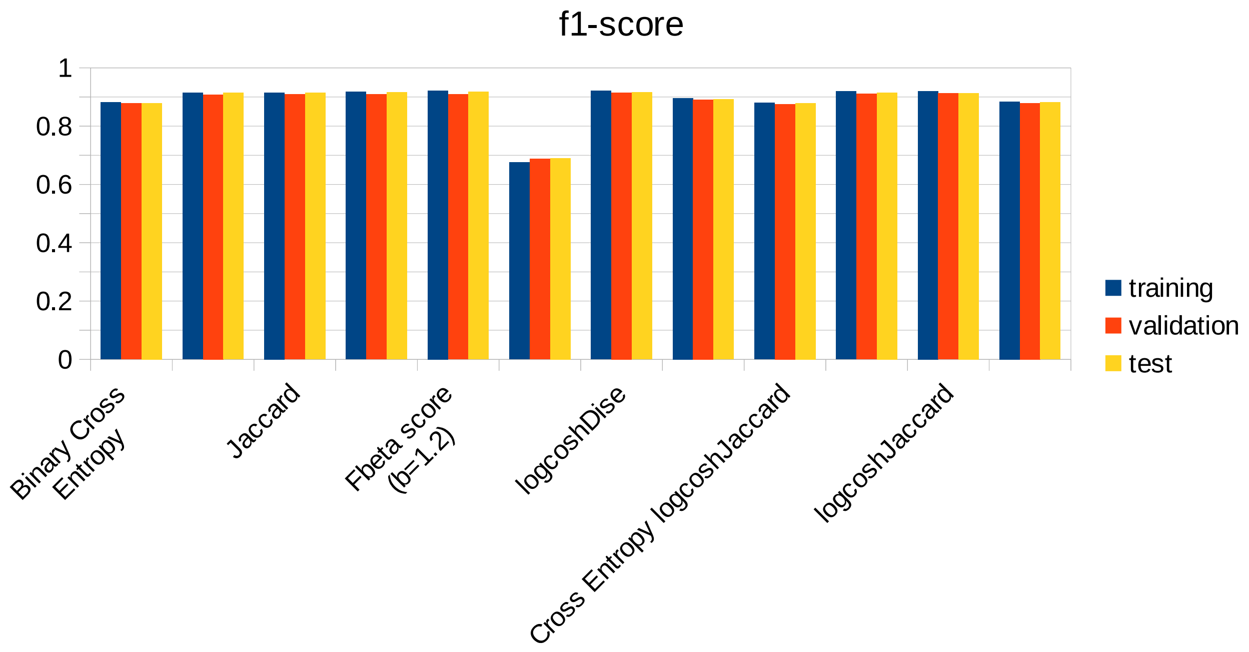

Figure 25: Comparison of the F1 score on the training/validation/test sets for the different loss functions

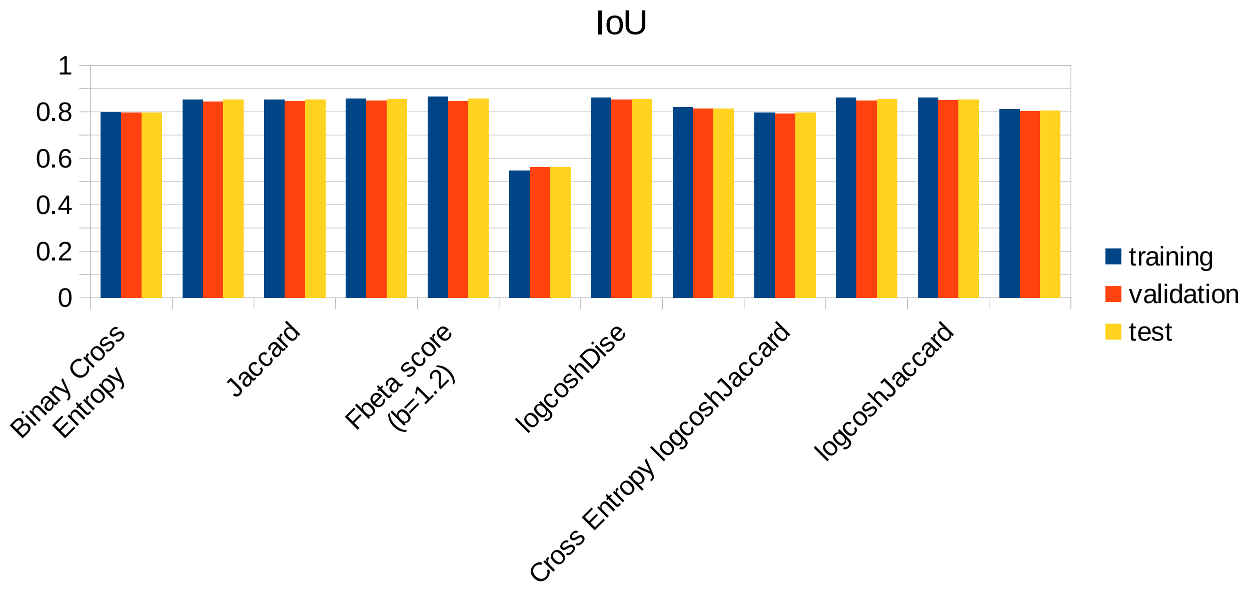

Figure 26: Comparison of the IoU on the training/validation/test sets for the different loss functions

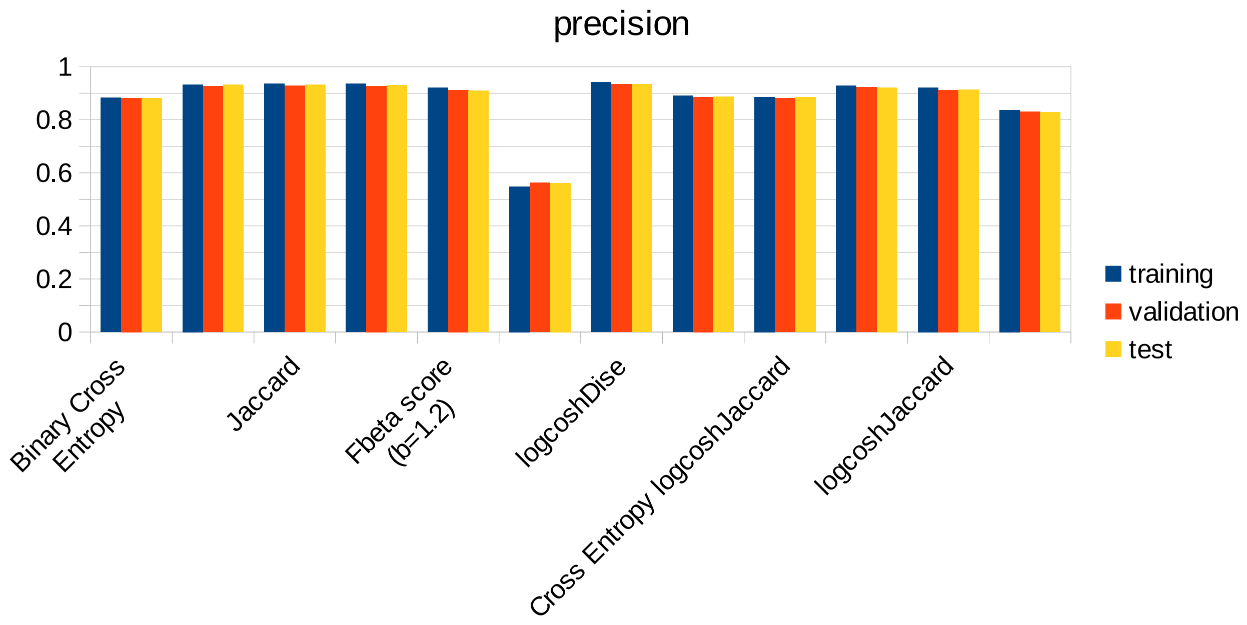

Figure 27: Comparison of the Precision on the training/validation/test sets for the different loss functions

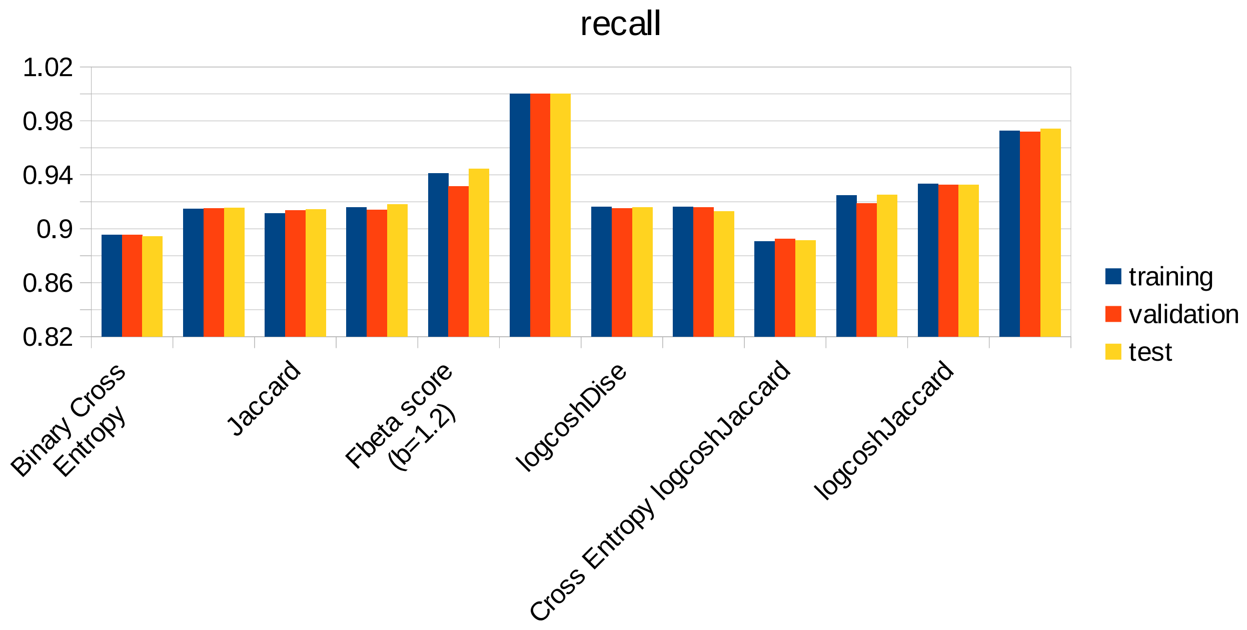

Figure 28: Comparison of the Recall on the training/validation/test sets for the different loss functions

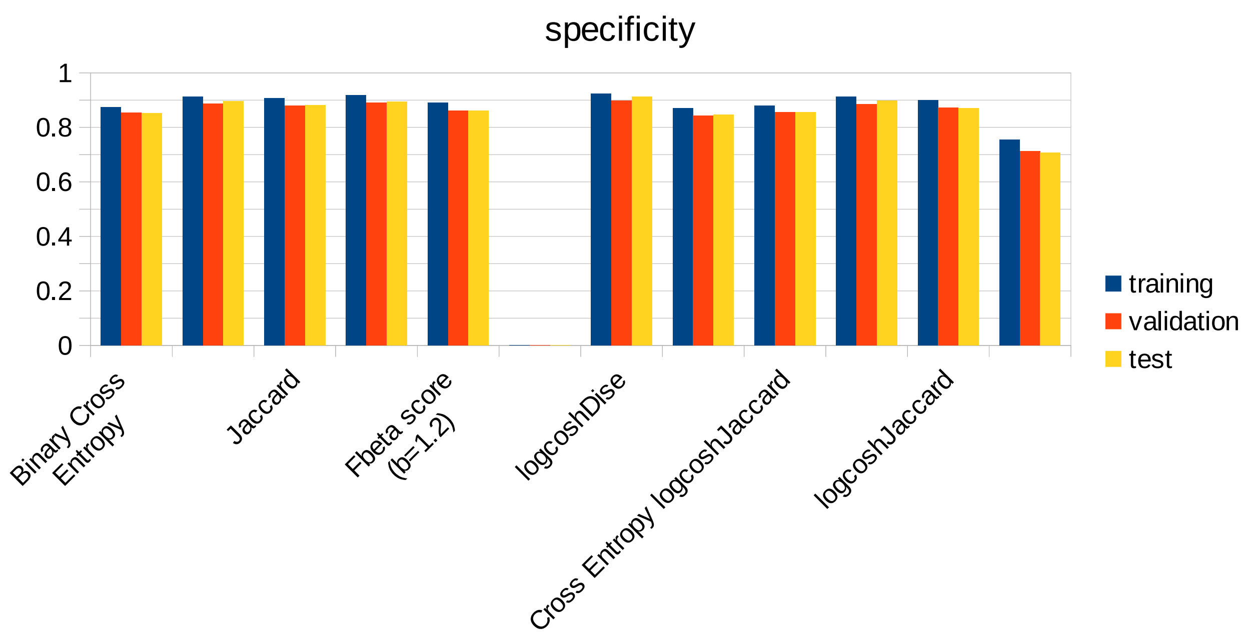

Figure 29: Comparison of the Specificity on the training/validation/test sets for the different loss functions

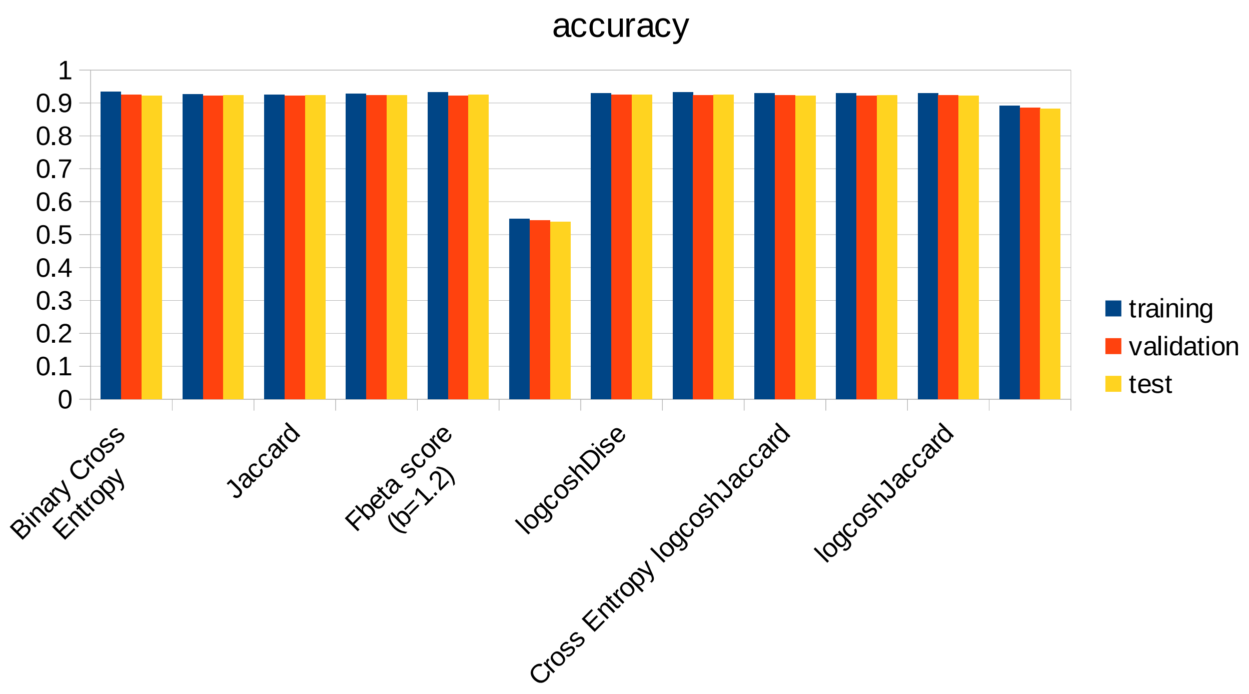

Figure 30: Comparison of the Accuracy on the training/validation/test sets for the different loss functions

From the above charts it can be inferred that the most suitable loss functions for cloud segmentation are: Dice, Jaccard, power Jaccard, Fbeta score, Fbeta score power Jaccard, LogCoshJaccard and LogCoshDice. Interestingly, the LogCoshDice had the best overall metric scores.

4.4 ImageNets

This is the final experiment where 50 networks of different sizes and performance on the ImageNet dataset have been compared, regarding their efficiency for cloud segmentation tasks. Bacause the networks are of various depth, the LAMB optimization algorithm has been employed for their training, in order to achieve an accelerated learning process on all of them. Additionally, they were trained utilizing the LogCoshDice loss function for 31 epochs, on the 60/20/20 dataset with a scheduled learning rate. The initial value of the learning rate was 0.001 and it decreased to 0.0005 after 10 epochs, to 0.0001 after 20 epochs and to 0.00005 after 30 epochs. Finally, the images were resized to 224×224 pixels, which is the default size for most of the networks.

Furthermore, the networks were all trained on four different conditions to determine the most preferable. Specifically, the types of training have been codenamed as A, B, C and D:

- **A**: The inputs have been preprocessed according to its network's specific kind of preprocessing operation. The networks' encoder was pretrained on the ImageNet dataset. Finally, during the training phase the networks' encoder was frozen, meaning that its weights did note get updated.
- **B**: The inputs have not been preprocessed in accordance to its networks' preprocessing operation. The networks' encoder was pretrained on the ImageNet dataset. Morever, during the training phase the networks' encoder was allowed to be trained and have its weights updated.
- **C**: The inputs have not been preprocessed according to its networks' preprocessing operation. The networks' encoder was pretrained on the ImageNet dataset. Finally, during the training phase the networks' encoder was frozen, meaning that its weights did note get updated.
- **D**: The inputs have not been preprocessed in accordance to its networks' preprocessing operation. The networks' encoder was initiated with random values on its weights. Morever,

during the training phase the networks' encoder was allowed to be trained and have its weights updated.

The results of the experiment are presented below in eight batches of six charts, corresponding to each of the eight different architecture families.

4.4.1 VGG

4.4.1.1 Training Set

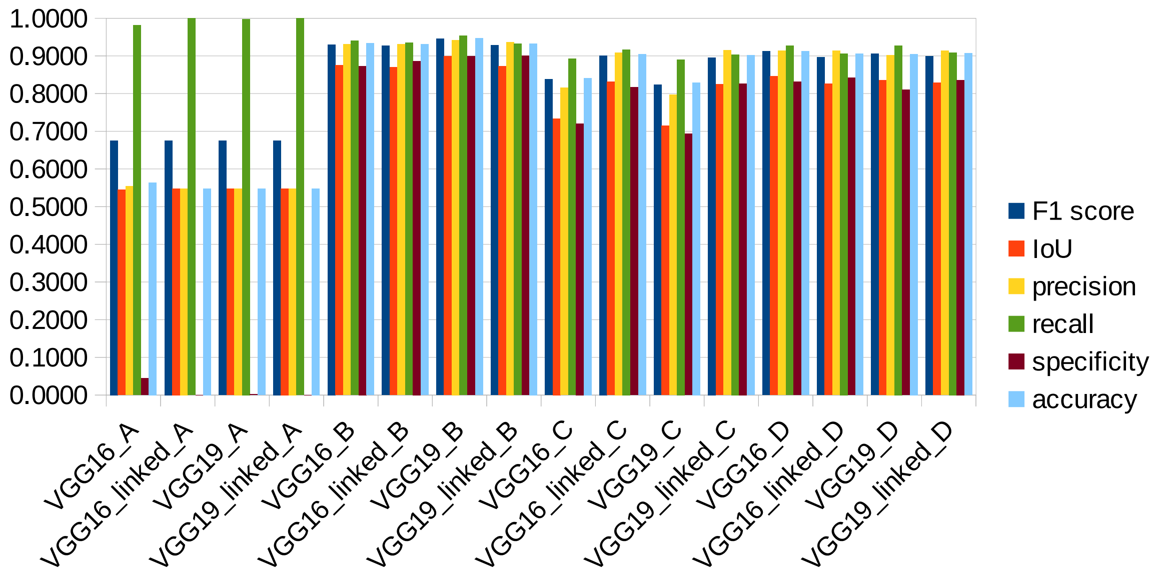

Figure 31: Comparison of the VGG architectures on the training set for all different conditions

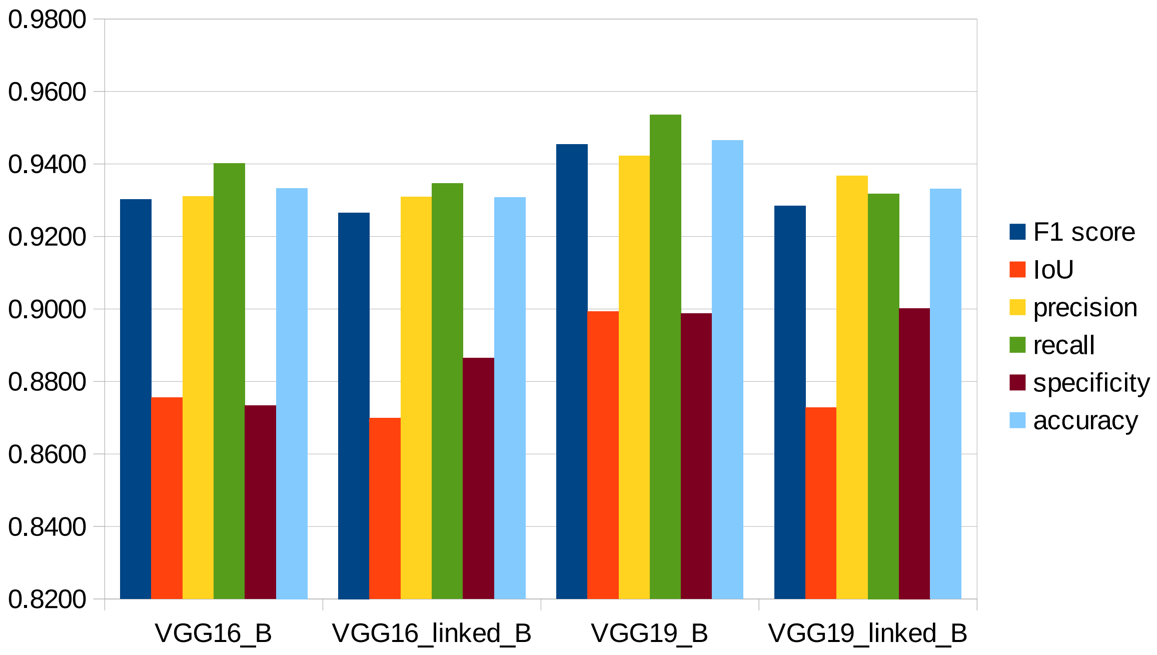

Figure 32: Comparison of the VGG architectures on the training set for the most favourable conditions

4.4.1.2 Validation Set

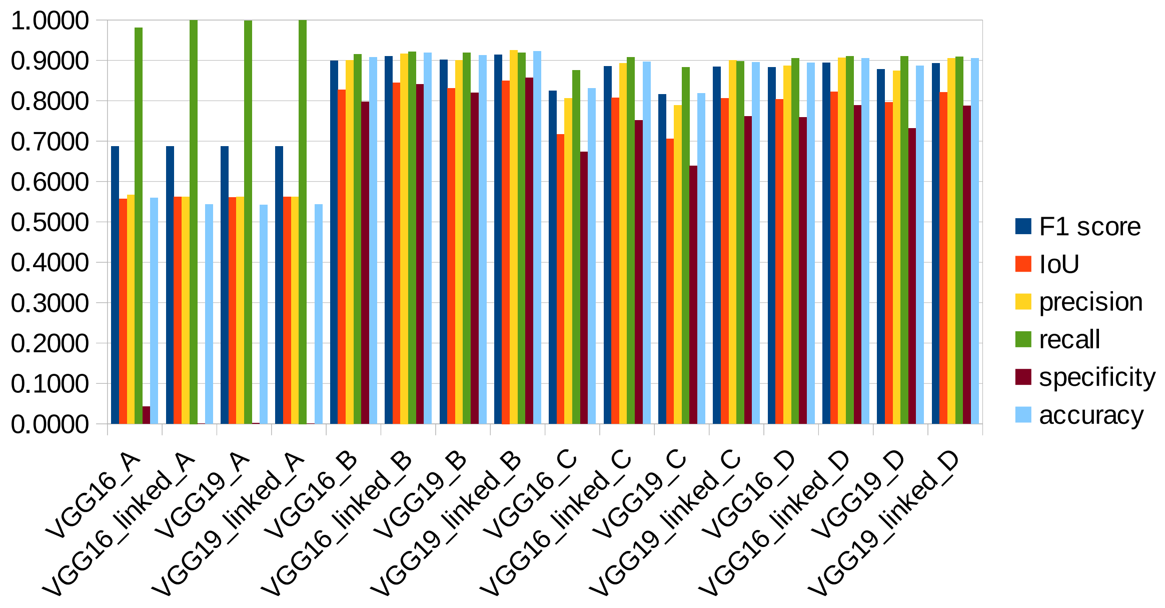

Figure 33: Comparison of the VGG architectures on the validation set for all different conditions

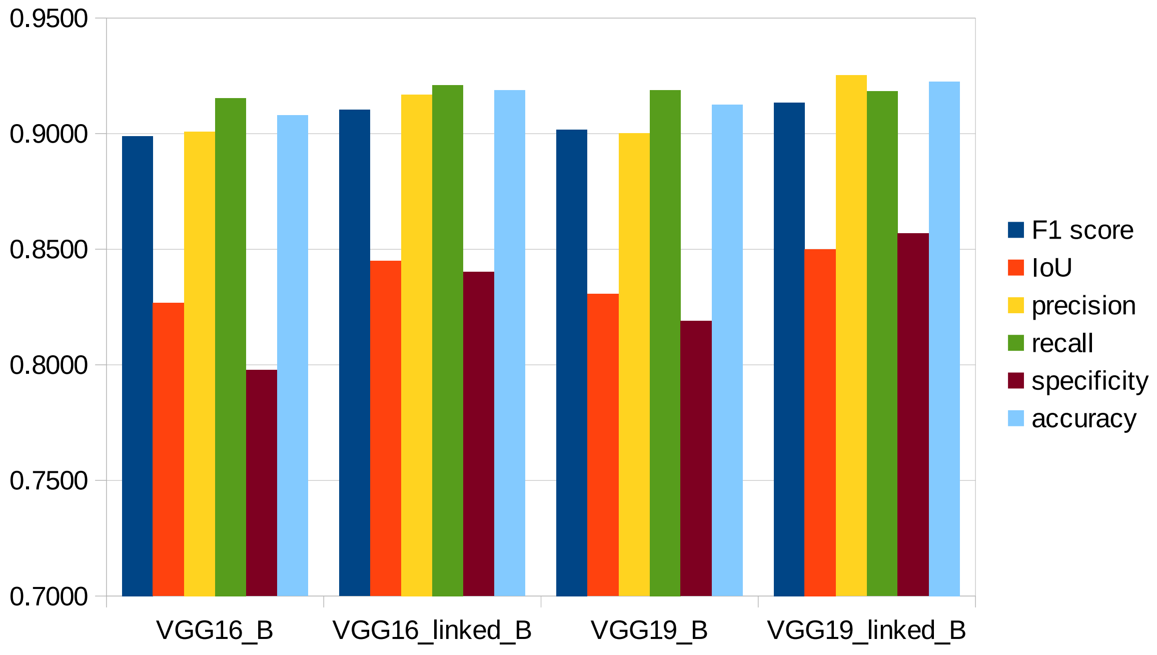

Figure 34: Comparison of the VGG architectures on the validation set for the most favourable conditions

4.4.1.3 Test Set

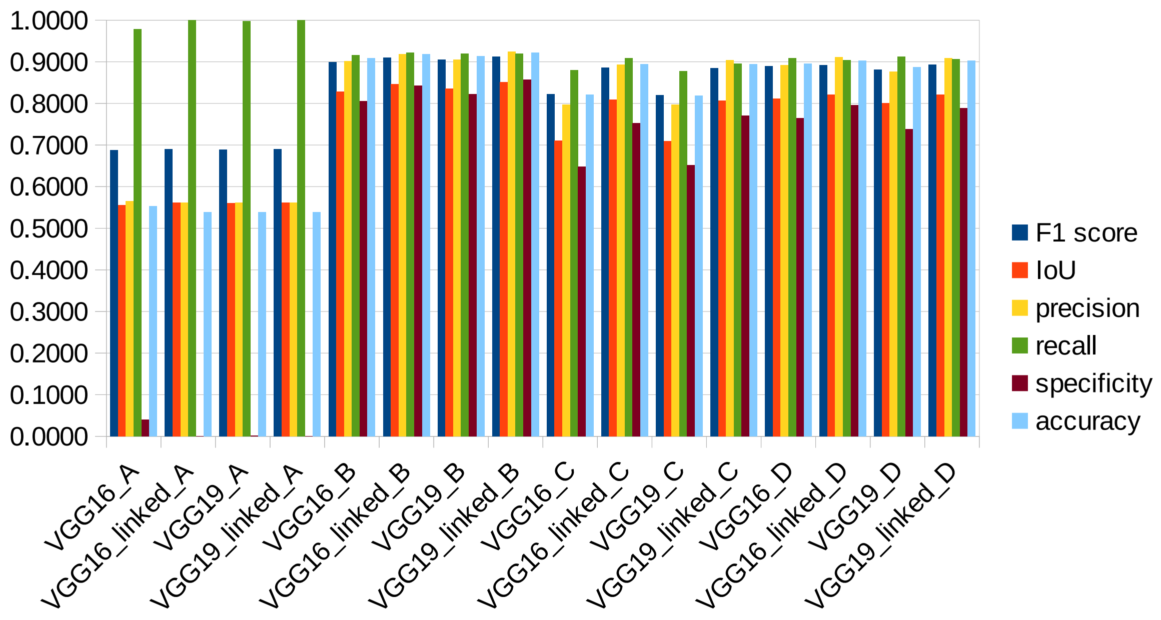

Figure 35: Comparison of the VGG architectures on the test set for all different conditions

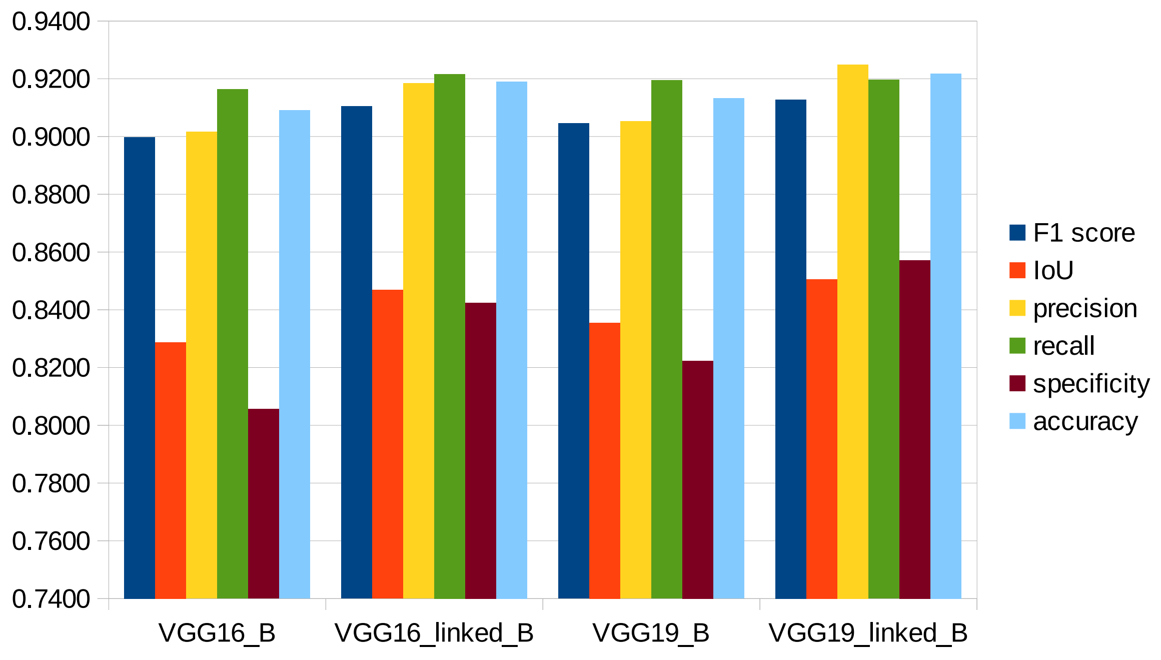

Figure 36: Comparison of the VGG architectures on the test set for the most favourable conditions

4.4.2 ResNet

4.4.2.1 Training Set

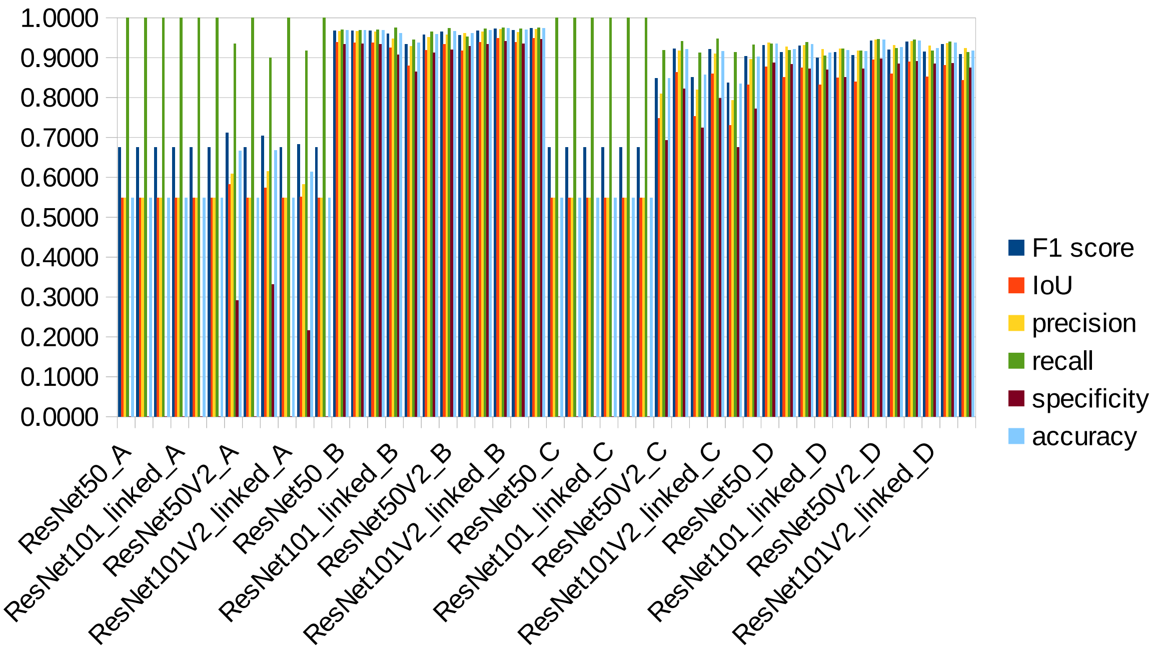

Figure 37: Comparison of the ResNet architectures on the training set for all different conditions

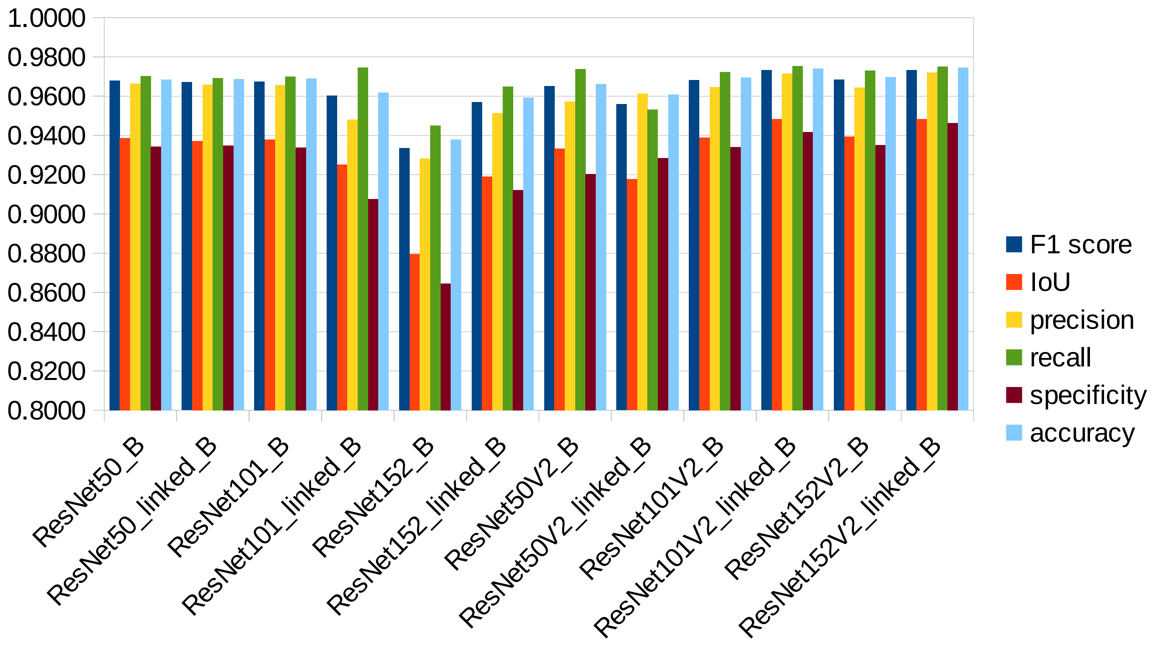

Figure 38: Comparison of the ResNet architectures on the training set for the most favourable conditions

4.4.2.2 Validation Set

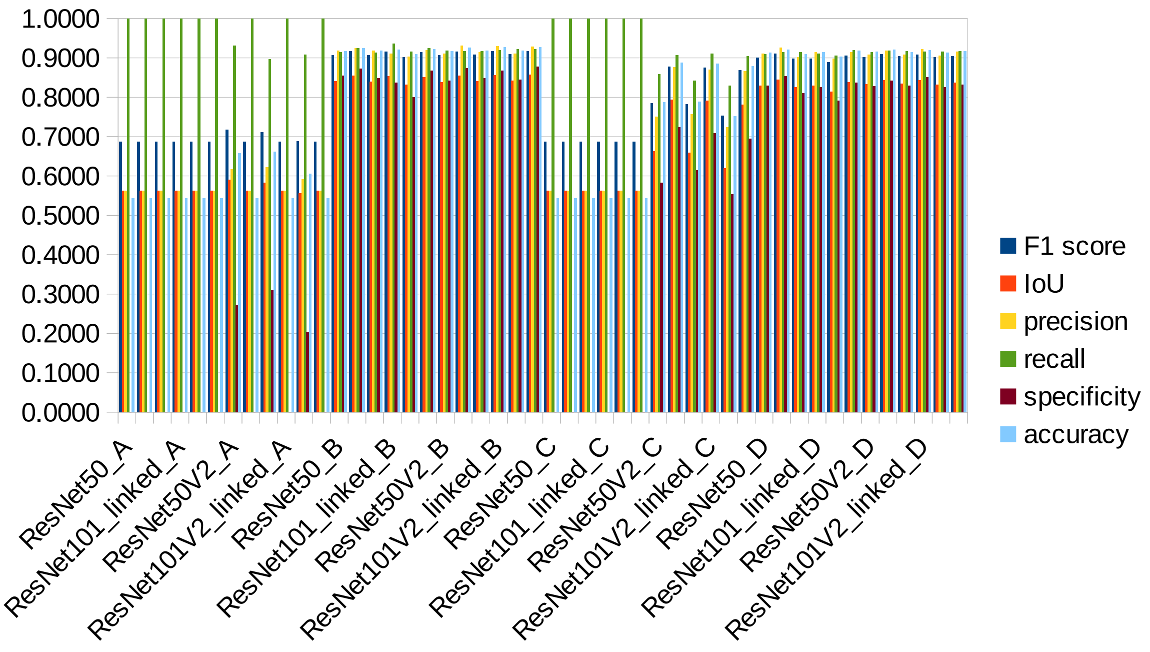

Figure 39: Comparison of the ResNet architectures on the validation set for all different conditions

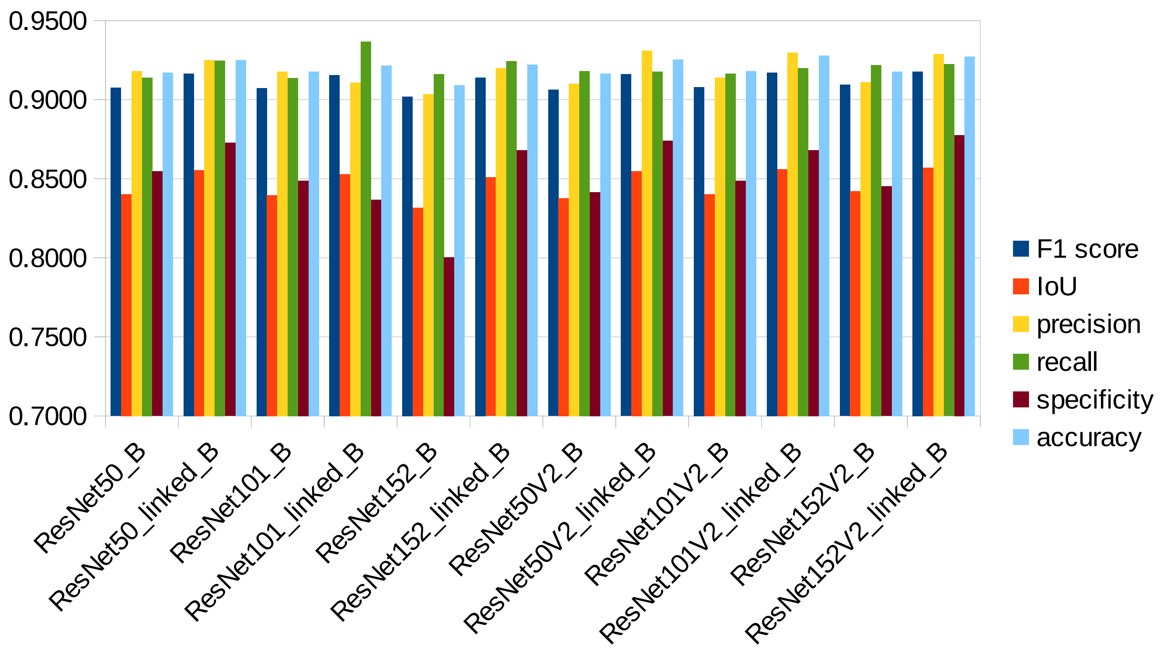

Figure 40: Comparison of the ResNet architectures on the validation set for the most favourable conditions

4.4.2.3 Test Set

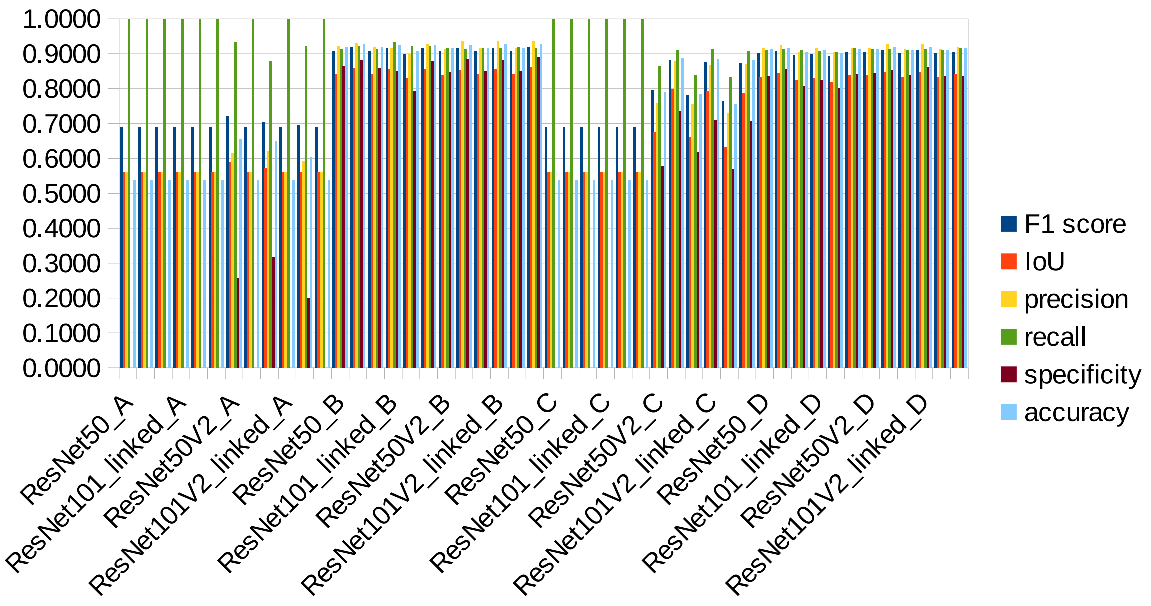

Figure 41: Comparison of the ResNet architectures on the test set for all different conditions

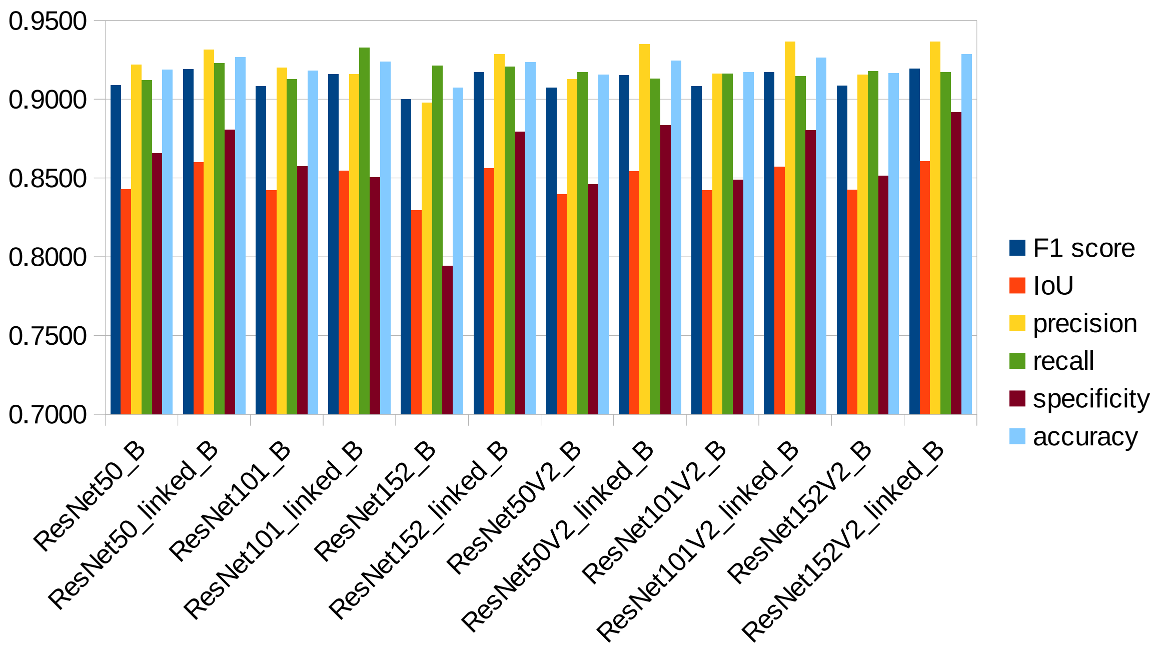

Figure 42: Comparison of the ResNet architectures on the test set for the most favourable conditions

4.4.3 Inception

4.4.3.1 Training Set

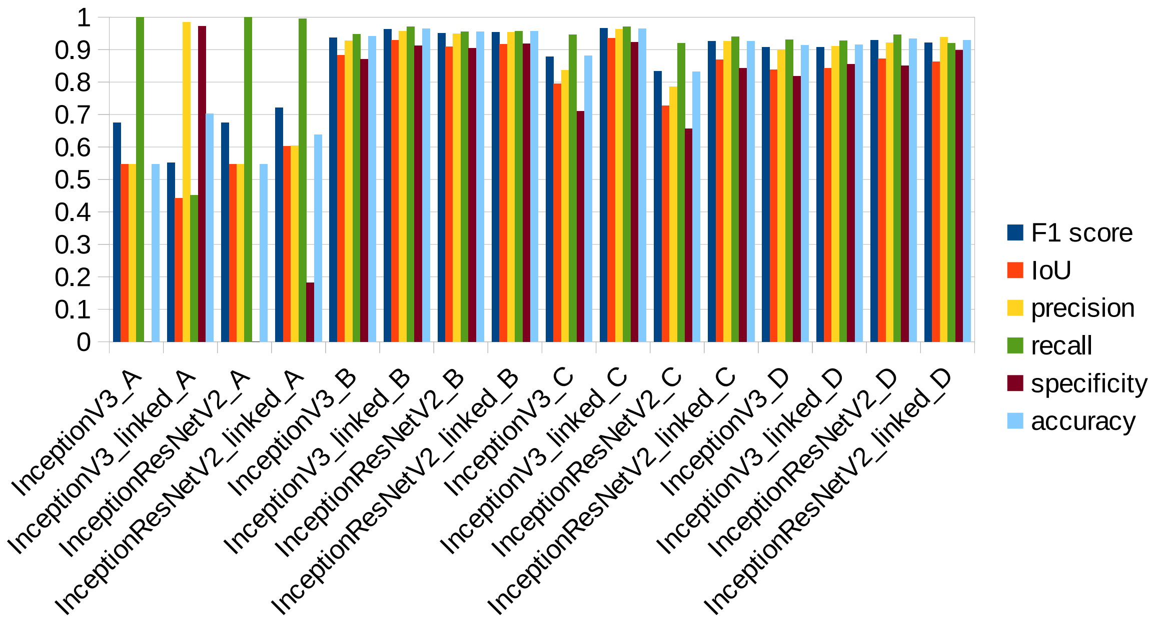

Figure 43: Comparison of the Inception architectures on the training set for all different conditions

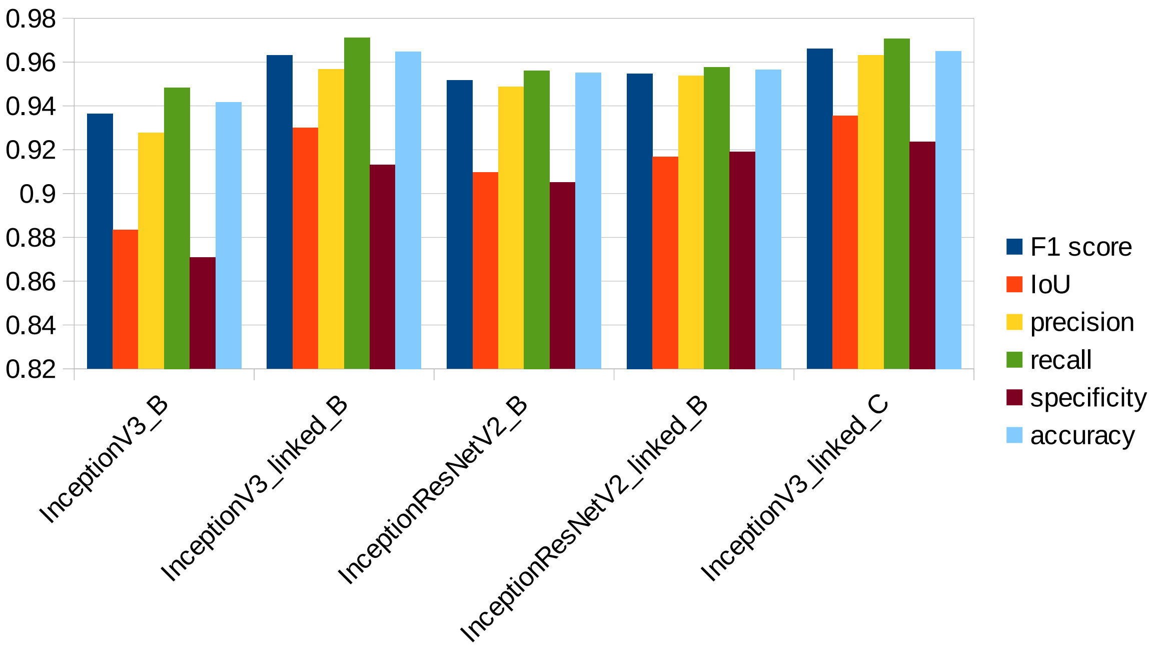

Figure 44: Comparison of the Inception architectures on the training set for the most favourable conditions

4.4.3.2 Validation Set

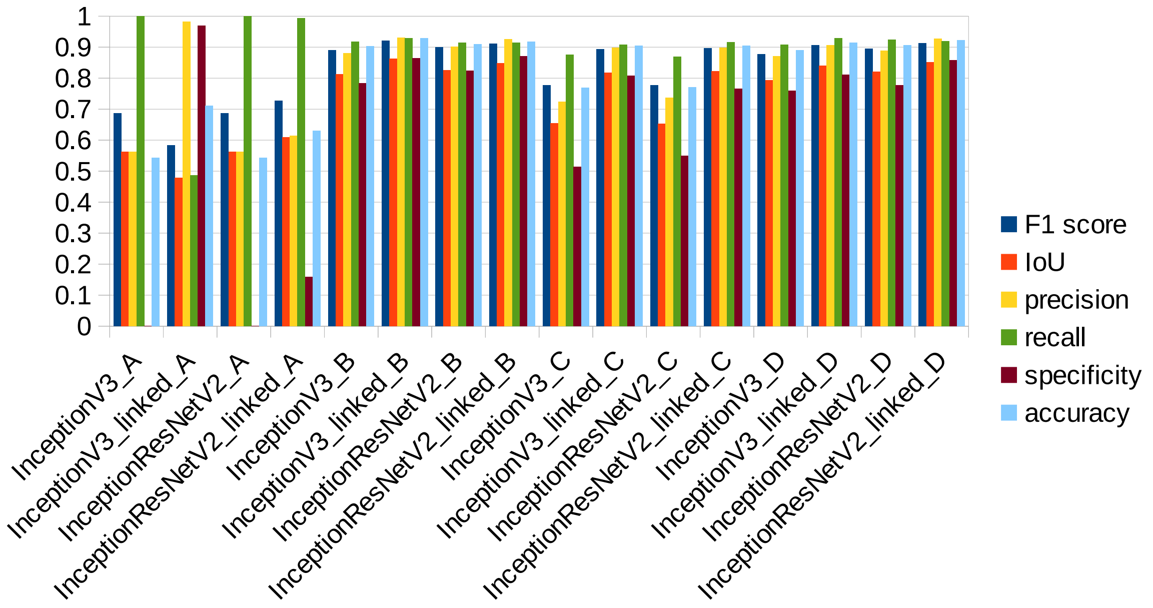

Figure 45: Comparison of the Inception architectures on the validation set for all different conditions

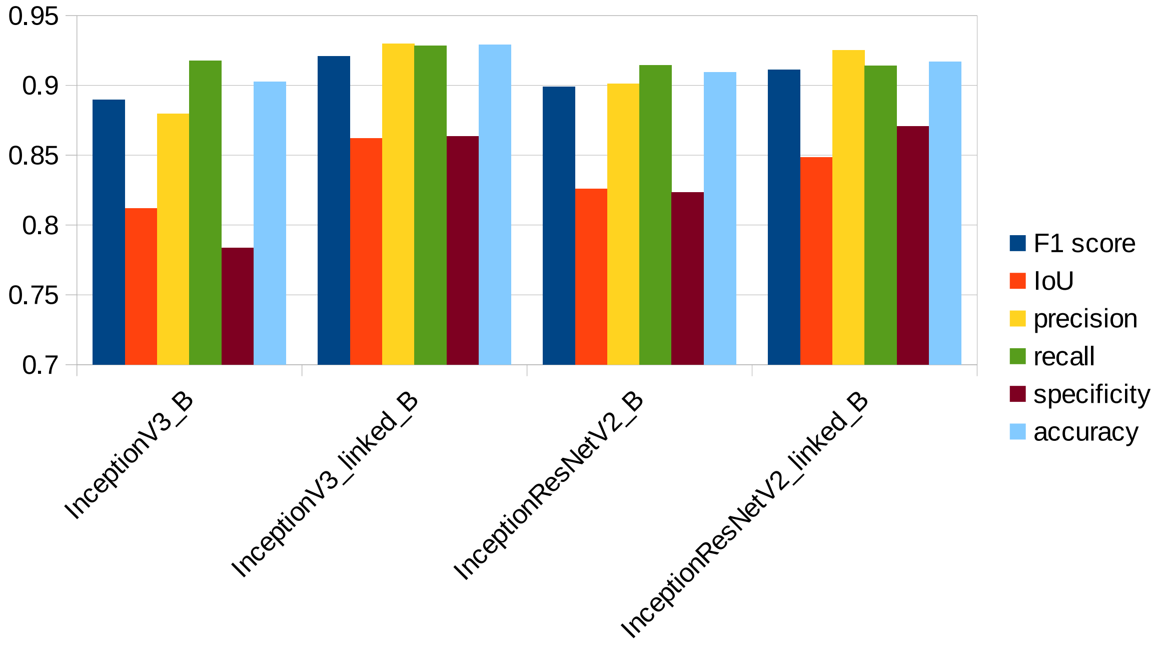

Figure 46: Comparison of the Inception architectures on the validation set for the most favourable conditions

4.4.3.3 Test Set

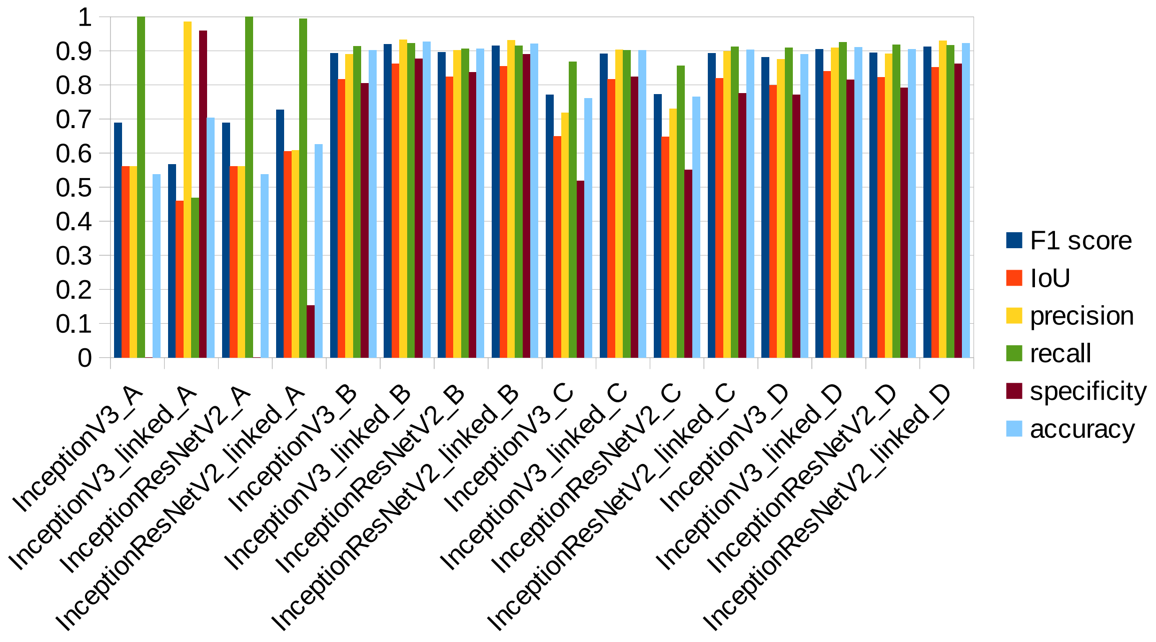

Figure 47: Comparison of the Inception architectures on the test set for all different conditions

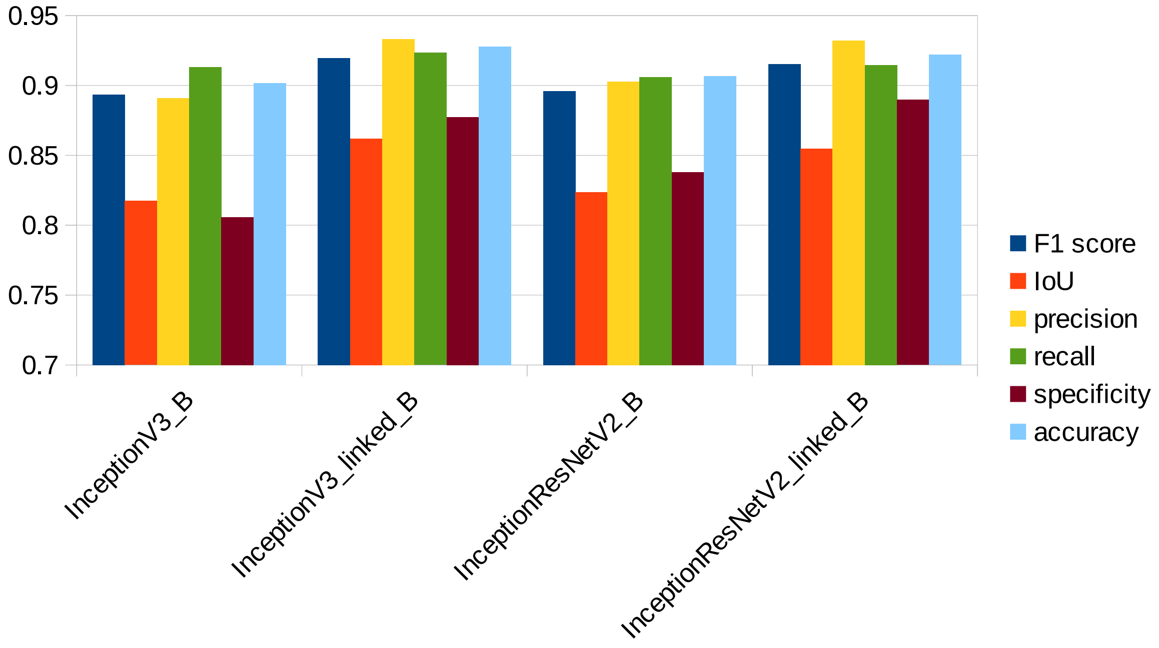

Figure 48: Comparison of the Inception architectures on the test set for the most favourable conditions

4.4.4 Xception

4.4.4.1 Training Set

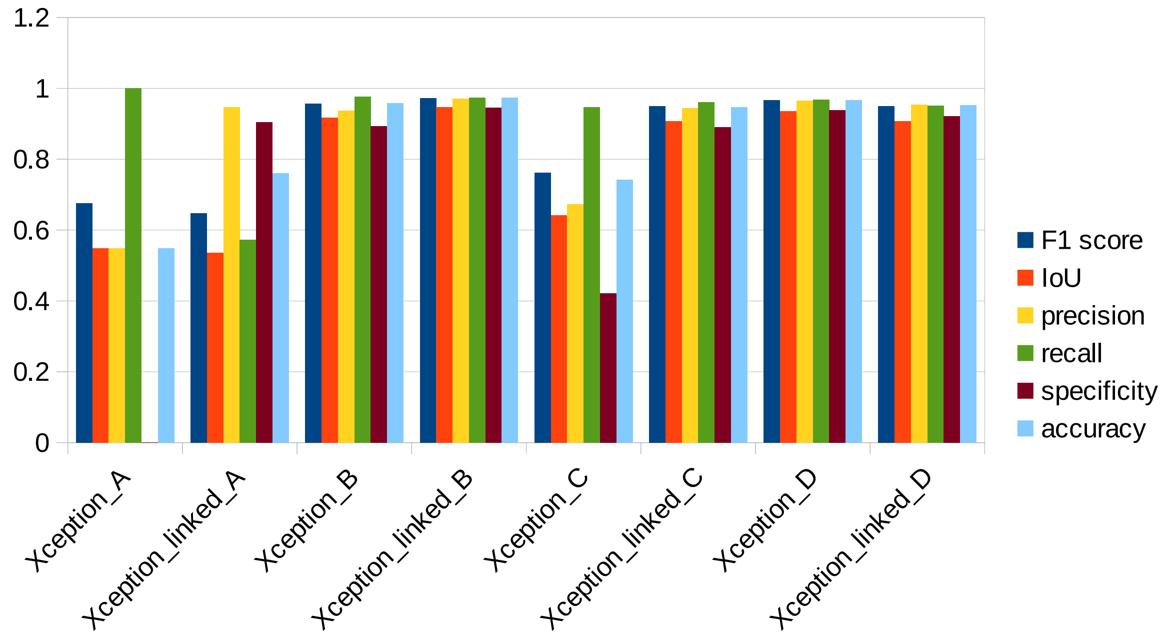

Figure 49: Comparison of the Xception architectures on the training set for all different conditions

4.4.4.2 Validation Set

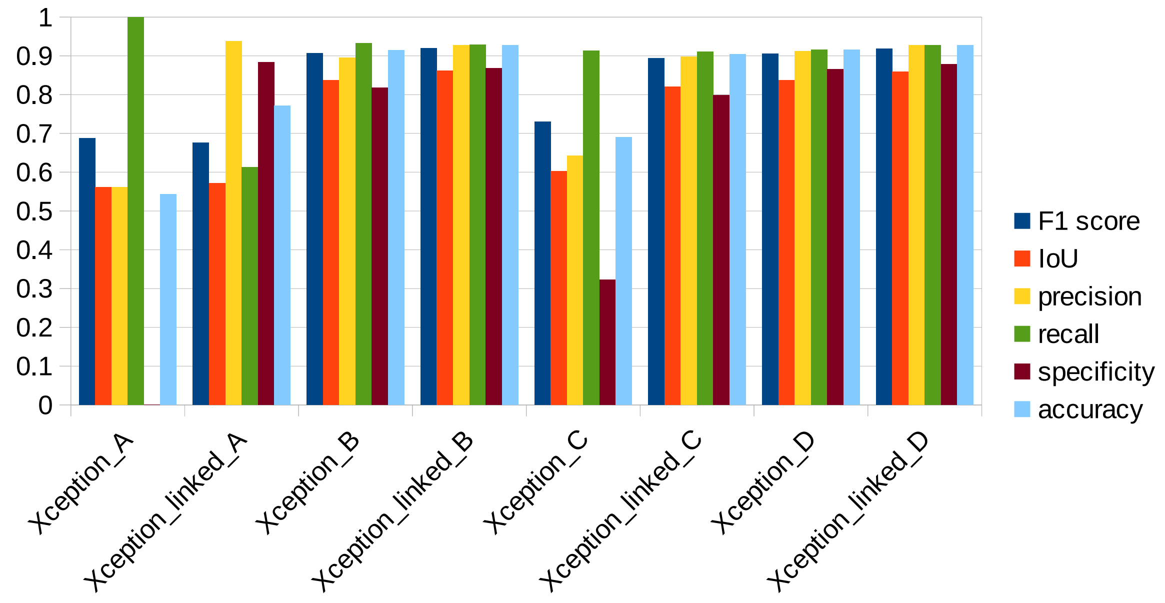

Figure 50: Comparison of the Xception architectures on the validation set for all different conditions

4.4.4.3 Test Set

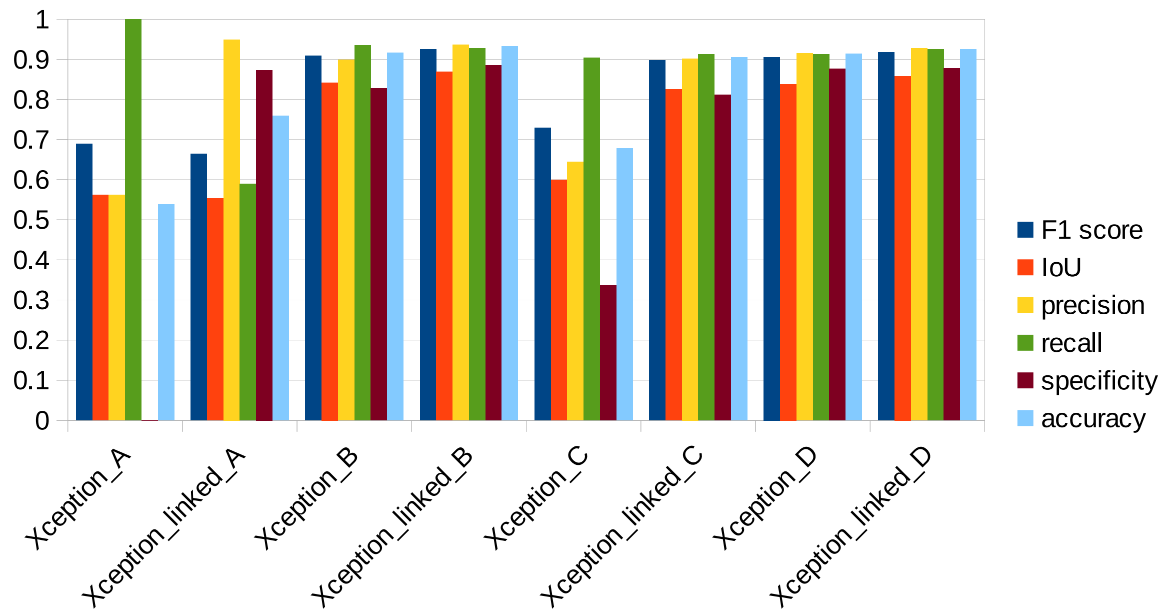

Figure 51: Comparison of the Xception architectures on the test set for all different conditions

4.4.5 NASNet

4.4.5.1 Training Set

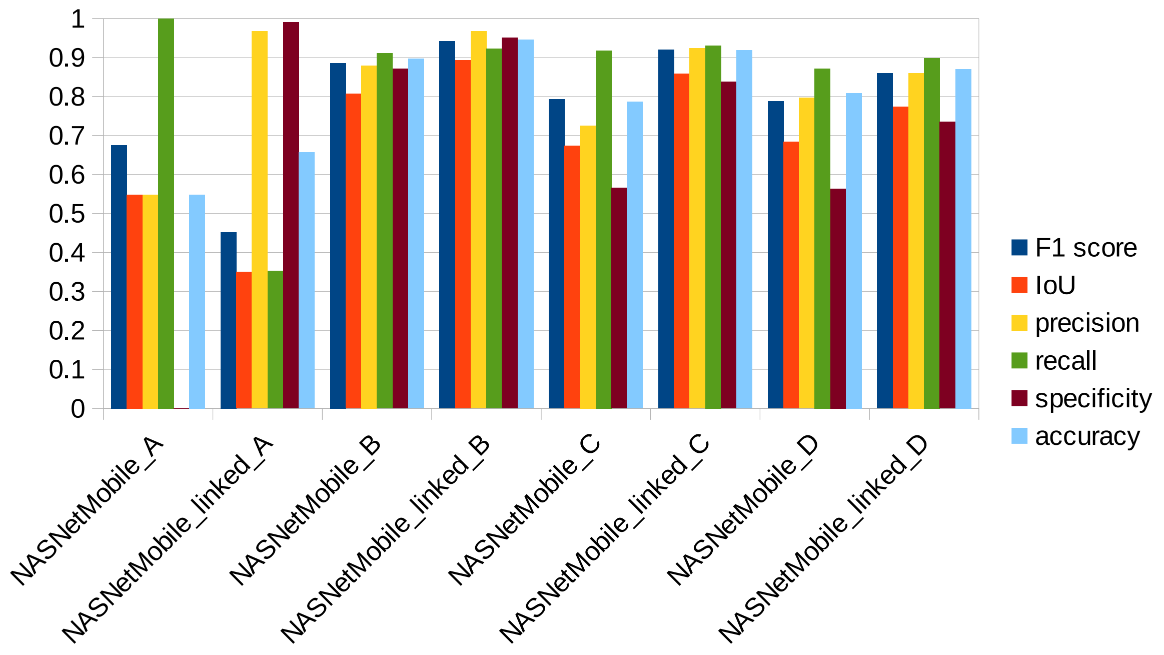

Figure 52: Comparison of the NASNet architectures on the training set for all different conditions

4.4.5.2 Validation Set

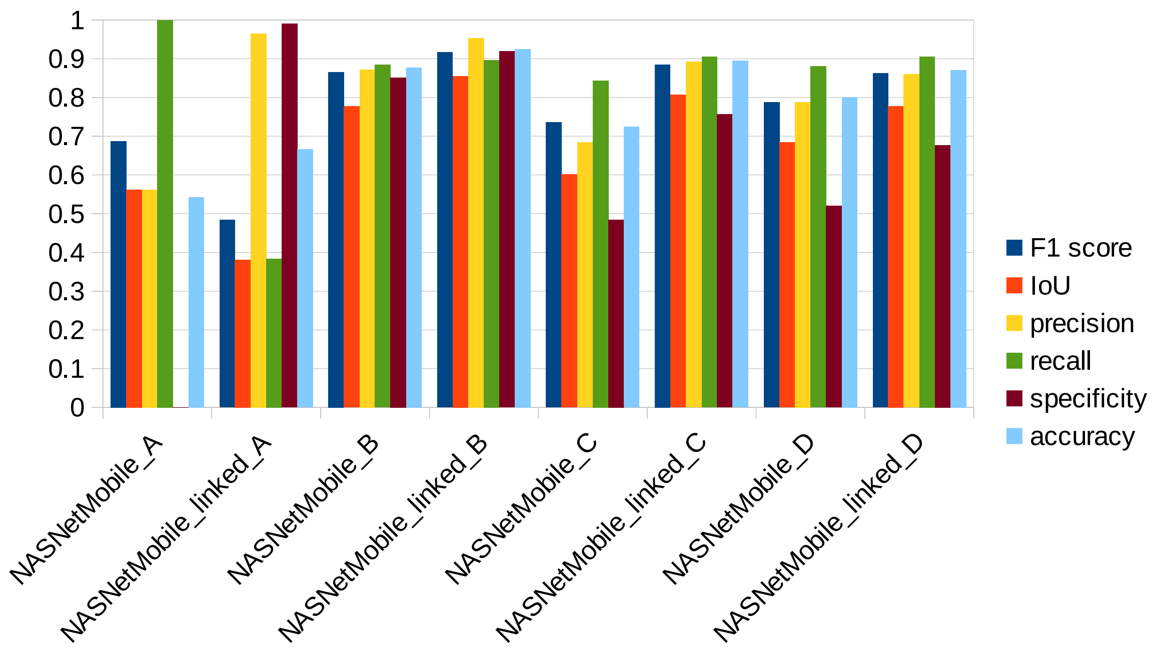

Figure 53: Comparison of the NASNet architectures on the validation set for all different conditions

4.4.5.3 Test Set

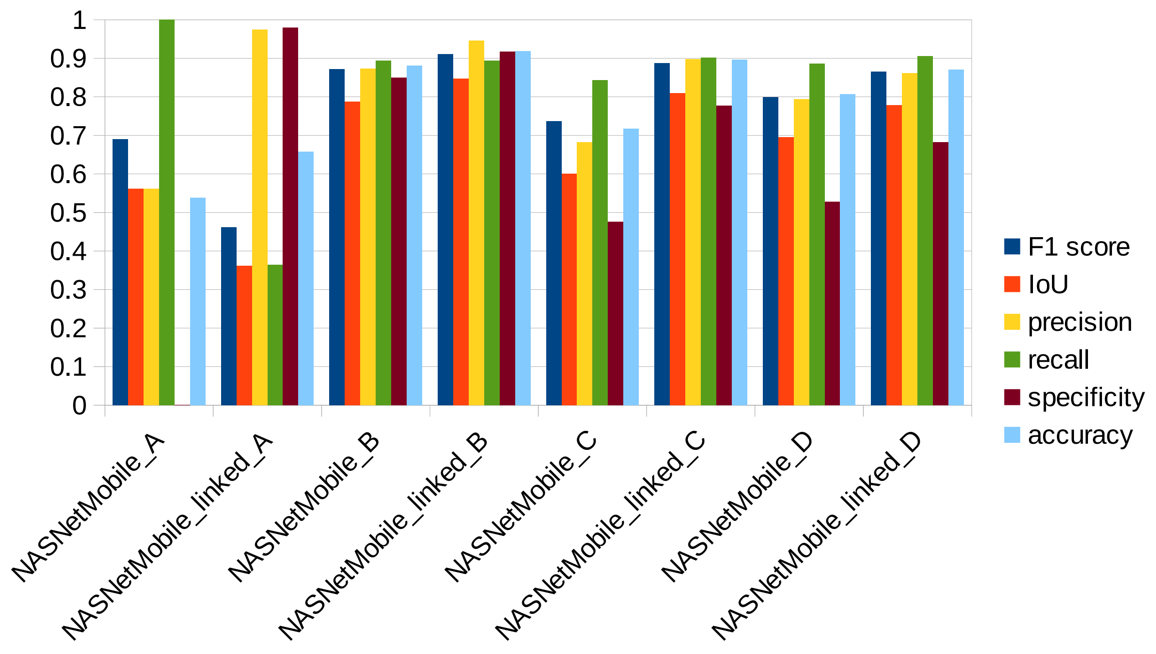

Figure 54: Comparison of the NASNet architectures on the test set for all different conditions

4.4.6 MobileNet

4.4.6.1 Training Set

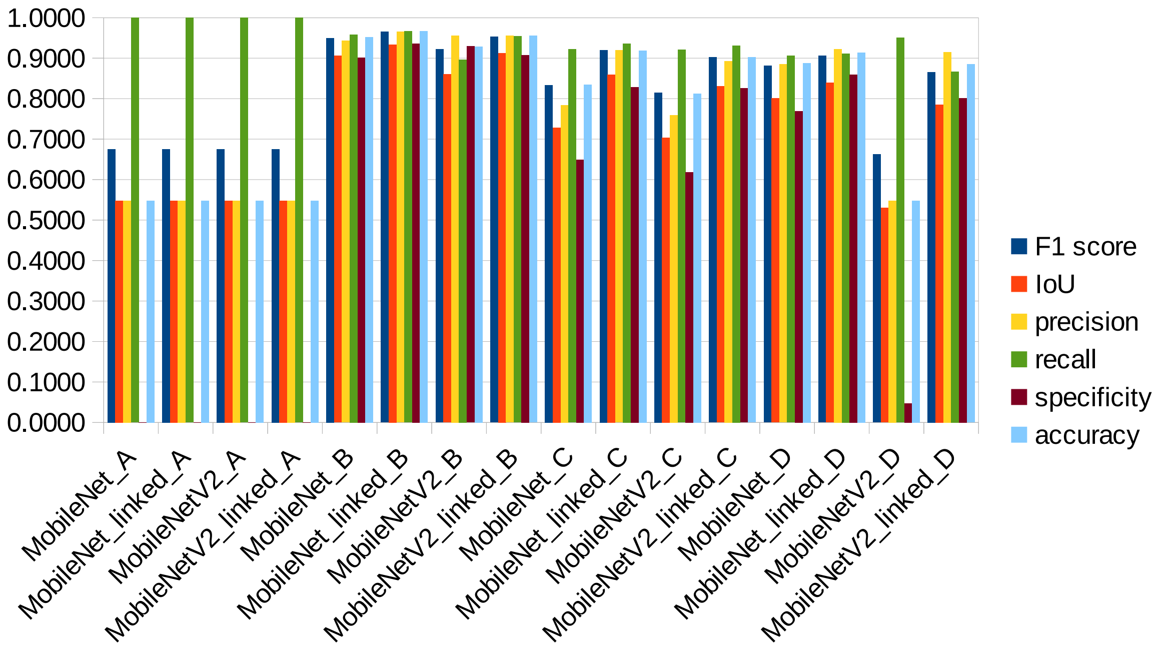

Figure 55: Comparison of the MobileNet architectures on the training set for all different conditions

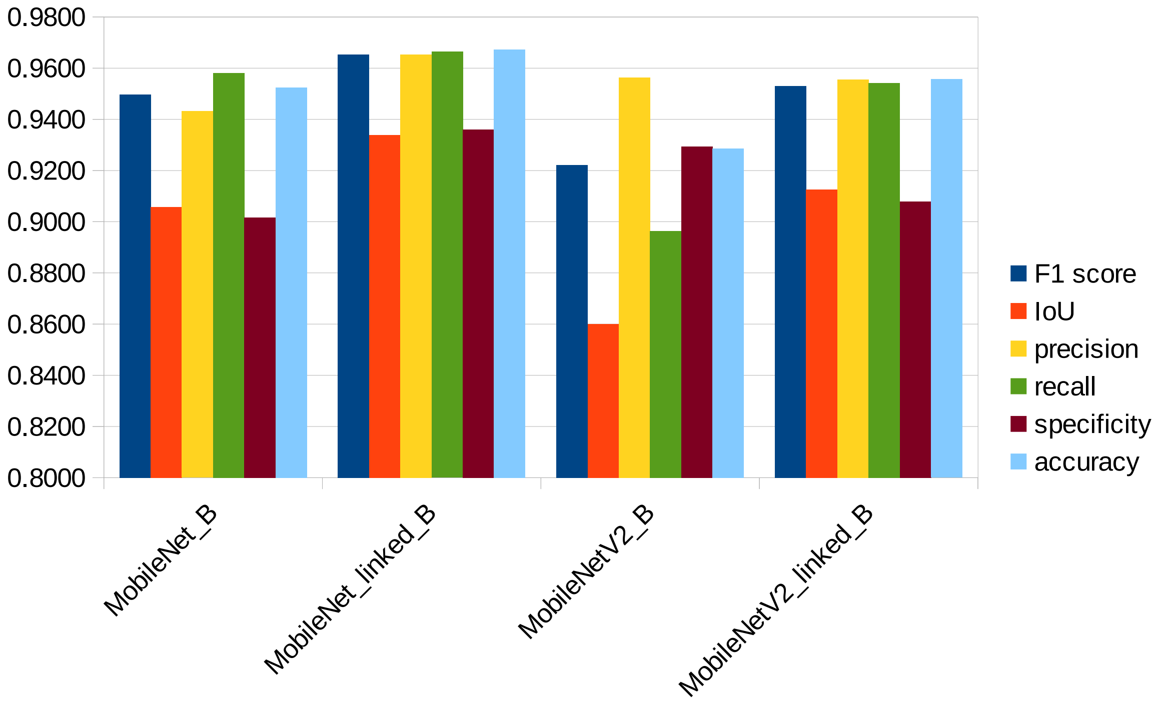

Figure 56: Comparison of the MobileNet architectures on the training set for the most favourable conditions

4.4.6.2 Validation Set

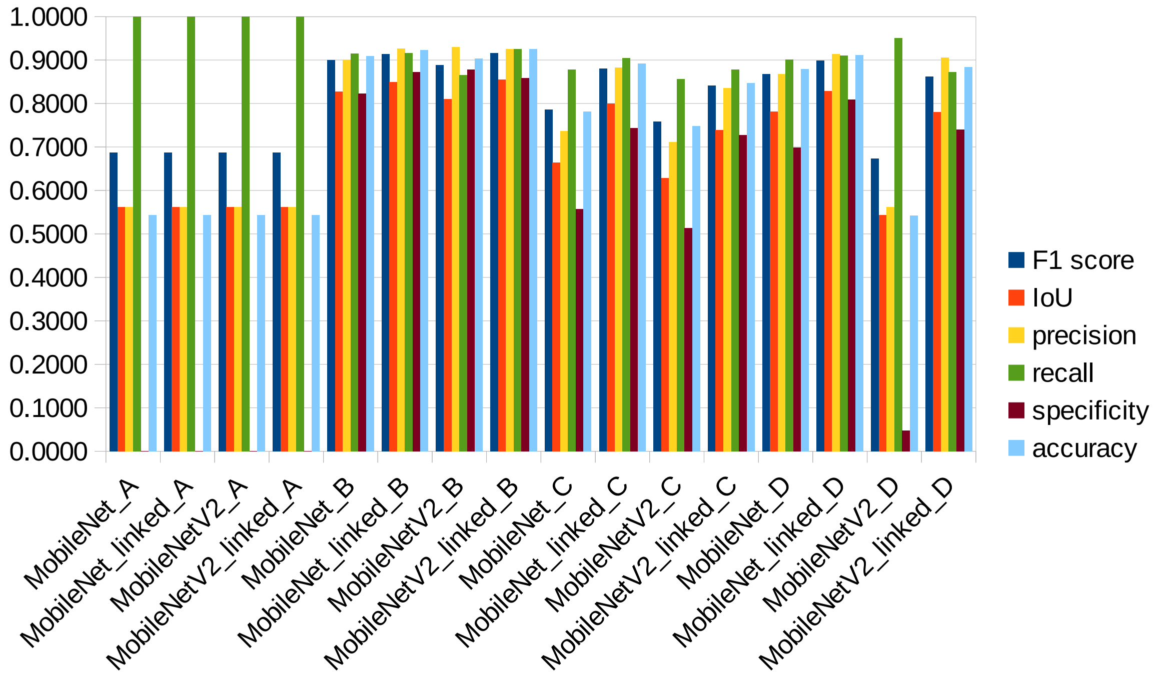

Figure 57: Comparison of the MobileNet architectures on the validation set for all different conditions

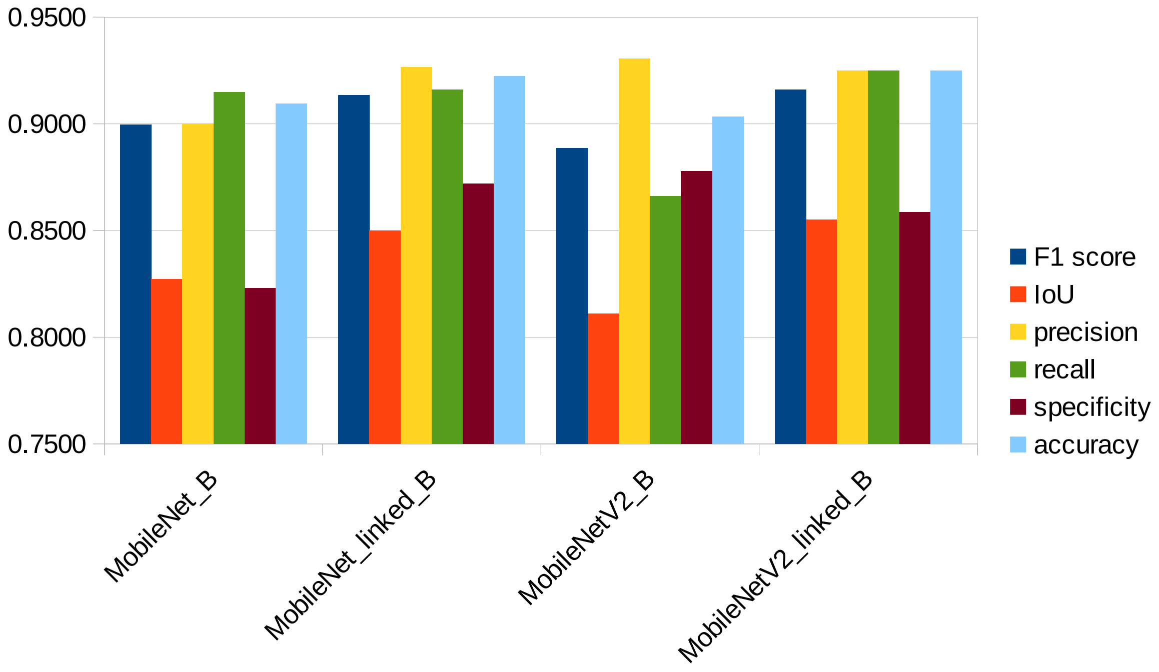

Figure 58: Comparison of the MobileNet architectures on the validation set for the most favourable conditions

4.4.6.3 Test Set

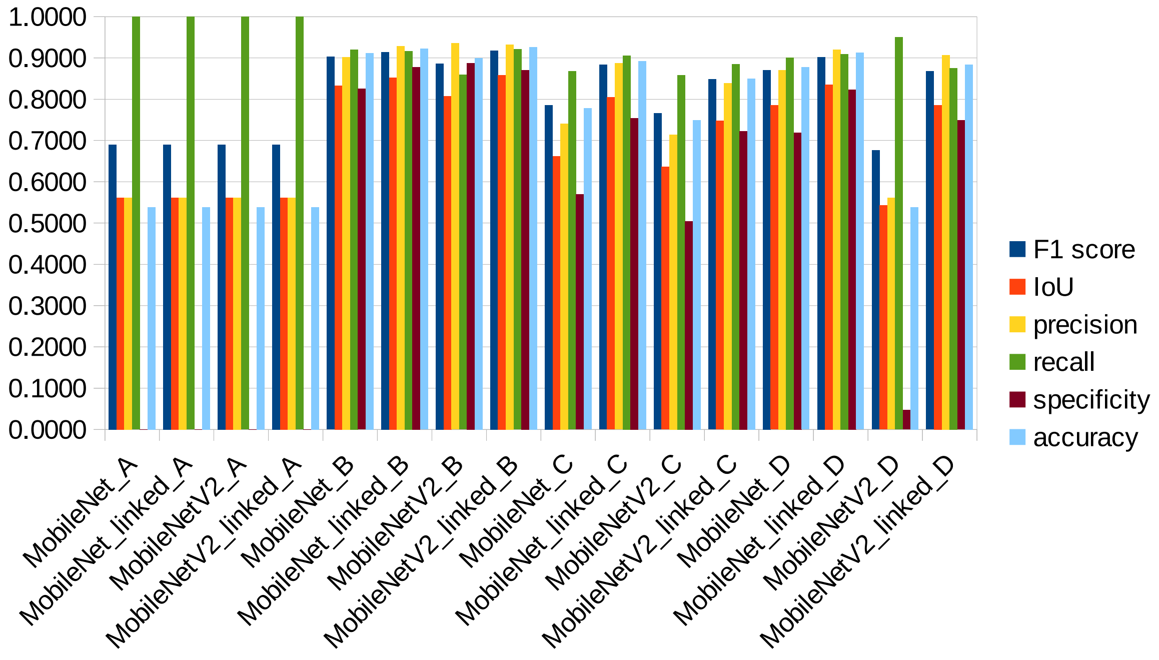

Figure 59: Comparison of the MobileNet architectures on the test set for all different conditions

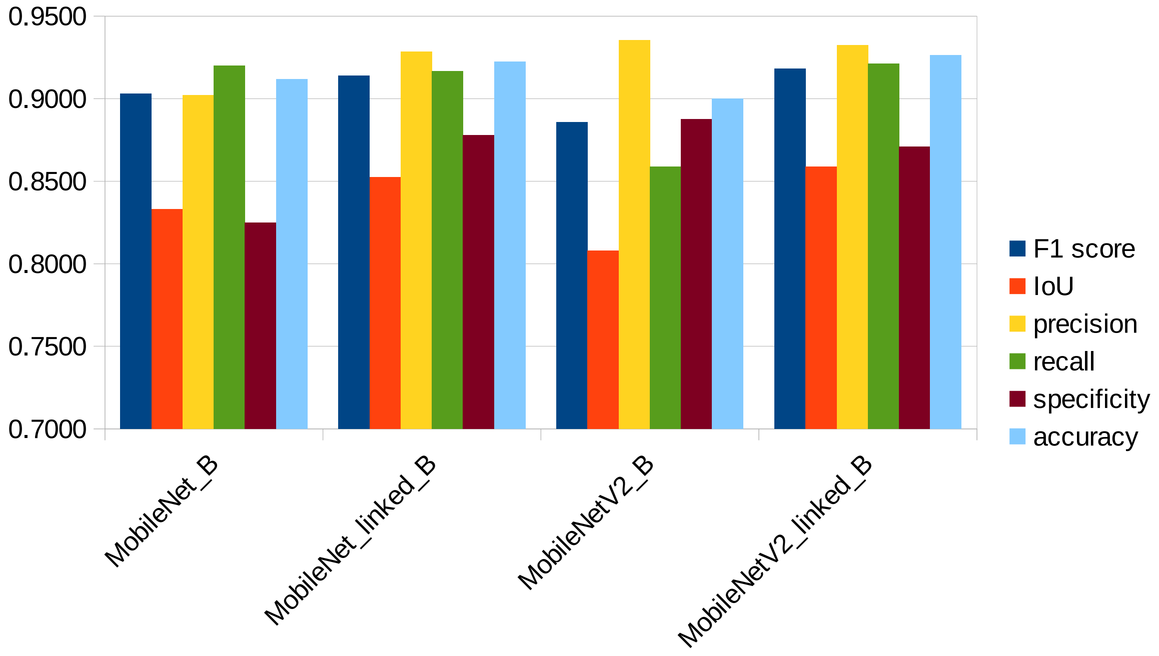

Figure 60: Comparison of the MobileNet architectures on the test set for the most favourable conditions

4.4.7 DenseNet

4.4.7.1 Training Set

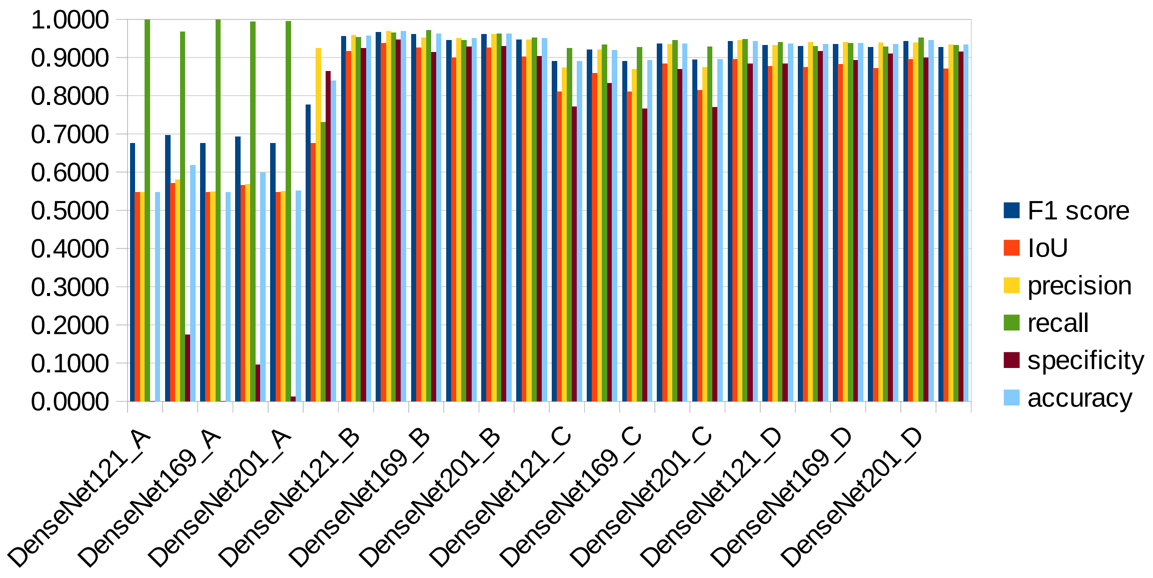

Figure 61: Comparison of the DenseNet architectures on the training set for all different conditions

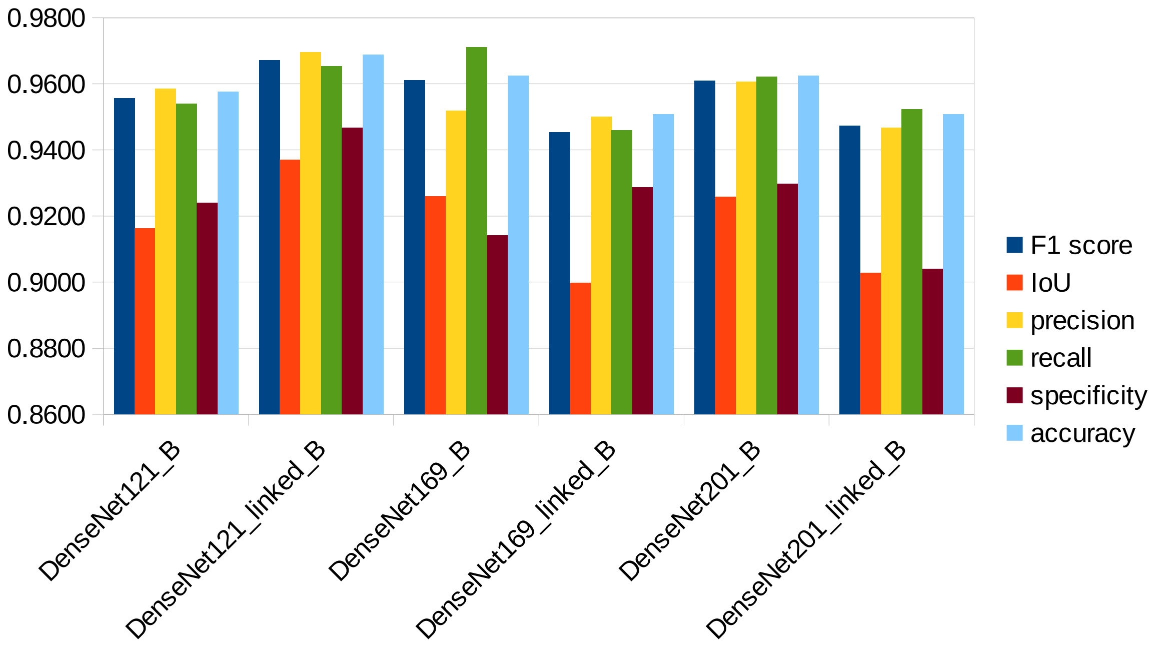

Figure 62: Comparison of the DenseNet architectures on the training set for the most favourable conditions

4.4.7.2 Validation Set

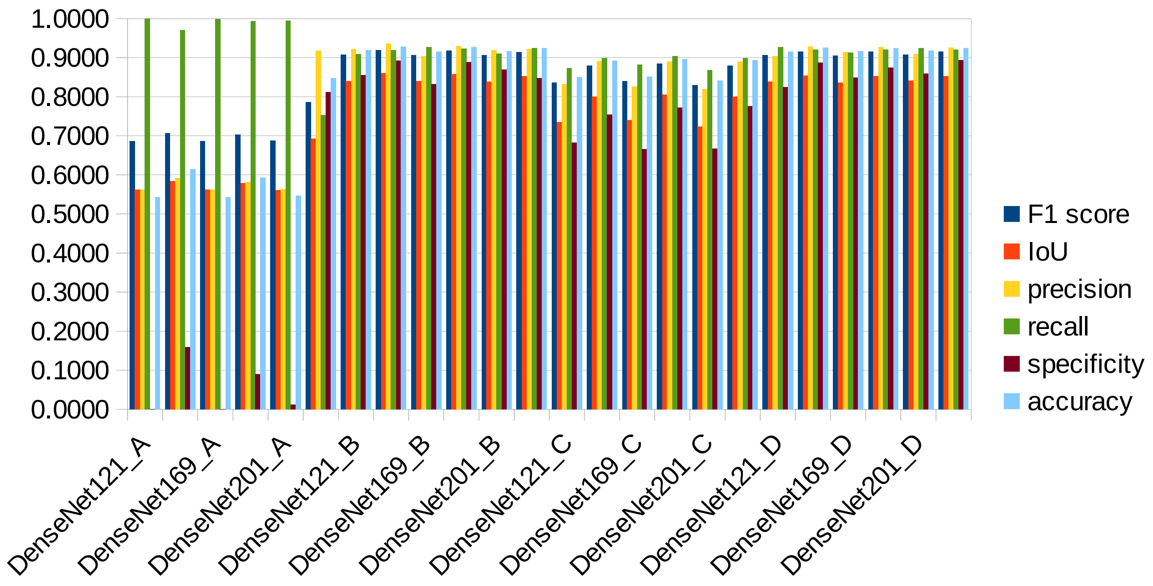

Figure 63: Comparison of the DenseNet architectures on the validation set for all different conditions

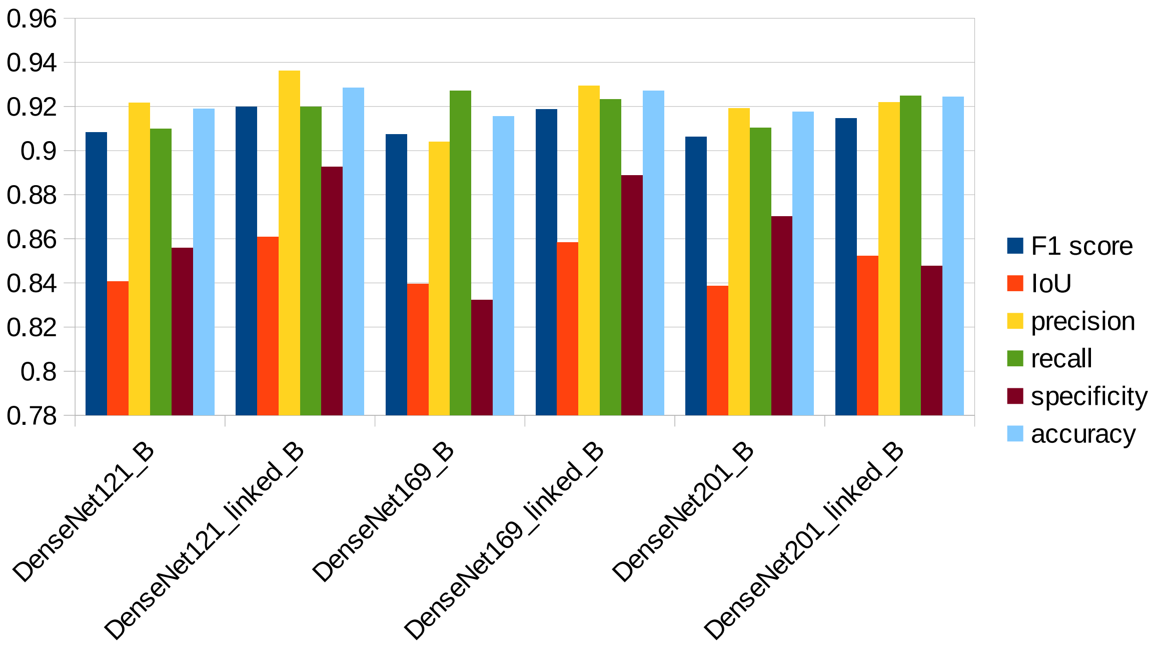

Figure 64: Comparison of the DenseNet architectures on the validation set for the most favourable conditions

4.4.7.3 Test Set

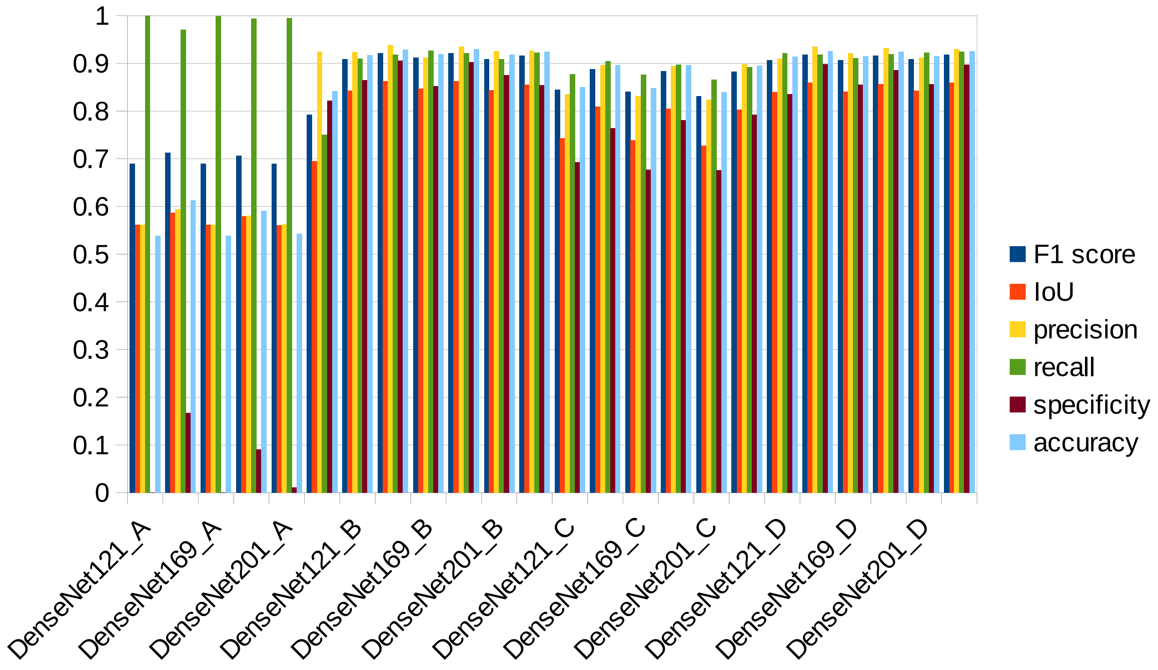

Figure 65: Comparison of the DenseNet architectures on the test set for all different conditions

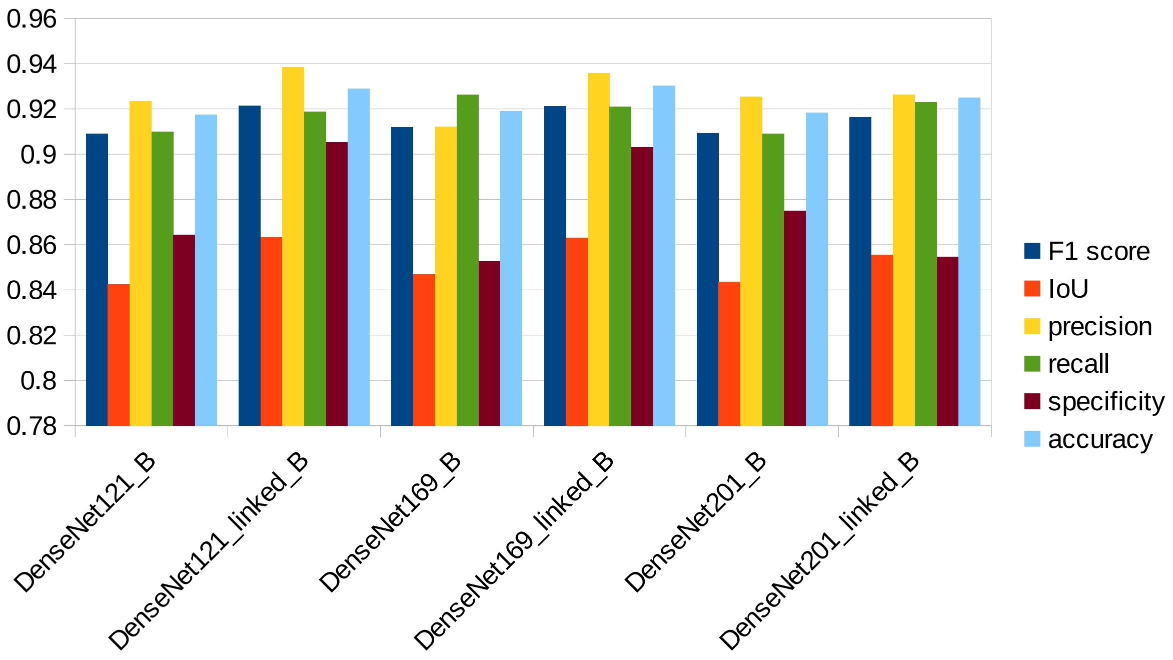

Figure 66: Comparison of the DenseNet architectures on the test set for the most favourable conditions

4.4.8 EfficientNet

4.4.8.1 Training Set

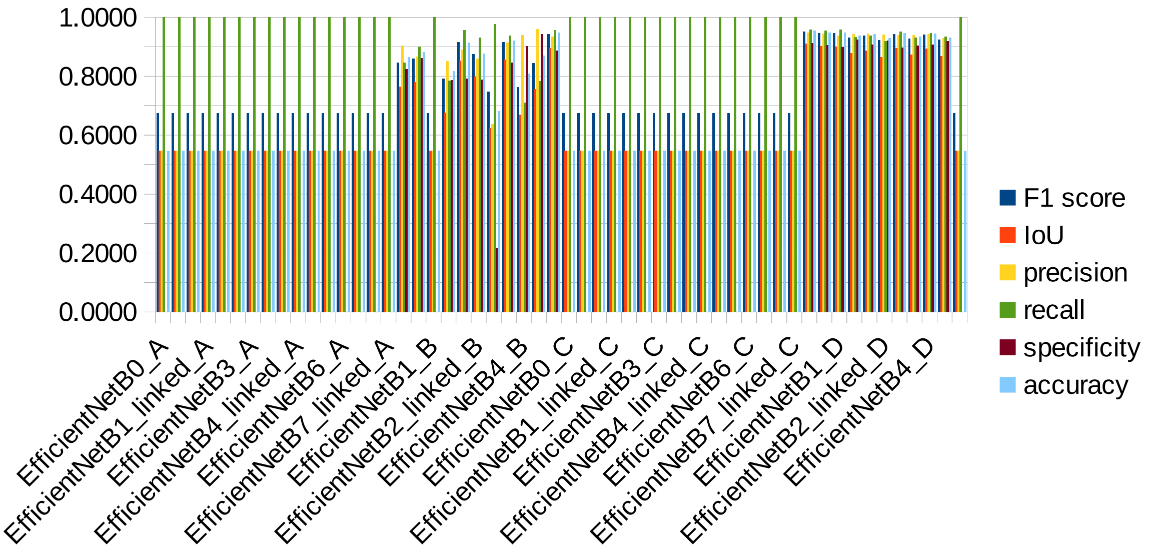

Figure 67: Comparison of the EfficientNet architectures on the training set for all different conditions

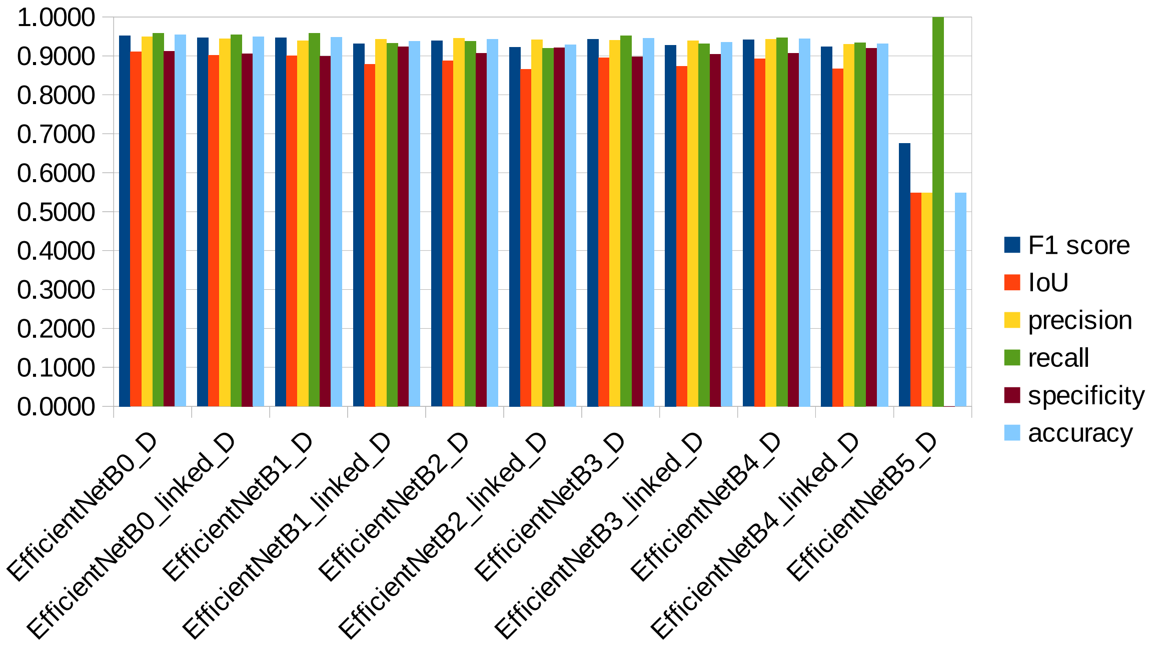

Figure 68: Comparison of the EfficientNet architectures on the training set for the most favourable conditions

4.4.8.2 Validation Set

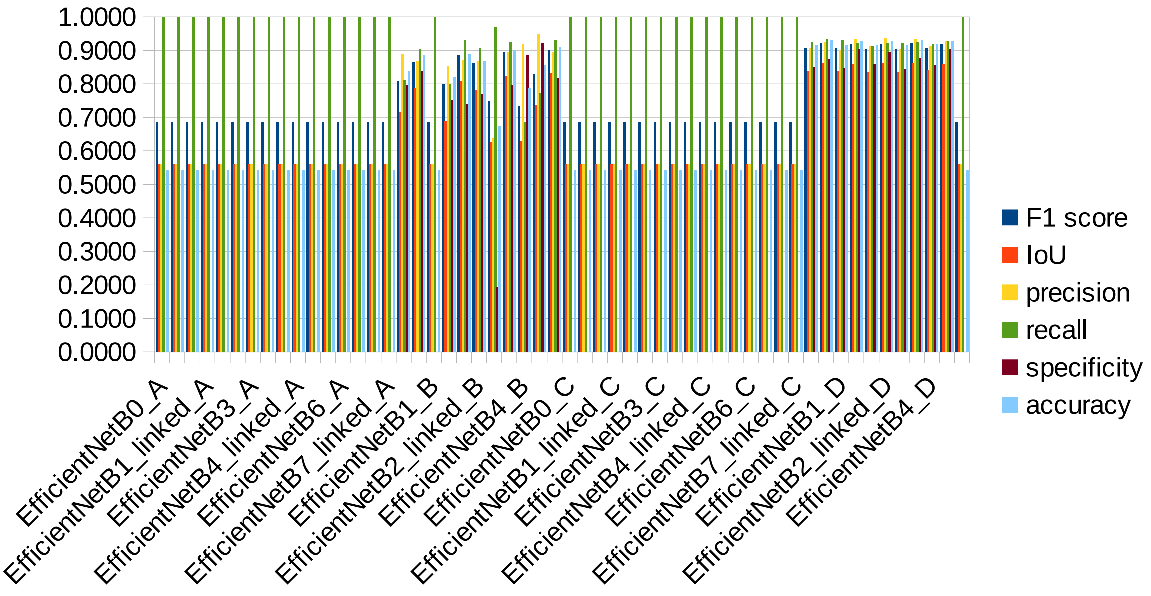

Figure 69: Comparison of the EfficientNet architectures on the validation set for all different conditions

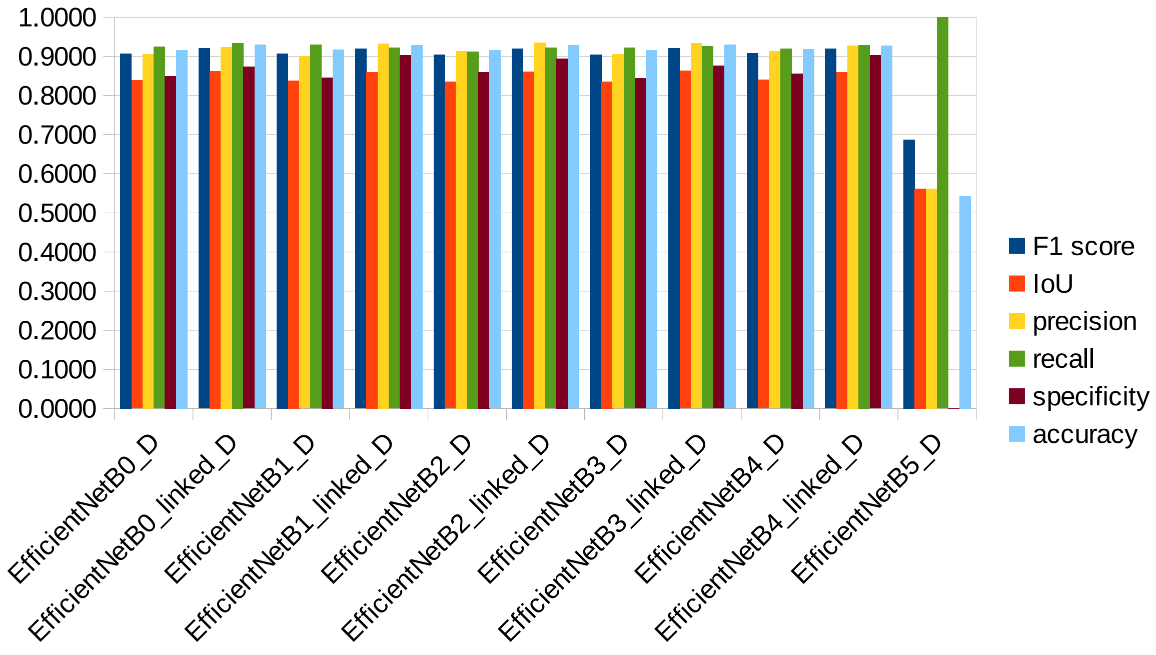

Figure 70: Comparison of the EfficientNet architectures on the validation set for the most favourable conditions

4.4.8.3 Test Set

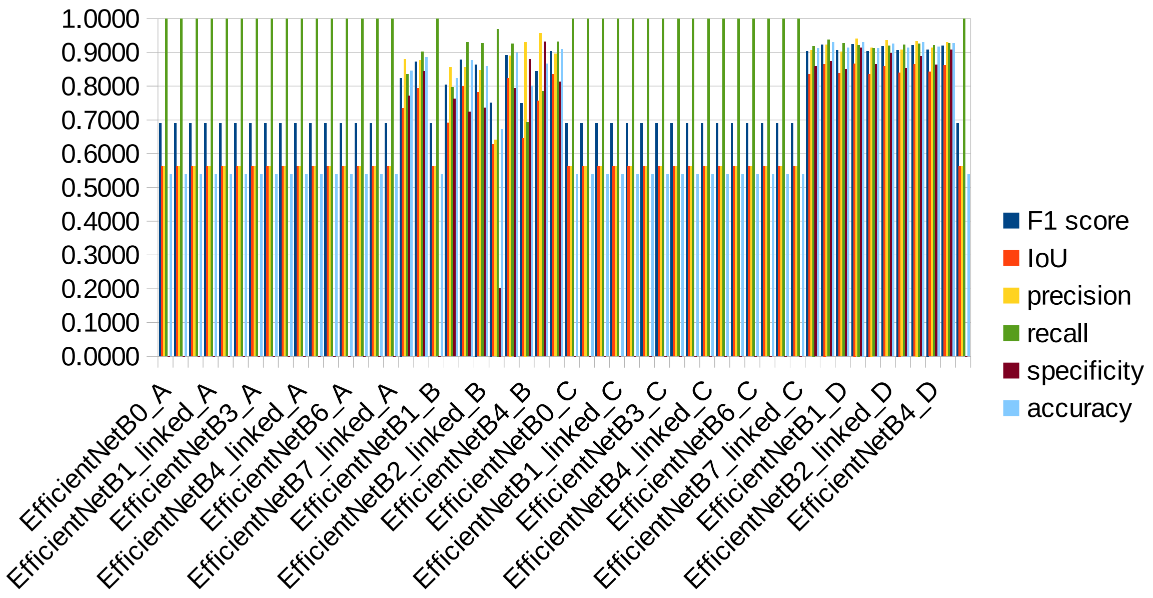

Figure 71: Comparison of the EfficientNet architectures on the test set for all different conditions

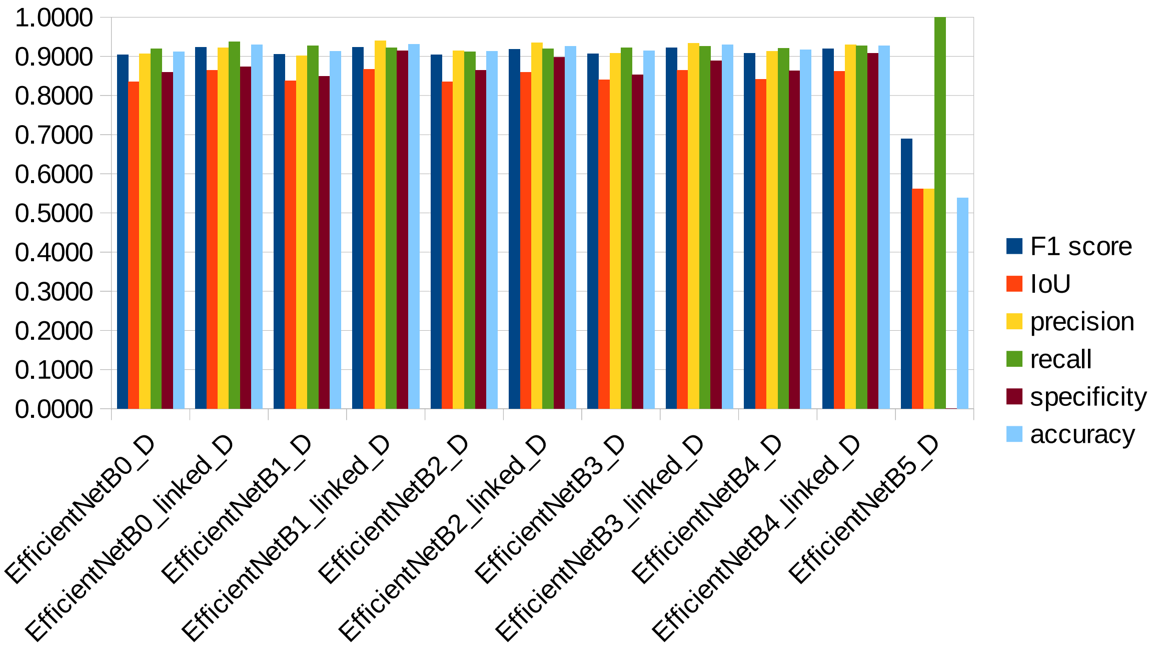

Figure 72: Comparison of the EfficientNet architectures on the test set for the most favourable conditions

4.4.9 Highest Performance Architectures

4.4.9.1 Training Set

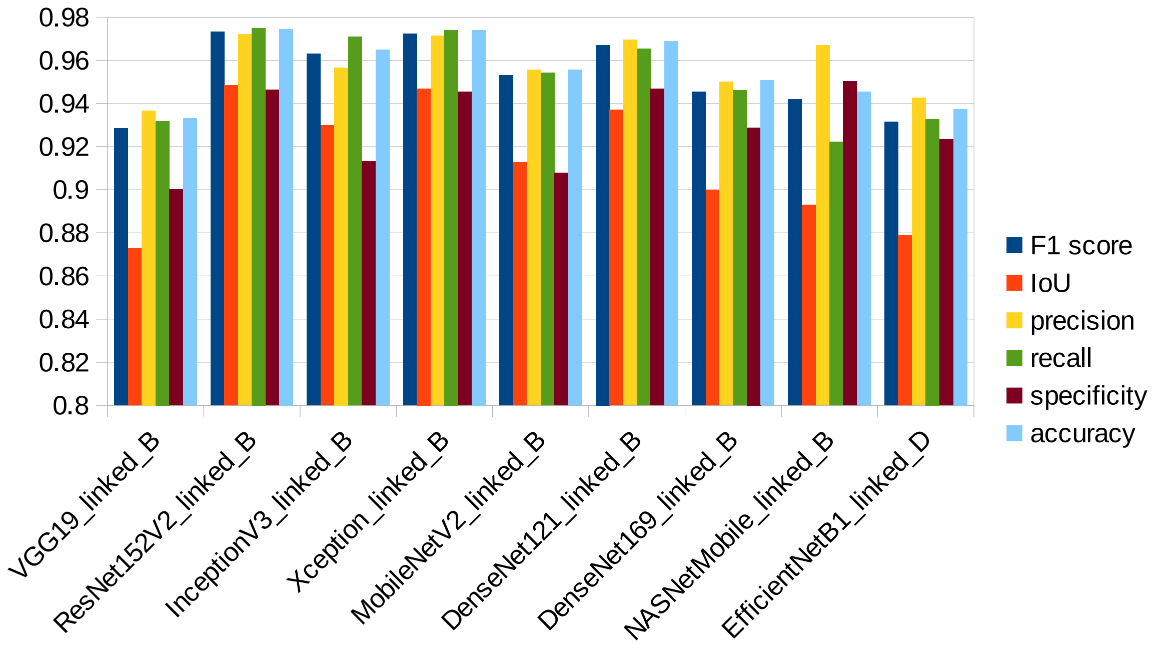

Figure 73: Comparison of the best architectures on the training set for their most favourable conditions

4.4.9.2 Validation Set

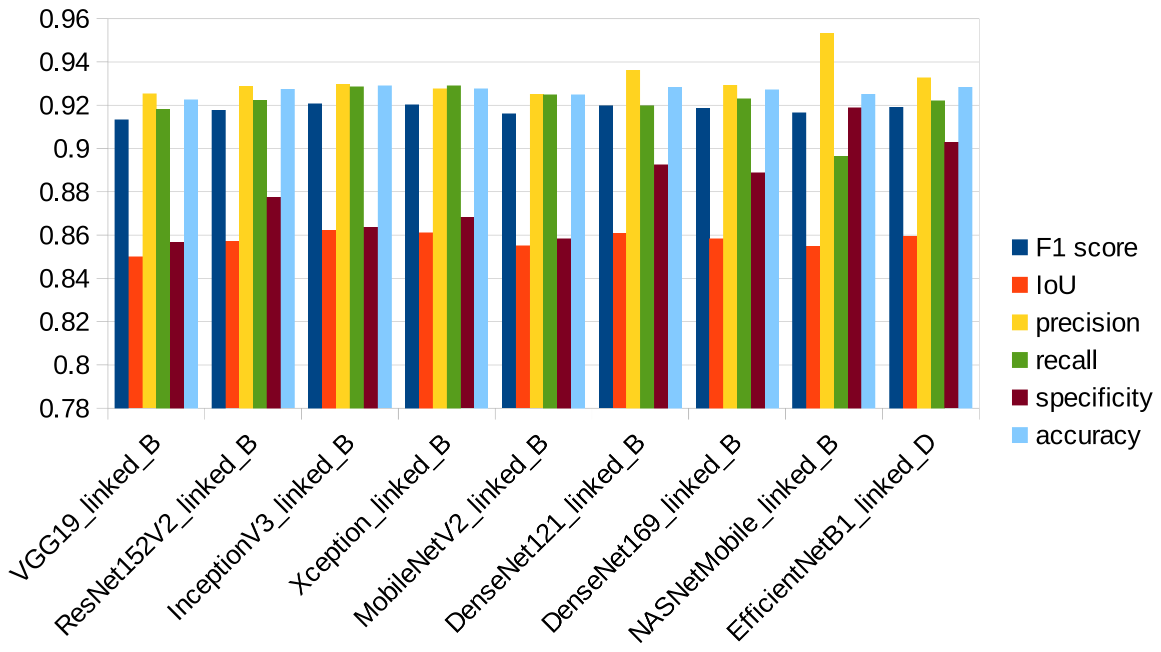

Figure 74: Comparison of the best architectures on the validation set for their most favourable conditions

4.4.9.3 *Test Set*

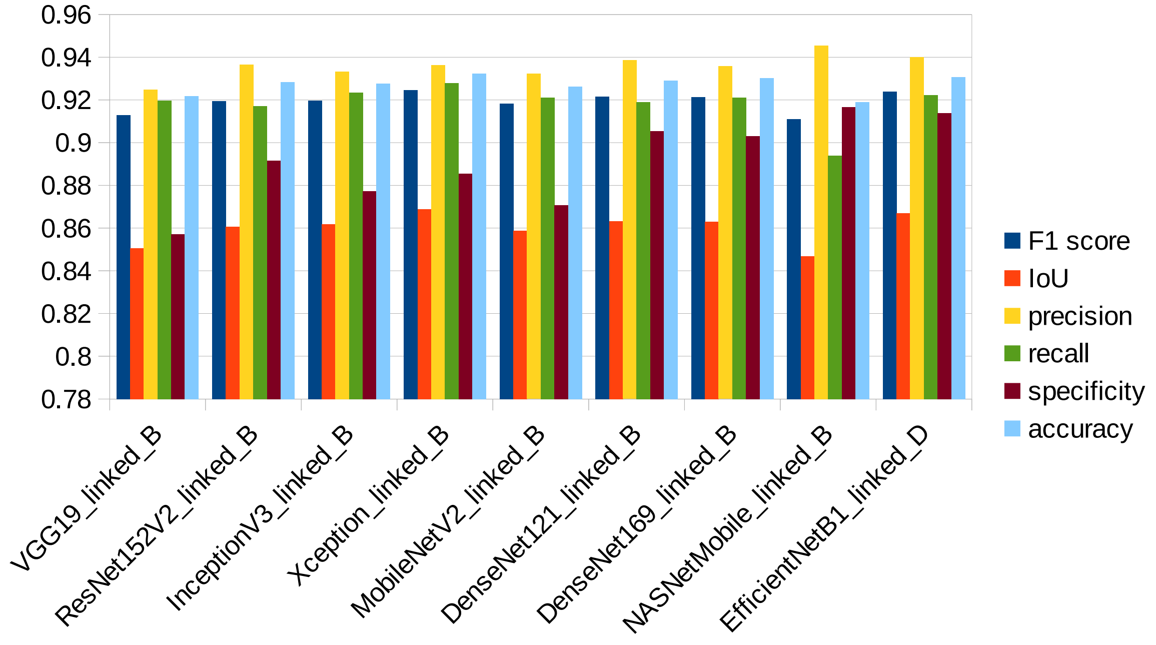

Figure 75: Comparison of the best architectures on the test set for their most favourable conditions

4.5 Sample of segmented Images

The sample of images illustrated below were all chosen arbitrarily from the test set.

4.5.1 RGB Images

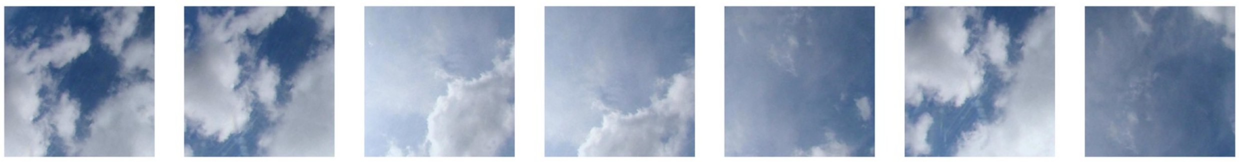

Figure 76: Sample of RGB images from the SWIMSEG dataset

4.5.2 Ground Truth Images

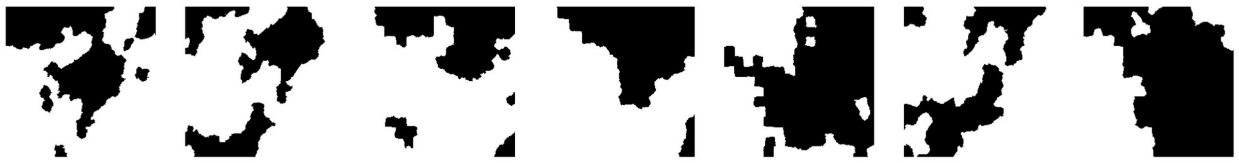

Figure 77: Ground truth images for the sample of RGB ones

4.5.3 Unet

Figure 78: Segmented images by the Unet architecture for the sample of RGB ones

4.5.4 A_Unet

Figure 79: Segmented images by the A_Unet architecture for the sample of RGB ones

4.5.5 D_Unet

Figure 80: Segmented images by the D_Unet architecture for the sample of RGB ones

4.5.6 W_Unet

Figure 81: Segmented images by the W_Unet architecture for the sample of RGB ones

4.5.7 R_Unet

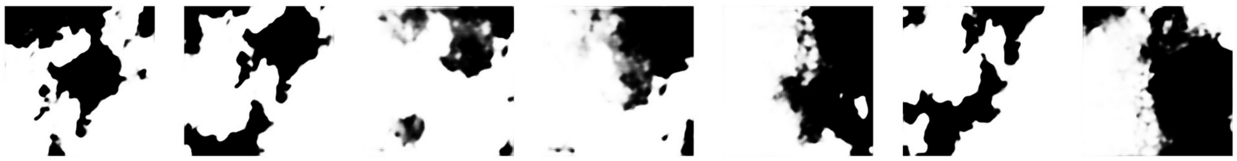

Figure 82: Segmented images by the R_Unet architecture for the sample of RGB ones

4.5.8 VGG19_linked

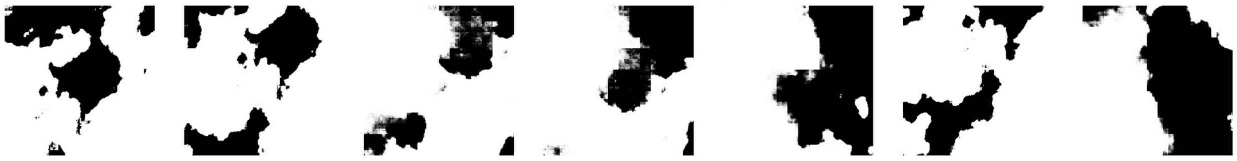

Figure 83: Segmented images by the VGG19_linked architecture for the sample of RGB ones

4.5.9 ResNet152V2_linked

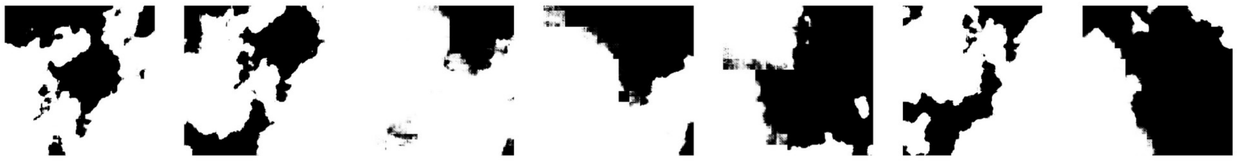

Figure 84: Segmented images by the ResNet152V2_linked architecture for the sample of RGB ones

4.5.10 InceptionV3_linked

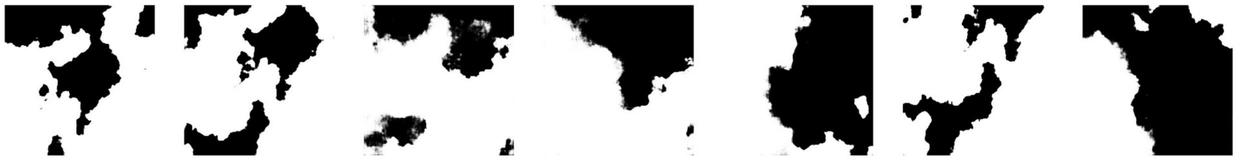

Figure 85: Segmented images by the InceptionV3_linked architecture for the sample of RGB ones

4.5.11 Xception_linked

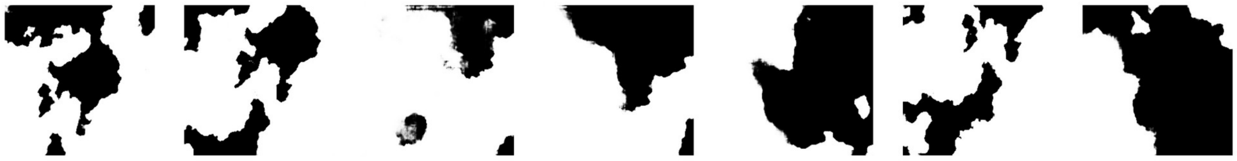

Figure 86: Segmented images by the Xception_linked architecture for the sample of RGB ones

4.5.12 MobileNetV2_linked

Figure 87: Segmented images by the MobileNetV2_linked architecture for the sample of RGB ones

4.5.13 DenseNet121_linked

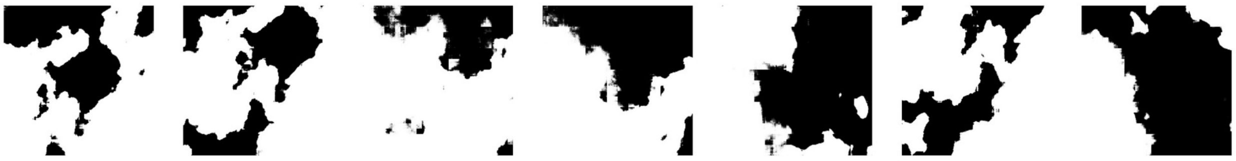
Figure 88: Segmented images by the DenseNet121_linked architecture for the sample of RGB ones

4.5.14 DenseNet169_linked

Figure 89: Segmented images by the DenseNet169 architecture for the sample of RGB ones

4.5.15 NASNetMobile_linked

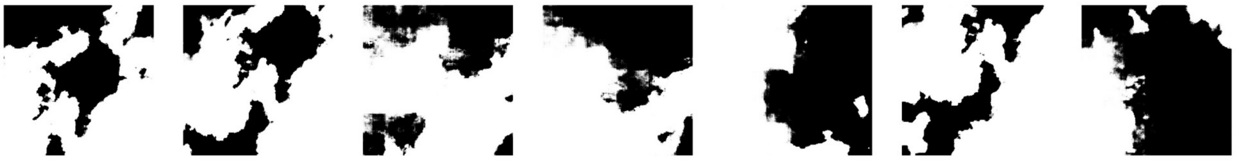
Figure 90: Segmented images by the NASNetMobile_linked architecture for the sample of RGB ones

4.5.16 EfficientNetB1_linked

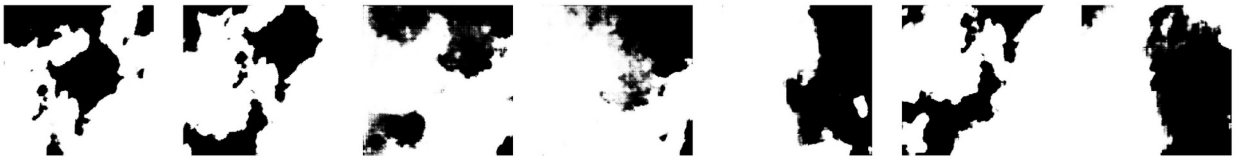
Figure 91: Segmented images by the EfficientNetB1_linked architecture for the sample of RGB ones

Chapter 5

5 Conclusions

A thorough investigation between several deep learning algorithms has been conducted in an attempt to determine the most suitable for the task of cloud image segmentation. Although, significant progress has been made in the field with the advent of deep learning algorithms, it still remains an open issue for the research community. That is because cloud classification is a rather difficult problem primarily due to its vague nature and secondly due to the lack of consistent data. The main purpose of this study is to make things more clear by providing information through the comparative analysis of existing algorithms as well as new variations inspired by successful architectures.

5.1 Specific Conclusions

- Comparing the dataset ratios it is impossible to reach a solid conclusion as far as the optimal ratio is concerned for a small dataset like SWIMSEG. What should really be inferred is that the number of images is inadequate for training very deep ones. Consequently, augmentation of the images should be considered as a possible solution to achieve better results and alleviate the problem of overfitting, observed in the very deep pretrained ones.
- Optimizers with adaptive learning rate seem to yield better results than conventional ones for semantic segmentation of cloud images.
- Loss functions who emphasize more on precision than recall tend to achieve greater results. This can be discerned from the scores of Fbeta (b=1.2), Dice and Tversky loss functions. Specifically, the Fbeta loss function emphasizes more on precision than recall and achieves higher score in all sets than Dice. On the contrary, Tversky loss function emphasizes more on recall and performs worse than Dice in all sets.
- Transfer learning on cloud segmentation, through networks pretrained on the ImageNet dataset, is possible. Although, most of the times it can lead to greater performance, this is not a general rule. A characteristic example is the case of EfficientNet networks, where the

models initialized with weights from the respective pretrained ones were outperformed by others with random initialization of weights.

- Very deep networks are more prone to overfitting than smaller ones. This is quite distinct between the VGG19_linked and ResNet152V2_linked architectures which have similar scores on the validation and test sets but a huge discrepancy between those on the training set.
- From the two implemented versions of fully convolutional neural networks with pretrained encoders, the one with integrated skip connections yielded better results than the other without them for all the pretrained networks. In more detail, the difference in overall performance was getting wider as the network's depth increased. Furthermore, the networks without skip connections were more susceptible to overfitting. That means skip connections play an important role on cloud segmentation and they should be included in networks designed for that purpose so as to enhance their performance.

5.2 Contributions

The original contributions of this thesis include:

- The development of five novel Unet variations for cloud image segmentation.
- Detailed evaluation for the use of pretrained models as backbones to encoder-decoder architectures designed for cloud image segmentation.
- A thorough investigation on the utilization of skip connections for cloud segmentation.

5.3 Future research

- Utilization of the albumentation package [56] for the augmentation of the images and evaluation of the networks on the new larger and more diverse dataset.
- Development of a new, consistent and larger dataset containing all weather conditions for cloud segmentation.
- Evaluation of the Unet variations on more datasets including both ground-based and satellite based ones.
- Further experimentation both on the architectures of the Unet variations and on the tweaking of the networks' hyperparameters.

6 Bibliography

1: F. Kreuwel, W. Knap, L. Visser, W. Sark, J. Arrelano, C. Heerwaarden, Analysis of high frequency photovoltaic solar energy fluctuations, 2020

2: A. Werkmeister, M. Lockhoff, M. Schrempf, K. Tohsing, B. Liley, G. Seckmeyer, Comparing satellite- to ground-based automated and manual cloud coverage observations - a case study, 2015

3: S. Mahajan, B. Fataniya, Cloud detection methodologies: variants and development-a review, 2019

4: S. Dev, S. Manandhar, Y. Lee, S. Winkler, Multi-label cloud segmentation using a deep network, 2019

5: S. Mohajerani, P. Saeedi, Cloud and cloud shadow segmentation for remote sensing imagery via filtered jaccard loss function and parametric augmentation, 2021

6: K. Zheng, J. Li, L. Ding, J. Yang, X. Zhang, X. Zhang, Cloud and snow segmentation in satellite images using encoder-decoder deep convolutional neural networks, 2021

7: Q. Li, W. Lu, J. Yang, A hybrid thresholding algorithm for cloud detection on ground-based color images, 2011

8: R. Shen, Y. Wang, R. Xing, D. Hua, M. Ma, Study on ultra-short-time power forecast of photovoltaic system based on ground-based cloud image recognition and key impact factors, 2020

9: X. Li, Z. Lu, Q. Zhou, Z. Xu, A cloud detection algorithm with reduction of sunlight interference in ground-based sky images, 2019

10: S. Dev, Y. H. Lee, S. Winkler, Color-based segmentation of sky/cloud images from ground-based cameras, 2016

11: S. Dev, Y. H. Lee, S. Winkler, Systematic study of color spaces and components for the segmentation of sky/cloud images, 2017

12: G. Terren-Serano, M. Martinez-Ramon, Comparative analysis of methods for cloud segmentation in ground-based infrared images, 2021

13: M. Hasenbalg, P. Kuhn, S. Wilbert, B. Nouri, A. Kazantzidis, Benchmarking of six cloud segmentation algorithms for ground based all-sky imagers, 2020

14: J. Long, E. Shelhammer, T. Darrel, Fully convolutional networks for semantic segmentation, 2015

15: K. Simonyan and A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2015

16: O. Ronneberger, P. Fischer, T. Brox, U-net: convolutional networks for biomedical image segmentation, 2015

17: S. Dev, A. Nautiyal, Y. H. Lee, S. Winkler, CloudSegNet: a deep network for nychttemeron cloud image segmentation, 2019

18: Q. Song, Z. Cui, P. Liu, An efficient solution for semantic segmentation of three ground-based cloud datasets, 2020

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

20: W. Xie, D. Liu, M. Yang, S. Chen, B. Wang, Z. Wang, Y. Xia, Y. Liu, Y. Wang, C. Zhang, Segcloud: a novel cloud image segmentation model using a deep convolutional neural network for ground-based all-sky-view camera observation, 2020

21: D. Hendrycks, K. Gimpel, Gaussian error linear units (gelus), 2020

22: L. C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, A. L. Yuille, Deeplab: Semantic image segmantation with deep convolutional nets, atrous convolution, and fully connected crfs, 2017

23: O. Oktay, J. Schlemper, L. Folgoc, M. Lee, M. Heinrich, K. Misawa, K. Mori, S. McDonagh, N. Hammerla, B. Kainz, B. Glocker, D. Rueckert, Attention u-net: learning where to look for the pancreas, 2018

24: T. Khanh, D. Dao, N. Ho, H. Yang, E. Baek, G. Lee, S. Kim, S. Yoo, Enhancing u-net with spatial-channel attention gate for abnormal tissue segmentation in medical imaging, 2020

25: B. Zhao, J. Soraghan, G. Caterina, D. Grose, Segmentation of head and neck tumours using modified u-net, 2019

33: M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L. Chen, MobileNetsV2: inverted residuals and linear bottlenecks, 2019

34: G. Huang, Z. Liu, L. Maaten, K. Weinberger, Densely connected convolutional networks, 2018

26: F. Ren, W. Liu, G. Wu, Feature reuse residual networks for insect pest recognition, 2019

27: K. He, X. Zhang, S. Ren, J. Sun, Identity mappings in deep residual networks, 2016

28: C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, 2014

29: C. Szegedy, V. Vanhoucke, S. Ioffe, J Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, 2015

30: C. Szegedy, S. Ioffe, V. Vanhoucke, A. Alemi, Inception-v4, inception-resnet and the impact of residual connections on learning, 2016

31: F. Chollet, Xception: deep learning with depthwise separable convolutions, 2016

32: A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: efficient convolutional neural networks for mobile vision applications, 2017

35: B. Zoph, V. Vasudevan, J. Shlens, Q. Le, Learning transferable architectures for scalable image recognition, 2018

36: B. Zoph, Q. Le, Neural architecture search with reinforcement learning, 2017

37: M. Tan, Q. Le, EfficientNet: rethinking model scaling for convolutional neural networks, 2020

38: M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, Q. Le, MnasNet: platform-aware neural architecture search for mobile, 2019

39: J. Hu, L. Shen, S. Albanie, G. Sun, E. Wu, Squeeze and excitation networks, 2019

40: G. Hinton, N. srivastava, K. Swersky, Neural networks for machine learning, Lecture 6a, overview of mini-batch gradient descent, 2018

41: J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, 2011

42: M. Zeiler, Adadelta: an adaptive learning rate method, 2012

43: D. Kingma, J. Ba, Adam: a method for stochastic optimization, 2017

44: H. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, Ad click prediction: a view from the trenches, 2013

45: T. Dozat, Incorporating nesterov momentum into adam,

46: I. Loshchilov, F. Hutter, Decoupled weight decay regularization, 2019

47: S. Ravi, T. Dinh, V. Lokhande, V. Singh, Constrained deep learning using conditional gradient and applications in computer vision, 2018

48: Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, C. Hsieh, Large batch optimization for deep learning: training bert in 76 minutes, 2020

49: Y. You, I. Gitman, B. Ginsbourg, Large batch training of convolutional networks, 2017

50: J. Duchi, Y. Singer, Efficient online and batch learning using forward backward splitting, 2009

51: L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, J. Han, On the variance of the adaptive learning rate and beyond, 2019

52: S. Reddi, M. Zaheer, D. Sachan, S. Kale, S. Kumar, Adaptive methods for nonconvex optimization, 2018

53: S. Jadon, A survey of loss functions for semantic segmentation, 2020

54: J. Ma, Segmentation loss odyssey, 2020

55: Y. Guo, X. Cao, B. Liu, M. Gao, Cloud detection for satellite imagery using attention-based u-net convolutional neural network, 2020

56: A. Buslaev, A. Parinov, E. Khvedchenya, V. I. Iglovikov, A. A. Kalinin, Albumentation: fast and flexible image augmentations, 2018

7 Appendix

7.1 Experimental results

7.1.1 Dataset results

dataset ratio	Net	set	f1-score	IoU	precision	recall	specifisity	accuracy
60/20/20	Unet	training	0.9192	0.8592	0.9200	0.9340	0.8895	0.9285
60/20/20	Unet	validation	0.9114	0.8474	0.9123	0.9297	0.8517	0.9240
60/20/20	Unet	test	0.9150	0.8546	0.9187	0.9305	0.8564	0.9227
60/20/20	A_Unet	training	0.9173	0.8566	0.9291	0.9215	0.9092	0.9304
60/20/20	A_Unet	validation	0.9110	0.8485	0.9203	0.9233	0.8775	0.9248
60/20/20	A_Unet	test	0.9161	0.8557	0.9253	0.9247	0.8831	0.9253
60/20/20	D_Unet	training	0.9204	0.8613	0.9325	0.9236	0.9119	0.9315
60/20/20	D_Unet	validation	0.9133	0.8504	0.9262	0.9190	0.8886	0.9259
60/20/20	D_Unet	test	0.9158	0.8564	0.9319	0.9184	0.8945	0.9264
60/20/20	W_Unet	training	0.9112	0.8475	0.9257	0.9171	0.8909	0.9238
60/20/20	W_Unet	validation	0.9095	0.8453	0.9194	0.9214	0.8527	0.9222
60/20/20	W_Unet	test	0.9087	0.8467	0.9230	0.9180	0.8590	0.9219
60/20/20	R_Unet	training	0.9089	0.8447	0.9260	0.9139	0.8970	0.9225
60/20/20	R_Unet	validation	0.9053	0.8397	0.9184	0.9167	0.8603	0.9207
60/20/20	R_Unet	test	0.9096	0.8470	0.9269	0.9145	0.8670	0.9202
65/15/20	Unet	training	0.9089	0.8449	0.9180	0.9223	0.8815	0.9233
65/15/20	Unet	validation	0.8982	0.8333	0.9253	0.9042	0.8690	0.9080
65/15/20	Unet	test	0.9179	0.8608	0.9274	0.9305	0.8524	0.9266
65/15/20	A_Unet	training	0.9167	0.8557	0.9085	0.9413	0.8761	0.9296
65/15/20	A_Unet	validation	0.9114	0.8505	0.9184	0.9266	0.8672	0.9165
65/15/20	A_Unet	test	0.9294	0.8757	0.9234	0.9482	0.8684	0.9354
65/15/20	D_Unet	training	0.9208	0.8617	0.9267	0.9299	0.9001	0.9323
65/15/20	D_Unet	validation	0.9086	0.8469	0.9323	0.9102	0.8881	0.9142
65/15/20	D_Unet	test	0.9261	0.8718	0.9358	0.9322	0.8781	0.9317
65/15/20	W_Unet	training	0.9032	0.8358	0.9111	0.9193	0.8644	0.9179
65/15/20	W_Unet	validation	0.8953	0.8282	0.9200	0.9039	0.8544	0.9065
65/15/20	W_Unet	test	0.9160	0.8566	0.9220	0.9315	0.8358	0.9235
65/15/20	R_Unet	training	0.9148	0.8532	0.9310	0.9173	0.9066	0.9287
65/15/20	R_Unet	validation	0.8994	0.8358	0.9356	0.8954	0.8996	0.9103
65/15/20	R_Unet	test	0.9189	0.8618	0.9365	0.9219	0.8808	0.9273
70/10/20	Unet	training	0.9075	0.8427	0.9172	0.9209	0.8893	0.9228
70/10/20	Unet	validation	0.9108	0.8482	0.9126	0.9308	0.8966	0.9241
70/10/20	Unet	test	0.8957	0.8250	0.8986	0.9210	0.8593	0.9105
70/10/20	A_Unet	training	0.9079	0.8445	0.9177	0.9220	0.9040	0.9259
70/10/20	A_Unet	validation	0.9121	0.8499	0.9211	0.9249	0.9205	0.9269
70/10/20	A_Unet	test	0.9023	0.8366	0.9126	0.9186	0.8855	0.9175
70/10/20	D_Unet	training	0.9184	0.8586	0.9315	0.9225	0.9041	0.9306
70/10/20	D_Unet	validation	0.9243	0.8661	0.9288	0.9327	0.9100	0.9307

70/10/20	D_Unet	test	0.9048	0.8382	0.9148	0.9176	0.8759	0.9170
70/10/20	W_Unet	training	0.9084	0.8451	0.9268	0.9138	0.8992	0.9245
70/10/20	W_Unet	validation	0.9152	0.8525	0.9257	0.9221	0.9048	0.9257
70/10/20	W_Unet	test	0.9001	0.8320	0.9133	0.9135	0.8725	0.9128
70/10/20	R_Unet	training	0.9096	0.8465	0.9324	0.9096	0.9027	0.9251
70/10/20	R_Unet	validation	0.9124	0.8495	0.9262	0.9189	0.9063	0.9252
70/10/20	R_Unet	test	0.8974	0.8286	0.9153	0.9079	0.8726	0.9123
70/15/15	Unet	training	0.9161	0.8556	0.9291	0.9219	0.9137	0.9302
70/15/15	Unet	validation	0.9098	0.8469	0.9276	0.9148	0.8875	0.9200
70/15/15	Unet	test	0.9183	0.8596	0.9264	0.9267	0.8962	0.9232
70/15/15	A_Unet	training	0.9167	0.8567	0.9248	0.9270	0.9132	0.9325
70/15/15	A_Unet	validation	0.9093	0.8454	0.9170	0.9241	0.8881	0.9217
70/15/15	A_Unet	test	0.9206	0.8618	0.9350	0.9203	0.9061	0.9254
70/15/15	D_Unet	training	0.9139	0.8518	0.9228	0.9243	0.8953	0.9280
70/15/15	D_Unet	validation	0.9107	0.8469	0.9224	0.9198	0.8630	0.9209
70/15/15	D_Unet	test	0.9150	0.8538	0.9177	0.9297	0.8586	0.9191
70/15/15	W_Unet	training	0.9098	0.8457	0.9175	0.9234	0.8859	0.9247
70/15/15	W_Unet	validation	0.9035	0.8377	0.9179	0.9139	0.8529	0.9172
70/15/15	W_Unet	test	0.9125	0.8494	0.9120	0.9301	0.8510	0.9162
70/15/15	R_Unet	training	0.9124	0.8502	0.9363	0.9096	0.9105	0.9273
70/15/15	R_Unet	validation	0.9051	0.8399	0.9325	0.9025	0.8753	0.9192
70/15/15	R_Unet	test	0.9146	0.8531	0.9309	0.9158	0.8771	0.9197
80/10/10	Unet	training	0.9072	0.8420	0.9254	0.9108	0.9081	0.9240
80/10/10	Unet	validation	0.9009	0.8369	0.9327	0.8998	0.8805	0.9089
80/10/10	Unet	test	0.9154	0.8567	0.9272	0.9264	0.8751	0.9272
80/10/10	A_Unet	training	0.9143	0.8523	0.9251	0.9201	0.9104	0.9299
80/10/10	A_Unet	validation	0.9093	0.8498	0.9412	0.9028	0.8940	0.9188
80/10/10	A_Unet	test	0.9157	0.8551	0.9269	0.9225	0.8898	0.9269
80/10/10	D_Unet	training	0.9125	0.8492	0.9177	0.9262	0.8981	0.9269
80/10/10	D_Unet	validation	0.9058	0.8428	0.9274	0.9112	0.8691	0.9117
80/10/10	D_Unet	test	0.9128	0.8521	0.9127	0.9364	0.8552	0.9252
80/10/10	W_Unet	training	0.9013	0.8323	0.9142	0.9123	0.8733	0.9168
80/10/10	W_Unet	validation	0.8966	0.8285	0.9298	0.8936	0.8482	0.9040
80/10/10	W_Unet	test	0.9047	0.8400	0.9115	0.9250	0.8353	0.9192
80/10/10	R_Unet	training	0.9128	0.8499	0.9267	0.9180	0.9099	0.9289
80/10/10	R_Unet	validation	0.9056	0.8422	0.9373	0.8995	0.8801	0.9133
80/10/10	R_Unet	test	0.9134	0.8524	0.9236	0.9256	0.8713	0.9257

7.1.2 Optimizer results

Optimizer	set	f1-score	IoU	precision	recall	specifisity	accuracy
Adadelta	training	0.4891	0.3340	0.5504	0.4861	0.5189	0.5126
Adadelta	validation	0.4983	0.3425	0.5646	0.4893	0.5174	0.5129
Adadelta	test	0.4991	0.3422	0.5646	0.4858	0.5207	0.5133
Adagrad	training	0.7696	0.6507	0.7487	0.8270	0.6710	0.8622
Adagrad	validation	0.7715	0.6545	0.7469	0.8324	0.6335	0.8606
Adagrad	test	0.7813	0.6630	0.7551	0.8363	0.6380	0.8617
Adam	training	0.9178	0.8573	0.9187	0.9329	0.8942	0.9306
Adam	validation	0.9111	0.8483	0.9115	0.9324	0.8589	0.9257
Adam	test	0.9142	0.8531	0.9162	0.9314	0.8816	0.9249
Adamax	training	0.9154	0.8534	0.9181	0.9276	0.8877	0.9308
Adamax	validation	0.9031	0.8365	0.9078	0.9214	0.8554	0.9201
Adamax	test	0.9110	0.8478	0.9147	0.9253	0.8620	0.9224
Ftrl	training	0.5020	0.3455	0.5484	0.5153	0.4847	0.5484
Ftrl	validation	0.5092	0.3516	0.5621	0.5153	0.4847	0.5430
Ftrl	test	0.5117	0.3529	0.5618	0.5153	0.4847	0.5388
Nadam	training	0.9146	0.8540	0.9250	0.9221	0.9086	0.9299
Nadam	validation	0.9061	0.8429	0.9175	0.9205	0.8785	0.9234
Nadam	test	0.9157	0.8550	0.9222	0.9264	0.8866	0.9254
RMSprop	training	0.9209	0.8631	0.9336	0.9237	0.9120	0.9333
RMSprop	validation	0.9101	0.8488	0.9244	0.9201	0.8792	0.9243
RMSprop	test	0.9215	0.8632	0.9268	0.9308	0.8877	0.9285
SGD	training	0.6244	0.4795	0.5835	0.7280	0.3871	0.6508
SGD	validation	0.6388	0.4969	0.5955	0.7396	0.3707	0.6484
SGD	test	0.6373	0.4915	0.5964	0.7298	0.3797	0.6472
AdamW	training	0.9110	0.8466	0.9370	0.9034	0.9152	0.9262
AdamW	validation	0.9075	0.8426	0.9304	0.9071	0.8846	0.9236
AdamW	test	0.9113	0.8483	0.9333	0.9078	0.8896	0.9226
ConditionalGradient	training	0.4954	0.3391	0.5484	0.5025	0.4975	0.5484
ConditionalGradient	validation	0.5024	0.3450	0.5621	0.5025	0.4975	0.5430
ConditionalGradient	test	0.5049	0.3463	0.5618	0.5025	0.4975	0.5388
LAMB	training	0.9178	0.8574	0.9261	0.9266	0.9077	0.9297
LAMB	validation	0.9076	0.8459	0.9159	0.9262	0.8770	0.9227
LAMB	test	0.9177	0.8572	0.9233	0.9282	0.8917	0.9255
LazyAdam	training	0.9203	0.8608	0.9287	0.9264	0.9080	0.9321
LazyAdam	validation	0.9120	0.8494	0.9224	0.9223	0.8832	0.9248
LazyAdam	test	0.9156	0.8549	0.9212	0.9281	0.8714	0.9238
ProximalAdagrad	training	0.6645	0.5206	0.6820	0.6784	0.5602	0.7850
ProximalAdagrad	validation	0.6685	0.5261	0.6861	0.6869	0.5315	0.7755
ProximalAdagrad	test	0.6760	0.5314	0.6896	0.6893	0.5347	0.7798
RectifiedAdam	training	0.9108	0.8454	0.9277	0.9100	0.9040	0.9265
RectifiedAdam	validation	0.9045	0.8381	0.9200	0.9113	0.8735	0.9228
RectifiedAdam	test	0.9086	0.8445	0.9262	0.9103	0.8799	0.9216
SGDW	training	0.5946	0.4415	0.5917	0.6506	0.4770	0.7109
SGDW	validation	0.6049	0.4528	0.6038	0.6553	0.4687	0.7123
SGDW	test	0.6079	0.4538	0.6050	0.6536	0.4692	0.7128
Yogi	training	0.9200	0.8593	0.9215	0.9310	0.9018	0.9343
Yogi	validation	0.9092	0.8437	0.9116	0.9264	0.8745	0.9243
Yogi	test	0.9122	0.8491	0.9154	0.9273	0.8737	0.9229

7.1.3 Loss function results

loss	set	f1-score	IoU	precision	recall	specificity	accuracy
Binary Cross Entropy	training	0.8812	0.7996	0.8825	0.8955	0.8736	0.9336
Binary Cross Entropy	validation	0.8785	0.7968	0.8821	0.8954	0.8532	0.9250
Binary Cross Entropy	test	0.8787	0.7967	0.8813	0.8942	0.8518	0.9225
Dise	training	0.9145	0.8528	0.9331	0.9146	0.9130	0.9264
Dise	validation	0.9080	0.8454	0.9262	0.9152	0.8876	0.9224
Dise	test	0.9143	0.8537	0.9327	0.9155	0.8952	0.9241
Jaccard	training	0.9147	0.8525	0.9357	0.9113	0.9062	0.9254
Jaccard	validation	0.9096	0.8464	0.9288	0.9136	0.8787	0.9227
Jaccard	test	0.9139	0.8525	0.9320	0.9144	0.8808	0.9232
power Jaccard (p=1.1)	training	0.9175	0.8578	0.9355	0.9158	0.9178	0.9287
power Jaccard (p=1.1)	validation	0.9096	0.8480	0.9274	0.9140	0.8908	0.9229
power Jaccard (p=1.1)	test	0.9160	0.8558	0.9307	0.9182	0.8937	0.9237
Fbeta score (b=1.2)	training	0.9222	0.8652	0.9201	0.9410	0.8900	0.9319
Fbeta score (b=1.2)	validation	0.9094	0.8477	0.9125	0.9314	0.8608	0.9219
Fbeta score (b=1.2)	test	0.9174	0.8582	0.9097	0.9444	0.8604	0.9242
Tversky	training	0.6756	0.5484	0.5484	1.0000	0.0000	0.5484
Tversky	validation	0.6874	0.5621	0.5621	1.0000	0.0000	0.5430
Tversky	test	0.6900	0.5618	0.5618	1.0000	0.0000	0.5388
logcoshDise	training	0.9213	0.8624	0.9414	0.9162	0.9229	0.9300
logcoshDise	validation	0.9149	0.8541	0.9346	0.9153	0.8987	0.9257
logcoshDise	test	0.9164	0.8556	0.9347	0.9158	0.9123	0.9246
Cross Entropy Dise	training	0.8961	0.8212	0.8909	0.9163	0.8694	0.9327
Cross Entropy Dise	validation	0.8911	0.8145	0.8860	0.9160	0.8430	0.9234
Cross Entropy Dise	test	0.8916	0.8150	0.8874	0.9129	0.8470	0.9244
Cross Entropy logcoshJaccard	training	0.8801	0.7975	0.8858	0.8907	0.8795	0.9298
Cross Entropy logcoshJaccard	validation	0.8761	0.7933	0.8815	0.8926	0.8563	0.9236
Cross Entropy logcoshJaccard	test	0.8794	0.7972	0.8850	0.8913	0.8551	0.9224
Fbeta score (b=0.9) power Jaccard (p=1.1)	training	0.9196	0.8614	0.9287	0.9247	0.9124	0.9301
Fbeta score (b=0.9) power Jaccard (p=1.1)	validation	0.9107	0.8483	0.9224	0.9189	0.8850	0.9227
Fbeta score (b=0.9) power Jaccard (p=1.1)	test	0.9150	0.8545	0.9218	0.9253	0.8981	0.9239
logcoshJaccard	training	0.9204	0.8613	0.9219	0.9334	0.8995	0.9289
logcoshJaccard	validation	0.9130	0.8502	0.9120	0.9327	0.8720	0.9236
logcoshJaccard	test	0.9138	0.8520	0.9137	0.9326	0.8706	0.9216
Fbeta score (b=1.4) Tversky	training	0.8847	0.8122	0.8369	0.9727	0.7555	0.8919
Fbeta score (b=1.4) Tversky	validation	0.8795	0.8044	0.8299	0.9718	0.7116	0.8854
Fbeta score (b=1.4) Tversky	test	0.8817	0.8053	0.8276	0.9742	0.7077	0.8828

7.1.4 Results of Imagenets with preprocessed inputs having pretrained & not trainable encoder

Network	Set	F1 score	IoU	precision	recall	specificity	accuracy
VGG16	training	0.6757	0.5445	0.5536	0.9814	0.0445	0.5642
VGG16	validation	0.6870	0.5573	0.5671	0.9808	0.0431	0.5600
VGG16	test	0.6880	0.5555	0.5656	0.9781	0.0398	0.5528
VGG16_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
VGG16_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
VGG16_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
VGG19	training	0.6748	0.5473	0.5481	0.9978	0.0017	0.5478
VGG19	validation	0.6866	0.5610	0.5618	0.9981	0.0011	0.5421
VGG19	test	0.6892	0.5608	0.5616	0.9977	0.0019	0.5384
VGG19_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
VGG19_linked	validation	0.6873	0.5619	0.5619	1.0000	0.0000	0.5429
VGG19_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet50	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet50	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet50	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet50_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet50_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet50_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet101	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet101	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet101	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet101_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet101_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet101_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet152	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet152	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet152	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet152_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet152_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet152_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet50V2	training	0.7121	0.5828	0.6096	0.9352	0.2920	0.6661
ResNet50V2	validation	0.7174	0.5899	0.6175	0.9311	0.2724	0.6582
ResNet50V2	test	0.7206	0.5907	0.6142	0.9318	0.2559	0.6542
ResNet50V2_linked	training	0.6754	0.5482	0.5482	1.0000	0.0001	0.5482
ResNet50V2_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0001	0.5429
ResNet50V2_linked	test	0.6898	0.5617	0.5617	1.0000	0.0001	0.5387
ResNet101V2	training	0.7045	0.5736	0.6152	0.8998	0.3321	0.6673
ResNet101V2	validation	0.7112	0.5827	0.6222	0.8973	0.3094	0.6621
ResNet101V2	test	0.7050	0.5731	0.6197	0.8793	0.3158	0.6505
ResNet101V2_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet101V2_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet101V2_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet152V2	training	0.6832	0.5508	0.5820	0.9171	0.2164	0.6141
ResNet152V2	validation	0.6885	0.5565	0.5914	0.9084	0.2025	0.6062
ResNet152V2	test	0.6958	0.5621	0.5926	0.9204	0.2003	0.6034
ResNet152V2_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet152V2_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429

ResNet152V2_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
InceptionV3	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
InceptionV3	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
InceptionV3	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
InceptionV3_linked	training	0.5526	0.4424	0.9850	0.4516	0.9722	0.7029
InceptionV3_linked	validation	0.5836	0.4778	0.9826	0.4873	0.9694	0.7102
InceptionV3_linked	test	0.5674	0.4602	0.9860	0.4690	0.9597	0.7045
InceptionResNetV2	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
InceptionResNetV2	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
InceptionResNetV2	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
InceptionResNetV2_linked	training	0.7212	0.6022	0.6053	0.9955	0.1818	0.6390
InceptionResNetV2_linked	validation	0.7271	0.6096	0.6142	0.9938	0.1589	0.6297
InceptionResNetV2_linked	test	0.7279	0.6062	0.6085	0.9941	0.1538	0.6255
Xception	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
Xception	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
Xception	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
Xception_linked	training	0.6481	0.5360	0.9473	0.5729	0.9038	0.7597
Xception_linked	validation	0.6765	0.5719	0.9381	0.6127	0.8835	0.7719
Xception_linked	test	0.6639	0.5536	0.9494	0.5896	0.8734	0.7594
MobileNet	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
MobileNet	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
MobileNet	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
MobileNet_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
MobileNet_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
MobileNet_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
MobileNetV2	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
MobileNetV2	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
MobileNetV2	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
MobileNetV2_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
MobileNetV2_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
MobileNetV2_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
DenseNet121	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
DenseNet121	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
DenseNet121	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
DenseNet121_linked	training	0.6971	0.5720	0.5809	0.9678	0.1742	0.6186
DenseNet121_linked	validation	0.7074	0.5836	0.5915	0.9704	0.1591	0.6149
DenseNet121_linked	test	0.7123	0.5868	0.5942	0.9707	0.1675	0.6134
DenseNet169	training	0.6753	0.5481	0.5482	0.9995	0.0007	0.5482
DenseNet169	validation	0.6872	0.5618	0.5620	0.9995	0.0007	0.5429
DenseNet169	test	0.6897	0.5616	0.5617	0.9995	0.0007	0.5386
DenseNet169_linked	training	0.6925	0.5664	0.5684	0.9934	0.0968	0.5987
DenseNet169_linked	validation	0.7034	0.5792	0.5813	0.9931	0.0909	0.5937
DenseNet169_linked	test	0.7066	0.5795	0.5810	0.9938	0.0906	0.5915
DenseNet201	training	0.6755	0.5473	0.5497	0.9950	0.0125	0.5519
DenseNet201	validation	0.6875	0.5612	0.5635	0.9952	0.0125	0.5474
DenseNet201	test	0.6897	0.5605	0.5630	0.9951	0.0111	0.5430
DenseNet201_linked	training	0.7768	0.6766	0.9252	0.7305	0.8646	0.8394
DenseNet201_linked	validation	0.7864	0.6931	0.9183	0.7527	0.8125	0.8478
DenseNet201_linked	test	0.7929	0.6949	0.9246	0.7502	0.8213	0.8413
NASNetMobile	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482

NASNetMobile	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
NASNetMobile	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
NASNetMobile_linked	training	0.4518	0.3503	0.9674	0.3522	0.9904	0.6569
NASNetMobile_linked	validation	0.4837	0.3816	0.9654	0.3839	0.9904	0.6657
NASNetMobile_linked	test	0.4611	0.3620	0.9736	0.3645	0.9795	0.6572
EfficientNetB0	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB0	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB0	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB0_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB0_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB0_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB1	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB1	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB1	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB1_linked	training	0.6754	0.5482	0.5482	0.9999	0.0002	0.5482
EfficientNetB1_linked	validation	0.6873	0.5619	0.5620	0.9998	0.0002	0.5429
EfficientNetB1_linked	test	0.6898	0.5616	0.5617	0.9999	0.0002	0.5386
EfficientNetB2	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB2	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB2	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB2_linked	training	0.6754	0.5481	0.5482	0.9998	0.0001	0.5481
EfficientNetB2_linked	validation	0.6873	0.5619	0.5619	0.9999	0.0001	0.5429
EfficientNetB2_linked	test	0.6898	0.5616	0.5617	0.9998	0.0001	0.5386
EfficientNetB3	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB3	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB3	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB3_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB3_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB3_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB4	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB4	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB4	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB4_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB4_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB4_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB5	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB5	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB5	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB5_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB5_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB5_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB6	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB6	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB6	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB6_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB6_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB6_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB7	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB7	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB7	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386

EfficientNetB7_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB7_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB7_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386

7.1.5 Results of Imagenets without preprocessed inputs having pretrained & trainable encoder

Network	Set	F1 score	IoU	precision	recall	specificity	accuracy
VGG16	training	0.9302	0.8755	0.9311	0.9401	0.8734	0.9333
VGG16	validation	0.8990	0.8267	0.9009	0.9154	0.7977	0.9080
VGG16	test	0.8998	0.8287	0.9016	0.9164	0.8056	0.9091
VGG16_linked	training	0.9266	0.8699	0.9310	0.9346	0.8864	0.9308
VGG16_linked	validation	0.9103	0.8449	0.9169	0.9210	0.8402	0.9187
VGG16_linked	test	0.9105	0.8469	0.9184	0.9217	0.8425	0.9190
VGG19	training	0.9454	0.8993	0.9422	0.9535	0.8988	0.9465
VGG19	validation	0.9018	0.8306	0.9001	0.9189	0.8190	0.9125
VGG19	test	0.9046	0.8355	0.9052	0.9195	0.8223	0.9132
VGG19_linked	training	0.9285	0.8728	0.9367	0.9318	0.9002	0.9332
VGG19_linked	validation	0.9134	0.8500	0.9253	0.9183	0.8568	0.9226
VGG19_linked	test	0.9128	0.8506	0.9249	0.9196	0.8572	0.9218
ResNet50	training	0.9680	0.9386	0.9664	0.9702	0.9342	0.9684
ResNet50	validation	0.9074	0.8403	0.9180	0.9141	0.8546	0.9171
ResNet50	test	0.9088	0.8428	0.9219	0.9121	0.8658	0.9187
ResNet50_linked	training	0.9671	0.9371	0.9659	0.9693	0.9349	0.9687
ResNet50_linked	validation	0.9165	0.8554	0.9250	0.9247	0.8727	0.9250
ResNet50_linked	test	0.9190	0.8601	0.9313	0.9228	0.8807	0.9266
ResNet101	training	0.9675	0.9378	0.9657	0.9700	0.9339	0.9689
ResNet101	validation	0.9072	0.8397	0.9178	0.9136	0.8488	0.9178
ResNet101	test	0.9082	0.8423	0.9200	0.9127	0.8575	0.9182
ResNet101_linked	training	0.9603	0.9251	0.9481	0.9747	0.9076	0.9618
ResNet101_linked	validation	0.9156	0.8530	0.9108	0.9367	0.8367	0.9215
ResNet101_linked	test	0.9159	0.8546	0.9158	0.9325	0.8503	0.9239
ResNet152	training	0.9336	0.8797	0.9282	0.9450	0.8646	0.9379
ResNet152	validation	0.9019	0.8318	0.9035	0.9162	0.8005	0.9092
ResNet152	test	0.9000	0.8294	0.8979	0.9213	0.7942	0.9073
ResNet152_linked	training	0.9570	0.9191	0.9513	0.9649	0.9123	0.9592
ResNet152_linked	validation	0.9140	0.8508	0.9200	0.9243	0.8680	0.9220
ResNet152_linked	test	0.9172	0.8562	0.9286	0.9206	0.8794	0.9234
ResNet50V2	training	0.9651	0.9334	0.9573	0.9739	0.9204	0.9661
ResNet50V2	validation	0.9062	0.8378	0.9102	0.9182	0.8415	0.9165
ResNet50V2	test	0.9072	0.8396	0.9127	0.9172	0.8460	0.9155
ResNet50V2_linked	training	0.9559	0.9177	0.9615	0.9531	0.9284	0.9608
ResNet50V2_linked	validation	0.9160	0.8548	0.9311	0.9177	0.8740	0.9254
ResNet50V2_linked	test	0.9154	0.8543	0.9350	0.9131	0.8834	0.9244
ResNet101V2	training	0.9682	0.9390	0.9646	0.9724	0.9341	0.9695
ResNet101V2	validation	0.9079	0.8403	0.9140	0.9166	0.8488	0.9180
ResNet101V2	test	0.9083	0.8422	0.9160	0.9161	0.8488	0.9172
ResNet101V2_linked	training	0.9732	0.9483	0.9715	0.9754	0.9416	0.9742
ResNet101V2_linked	validation	0.9170	0.8560	0.9299	0.9201	0.8681	0.9278
ResNet101V2_linked	test	0.9171	0.8571	0.9364	0.9147	0.8804	0.9265
ResNet152V2	training	0.9684	0.9395	0.9643	0.9731	0.9351	0.9698
ResNet152V2	validation	0.9094	0.8421	0.9111	0.9218	0.8451	0.9178
ResNet152V2	test	0.9086	0.8424	0.9154	0.9177	0.8515	0.9164
ResNet152V2_linked	training	0.9733	0.9485	0.9720	0.9750	0.9463	0.9746

ResNet152V2_linked	validation	0.9177	0.8571	0.9289	0.9223	0.8775	0.9274
ResNet152V2_linked	test	0.9194	0.8606	0.9366	0.9171	0.8916	0.9284
InceptionV3	training	0.9364	0.8833	0.9277	0.9484	0.8708	0.9416
InceptionV3	validation	0.8898	0.8121	0.8798	0.9177	0.7835	0.9025
InceptionV3	test	0.8934	0.8174	0.8908	0.9132	0.8058	0.9017
InceptionV3_linked	training	0.9630	0.9299	0.9567	0.9710	0.9132	0.9648
InceptionV3_linked	validation	0.9208	0.8622	0.9297	0.9285	0.8637	0.9290
InceptionV3_linked	test	0.9197	0.8619	0.9332	0.9234	0.8772	0.9277
InceptionResNetV2	training	0.9517	0.9096	0.9488	0.9561	0.9051	0.9551
InceptionResNetV2	validation	0.8990	0.8261	0.9011	0.9143	0.8235	0.9095
InceptionResNetV2	test	0.8961	0.8236	0.9028	0.9061	0.8379	0.9067
InceptionResNetV2_linked	training	0.9548	0.9168	0.9538	0.9577	0.9191	0.9564
InceptionResNetV2_linked	validation	0.9112	0.8486	0.9250	0.9142	0.8707	0.9169
InceptionResNetV2_linked	test	0.9154	0.8548	0.9320	0.9147	0.8897	0.9219
Xception	training	0.9562	0.9174	0.9372	0.9771	0.8932	0.9584
Xception	validation	0.9065	0.8376	0.8957	0.9331	0.8178	0.9141
Xception	test	0.9086	0.8422	0.8986	0.9350	0.8280	0.9162
Xception_linked	training	0.9724	0.9469	0.9713	0.9741	0.9454	0.9740
Xception_linked	validation	0.9202	0.8612	0.9276	0.9292	0.8682	0.9277
Xception_linked	test	0.9246	0.8689	0.9364	0.9280	0.8855	0.9324
MobileNet	training	0.9496	0.9058	0.9432	0.9582	0.9016	0.9525
MobileNet	validation	0.8997	0.8271	0.8999	0.9150	0.8230	0.9095
MobileNet	test	0.9030	0.8329	0.9022	0.9200	0.8249	0.9117
MobileNet_linked	training	0.9653	0.9339	0.9653	0.9664	0.9360	0.9673
MobileNet_linked	validation	0.9134	0.8499	0.9266	0.9160	0.8720	0.9225
MobileNet_linked	test	0.9140	0.8524	0.9284	0.9167	0.8780	0.9223
MobileNetV2	training	0.9221	0.8600	0.9563	0.8964	0.9293	0.9287
MobileNetV2	validation	0.8886	0.8109	0.9306	0.8660	0.8778	0.9034
MobileNetV2	test	0.8859	0.8079	0.9354	0.8587	0.8874	0.8999
MobileNetV2_linked	training	0.9530	0.9126	0.9556	0.9541	0.9079	0.9556
MobileNetV2_linked	validation	0.9161	0.8550	0.9251	0.9249	0.8585	0.9249
MobileNetV2_linked	test	0.9183	0.8587	0.9322	0.9212	0.8708	0.9262
DenseNet121	training	0.9557	0.9164	0.9586	0.9540	0.9240	0.9577
DenseNet121	validation	0.9082	0.8406	0.9216	0.9098	0.8559	0.9188
DenseNet121	test	0.9090	0.8424	0.9233	0.9098	0.8644	0.9175
DenseNet121_linked	training	0.9671	0.9371	0.9696	0.9653	0.9467	0.9688
DenseNet121_linked	validation	0.9197	0.8609	0.9362	0.9198	0.8926	0.9284
DenseNet121_linked	test	0.9215	0.8632	0.9385	0.9189	0.9054	0.9290
DenseNet169	training	0.9611	0.9260	0.9520	0.9712	0.9141	0.9624
DenseNet169	validation	0.9074	0.8396	0.9039	0.9270	0.8323	0.9156
DenseNet169	test	0.9119	0.8470	0.9121	0.9262	0.8526	0.9191
DenseNet169_linked	training	0.9454	0.8999	0.9501	0.9461	0.9288	0.9508
DenseNet169_linked	validation	0.9187	0.8584	0.9293	0.9231	0.8888	0.9272
DenseNet169_linked	test	0.9213	0.8631	0.9357	0.9210	0.9030	0.9302
DenseNet201	training	0.9610	0.9259	0.9607	0.9621	0.9298	0.9625
DenseNet201	validation	0.9062	0.8386	0.9193	0.9103	0.8702	0.9176
DenseNet201	test	0.9094	0.8437	0.9254	0.9092	0.8750	0.9184
DenseNet201_linked	training	0.9474	0.9028	0.9467	0.9523	0.9041	0.9508
DenseNet201_linked	validation	0.9146	0.8522	0.9219	0.9249	0.8477	0.9245
DenseNet201_linked	test	0.9163	0.8556	0.9262	0.9231	0.8547	0.9250

NASNetMobile	training	0.8861	0.8068	0.8794	0.9112	0.8710	0.8968
NASNetMobile	validation	0.8655	0.7775	0.8717	0.8841	0.8517	0.8773
NASNetMobile	test	0.8721	0.7875	0.8735	0.8933	0.8491	0.8810
NASNetMobile_linked	training	0.9418	0.8929	0.9669	0.9222	0.9503	0.9453
NASNetMobile_linked	validation	0.9166	0.8550	0.9533	0.8965	0.9189	0.9251
NASNetMobile_linked	test	0.9111	0.8468	0.9454	0.8940	0.9168	0.9189
EfficientNetB0	training	0.8460	0.7645	0.9035	0.8468	0.8250	0.8645
EfficientNetB0	validation	0.8083	0.7146	0.8885	0.8110	0.7972	0.8385
EfficientNetB0	test	0.8237	0.7343	0.8801	0.8353	0.7711	0.8457
EfficientNetB0_linked	training	0.8598	0.7803	0.8661	0.8999	0.8610	0.8823
EfficientNetB0_linked	validation	0.8659	0.7876	0.8689	0.9044	0.8368	0.8857
EfficientNetB0_linked	test	0.8723	0.7940	0.8763	0.9012	0.8438	0.8864
EfficientNetB1	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB1	validation	0.6874	0.5620	0.5621	0.9997	0.0007	0.5433
EfficientNetB1	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB1_linked	training	0.7915	0.6766	0.8520	0.7856	0.7869	0.8169
EfficientNetB1_linked	validation	0.8000	0.6882	0.8532	0.7996	0.7527	0.8209
EfficientNetB1_linked	test	0.8039	0.6914	0.8557	0.7961	0.7625	0.8225
EfficientNetB2	training	0.9160	0.8528	0.8910	0.9559	0.7921	0.9148
EfficientNetB2	validation	0.8858	0.8090	0.8701	0.9296	0.7402	0.8893
EfficientNetB2	test	0.8780	0.7990	0.8566	0.9302	0.7239	0.8772
EfficientNetB2_linked	training	0.8755	0.7984	0.8605	0.9318	0.7888	0.8762
EfficientNetB2_linked	validation	0.8611	0.7814	0.8666	0.9063	0.7690	0.8671
EfficientNetB2_linked	test	0.8636	0.7817	0.8466	0.9267	0.7357	0.8584
EfficientNetB3	training	0.7485	0.6241	0.6370	0.9773	0.2158	0.6816
EfficientNetB3	validation	0.7488	0.6251	0.6396	0.9702	0.1937	0.6730
EfficientNetB3	test	0.7510	0.6270	0.6414	0.9691	0.2030	0.6717
EfficientNetB3_linked	training	0.9159	0.8561	0.9142	0.9370	0.8454	0.9205
EfficientNetB3_linked	validation	0.8965	0.8248	0.8951	0.9245	0.7969	0.9010
EfficientNetB3_linked	test	0.8911	0.8227	0.8905	0.9253	0.7940	0.8994
EfficientNetB4	training	0.7629	0.6690	0.9405	0.7104	0.9020	0.8080
EfficientNetB4	validation	0.7325	0.6302	0.9189	0.6854	0.8850	0.7862
EfficientNetB4	test	0.7493	0.6450	0.9305	0.6931	0.8792	0.8018
EfficientNetB4_linked	training	0.8450	0.7560	0.9594	0.7833	0.9430	0.8704
EfficientNetB4_linked	validation	0.8297	0.7365	0.9481	0.7736	0.9204	0.8551
EfficientNetB4_linked	test	0.8438	0.7561	0.9569	0.7847	0.9319	0.8665
EfficientNetB5	training	0.9435	0.8962	0.9345	0.9572	0.8862	0.9482
EfficientNetB5	validation	0.9022	0.8327	0.8943	0.9315	0.8172	0.9107
EfficientNetB5	test	0.9038	0.8351	0.8955	0.9313	0.8125	0.9099

7.1.6 Results of Imagenets without preprocessed inputs having pretrained & not trainable encoder

Network	Set	F1 score	IoU	precision	recall	specificity	accuracy
VGG16	training	0.8379	0.7339	0.8157	0.8923	0.7203	0.8412
VGG16	validation	0.8250	0.7168	0.8057	0.8749	0.6731	0.8303
VGG16	test	0.8220	0.7108	0.7965	0.8803	0.6481	0.8212
VGG16_linked	training	0.9009	0.8313	0.9093	0.9170	0.8169	0.9040
VGG16_linked	validation	0.8851	0.8075	0.8931	0.9072	0.7509	0.8958
VGG16_linked	test	0.8865	0.8089	0.8931	0.9088	0.7521	0.8947
VGG19	training	0.8234	0.7151	0.7969	0.8903	0.6936	0.8293
VGG19	validation	0.8156	0.7059	0.7889	0.8825	0.6386	0.8180
VGG19	test	0.8197	0.7093	0.7967	0.8778	0.6520	0.8183
VGG19_linked	training	0.8957	0.8243	0.9155	0.9035	0.8268	0.9015
VGG19_linked	validation	0.8836	0.8056	0.9003	0.8982	0.7613	0.8949
VGG19_linked	test	0.8845	0.8071	0.9039	0.8959	0.7712	0.8946
ResNet50	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet50	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet50	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet50_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet50_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet50_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet101	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet101	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet101	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet101_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet101_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet101_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet152	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet152	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet152	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet152_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
ResNet152_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
ResNet152_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
ResNet50V2	training	0.8491	0.7481	0.8092	0.9184	0.6935	0.8492
ResNet50V2	validation	0.7854	0.6633	0.7503	0.8589	0.5823	0.7869
ResNet50V2	test	0.7953	0.6752	0.7573	0.8636	0.5768	0.7896
ResNet50V2_linked	training	0.9229	0.8630	0.9176	0.9420	0.8220	0.9208
ResNet50V2_linked	validation	0.8777	0.7940	0.8769	0.9069	0.7243	0.8881
ResNet50V2_linked	test	0.8808	0.7994	0.8777	0.9099	0.7346	0.8884
ResNet101V2	training	0.8514	0.7526	0.8192	0.9130	0.7239	0.8569
ResNet101V2	validation	0.7827	0.6592	0.7563	0.8423	0.6151	0.7881
ResNet101V2	test	0.7822	0.6599	0.7556	0.8374	0.6169	0.7855
ResNet101V2_linked	training	0.9211	0.8602	0.9095	0.9478	0.7988	0.9162
ResNet101V2_linked	validation	0.8750	0.7912	0.8706	0.9103	0.7085	0.8856
ResNet101V2_linked	test	0.8771	0.7933	0.8679	0.9137	0.7087	0.8840
ResNet152V2	training	0.8372	0.7300	0.7939	0.9136	0.6758	0.8348
ResNet152V2	validation	0.7534	0.6203	0.7233	0.8288	0.5532	0.7518
ResNet152V2	test	0.7646	0.6333	0.7309	0.8338	0.5687	0.7555
ResNet152V2_linked	training	0.9031	0.8321	0.8962	0.9319	0.7716	0.9021

ResNet152V2_linked	validation	0.8682	0.7816	0.8668	0.9047	0.6945	0.8787
ResNet152V2_linked	test	0.8724	0.7876	0.8693	0.9088	0.7056	0.8810
InceptionV3	training	0.8793	0.7959	0.8372	0.9470	0.7107	0.8819
InceptionV3	validation	0.7773	0.6541	0.7234	0.8761	0.5144	0.7689
InceptionV3	test	0.7711	0.6498	0.7188	0.8681	0.5198	0.7614
InceptionV3_linked	training	0.9661	0.9354	0.9632	0.9706	0.9237	0.9650
InceptionV3_linked	validation	0.8929	0.8170	0.8986	0.9075	0.8081	0.9048
InceptionV3_linked	test	0.8915	0.8172	0.9037	0.9022	0.8239	0.9018
InceptionResNetV2	training	0.8333	0.7279	0.7870	0.9197	0.6569	0.8329
InceptionResNetV2	validation	0.7779	0.6534	0.7361	0.8688	0.5498	0.7711
InceptionResNetV2	test	0.7729	0.6489	0.7297	0.8571	0.5516	0.7660
InceptionResNetV2_linked	training	0.9270	0.8698	0.9261	0.9401	0.8434	0.9263
InceptionResNetV2_linked	validation	0.8960	0.8223	0.8981	0.9165	0.7650	0.9051
InceptionResNetV2_linked	test	0.8937	0.8201	0.8987	0.9117	0.7759	0.9031
Xception	training	0.7615	0.6413	0.6733	0.9471	0.4214	0.7414
Xception	validation	0.7307	0.6025	0.6423	0.9131	0.3225	0.6907
Xception	test	0.7294	0.5990	0.6448	0.9043	0.3359	0.6779
Xception_linked	training	0.9496	0.9067	0.9437	0.9606	0.8904	0.9472
Xception_linked	validation	0.8943	0.8200	0.8981	0.9107	0.7989	0.9044
Xception_linked	test	0.8980	0.8251	0.9020	0.9122	0.8118	0.9056
MobileNet	training	0.8338	0.7287	0.7843	0.9217	0.6494	0.8347
MobileNet	validation	0.7857	0.6647	0.7368	0.8784	0.5571	0.7822
MobileNet	test	0.7855	0.6620	0.7407	0.8674	0.5693	0.7778
MobileNet_linked	training	0.9203	0.8593	0.9194	0.9360	0.8289	0.9187
MobileNet_linked	validation	0.8805	0.7995	0.8828	0.9050	0.7434	0.8916
MobileNet_linked	test	0.8836	0.8042	0.8868	0.9060	0.7543	0.8919
MobileNetV2	training	0.8151	0.7035	0.7596	0.9206	0.6181	0.8128
MobileNetV2	validation	0.7584	0.6286	0.7119	0.8566	0.5138	0.7487
MobileNetV2	test	0.7659	0.6357	0.7146	0.8586	0.5041	0.7492
MobileNetV2_linked	training	0.9023	0.8303	0.8927	0.9313	0.8257	0.9023
MobileNetV2_linked	validation	0.8409	0.7387	0.8362	0.8779	0.7275	0.8472
MobileNetV2_linked	test	0.8481	0.7477	0.8388	0.8847	0.7228	0.8500
DenseNet121	training	0.8909	0.8102	0.8734	0.9245	0.7710	0.8907
DenseNet121	validation	0.8367	0.7356	0.8322	0.8729	0.6833	0.8506
DenseNet121	test	0.8447	0.7436	0.8351	0.8777	0.6929	0.8503
DenseNet121_linked	training	0.9202	0.8590	0.9210	0.9343	0.8331	0.9195
DenseNet121_linked	validation	0.8799	0.8002	0.8917	0.8990	0.7548	0.8929
DenseNet121_linked	test	0.8874	0.8097	0.8959	0.9043	0.7639	0.8958
DenseNet169	training	0.8910	0.8101	0.8703	0.9276	0.7665	0.8934
DenseNet169	validation	0.8409	0.7400	0.8261	0.8820	0.6656	0.8514
DenseNet169	test	0.8412	0.7389	0.8313	0.8764	0.6776	0.8484
DenseNet169_linked	training	0.9361	0.8840	0.9348	0.9456	0.8696	0.9363
DenseNet169_linked	validation	0.8851	0.8057	0.8898	0.9042	0.7726	0.8963
DenseNet169_linked	test	0.8837	0.8046	0.8945	0.8970	0.7812	0.8960
DenseNet201	training	0.8937	0.8150	0.8746	0.9286	0.7706	0.8953
DenseNet201	validation	0.8295	0.7239	0.8197	0.8684	0.6669	0.8420
DenseNet201	test	0.8317	0.7270	0.8236	0.8662	0.6764	0.8401
DenseNet201_linked	training	0.9434	0.8965	0.9454	0.9482	0.8844	0.9424
DenseNet201_linked	validation	0.8804	0.8001	0.8901	0.8984	0.7762	0.8934
DenseNet201_linked	test	0.8831	0.8030	0.8984	0.8922	0.7925	0.8950

NASNetMobile	training	0.7923	0.6733	0.7253	0.9180	0.5659	0.7865
NASNetMobile	validation	0.7359	0.6016	0.6839	0.8438	0.4849	0.7242
NASNetMobile	test	0.7371	0.6007	0.6825	0.8432	0.4753	0.7173
NASNetMobile_linked	training	0.9196	0.8586	0.9241	0.9309	0.8379	0.9181
NASNetMobile_linked	validation	0.8851	0.8067	0.8922	0.9052	0.7561	0.8948
NASNetMobile_linked	test	0.8870	0.8094	0.8975	0.9015	0.7772	0.8958
EfficientNetB0	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB0	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB0	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB0_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB0_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB0_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB1	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB1	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB1	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB1_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB1_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB1_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB2	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB2	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB2	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB2_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB2_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB2_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB3	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB3	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB3	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB3_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB3_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB3_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB4	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB4	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB4	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB4_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB4_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB4_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB5	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB5	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB5	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB5_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB5_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB5_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB6	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB6	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB6	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB6_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB6_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB6_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB7	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB7	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429

EfficientNetB7	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386
EfficientNetB7_linked	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB7_linked	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB7_linked	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386

7.1.7 Results of Imagenets without preprocessed inputs having untrained & trainable encoder

Network	Set	F1 score	IoU	precision	recall	specificity	accuracy
VGG16	training	0.9122	0.8461	0.9136	0.9272	0.8315	0.9125
VGG16	validation	0.8825	0.8029	0.8870	0.9050	0.7588	0.8944
VGG16	test	0.8890	0.8113	0.8917	0.9095	0.7644	0.8959
VGG16_linked	training	0.8969	0.8259	0.9137	0.9064	0.8418	0.9057
VGG16_linked	validation	0.8943	0.8223	0.9069	0.9098	0.7891	0.9056
VGG16_linked	test	0.8925	0.8205	0.9106	0.9037	0.7958	0.9027
VGG19	training	0.9057	0.8356	0.9022	0.9275	0.8101	0.9050
VGG19	validation	0.8780	0.7956	0.8748	0.9096	0.7312	0.8866
VGG19	test	0.8817	0.8002	0.8766	0.9124	0.7387	0.8867
VGG19_linked	training	0.8987	0.8289	0.9139	0.9093	0.8359	0.9071
VGG19_linked	validation	0.8927	0.8211	0.9056	0.9095	0.7878	0.9047
VGG19_linked	test	0.8931	0.8208	0.9082	0.9069	0.7889	0.9027
ResNet50	training	0.9315	0.8769	0.9375	0.9348	0.8873	0.9355
ResNet50	validation	0.9002	0.8293	0.9103	0.9096	0.8291	0.9131
ResNet50	test	0.9020	0.8330	0.9152	0.9091	0.8370	0.9120
ResNet50_linked	training	0.9140	0.8508	0.9278	0.9182	0.8836	0.9217
ResNet50_linked	validation	0.9102	0.8450	0.9255	0.9146	0.8538	0.9207
ResNet50_linked	test	0.9073	0.8442	0.9242	0.9144	0.8559	0.9173
ResNet101	training	0.9304	0.8753	0.9317	0.9387	0.8727	0.9341
ResNet101	validation	0.8976	0.8254	0.9015	0.9143	0.8101	0.9097
ResNet101	test	0.8965	0.8248	0.9040	0.9112	0.8062	0.9054
ResNet101_linked	training	0.8996	0.8317	0.9218	0.9050	0.8698	0.9130
ResNet101_linked	validation	0.8983	0.8293	0.9146	0.9102	0.8250	0.9147
ResNet101_linked	test	0.8981	0.8309	0.9169	0.9082	0.8246	0.9102
ResNet152	training	0.9138	0.8496	0.9220	0.9220	0.8504	0.9184
ResNet152	validation	0.8896	0.8139	0.8982	0.9057	0.7912	0.9026
ResNet152	test	0.8918	0.8184	0.9051	0.9036	0.8005	0.9009
ResNet152_linked	training	0.9066	0.8401	0.9174	0.9177	0.8729	0.9160
ResNet152_linked	validation	0.9054	0.8386	0.9147	0.9189	0.8366	0.9184
ResNet152_linked	test	0.9045	0.8391	0.9163	0.9171	0.8402	0.9141
ResNet50V2	training	0.9427	0.8954	0.9455	0.9462	0.8972	0.9457
ResNet50V2	validation	0.9023	0.8326	0.9087	0.9149	0.8284	0.9153
ResNet50V2	test	0.9050	0.8381	0.9167	0.9125	0.8452	0.9135
ResNet50V2_linked	training	0.9198	0.8596	0.9319	0.9233	0.8844	0.9258
ResNet50V2_linked	validation	0.9092	0.8434	0.9188	0.9187	0.8424	0.9207
ResNet50V2_linked	test	0.9100	0.8462	0.9266	0.9135	0.8524	0.9185
ResNet101V2	training	0.9397	0.8897	0.9410	0.9445	0.8918	0.9421
ResNet101V2	validation	0.9038	0.8342	0.9083	0.9165	0.8300	0.9147
ResNet101V2	test	0.9018	0.8334	0.9128	0.9107	0.8383	0.9114
ResNet101V2_linked	training	0.9152	0.8528	0.9305	0.9175	0.8849	0.9233
ResNet101V2_linked	validation	0.9088	0.8437	0.9223	0.9163	0.8503	0.9192
ResNet101V2_linked	test	0.9090	0.8459	0.9267	0.9134	0.8611	0.9181
ResNet152V2	training	0.9339	0.8811	0.9360	0.9404	0.8855	0.9374
ResNet152V2	validation	0.9018	0.8315	0.9058	0.9161	0.8253	0.9130
ResNet152V2	test	0.9030	0.8342	0.9132	0.9117	0.8370	0.9114
ResNet152V2_linked	training	0.9087	0.8430	0.9239	0.9144	0.8749	0.9174

ResNet152V2_linked	validation	0.9047	0.8374	0.9162	0.9164	0.8313	0.9175
ResNet152V2_linked	test	0.9056	0.8403	0.9201	0.9146	0.8364	0.9147
InceptionV3	training	0.9073	0.8380	0.8985	0.9306	0.8184	0.9144
InceptionV3	validation	0.8769	0.7939	0.8709	0.9086	0.7599	0.8903
InceptionV3	test	0.8814	0.8001	0.8757	0.9088	0.7717	0.8898
InceptionV3_linked	training	0.9084	0.8438	0.9114	0.9284	0.8561	0.9157
InceptionV3_linked	validation	0.9057	0.8397	0.9065	0.9295	0.8102	0.9149
InceptionV3_linked	test	0.9052	0.8398	0.9103	0.9253	0.8154	0.9113
InceptionResNetV2	training	0.9289	0.8730	0.9211	0.9467	0.8506	0.9341
InceptionResNetV2	validation	0.8956	0.8215	0.8881	0.9245	0.7770	0.9063
InceptionResNetV2	test	0.8956	0.8228	0.8924	0.9180	0.7919	0.9047
InceptionResNetV2_linked	training	0.9212	0.8632	0.9392	0.9202	0.8983	0.9296
InceptionResNetV2_linked	validation	0.9129	0.8507	0.9275	0.9191	0.8586	0.9227
InceptionResNetV2_linked	test	0.9127	0.8518	0.9299	0.9172	0.8629	0.9222
Xception	training	0.9664	0.9359	0.9659	0.9681	0.9390	0.9674
Xception	validation	0.9060	0.8373	0.9119	0.9157	0.8649	0.9164
Xception	test	0.9058	0.8383	0.9152	0.9127	0.8765	0.9142
Xception_linked	training	0.9492	0.9072	0.9534	0.9514	0.9213	0.9525
Xception_linked	validation	0.9189	0.8594	0.9272	0.9271	0.8789	0.9276
Xception_linked	test	0.9180	0.8583	0.9281	0.9251	0.8774	0.9256
MobileNet	training	0.8814	0.8009	0.8848	0.9057	0.7692	0.8883
MobileNet	validation	0.8678	0.7813	0.8672	0.9016	0.6982	0.8800
MobileNet	test	0.8708	0.7852	0.8706	0.9008	0.7192	0.8774
MobileNet_linked	training	0.9056	0.8389	0.9225	0.9114	0.8595	0.9142
MobileNet_linked	validation	0.8986	0.8286	0.9133	0.9101	0.8098	0.9122
MobileNet_linked	test	0.9020	0.8356	0.9202	0.9097	0.8231	0.9126
MobileNetV2	training	0.6623	0.5301	0.5477	0.9506	0.0473	0.5478
MobileNetV2	validation	0.6739	0.5432	0.5615	0.9509	0.0478	0.5426
MobileNetV2	test	0.6761	0.5428	0.5610	0.9500	0.0468	0.5383
MobileNetV2_linked	training	0.8658	0.7850	0.9148	0.8665	0.8017	0.8851
MobileNetV2_linked	validation	0.8618	0.7809	0.9054	0.8723	0.7399	0.8837
MobileNetV2_linked	test	0.8673	0.7858	0.9067	0.8750	0.7488	0.8839
DenseNet121	training	0.9325	0.8780	0.9324	0.9407	0.8839	0.9367
DenseNet121	validation	0.9069	0.8385	0.9040	0.9274	0.8248	0.9161
DenseNet121	test	0.9064	0.8396	0.9101	0.9214	0.8351	0.9142
DenseNet121_linked	training	0.9297	0.8749	0.9405	0.9302	0.9175	0.9356
DenseNet121_linked	validation	0.9160	0.8541	0.9280	0.9203	0.8877	0.9254
DenseNet121_linked	test	0.9183	0.8592	0.9348	0.9189	0.8984	0.9252
DenseNet169	training	0.9354	0.8822	0.9402	0.9373	0.8933	0.9383
DenseNet169	validation	0.9054	0.8365	0.9149	0.9134	0.8497	0.9171
DenseNet169	test	0.9072	0.8407	0.9209	0.9116	0.8557	0.9151
DenseNet169_linked	training	0.9276	0.8720	0.9394	0.9286	0.9102	0.9345
DenseNet169_linked	validation	0.9151	0.8532	0.9278	0.9209	0.8742	0.9243
DenseNet169_linked	test	0.9163	0.8563	0.9317	0.9193	0.8857	0.9248
DenseNet201	training	0.9435	0.8958	0.9394	0.9523	0.9003	0.9462
DenseNet201	validation	0.9082	0.8410	0.9095	0.9239	0.8598	0.9184
DenseNet201	test	0.9084	0.8425	0.9118	0.9224	0.8568	0.9155
DenseNet201_linked	training	0.9269	0.8707	0.9338	0.9327	0.9149	0.9343
DenseNet201_linked	validation	0.9151	0.8531	0.9266	0.9213	0.8933	0.9244
DenseNet201_linked	test	0.9189	0.8597	0.9297	0.9250	0.8971	0.9253

NASNetMobile	training	0.7878	0.6839	0.7967	0.8717	0.5638	0.8086
NASNetMobile	validation	0.7878	0.6840	0.7882	0.8806	0.5200	0.8009
NASNetMobile	test	0.7991	0.6946	0.7943	0.8863	0.5276	0.8068
NASNetMobile_linked	training	0.8597	0.7736	0.8602	0.8986	0.7346	0.8705
NASNetMobile_linked	validation	0.8625	0.7777	0.8594	0.9057	0.6769	0.8709
NASNetMobile_linked	test	0.8649	0.7780	0.8615	0.9056	0.6826	0.8697
EfficientNetB0	training	0.9522	0.9104	0.9488	0.9580	0.9117	0.9544
EfficientNetB0	validation	0.9069	0.8387	0.9057	0.9244	0.8497	0.9164
EfficientNetB0	test	0.9038	0.8351	0.9067	0.9190	0.8592	0.9116
EfficientNetB0_linked	training	0.9466	0.9016	0.9437	0.9548	0.9061	0.9488
EfficientNetB0_linked	validation	0.9208	0.8622	0.9235	0.9340	0.8734	0.9293
EfficientNetB0_linked	test	0.9228	0.8653	0.9227	0.9377	0.8739	0.9304
EfficientNetB1	training	0.9461	0.9001	0.9385	0.9577	0.8995	0.9486
EfficientNetB1	validation	0.9069	0.8384	0.9001	0.9293	0.8461	0.9166
EfficientNetB1	test	0.9061	0.8385	0.9014	0.9277	0.8495	0.9130
EfficientNetB1_linked	training	0.9315	0.8789	0.9425	0.9328	0.9233	0.9374
EfficientNetB1_linked	validation	0.9192	0.8595	0.9327	0.9222	0.9029	0.9285
EfficientNetB1_linked	test	0.9239	0.8670	0.9400	0.9221	0.9140	0.9307
EfficientNetB2	training	0.9386	0.8874	0.9451	0.9374	0.9074	0.9431
EfficientNetB2	validation	0.9042	0.8349	0.9137	0.9123	0.8599	0.9155
EfficientNetB2	test	0.9043	0.8358	0.9144	0.9121	0.8649	0.9129
EfficientNetB2_linked	training	0.9228	0.8651	0.9415	0.9201	0.9210	0.9293
EfficientNetB2_linked	validation	0.9200	0.8612	0.9354	0.9218	0.8942	0.9279
EfficientNetB2_linked	test	0.9183	0.8592	0.9355	0.9195	0.8973	0.9257
EfficientNetB3	training	0.9434	0.8954	0.9398	0.9514	0.8973	0.9459
EfficientNetB3	validation	0.9046	0.8356	0.9051	0.9220	0.8436	0.9154
EfficientNetB3	test	0.9070	0.8399	0.9076	0.9227	0.8532	0.9142
EfficientNetB3_linked	training	0.9274	0.8734	0.9395	0.9305	0.9045	0.9347
EfficientNetB3_linked	validation	0.9212	0.8630	0.9334	0.9254	0.8761	0.9297
EfficientNetB3_linked	test	0.9216	0.8645	0.9336	0.9264	0.8884	0.9301
EfficientNetB4	training	0.9414	0.8933	0.9434	0.9461	0.9076	0.9447
EfficientNetB4	validation	0.9076	0.8402	0.9125	0.9189	0.8558	0.9181
EfficientNetB4	test	0.9085	0.8422	0.9132	0.9210	0.8633	0.9167
EfficientNetB4_linked	training	0.9242	0.8676	0.9297	0.9341	0.9194	0.9317
EfficientNetB4_linked	validation	0.9193	0.8600	0.9277	0.9286	0.9027	0.9272
EfficientNetB4_linked	test	0.9194	0.8617	0.9301	0.9274	0.9078	0.9273
EfficientNetB5	training	0.6754	0.5482	0.5482	1.0000	0.0000	0.5482
EfficientNetB5	validation	0.6873	0.5620	0.5620	1.0000	0.0000	0.5429
EfficientNetB5	test	0.6898	0.5617	0.5617	1.0000	0.0000	0.5386

7.2 Source code

The full source code for this thesis is available on GitHub[15].

15 https://github.com/ptziolos/UoA/tree/main/Master/Cloud_Segmentation