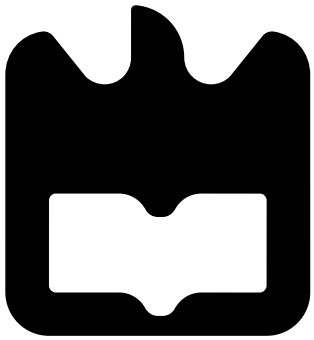
University of Aveiro Department of Mechanical Engineering
2018

Pedro Santos da Conceição

Wind Turbine Blade Inspection
Inspeção de Pás de Turbinas Eólicas



Pedro Santos da Conceição

Wind Turbine Blade Inspection
Inspeção de Pás de Turbinas Eólicas

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Automação Industrial, realizada sob a orientação científica de Miguel Armando Riem de Oliveira, Professor do Departamento de Mecânica da Universidade de Aveiro

o júri / the jury

presidente / president

Professor Doutor José Paulo Oliveira dos Santos
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Professora Doutora Pétia Georgieva Georgieva
Professora Auxiliar da Universidade de Aveiro (arguente)

Professor Doutor Miguel Armando Riem de Oliveira
Professor Auxiliar da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Quero agradecer a todos os que me apoiaram e fizeram parte deste longo percurso .

Em primeiro lugar à minha família, os meus pais Pedro e Maria Gabriela e as minhas irmãs Patrícia e Cláudia, e à minha namorada Joana por todo o apoio, aconselhamento e motivação e por estarem sempre ao meu lado.

Agradecer à Universidade de Aveiro e à NHL Hogeschool por todas as condições proporcionadas. Sem estas, não seria possível atingir os objectivos propostos.

Aos meus professores, nomeadamente, aos meus orientadores Miguel Riem e Ioannis Katramados e também ao Klaas Dijkstra.

Ao António, Armindo, Rúben e restantes colegas por todo o apoio e disponibilidade ao longo de todo o mestrado. Ao Tomás Osório por toda a ajuda na fase inicial.

A todos, muito obrigado!

Resumo

Devido a preocupações climatéricas, energia renovável é uma importante fonte energética. Energia eólica desempenha um importante papel na produção de energias renováveis e é produzida por geradores eólicos. Estes estão sujeitos a danos e requerem inspecção regular. Esta tarefa é realizada manualmente e é um procedimento demorado, perigoso e dispendioso. Por estas razões, uma abordagem diferente e automática é necessária.

Nos recentes anos, a área da Inteligência Artificial tem crescido muito e já desempenha um papel importante no nosso dia-a-dia.*Deep Learning*, um dos ramos da Inteligência Artificial, usa dados de forma a aprender padrões nos quais se baseia para tomar decisões. É já usada em muitas aplicações provando ser uma alternativa a ter em conta para desempenhar diferentes tarefas. Neste trabalho, propomos um modelo capaz de detectar áreas com dano nas pás de geradores eólicos baseado em *Deep Learning*.

O *dataset* disponível é composto por oitenta imagens de pás de geradores eólicos tiradas por especialistas. As áreas com dano nestas imagens foram anotadas por um especialista. Testamos duas arquitecturas diferentes para segmentação de objectos, para segmentar as pás nas imagens, e duas arquitecturas diferentes para detecção de objectos, para colocar uma *bounding box* na área com dano.

Quanto à segmentação, U-Net e DeepLabv3 produziram resultados promissores para serem usados em aplicações no mundo real. No entanto, a segmentação mostrou não ser totalmente correcta com algumas áreas das pás a não serem correctamente segmentadas sendo necessária mais pesquisa futura. Quanto à detecção, RetinaNet teve um desempenho superior que a Faster R-CNN. No entanto, é recomendável continuar o desenvolvimento do modelo de modo a aumentar a eficácia do mesmo.

Abstract

Due to climate concerns, renewable energy is an important energy resource. Wind energy plays a big role in renewable energy production and is produced by wind turbines. These are prone to damage and require inspection at all time. This task is done manually and it is a time-consuming, dangerous and expensive procedure. Because of this reasons, a different and automatic approach is needed.

In recent years, Artificial Intelligence has improved a lot and already takes an important role in our daily lives. Deep Learning, one of Artificial Intelligence's fields, uses data to learn patterns for use in decision making. It is already used in many applications and has proven to be reliable in different tasks. In this work we propose a model capable of detecting damaged areas in wind turbine blades based in Deep Learning.

The available dataset was made by eighty wind turbine blade images taken by experts. The damaged areas in these images were annotated by an expert. We try two different architectures for object segmentation, to segment the blades in the images, and two different architectures for object detection, to place a bounding box around the damaged area.

Regarding object segmentation, U-Net or DeepLabv3 produced promising results to be used in a real-world application. Although, the segmentation revealed to be faulty with several areas of the blades not being correctly segmented and further research is recommended. Regarding object detection, RetinaNet performed better than Faster R-CNN. However, it is recommended to continue the research in order to increase the accuracy of the model.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this Chapter we introduce the problem and the approach we decided to take in order to overcome that problem.

1.1 Motivation

Due to climate concerns, renewable energy is an important energy resource. In Portugal, wind is responsible for a large amount of the total energy generated by renewable sources. In March 2018, the electrical renewable energy production exceeded in 3.6% the electrical energy consumption. 42% of the total renewable energy produced was generated by wind [14]. Portugal will also connect an offshore wind turbine farm to the country's power grid [3] and expects a growth of 8.3% in wind energy production until 2027 [4].

To generate wind energy, wind turbines are used. These are prone to damage, once they are exposed to the natural elements. This deeply affects their efficiency. Currently, the inspection of wind turbine blades must be done at all times by people who climb the wind turbine and inspect the blades up close making this a time-consuming, dangerous and expensive procedure. While in land, a wind turbine blade inspection is already difficult. At offshore farms, this task is too risky to be done manually. Therefore an alternative approach is necessary.

In recent years, Artificial Intelligence has improved a lot and already takes an important role in our daily lives. Deep Learning, one of Artificial Intelligence's fields, uses data to learn patterns for use in decision making. In industry, many tasks are already performed with the help of, sometimes even exclusively by, Artificial Intelligence. This automation of processes leads to work safety, better task performance (higher quality of product and speed of production), thus more profit.

This dissertation is part of NHL's Centre of Expertise in Computer Vision and Data Science project Inspection with Automated UAVs using Computer vision (IWAUC) and will focus on this problem, by proposing an algorithm for automatic inspection of blades and damage detection to be done offline, by images taken from a camera mounted on a drone. This is a complex problem because the environmental conditions such as weather and light are not constant and therefore the inspection device must be able to work in too many different and unexpected scenarios. This system in itself is already complex, because it must be able not only to perform segmentation of the blade in the image it captures, but also to delineate the damages with a bounding box. To make this possible, multiple neural network models

with different architectures have been studied and implemented.

1.2 Objectives

The main goal of this work is to develop a deep learning based model capable of detecting damages in images of wind turbine blades. To do so, this model must be able to segment the blade from the image and then to detect damages in the selected region of interest. We also test an approach without segmentation in order to evaluate its impact in detecting damages.

In this work, two different models with different hyperparameters will be tested for the segmentation and object detection network. The segmentation networks tested in this work are U-Net and DeepLabV3. As for the object detection task, Faster RCNN and RetinaNet were tested.

1.3 Organization

In Chapter 2, an introduction and brief explanation of important deep learning concepts is done. The deep learning architectures and models used in this work will also be a topic of discussion. In Chapter 3, all the hardware and software used in this work is described. Afterwards, the selected and used data is described in Chapter 3.3. The proposed solution and all the considerations are presented in Chapter 4. Chapter 5 describes all the tests and respective results are discussed. Finally, in Chapter 6 conclusions are drawn and future work is proposed.

Chapter 2

Background and Related Work

In Chapter 2 we explain some basic yet important concepts of Deep Learning. First we explain the different types of layers and its characteristics. After, we explain in detail the architectures we used in this work as well as the metrics used to measure the performance of the models.

2.1 Introduction to Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model inspired by the way biological neural networks in the human brain process information. Such model is capable of progressively improving performance on a specific task by considering examples without task-specific programming, as defined by [27].

An ANN is a set of multiple artificial neurons arranged in different layers. Typically, these layers are the input layer, the hidden layer(s) and the output layer. There are connections between layers connecting the neurons so there is information flow from the input to the output. A more detailed explanation will follow in 2.1.2.

2.1.1 Artificial Neuron Model

The basic unit of computation in an ANN is the artificial neuron. Its mathematical model is based on the biological neuron as exemplified in Figure 2.1. The biologic neuron gets electrical impulses as inputs on the dendrites. Then, the nucleous processes the impulses and generates another that is propagated to the next neuron(s) by the axon terminals. In a similar way, the artificial neuron receives one or more inputs that are individually weighted. The weighted inputs are then summed up. This sum is passed through a non-linear function (known as activation function) and the result will be passed to neurons in the next layer.

2.1.2 Artificial Neural Network

An ANN is, as explained before, a set of artificial neurons arranged in three different layers. The number of artificial neurons by layer as well as the number of hidden layers are variable. The input layer provides information from the outside to the network without performing any computation on the given data. The data is passed to the first hidden layer.

The hidden layer(s) perform computation on the given data passing the output of each one of them to next until reaching the output layer. In these layers complex and abstract

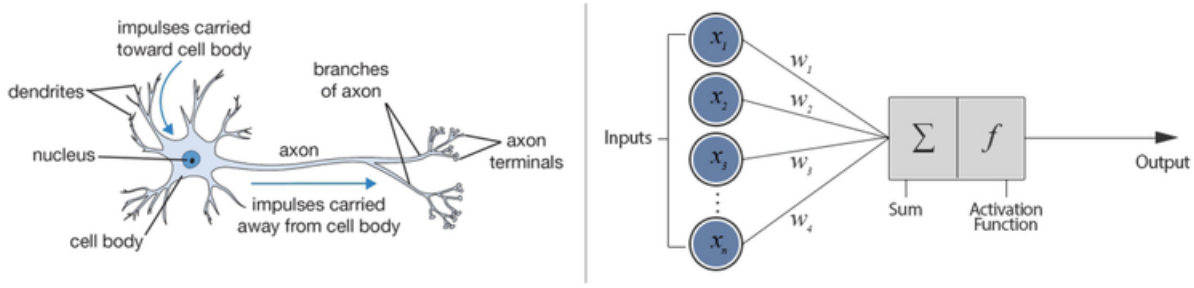


Figure 2.1: Biological neuron model and equivalent mathematical model from [18].

patterns and decisions may be found. The deeper the network is, the more complex and abstract these patterns and decisions are. For example, in convolutional neural networks, in the first hidden layers simple features like edges are processed. However, in deeper hidden layers, more sophisticated features, like faces, animals and other kind of objects, emerge. The output layer transfers the information from the network to the outside world.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of ANNs used to analyse and extract information from images. CNNs will be widely used in this work in order to successfully do the blade segmentation and the damage detection.

The input of a CNN is an image that can also be referred as a tri-dimensional matrix with dimensions $width \times height \times channels$. For example, for an RGB image, the dimensions would be $width \times height \times 3$, where 3 is the number of the R (red), G (green) and B (blue) channels. The output is a tri-dimensional matrix with size $1 \times 1 \times number\ of\ classes$. In other words, the output is a single vector with depth equal to the number of classes in which each channel is a score for each class.

The hidden layers of a CNN typically consist of convolutional layers Subsection 2.2.1 and pooling layers Subsection 2.2.2. The special feature of CNNs when compared to other kinds of ANNs is the convolutional layer, hence the name Convolutional Neural Network. This layer makes image processing a possible and relative fast process while maintaining spacial information.

2.2.1 Convolutional Layer

The convolutional layer applies a convolution operation to the input passing the result as output to the next layer. This operation extracts semantic information of images, but eliminates spatial information every time it is performed. Figure 2.2 shows a convolution performed by a 3×3 kernel.

This layer can have several different kernels (convolutional filters) producing a new output matrix for each of them, given an input matrix. The number of kernels per layer is an adjustable parameter as well as the dimensions of the kernels. Despite this, the most commonly used kernel dimensions are 3×3 and 5×5.

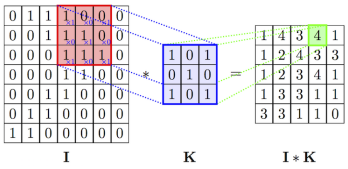


Figure 2.2: Example of a convolution with a 3×3 kernel from [25].

2.2.2 Pooling Layer

Pooling layers perform a downsampling operation on the input. There are different types of pooling layers such as max-pooling, average pooling and L2-norm pooling.

For example, the max-pooling layer has a kernel that outputs the maximum value for a given position in the matrix. This operation reduces the *width* and *height* dimensions of the output in relation to the input. This reduction of the spatial size of the image allows the reduction of parameters and, therefore, the reduction in the number of calculations made by the network. It also helps in controlling overfitting. By only outputting the maximum value in the kernel's receptive field, max-pooling drives the network to learn the most general patterns of a given dataset while discarding other single-image specific patterns.

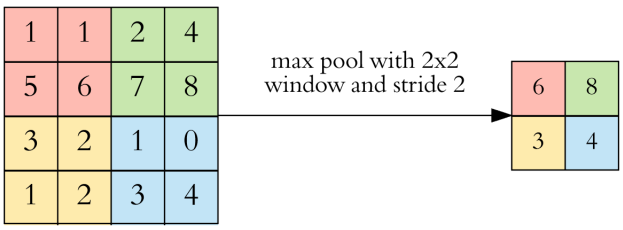


Figure 2.3: Max-pooling operation from [6].

2.2.3 Overfitting

In statistics, overfitting is "the production of an analysis which corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably." according to [7].

When training a CNN, this aspect must be taken into account, otherwise the model will lose its generalisation capability and, consequently, the intended capability of prediction.

Overfitting usually occurs when the training dataset is small therefore having low variability. This leads to a failed attempt to learn the general features of a given object. One way to avoid overfitting is by using the dropout technique [23]. This technique is only used during training and deactivates random neurons forcing different neurons to learn the same general concept of a given object.

2.2.4 Loss Function

The output of a loss function is the error of the training. The lower the value is, the better is the network predicting. In other words, one can say that the loss represents the difference between the output and the desired output. There are different functions that can be used as loss functions.

Binary Cross Entropy is used for one class predictions only. In this work this function is used in both in 2.4.1 and 2.4.2. Binary Cross Entropy is a one class loss and therefore it can only be used in one class predictions.

The function is described in (2.1) where y stands for the ground truth labels and x stands for the corresponding probabilities predicted by the network.

$$g(x, y) = -\frac{1}{N} \sum y \times log(x) + (1 - y) \times log(1 - x) \tag{2.1}$$

The **L1** function calculates the distance between the target value y and the estimated value x as seen in (2.2).

$$g(x, y) = \sum |y - x| \tag{2.2}$$

Smooth L1 error can be thought of as a smooth version of (2.2). It uses a squared term if the squared element-wise error is smaller than 1 and L1 distance otherwise. It is less sensitive to outliers than the L1 function.

$$g(x, y) = \begin{cases} 0.5 \times (x - y)^2 & 0.5 \times (|\text{x}-\text{y}|) < 1 \\ |x - y| - 0.5 & \text{otherwise} \end{cases} \tag{2.3}$$

2.2.5 Backpropagation

The training of a CNN consists basically on the adjustment of all the weights of the kernels in the network. To do so, backpropagation is used. Backpropagation is an efficient method for computing gradients required to perform gradient-based optimization of the weights in neural networks.

The weights of the kernels of the entire network are randomly initialised. Therefore, the CNN cannot make correct predictions for a given input. In training, an image is given as input and the output is compared with the ground truth, which is the pretended output. With this, an error is calculated and the weights are adjusted to make this error as low as possible. In the first training iterations, this error is usually very high and decreases as the training proceeds and the weights are adjusted. There are two main phases in this process. In the first phase, an input is propagated through the network generating an output. Using a loss function, we use this output to calculate the error. Then, backward propagation is done in order to calculate the influence that each weight has in the final error. In the second phase, considering the previously calculated influence of each weight, the weights are adjusted to minimise the loss function value and so minimise the error.

2.2.6 Activation Function

The activation function of a neuron is a function that generates an output given an input or a set of inputs. The purpose of the activation function is to introduce non-linearity into the output of a neuron, allowing networks to solve non-linear problems by using only a small number of neurons. This an important aspect, because the network must be able to learn non-linear representations of the real world.

The **sigmoid function**, Equation (2.4), has a very steep slope between -2 and 2. This means that any small changes in the values of X in that region will cause values of Y to change significantly, bringing Y values to either end of the curve and saturating these values. Although this behaviour helps making clear distinctions on prediction, it also has a downside. When Y values saturate, they tend to respond poorly to changes in X. This leads to small or nonexistent gradient and makes it hard to change the influence of a given neuron to the network. As a consequence, the network might stop learning or learning at a very slow rate.

$$f(x) = 1/(1 + e^x) \tag{2.4}$$

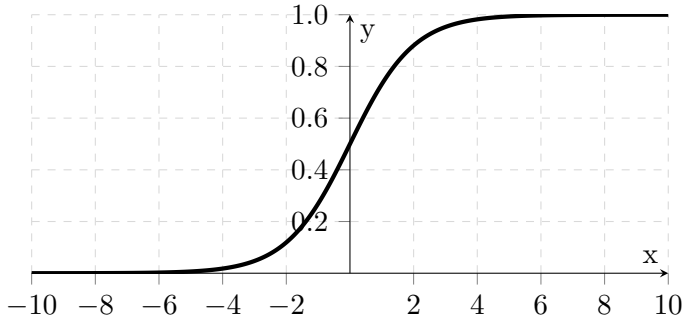


Figure 2.4: Plot of Sigmoid function.

Rectified Linear Unit (ReLU) is one of the most used activation functions because of its low computing cost.

This function outputs the input if it is higher than zero and outputs zero if the input is less than zero. In other words, the activation is simply thresholded at zero. This allows ReLU to be a low computing demanding function when compared to other popular activation functions. Unlike the sigmoid activation function, Y values do not saturate for positive X values.

The disadvantage of this function is that for negative values of X the gradient can go towards 0, preventing the neurons that go into that state of responding to variations in error. This is called dying ReLu problem. This problem can prevent several neurons of responding, leading to passiveness of a substantial part of the network.

$$f(x) = max(0, x) \tag{2.5}$$

Figure 2.5 shows the plot of function (2.5).

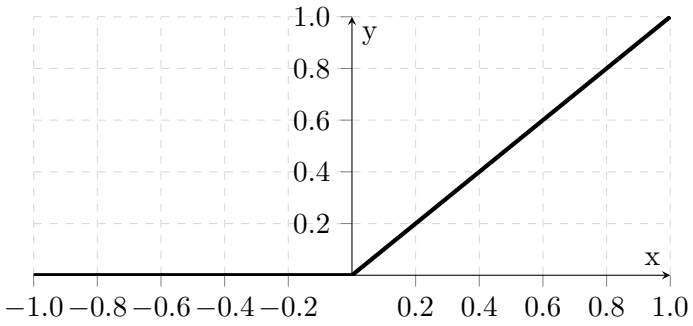


Figure 2.5: Plot of ReLU function.

Exponential Linear Unit (ELU) solves the dying ReLU problem by computing negative X values differently as (2.6) shows. It smooths the negative X values instead of thresholding to 0.

$$f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \tag{2.6}$$

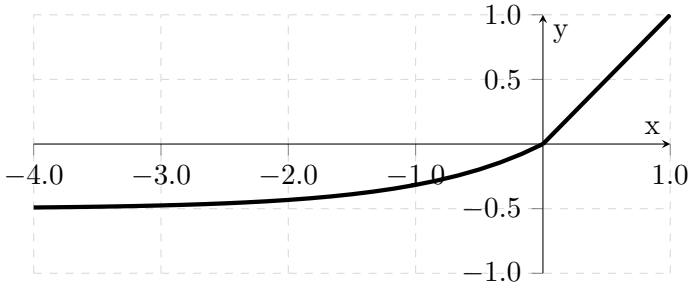


Figure 2.6: Plot of ELU function (with $\alpha = 0.5$).

2.2.7 Hyperparameter Tunning

The training is ser by hyperparameters. Hyperparameters are all the non-trainable parameters and are defined manually.

Examples of hyperparameters are:

- Batch size - number of samples to be propagated through the network by iteration;
- Steps - number of iterations per epoch;
- Number of epochs - number of times the dataset is going to be fed to the network;
- Learning rate;
- Optimiser;
- Loss function;
- Activation function;
- Number and size of kernels.

All these parameters must be tuned manually. While definition of these parameters is not subject to strict rules and is based mostly experience and intuition, we can only use general guidelines. A good choice of hyperparameters will help to achieve better results. Therefore, hyperparameter tuning is an extremely important yet complex and difficult task of neural network training.

2.2.8 Examples of CNNs

VGG16 and ResNet are two of the most accurate CNN architectures[10] . These networks perform image classification which means that they label an entire image. Even though the current research is more focused in object detection and segmentation, these networks still play an important role as backbones by extracting features from images.

VGGNet

VGGNet, the runner up in the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2014 [20], was developed by Simonyan and Zisserman from the University of Oxford [22]. This network has a very simple architecture consisting of consecutive convolutional layers with 3×3 sized filters with pooling layers in between. There are different versions of this architecture being VGG16 is one among different existing versions of this network. The name comes after the fact this architecture has 13 convolutional layers plus 3 fully connected layers as shown in Table 2.1.

VGG16 layers
3×3 $Conv$, 64
3×3 $Conv$, 64
max - pool
3×3 $Conv$, 128
3×3 $Conv$, 128
max - pool
3×3 $Conv$, 256
3×3 $Conv$, 256
3×3 $Conv$, 256
max - pool
3×3 $Conv$, 512
3×3 $Conv$, 512
3×3 $Conv$, 512
max - pool
3×3 $Conv$, 512
3×3 $Conv$, 512
3×3 $Conv$, 512
max - pool
FC-4096
FC-4096
FC-1000
softmax

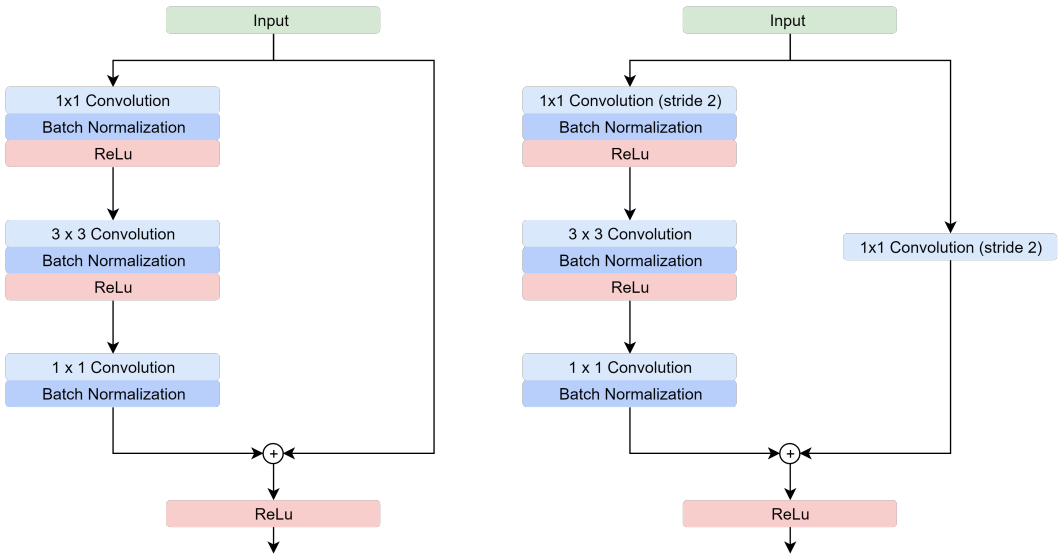
Table 2.1: VGG16 architecture.

Despite having only 16 layers with trainable parameters, it is a massive network with 138 million parameters and achieved a top-5 error of 7.3% in ILSVRC 2014. For that reason it is still a popular and widely used network.

Residual Net

Residual Net or ResNet [10] was the winner of ILSVRC 2015 with only 3.6% top-5 error, beating human-level performance. It is also a very popular network.

This network has a different approach from VGG16 being composed by a sequence of modules called **residual modules** shown in Figure 2.7. This block is composed by three convolutions, three ReLU activations and three batch normalization layers, having a total of nine layers per block. It introduced skip connections between modules that solved the issue of vanishing gradient in deep architectures. During the back-propagation, every time the network reaches such a depth that reduces the signal originated from the end of the network and required to change the weights, vanishing gradient occurs.



(a) Residual module with identity shortcut.

(b) Residual module with projection shortcut

Figure 2.7: ResNet residual modules.

All the convolutions have stride 1, except the projection shortcut convolution and the first convolution of the common path between Figure 2.7 (a) and Figure 2.7 (b). This is used in order to decrease the width and height dimensions while increasing the depth of the feature map.

ResNet is built by placing residual modules on top of each other therefore being possible to create shallower or deeper versions of this architecture. In the original paper [10] different versions of ResNet are shown which are described but here we only present ResNet-50 and ResNet-101 in Table 2.2.

50-layer	**101-layer**
7×7 Conv, 64, stride 2	
3×3 max-pool, stride 2	
$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 64 \\ 3 \times 3 & \text{Conv}, & 64 \\ 1 \times 1 & \text{Conv}, & 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 64 \\ 3 \times 3 & \text{Conv}, & 64 \\ 1 \times 1 & \text{Conv}, & 256 \end{bmatrix} \times 3$
$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 128 \\ 3 \times 3 & \text{Conv}, & 128 \\ 1 \times 1 & \text{Conv}, & 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 128 \\ 3 \times 3 & \text{Conv}, & 128 \\ 1 \times 1 & \text{Conv}, & 512 \end{bmatrix} \times 4$
$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 256 \\ 3 \times 3 & \text{Conv}, & 256 \\ 1 \times 1 & \text{Conv}, & 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 256 \\ 3 \times 3 & \text{Conv}, & 256 \\ 1 \times 1 & \text{Conv}, & 1024 \end{bmatrix} \times 23$
$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 512 \\ 3 \times 3 & \text{Conv}, & 512 \\ 1 \times 1 & \text{Conv}, & 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1 & \text{Conv}, & 512 \\ 3 \times 3 & \text{Conv}, & 512 \\ 1 \times 1 & \text{Conv}, & 2048 \end{bmatrix} \times 3$
average pool, 1000-d fc, softmax	

Table 2.2: ResNet-50 and ResNet-101 architectures. Building blocks are shown in brackets with the number of blocks stacked.

The architecture of ResNet-50, shown in Figure 2.8, is a stack of 17 residual modules and has a total of 50 layers. The identity shortcuts are represented by solid line shortcuts and the projection shortcuts are represented by dotted shortcuts in Figure 2.8. The architecture of ResNet-101 would be similar but instead of having only 6 blocks in the third stage it would have 23. Due to this fact ResNet-101 is a deeper architecture and thus an architecture that demands more memory and computational power. In each stage of this architecture the width and height are halved and the depth of the feature map is doubled. For example, for an input size of 224, the size of the feature map after the max-pooling layer would be 112 due to the first convolution and max-pooling layer. This feature map would be fed to the first stack of blocks (first stage) resulting in an output feature map of size 56. In the second, third, and forth blocks, the size of the feature map would be, respectively, 28, 14 and 7. In the last layer of the network the probabilities off all the classes are calculated resulting in the final output of the network with size 1 and depth equal to the number of classes.

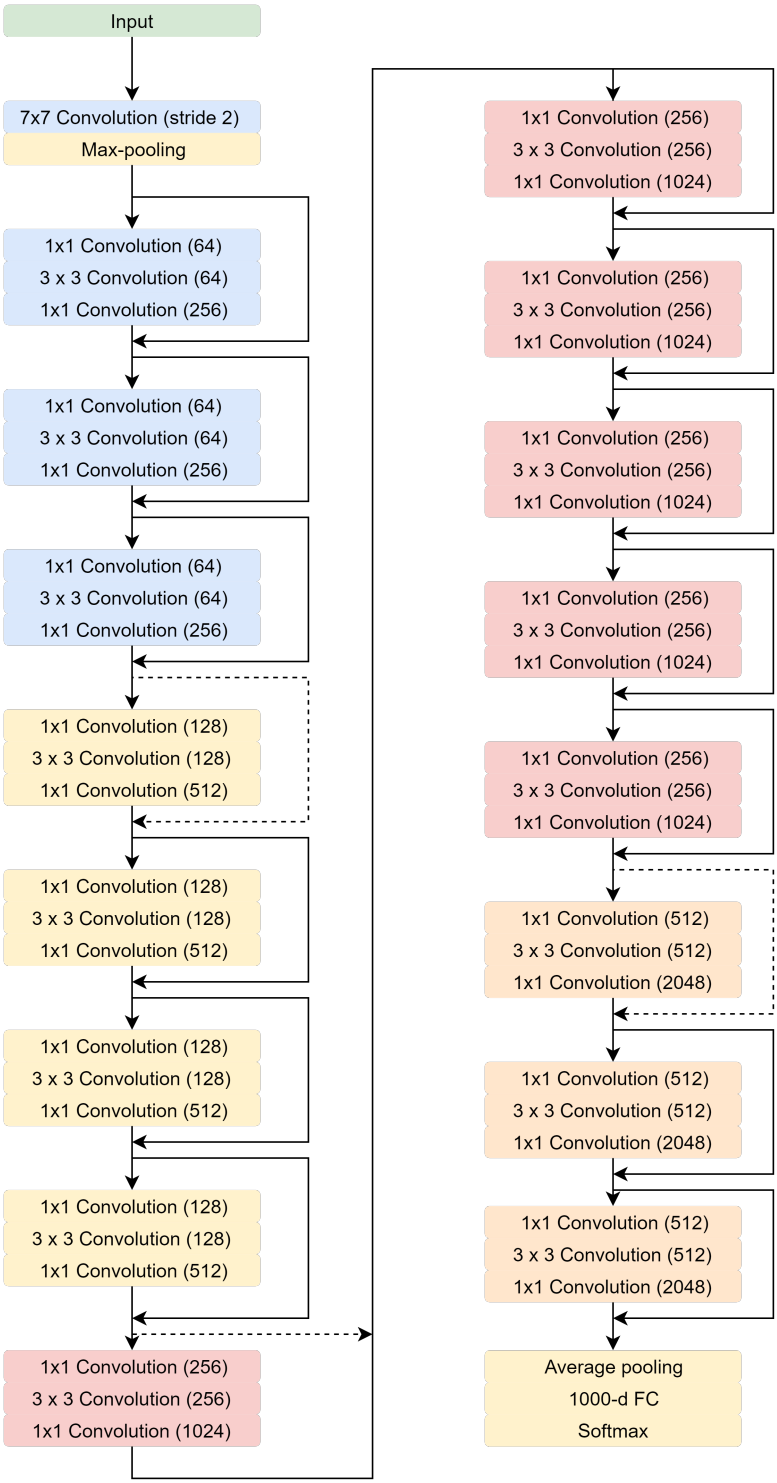


Figure 2.8: ResNet architecture.

2.3 Fully Convolutional Networks

The goal of this kind of networks is to determine what objects are present and where those objects are in the image. To achieve Fully Convolutional Networks (FCN), a modified version of CNNs is used. In order to achieve the pixel-wise segmentation, the fully connected layer present in CNNs was replaced by a convolutional layer enabling the output of segmentation maps. By doing this, the spatial information is not discarded by the fully connected layer and can be associated with class confidence information generating a heatmap for each class as output as seen in Figure 2.9.

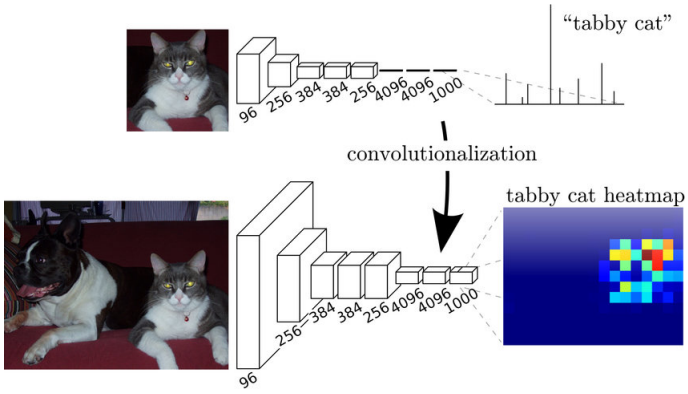


Figure 2.9: Difference between CNN and FCN and its outputs [1].

2.4 Object Segmentation

Object segmentation, or semantic segmentation, refers to understanding an image at pixel level by assigning an object class for each pixel. This means that the object not only should be found and classified but also to delineated by the network. This is currently a big challenge in deep learning applied to computer vision, because too much spatial information of the image is lost while performing convolutions. This loss makes is it difficult to delineate very detailed contours of the detected objects and having a fine segmentation.

The heatmaps that FCNs generate have smaller dimensions when compared to the input image and therefore must be upscaled to coincide with the input image dimensions, so that a correct segmentation can be obtained. Next, we show the techniques used by each of the architectures.

2.4.1 U-Net

U-Net is a popular image segmentation network in deep learning named after the network "U" shape as can be seen in Figure 2.10. In 2015, Ronneberger et al. [19] designed this network for biomedical image segmentation purposes, having won ISBI cell tracking challenge 2015. Their goal was to achieve a good image segmentation using a small dataset where data augmentation would be performed.

The network consists of an enconding part to capture context followed by a symmetrical decoding part that enables precise localization.

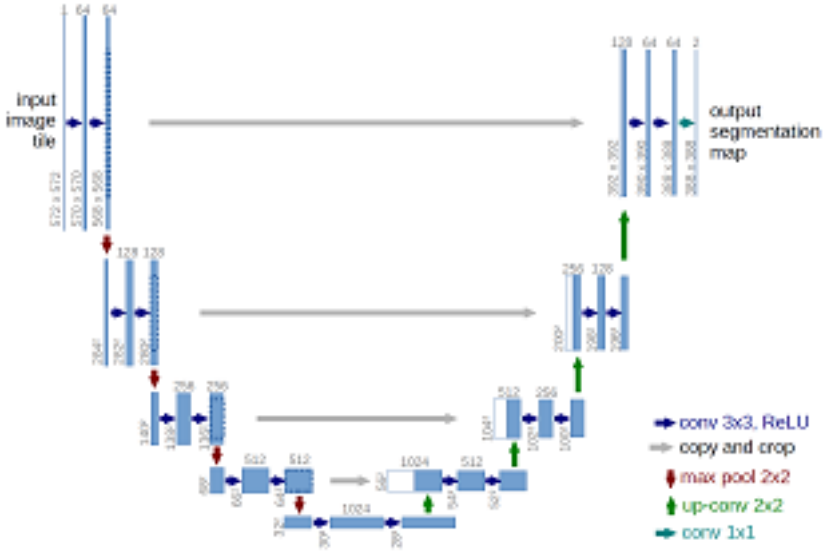
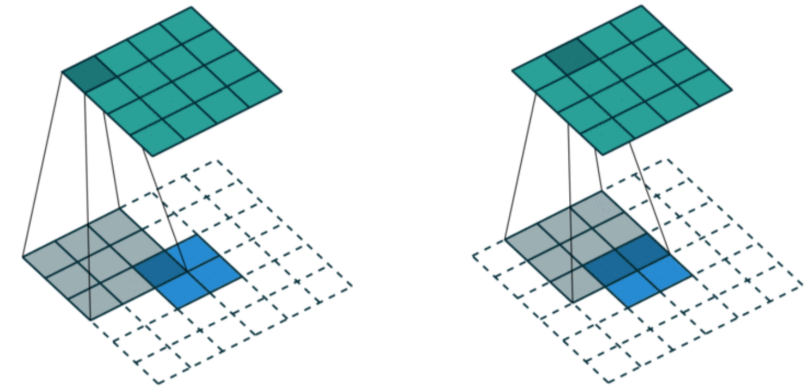


Figure 2.10: U-Net architecture example for 32x32 pixels in the lowest resolution [19].

The enconding, or contracting part, consists of the repeated application of unpadded convolutions each followed by a ReLU and max-pooling operations for downsampling. At every step, spatial information is lost and semantic information is created. This is the convolutional part of the network.

The decoding part consists of an upsampling of the feature map followed by a convolution that halves the number of feature channels, a concatenation with the corresponding feature

map from the contracting part and two convolutions, each followed by a ReLU. The upsampling convolution is achieved by using transposed convolution. Transposed convolution is very similar to a usual convolution but generates an output that bigger, both in width and height, than the input. Figure 2.11 helps understanding that the kernel is multiplied time at a time by the input pixel values originating a bigger output image.



(a) Transposed convolution for the first pixel.

(b) Transposed convolution for the first pixel.

Figure 2.11: Example of transposed convolution.

U-Net was chosen in this work as the segmentation network as it has proven to be reliable on small datasets in the original publication. Furthermore it is a simple, easy and fast to train network and has the advantage of not relying on pre-trained networks.

2.4.2 DeepLabv3

DeepLabv3 is an object segmentation network developed by Google and released in December 2017 [5]. This network is much more complex than U-Net and performs better object segmentation.

DeepLabv3 offers a different approach from the encoder-decoder explained in 2.4.1 that enables learning multi-scale contextual features. In this network, a modified version of ResNet is used as a feature extractor network. In order to achieve multi-scale feature learning, regular convolutions are replaced by atrous convolutions, with unit rate 2, in the last block of ResNet-50, as Figure 2.12 shows. By adding more blocks after this one, the unit rate will increase by a factor of 2 in each next block. The different unit rates of the atrous convolutions allow to capture multi-scale context. This change increases the field of view of the convolutions.

Atrous Spatial Pyramid Pooling (ASPP) is used on top of this block as an attempt to classify regions.

Atrous, or dilated, convolutions are convolutions with a factor that expands the filter's footprint. For instance, a 3x3 convolution filter with dilation rate 1 produces the same output as a normal convolution. However, if the dilation rate is 2, the filter becomes a 5x5 filter with columns 2 and 3 and rows 2 and 3 filled with zeros as illustrated in Figure 2.13. For both cases, only nine cells of both filters are taken into account.

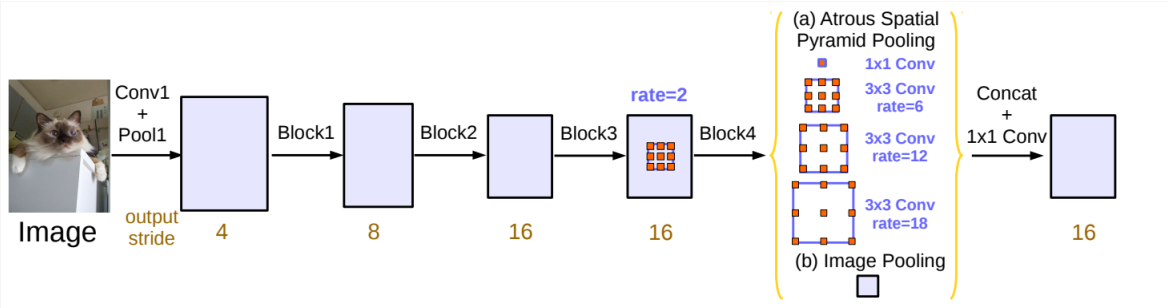


Figure 2.12: DeepLabv3 architecture [5].

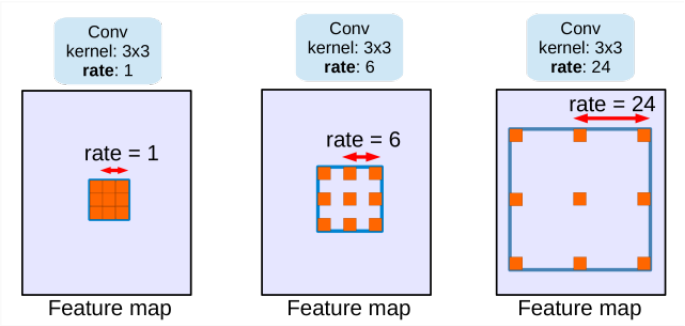


Figure 2.13: Example of atrous convolution [5].

The dilation rate must be tuned according to the size of the feature maps. If the dilation rate is excessively high, the atrous convolution might work as a 1×1 convolution, and so may have a negative impact on the efficiency of atrous convolutions. This happens when the dilation rate is equal or higher than the highest dimension of the image. When doing the convolution, the kernel will have a maximum of one weight over the image at a time, therefore behaving like a 1×1 convolution. In order to tune the dilation rate we must understand the concept of **output stride**.

Output stride is the ratio of input image spatial resolution to final output resolution. The lower this ratio is, the finer the segmentation will be at the cost of more computation power. For example, an output feature map would be 32x32 for an output stride of 16 and a 512x512 input image. Considering this example and a 3x3 filter, a dilation rate of 32 would make the filter behave like a 1x1 filter. The image would be smaller than the filter resulting that, at each change of position of the filter, only one of the weights would be above the image.

Using the Multi-Grid Method, DeepLabv3 proposes different dilation rates for each convolution of the last block of ResNet-50. The final atrous rate for the convolutional layer is equal to the multiplication of the unit rate and the corresponding rate. For example, for Multi-Grid $= (1, 2, 4)$ and unit rate 2, the three convolutions inside the block would have dilation rates $= 2 \times (1, 2, 4) = (2, 4, 8)$. This approach allows an expanded field of view without increasing the number of parameters.

The Atrous Spatial Pyramid Pooling applies four parallel atrous convolutions - 1x1 convolution and three 3x3 convolutions with dilation rates 6, 12 and 18 - on top of the feature map. ASPP with different atrous rates effectively captures multi-scale information and context of the image.

Since the output stride of the network is 16, and we want the network to generate a mask for the input image, it is necessary to resize the output to match the dimensions of the input image and reach a result that makes sense. In order to do this, several different techniques can be applied, such as bilinear interpolation, transposed convolution and dense upsampling convolution. Bilinear interpolation takes into account the pixels in the neighbourhood. As we can see in Figure 2.14, the closest pixels of the lowest resolution image have a bigger influence on the new higher resolution image that the further pixels.

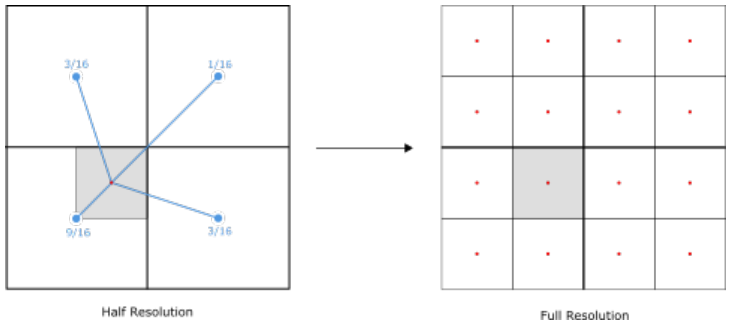


Figure 2.14: Bilinear upsampling example.

Dense upsampling convolution (DUC) [26] has a completely different approach. DUC, unlike bilinear interpolation or transpose convolution, has the ability to learn like any other layer in the network therefore achieving better results. An image of size $H \times W \times C$, where H is height, W is width and C are the image channels, is fed into ResNet resulting an output of size $h \times w \times c$, where $h = H/d$ and $w = w/d$, and d is the downsampling factor. DUC uses this output, or feature map, from ResNet, and performs a convolution to produce a feature map of dimensions $h \times w \times (d^2 \times L)$, where L is the total number of classes in the semantic segmentation task. In the last step, a convolution with kernel size 1 and stride 1, is made over this feature map. This allows each layer of the convolution to learn a prediction for each pixel in the output image.

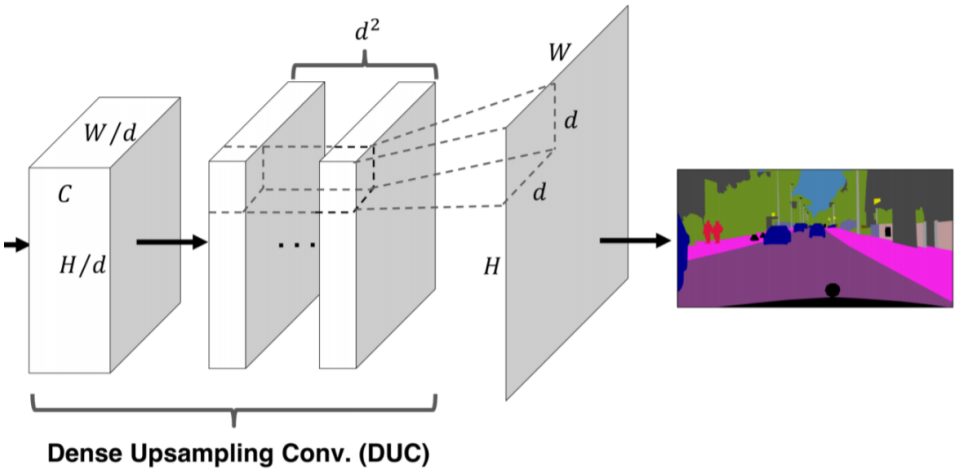


Figure 2.15: Dense upsampling convolution flow, from [26].

2.4.3 Metrics

In order to evaluate how close the model is to the ground truth, in object segmentation, we used the SrensenDice coefficient or just Dice coef. Dice coef is the coefficient of similarity of, in this case, the prediction and the ground truth and varies between 0 and 1 being 1 total similarity. It is defined in (2.7) where A is the predictiond and B is the ground truth.

$$Dice(A, B) = \frac{2|A \cdot B|}{|A| + |B|} \tag{2.7}$$

It can also be expressed as in (2.8), when applied to boolean data, where TP is true positive, FP is false positives and FN is false negative.

$$Dice = \frac{2TP}{2TP + FP + FN} \tag{2.8}$$

2.5 Object Detection

The goal of object detection is to classify an object and estimate the coordinates of the bounding box that encloses the object. There are a few different approaches but we propose two given their accuracy. Even though image processing is going to be performed by a drone, we tried to focus more on accuracy than processing speed. For this reason Faster RCNN and RetinaNet, two of the most accurate object detection networks [13], are explained below.

2.5.1 Faster RCNN

In June 2015 Faster RCNN (F-RCNN) was published by a Microsoft Research team. It is the third iteration of R-CNN which used an algorithm called Selective Search to propose possible regions of interest and a standard CNN to classify and adjust them. The second iteration was Fast R-CNN [9], published in early 2015, where a technique called Region of Interest Pooling allowed for sharing expensive computations and made the model much faster.

In F-RCNN the main feature was the replacement of the slow Selective Search algorithm with a fast neural net allowing this architecture to be 250 times faster than the first iteration while maintaining a high accuracy. The Region Proposal Network (RPN) was therefore created improving the speed of this network while keeping the accuracy of previous iterations. This change makes this network faster, yet accurate relatively to the two previous versions. Although it is not the fastest among the available architectures, it is one of the most accurate and was, therefore, chosen.

To perform object detection, F-RCNN has several different steps resulting in a complex architecture, as seen in Figure 2.16.

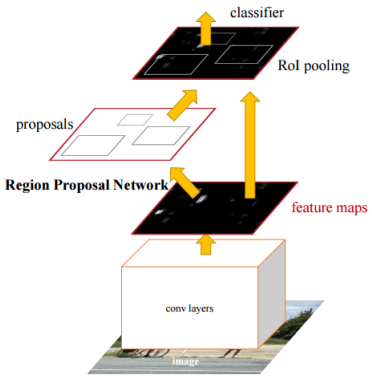


Figure 2.16: Faster RCNN architecture [17].

The input images first go through a pre-trained CNN resulting a convolutional feature map. This CNN, also called feature extractor or backbone, can be VGG or ResNet for instance. By having a pre-trained CNN, usually trained on a big dataset, it is possible to fine-tune the network (make small adjustment on all the weights) for a smaller dataset. By doing this, rather than training the network from scratch, a lot of time can be saved and the network can be trained with a smaller dataset with different classes. Since a lot of patterns and features are already learned, only few adjustments are needed to accurately predict the new classes.

The next step is the RPN that generates a predefined number of bounding boxes that may contain objects. The RPN slides a n x n window across the convolutional feature map. At each sliding window location, several anchors are generated with different shapes and ratios, the multiple region proposals - a set of k anchors with the same centre, different aspect ratios and three different scales - and classified as object or not object. Therefore the regression layer has $4 \times k$ outputs encoding the coordinates of k boxes and the classification layer outputs $2 \times k$ scores that estimate the probability of object or not object for each proposal. It does not take into account what kind of object is but only if there is an object or not. The shapes and ratios must be adjusted depending on the kind of objects the network is trained to find. For example, for cars the bounding boxes would be more like a square and for lamps the bounding boxes would have a high height/width ratio.

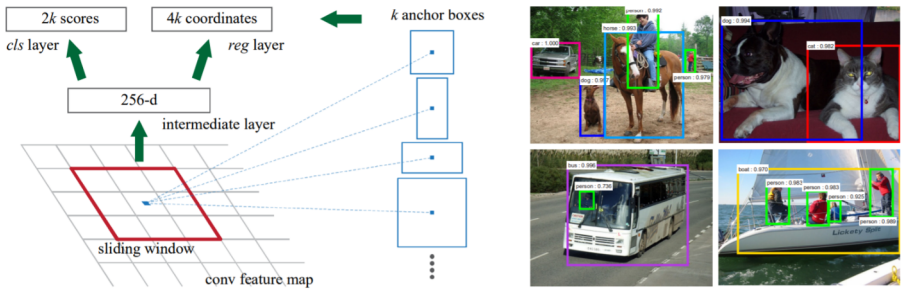


Figure 2.17: Left: Region Proposal Network (RPN). Right: Example of detections using RPN proposals on PASCAL VOC 2007 test [17].

Using the features extracted by the CNN and the bounding boxes with relevant objects, the third step is the Region of Interest Pooling (RoIP) that extracts the features which correspond to the relevant objects into a new tensor. The RoIP extracts the tensor respective to each bounding box from the feature map computed by the feature extractor. Instead of running each bounding box in another network in order to classify the object, the RoIP reuses the convolutional feature map an so additional computation is not necessary. RoIP uses hard negative mining [2] in order to achieve better results. Two intersection over union (IoU) (explained in 2.5.3) limits are defined, for example, 0.1 and 0.5. With hard negative mining, a predicted bounding box is only considered a good prediction when it overlaps the ground truth bounding box in, at least, 50%. If the IoU is less than 10%, then this prediction is discarded. However, if the IoU of this prediction is between 10% and 50% it is considered a hard negative example. Hard negative examples are used in hard negative mining [21]. This is a technique that aims to balance classes in training. While training, the network produces several different bounding boxes originating true and false predictions. This technique uses the false predictions as negative examples reinforcing the learning of the network. This approach gives extra knowledge to the network and has a positive effect on the final accuracy of the model.

The final step of F-RCNN is the Region-based Convolutional Neural Network (R-CNN) that classifies the object in each bounding box using the extracted convolutional feature map. R-CNN also adjust the coordinates of the bounding boxes to better fit the objects.

2.5.2 RetinaNet

RetinaNet, one of the most recent object detection networks, was published in 2017. RetinaNet, unlike Faster RCNN, is a one stage detector but it matches Faster RCNN's Common Objects in Context Average Precision (COCO AP).

The main feature of this architecture is the **focal loss**, a loss function that eliminates imbalance during train. This way the scaling factor can automatically downweight the contribution of easy examples during training and rapidly focus the model on hard examples.

As seen in the Figure 2.18 the one-stage RetinaNet network architecture [13] uses a Feature Pyramid Network (FPN) (b) on top of a feedforward ResNet architecture (a) to generate a rich, multiscale convolutional feature pyramid. After this two subnetworks are attached, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d). The developed loss function eliminates the accuracy gap between this one-stage detector and two-stage detectors like Faster R-CNN while running at faster speeds.

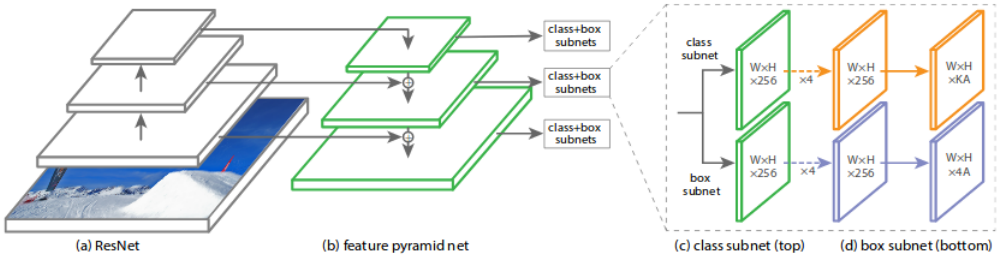


Figure 2.18: RetinaNet architecture [13].

FPN makes it easier to detect the same object at different scales by using different scaled feature maps. By generating multi-scale feature maps through convolutions on the bottom-up pathway, it increases semantic value as Figure 2.19 shows. "The top-down pathway hallucinates higher resolution features by upsampling spatially coarser, but semantically stronger, feature maps from higher pyramid levels. These features are then enhanced with features from the bottom-up pathway via lateral connections. Each lateral connection merges feature maps of the same spatial size from the bottom-up pathway and the top-down pathway. The bottom-up feature map is of lower-level semantics, but its activations are more accurately localised as it was subsampled fewer times." as explained in the original paper [12].

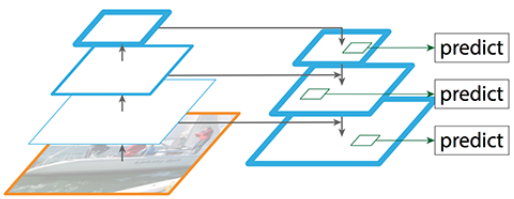


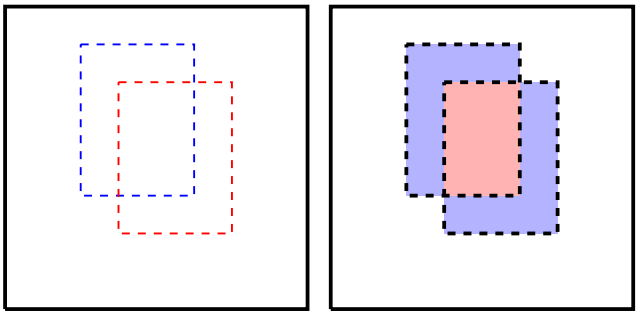
Figure 2.19: FPN diagram [12].

2.5.3 Metrics

In object detection, the perfect model is that in which the predicted bounding boxes totally overlap the ground truth bounding boxes. To measure this overlap we use the mean average precision (mAP) of the intersection over union (IoU). IoU is matematically defined as the ratio between the area of overlap and the area of reunion as shown in (2.9). When IoU is higher than a certain user-defined threshold, it is considered a true positive.

$$IoU(A, B) = \frac{A \cap B}{A \cup B} \tag{2.9}$$

In Figure 2.20 we show an example of an IoU between two bounding boxes. Assuming the red box represents the ground truth, with coordinates $(1, 3.5)$ and $(2.5, 1.5)$, and the blue box represents the prediction, with coordinates $(1.5, 3)$ and $(3, 1)$, we can compute the IoU. Therefore, IoU for this prediction would be $(intersection)/(1.5\times3)+(2.5\times1.5)-intersection)$, where $intersection = (2.5 - 1.5) \times (3 - 1.5)$. As a result, this prediction would be considered a correct prediction only if the IoU threshold was higher than $1.5/(6 - 1.5) = 0.33$.



(a) Red and blue bounding boxes represent ground truth and prediction, respectively.

(b) Red and blue are represent intersection and union, respecitevely, of the bounding boxes.

Figure 2.20: Example of IoU.

This metric is calculated to all bounding boxes in a given class. Then, the precision of each class is calculated by diving the correct predicted bounding boxes by the bounding boxes in the ground truth. When a model is a multi-class predictor, it calculates the precision for each class and divides by the number of classes. This is called average precision. Equation (2.10) shows the mathematical expression of precision, where TP is True Positives and FP is False Positives.

$$Precision = \frac{|GroundTruth \cap Predictions|}{Predictions} = \frac{TP}{TP + FP} \tag{2.10}$$

However, *precision* only takes into account the percentage of correct predictions. If, for example, the network would only predicted one bounding box out of 100 ground truth bounding boxes and it was a correct prediction, the precision would be 100%. However, the network would have missed 99 bounding boxes. As we can see by this example, this metric does not reflect how good a model is. Therefore, *recall*, equation (2.11), is also computed.

Recall, on the other hand, is the percentage of ground truth bounding boxes that the network correctly predicted.

$$Recall = \frac{|GroundTruth \cap Predictions|}{GroundTruth} = \frac{TP}{TP + FN} \tag{2.11}$$

In order to have a single metric that can inform about the previous two metrics, $F_1 score$, Equation (2.12), is used. $F_1 score$ is the weighted average of precision and recall that takes into account both false positives and false negatives. If the network, eventually, does not predict any bounding box, both TP and FP are zero. In this case, precision is undefined. However, we assume that the precision is zero and, therefore, $F_1 score$ is zero in these cases.

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{2.12}$$

Chapter 3

Hardware and Software

We describe the hardware and the software we used to implement the models in this Chapter. We also describe the dataset available for this work.

3.1 Hardware

Deep Frisian is a supercomputer from NHL Stenden Centre of Expertise in Computer Vision & Data Science.

Deep Frisian has:

- 44 Xeon cores;
- 512 GB RAM;
- 98 TB storage;
- 4 Nvidia Tesla P100 GPUs (14000 GPU cores);
- Up to 85 Teraflops of computing power.

Due to its characteristics, Deep Frisian allows training complex and high computing demanding networks.

3.2 Software

Using GNU Image Manipulation Program (GIMP), a free and open-source image editor, annotations for the segmentation network were generated. The annotations for this kind of networks are images with the same height and width as the original images and with as many channels as classes there are to train. Each channel of the generated image is a mask that corresponds to a certain label. For the object detection network training, bounding box coordinates are needed. LabelImg is an open-source graphical image annotation tool that allows fast and easy annotation, generating an ".xml" file with all the annotated bounding boxes in an image and the associated labels.

Anaconda is a free and open source Python distribution focused on data science and machine learning related applications. Because of the easy debug and fast experimentation, as well as the important machine learning-related libraries it offers, this is a great tool to use

in data science applications. Keras, a high-level neural networks API, is capable of running on top of Tensorflow (an open-source software library developed by Google for machine learning) reducing programming complexity for the developer. It is an extended library with most models already implemented as well as many important functions which allow the user fast and easy experimentation.

PyCharm is an Integrated Development Environment that integrates the Anaconda distribuition making debugging an easy task. It is also a useful tool for deployment which is an important feature since Deep Frisian was used remotely.

CUDA, an Nvidia toolkit that allows parallel processing on an Nvidia GPU, was used. Enabling parallel processing on a GPU drastically improves the training time as well as the prediction time of an ANN when compared to CPU processing.

3.3 Dataset

Dataset in deep learning is the data available to train an ANN. Dataset plays an important role when training a new model because the quality of the labelling has an impact on how well the ANN learns what is intended. Another important aspect is the size and variability of the dataset. A big dataset with high variability will allow the network to learn different representations of the same class in different contexts leading to a better prediction model.

The first step in deep learning is to ensure the dataset meets the above criteria.

The data available for this project was formed by 80 pictures taken by a drone. This images were 7952 pixels width by 5304 pixels height with 3 channels depth.

All the pictures were taken the same day so there is almost no variability in weather and lighting conditions. Apart from that, the number of images available to train the network is insufficient and so data augmentation was needed.

Data augmentation is the process of creating new data from already available data. There are several operations that can be performed on the original images such as rotation, flipping, translation, resizing, zooming, contrast and brightness changes, skewing and so on. Data augmentation has been proved to increase the accuracy of predictions [15] and in small datasets it is an efficient way of generating new data. In Dataset Preparation it is described how data augmentation was performed in this work.

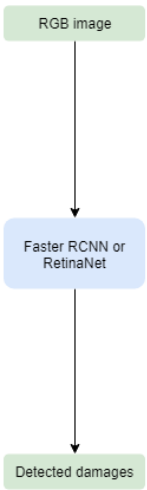
Chapter 4

Proposed Solution

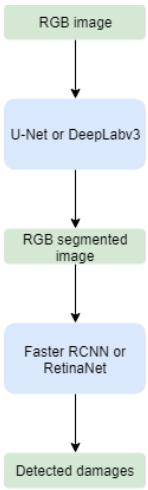
Next, we describe the approach we used and the steps we took in this work. We also show how we prepared the dataset and the considerations wee took when implementing the different architectures.

4.1 Overall Proposed Model

The main goal of this work is to create a model that can accurately detect damaged areas in windmill blades from RGB images. In order to achieve this, the object detection networks Faster RCNN and RetinaNet will be evaluated. However, in most images the region of interest (the blade) is just a small portion of the image. In these images almost every pixel corresponds to background, which means that it is not relevant for the detection of damage. Taking this aspect into account, two models are proposed: one with blade segmentation using U-Net and DeepLabv3 for that purpose and another one without. These two approaches are shown in a diagram in Figure 4.1.



(a) Model with damage detection only.



(b) Model with blade segmentation and damage detection.

Figure 4.1: Diagram of the two different proposed models.

4.2 Framework

The framework of this work is represented in Figure 4.2 and can be divided into three main steps. The first step is to create ground truth by annotating and labelling all the available data. This is an important task because the quality of such annotations has an impact on the performance of the network. Badly annotated pictures will lead to a poor learning thus making incorrect predictions. After, given the dataset has few images, data augmentation is performed since ANNs need a large number of information to be trained with. All the newly generated data is then divided into train, validation and test images. The second step is the network setup and training step. It is divided in the two different approaches explained before in Figure 4.1. The third and final step is to use the differently trained models to predict on the test images in order to measure their accuracy. We will be possible to assess the impact that segmentation has on the damage detection accuracy. We will also analyse the results of the object detection networks and assess which of them works best.

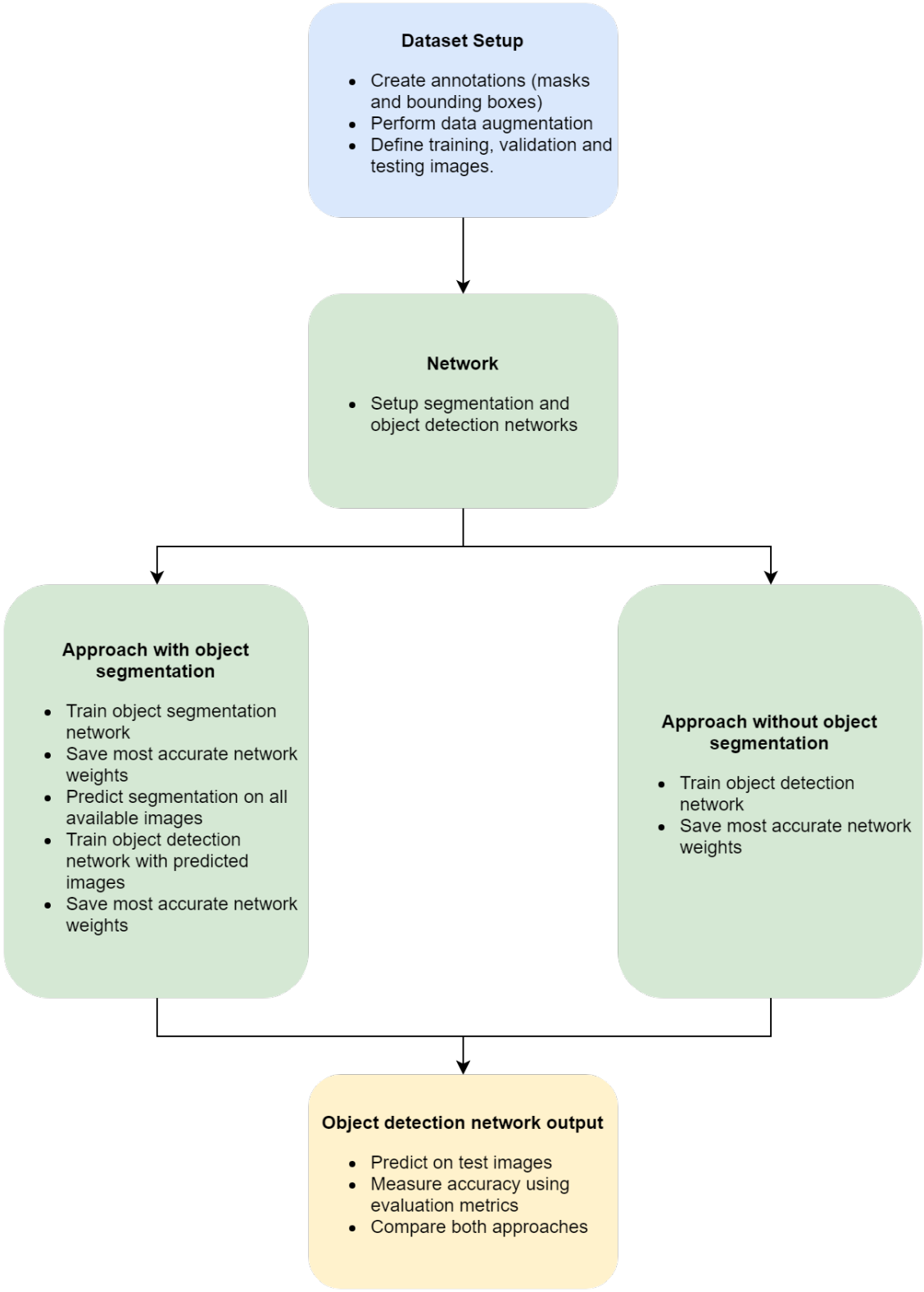


Figure 4.2: Framework diagram.

4.3 Dataset Preparation

Ground truth are the annotations to which the network will compare its output and compute the loss in order to adjust itself and learn what is pretended. In order to successfully train a segmentation network, it is required to create ground truth images. These images are pixel-wise annotations of all the classes we want the network to learn and are fed to the network in the training stage. In this work there was only one class - blade region - since it was only required to know if a pixel belonged to the windmill blade or not. Binary images were therefore generated where the white pixels, maximum pixel value, represent the blade and the black pixels, minimum pixel value, represent everything else.

Using GIMP, one image or mask, was made for each picture where the white pixels represent the area of interest, the windmill blade, and the black pixels represent everything else, the background as seen in Figure 4.3. In the ground truth, we only considered the nearest windmill blade while annotating and considered the remaining of the image as background.

(a) Picture taken by drone. (b) Corresponding mask.

Figure 4.3: Example of picture taken by drone and the corresponding ground truth mask.

For the object detection network, in which the goal is to detect the damages, the ground truth are bounding boxes with an associated label depending on the type of damage. The labelling was done by a windmill blade expert using LabelImg [24]. This software is available at [24]. The expert annotated three different classes of damage: 'SD', 'Erosion' and 'Appendage'.

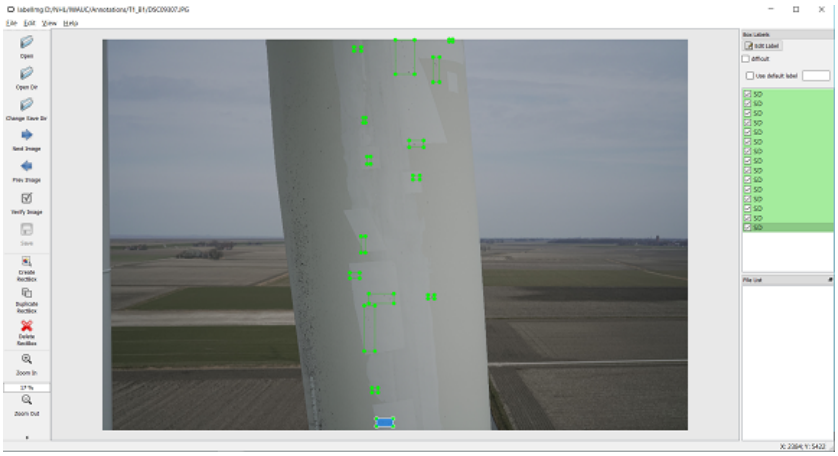


Figure 4.4: Example of annotated picture using labelImg

Since there were only 80 pictures taken it was important to perform data augmentation in order to have sufficient data to train the network. The pictures available were extremely large, as said in Section Dataset, when compared to the typical sizes (smaller than 512 by 512 pixels). However, by resizing them this much (more than ten times smaller) a lot of information, such as small damages, important for the object detection networks, would be lost. Hence another approach was used. Taking advantage of the big pictures dimensions all of them were cropped in smaller 1024 by 1024 overlapping patches. Figure 4.5 exemplifies how the cropping was done.

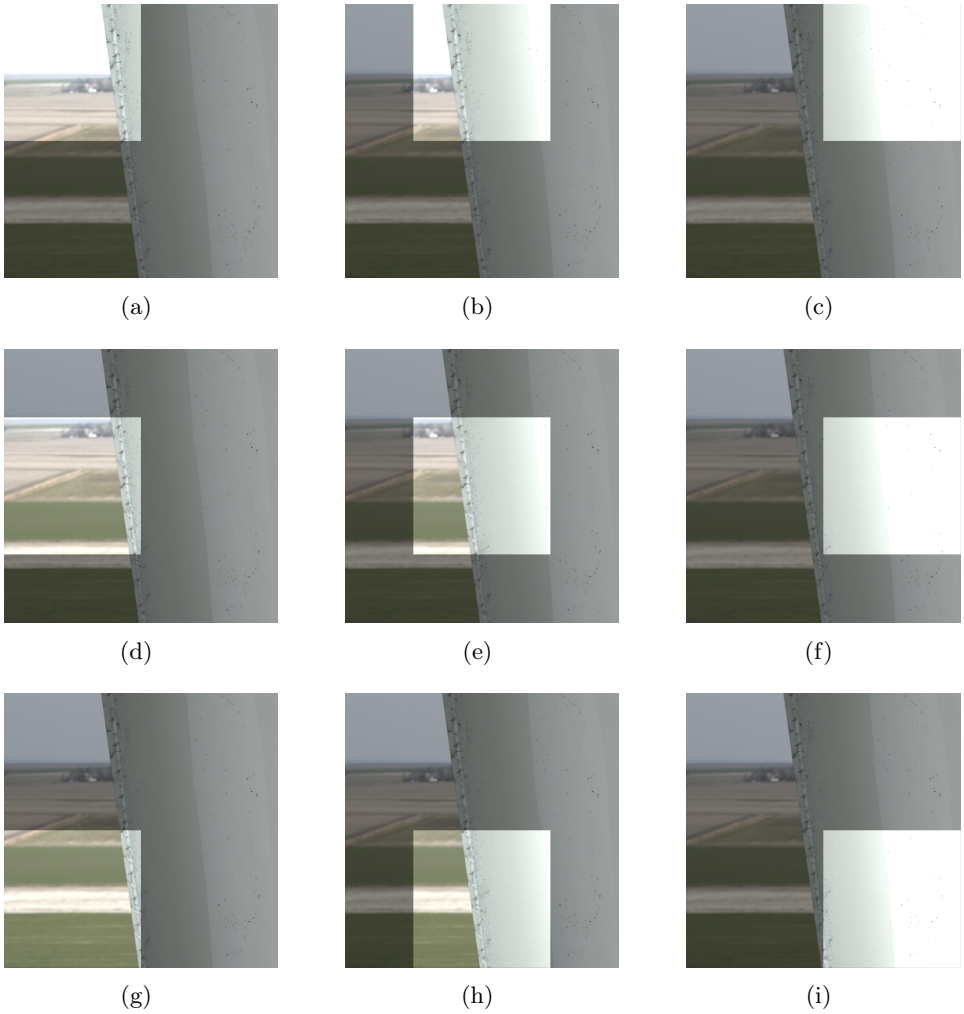


Figure 4.5: Example of picture cropping.

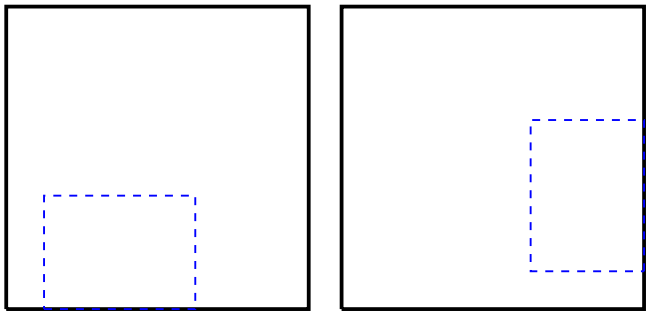
In Figure 4.5 the light area represents the cropped patch. Each patch has a size of 1024×1024 pixels and between each one there is a 512 pixels step causing the patches to overlay. With this approach it is possible to have 126 patches from a single picture.

In most pictures, the background is responsible for the majority of the given information. By cropping all the pictures as explained above this would create an unbalanced dataset, meaning there would be too much information about background and lack of information about the windmill blades. For this reason, all the patches that contained a damaged region,

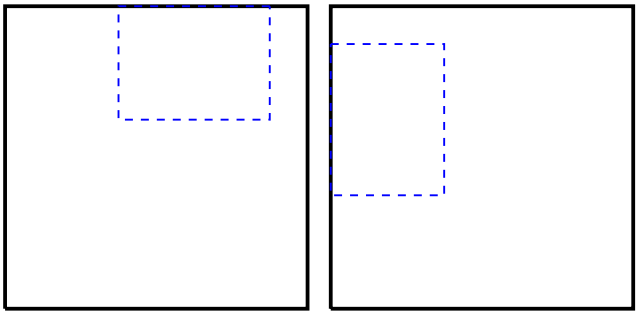
were rotated with angles of 90°. This way, there is no information getting lost. If the rotation angle was another, the rotated patch would not fill the initial dimensions. The corners of the original patch would be outside the new rotated patch and the corners of the new patch would not have image. By rotating the patch with angles of 90°, the rotated patch always fits the original patch dimensions.

Rotating the patches with damage also helps the network to learn different representations of the same object, damage, thus becoming rotation invariant. Note that there is no limit for the number of labelled damages in one patch meaning the number for labelled damages can be zero when are no labels or any other number depending on how much damaged regions there are in that patch.

By cropping and rotating the images, the original bounding boxes from the full sized images must be adjusted. In Figure 4.6 we show an example of how a generic bounding box was adjusted depending on the rotation of the tile. First we would check if there was a bounding box in the tile, completely or partially, or if the tile was inside a bounding box. In the last two cases the bounding box limits would be adjusted in order to fit inside the tile. After having redefined the coordinates, we performed rotations and adjusted the coordinates, once again, so they would still contain the object. Considering Figure 4.6(a) and the upper left point as (x_{min}, y_{min}) and the bottom right point as (x_{max}, y_{max}), the coordinates for Figure 4.6(b) would be $(y_{min}, 1024-x_{max})$ and $(y_{max}, 1024-x_{min})$. For Figure 4.6(c) and 4.6(d), the coordinates for the first tile would be $(1024-x_{max}, 1024-y_{max})$ and $(1024-x_{min}, 1024-y_{min})$ and for the second tile would be $(1024-y_{max}, x_{min})$ and $(1024-y_{min}, x_{max})$



(a) Original position. (b) 90° Rotation



(c) 180° Rotation (d) 270° Rotation

Figure 4.6: Visual example of bounding box adjustment depending on the rotation of the tile.

Since we used CSV files as input for the training scripts of the object detection networks, all the bounding boxes coordinates as well as the respective label were saved into the file. Each line of the CSV file should contain the information of each image. For a given image without the bounding box the line in the CSV file should be "path_to_image,,,,,". In the case of a bounding box the line the should be "path_to_image, x_min, y_min, x_max, y_max, class". In case of more than one bounding box per image, a new line should be added for each bounding box with the same path and respective coordinates. The python code developedd and used to generate the ground truth is shown in Appendix B.

When training, the goal of the network is to adjust its weights to better fit the output in relation to the ground truth. This is saying that the network tries to adjust itself to the training dataset, to a particular set of images. However, we want generalization of the dataset features in order to better predict those same features on new images. Therefore we split the dataset in training set, validation set and testin set. The testing set was composed by tiling 16 of the original images without overlaping, resulting in 560 new images. The remaining 64 images were cropped with overlaping tiles resulting in 10,125 images with 3,196 annotations of 'SD', 700 annoatations of 'Erosion' and 144 annotations of 'Appendage'. From this images, we split again randomly with a ratio of $8:2$ creating, this way, the training set and validation set.

For evaluating the accuracy of the model, we used the testing set. These images were not previously seen by the network and, therefore, the model is not adjusted to this set of images. This helps to have a glance of how the model may respond in a real case scenario.

4.4 Object Segmentation

To achieve the best possible object segmentation, U-Net and DeepLabv3 with different input image sizes were tested. The previous obtained patches were resized to 512×512, 256×256, 128×128 and 64×64 pixel images. With the bigger images it is possible to get all the details of the image and no information is lost however processing time will be slower. On the other hand, with the smaller images, the details are lost but the big objects in the image remain and the processing time is faster.

In the particular case of this work, regarding object segmentation, small details don't matter since the goal is to perform segmentation of the blades which are large objects in the input images.

The architectures of the networks must be adapted for each input size. It is explained how the networks were set up and all the considerations made in the next subsections .

4.4.1 U-Net

U-Net was designed using Keras and was based on the original paper [19]. For the 1024×1024 pixel images the used architecture is the one shown in Figure 4.7. The consideration behind the different architectures tested for the different image input sizes is that in the lowest resolution the feature map should be 16×16. While other approaches might accomplish better results, this was the rule set in order to establish a general trait to the different architectures. Every other aspects remain the same.

Unlike the original version, padded convolutions were used, instead of unpadded convolutions, in order to maintain the spacial dimension and to compute an output with the same dimensions as the input. Padded convolutions add frame of zeros around the input producing an output of the same size. The spatial information is halved in each block resulting in a 16×16 feature map in the lowest resolution while the semantic information is doubled.

At the final layer a one by one convolution using sigmoid function as activation function is used to map each 16-component feature vector to the desired number of classes. In this case the number of classes is one since the only class is "windmill blade". Everything else is considered background and, therefore, does not belong to any class.

In this work we test the same architecture with two different activation functions - ReLu and ELU - to assert which one produces the most accurate model. In Apppendix A we show the implementation of U-Net using ELU as activation function.

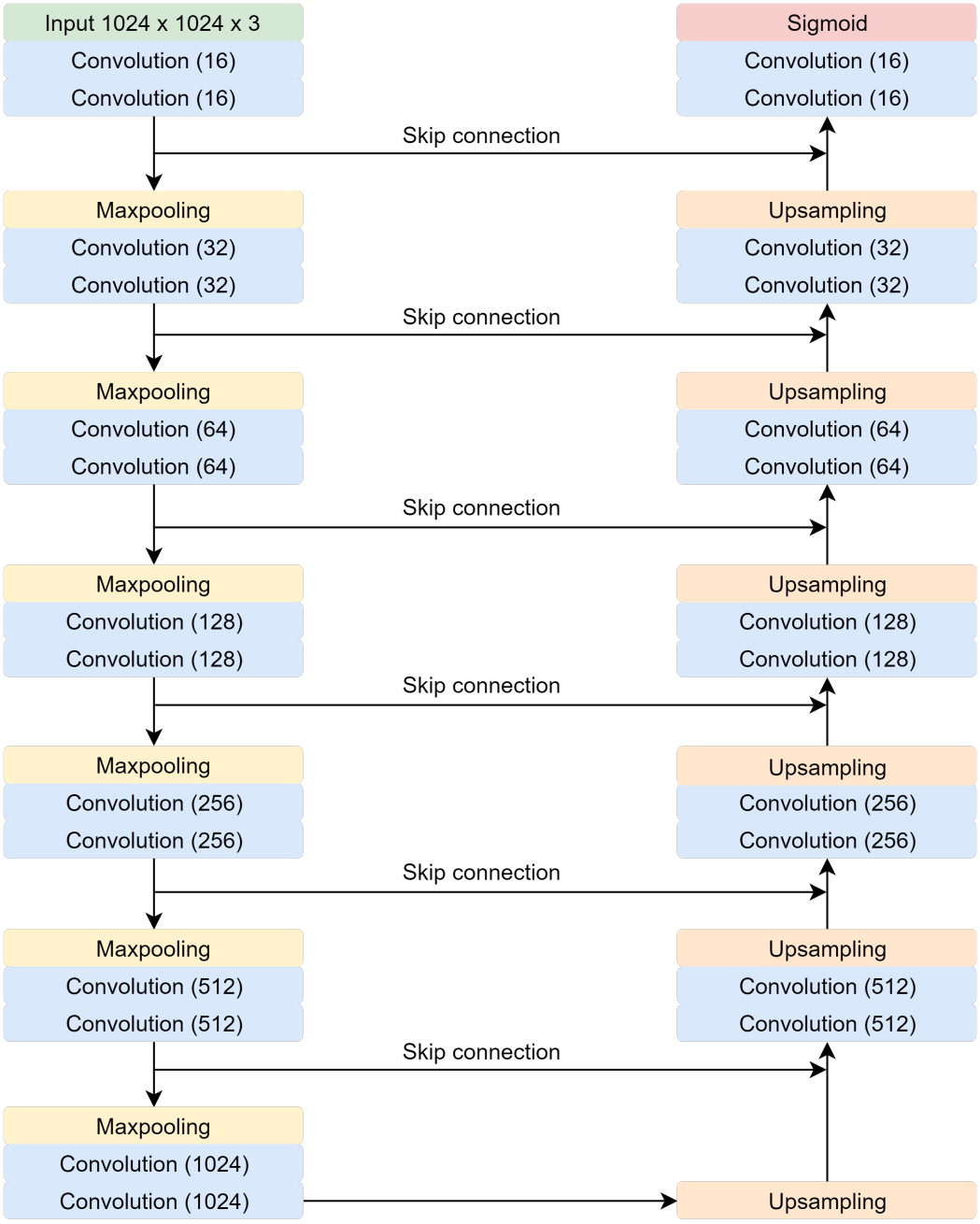


Figure 4.7: U-Net architecture for 1024 by 1024 pixel sized input image.

All the convolutional layers use $3 \times 3 \times depth$ sized kernels, meaning that the receptive field is 3×3 in every layer. The number of kernels, that is also the depth of the output, by layer, is indicated in Figure 4.7 between brackets. The deeper the layer is, the deeper the kernels are in order to make possible the extraction of more complex and specific information and patterns. From here, it is clear that a deeper network requires a larger number of trainable parameters.

The number of parameters is calculated with $(Kernel\ height \times Kernel\ width \times Input\ depth + 1) \times Number\ of\ kernels$. For example, in the first convolution layer there are 16 filters. Since the kernel is 3×3 and the input image depth is 3, the total number of trainable parameters for this layer is $(3 \times 3 \times 3 + 1) \times 16 = 448$ parameters. Since in U-Net the only layers with trainable parameters are convolutional layers, by applying the previous rule to every convolutional layer in the network, the total number of trainable parameters is $31,458,433$.

There are a large number of hyperparameters which need to be defined before training the network. The hyperparameters were set by taking into account the typical and commonly used hyperparameters in U-Net implementations since there is no strict rule to set them. Moreover some parameters, like batch size and number of filters in each layer, were tuned by trial and error. The hyperparameters used in all U-Net architectures implemented in this work are shown in Table 4.1.

Hyperparameter	**Setting**
Activation function	ReLU or ELU
Loss function	Binary Cross Entropy
Convolution border mode	Same
Stride	1
Kernel size	3×3
Optimizer	Adam
Initial learning rate	0,0024
Batch size	8

Table 4.1: Hyperparameters used in all U-Net architectures implemented.

As mentioned in Section 2.2.4, the loss function used in training is binary cross entropy. This loss function calculates the error in each pixel of the image by using Equation (2.1). The loss, in this case, is the average error of every pixel in a prediction.

In order to maintain the 16×16 feature map in the lowest resolution, some changes were made in the architecture to make 512×512 images training possible. The last block of the enconding part and the first block of the decoding part were removed as shown in Figure 4.8. Since this implementation is not as deep as the previous one, the number of parameters is lower. In this case, the number of parameters is **7,862,401**. Also, because the size of the input image is 512 instead of 1024, this version is faster to train and predict and demands less computation.

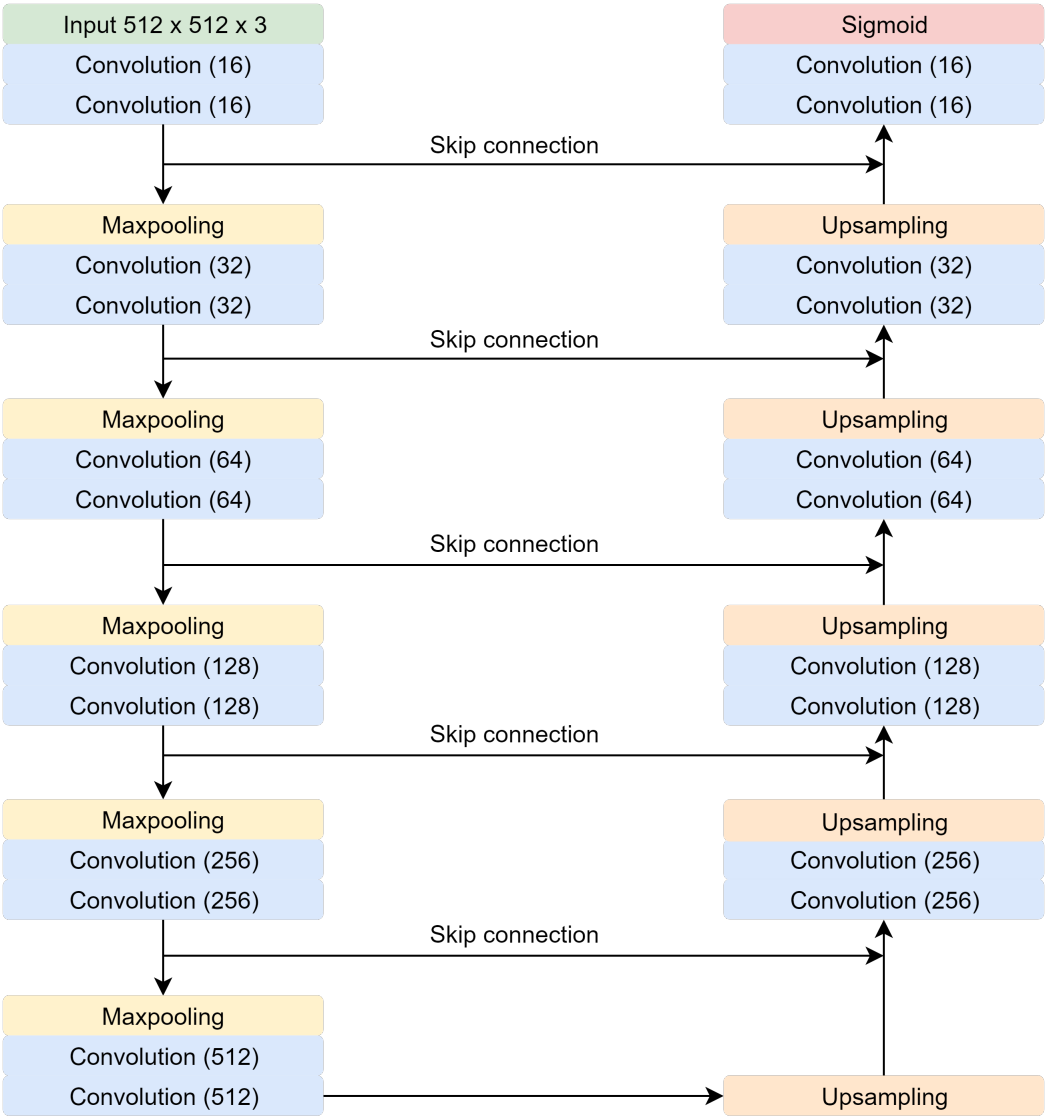


Figure 4.8: U-Net architecture for 512 by 512 pixel sized input image.

By keeping the lowest resolution feature map at 16×16, the smaller the input image size is, the shallower will the network be. As a consequence, there will be much less trainable parameters and the number of calculations drastically decreases for small input image sizes thus making the network faster when compared to big sized input images. The total number of parameters for each U-Net implementation are shown in table 4.2.

Input Size	**Trainable params**	**Non-trainable params**	**Total**
1024^2	31 458 433	0	31 458 433
512^2	7 862 401	0	7 862 401
256^2	1 962 625	0	1 96 2625
128^2	487 2971	0	487 297
64^2	118 273	0	118 273

Table 4.2: Parameters for each U-Net implementation.

The of the number of parameters affects the time each model will need to train and predict. The difference of parameters between each model will make the models with fewer parameters to be faster than the models with more parameters. However, we cannot assess yet how each one will perform.

4.4.2 DeepLabv3

To test DeepLabv3, an implementation available at GitHub [16] was used. There are more hyperparameters, when compared to U-Net, that need to be set such as the multi-grid parameters and the number of blocks of the model. All the choices we made are based on the original paper [5] while keeping in mind our limited resources such as hardware and time. By training a very deep and complex network with big images, such as 1024×1024 pixel images, a lot of GPU memory is required. Furthermore, the time necessary to train and to predict a single image increases with the complexity of the network.

The settings we used are explained next and were chosen taking into account the best results the original paper got. As feature extractor, or backbone, we decided to use ResNet-50 instead of ResNet-101. In the original paper it is shown that, for the same conditions, the performance of ResNet-50 is not very different from ResNet-101. For example, with output stride 16 and with 6 blocks, the accuracy of ResNet-50 is only 1% lower than the accuracy of ResNet-101. The computational and memory cost of ResNet-101 is too high for a 1% improvement in accuracy when compared to ResNet-50. By building such network with 6 blocks, the dilated convolutions will have rate 2, 4 and in blocks 4, 5 and 6, respectively. The first 3 blocks will be just like the original ResNet-50 implementation. The chosen output stride was 16 rather than 8. Again, the performance improvement of using output stride 8 was just 1% higher than when using output stride 16 at the cost of more memory usage. Regarding the Multi-Grid Method the best accuracy, using 6 blocks, was achieved with Multi-Grid (1,2,1). Thus we choose also this Multi-Grid.

Hyperparameter	**Setting**
Loss function	Binary Cross Entropy
Optimizer	Adam
Initial learning rate	0,0024
Batch size	8
Backbone	ResNet-50
Blocks	6
Output stride	16
Multi-Grid	$(1, 2, 1)$
Upsampling	UC, bilinear interpolation or transposed convolution

Table 4.3: Hyperparameters used in all DeepLabv3 architectures implemented.

Deconv Method	**Trainable params**	**Non-trainable params**	**Total**
Transposed Convolution	74 528 256	101 248	74 629 504
Bilinear Interpolation	74 266 369	101 248	74 367 617
DUC	74 856 706	101 760	74 958 466

Table 4.4: Parameters for each DeepLabv3 implementation.

Three different upsampling techniques, UC, bilinear interpolation and transposed convolution, were tested in order to assess which yields the best segmentation result. In table 4.4 are shown the total number of parameters for each different tested version of DeepLabv3. In DeepLabv3 the size of the input image does not affect the number of parameters of the network.

Comparing to U-Net, DeepLabv3 has much more parameters. This is due to the fact that DeepLabv3 uses ResNet-50 as backbone, which by itself has already more than 25 million parameters, thus greatly increasing the number of total parameters in this architecture.

4.5 Object Detection

Regarding object detection, Faster RCNN and RetinaNet were tested. In these networks only the original 1024×1024 tiles were used. This consideration comes from the fact that damages on windmill blades can be very small. By resizing the image, like we did for the segmentation networks, these small damages could simply vanish from the image thus resulting in faulty predictions.

All the considerations for each network are explained next.

4.5.1 Faster RCNN

In this work, F-RCNN was tested with ResNet-50 and VGG as feature extractors. The implementation used is available in GitHub [11]. In F-RCNN it is necessary to specify the anchor boxes, their scales and ratios, as well as the number of RoIs per image. We defined three different anchor boxes ratios, represented in Figure 4.9, (1×1), (1×2) and (2×1) and the anchor boxes scales 8, 32 and 128. Therefore, for each anchor in the image there will be 9 anchor boxes of sizes (8×8), (8×16), (16×8), (32×32), (32×64), (64×32), (128×128), (128×256) and (256×128). We also defined the stride of the anchors as 16 which means an anchor is set and the 9 anchor boxes are generated each 16 pixels in x and y directions. These parameters were choosen in order to best fit the ground truth bounding boxes.

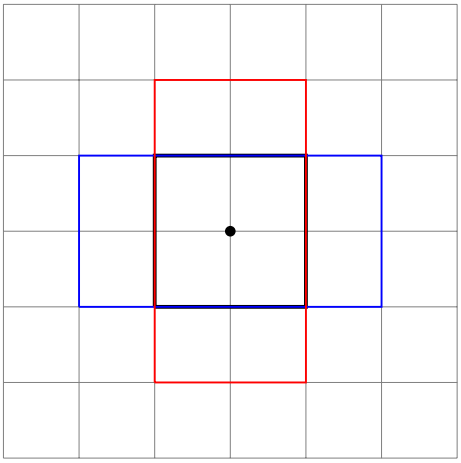


Figure 4.9: Anchor boxes example.

In order to limit the computational power needed, only 32 random ROIs per image of the RPN are sent to the classifier while training in each iteration. We also defined a minimum IoU of 0.5 meaning that a predicted bounding box is only considered a good prediction when overlaps at least 50% of the ground truth bounding box. If the IoU is less than 10%, the bounding box is considered a wrong prediction and if the IoU is between 50% and 10% it is considered an hard negative example. The number of parameters for the different Faster RCNN implementations are described in table 4.5.

Backbone	Trainable params	Non-trainable params	Total
ResNet-50	28 235 955	106 240	28 342 195
VGG16	136 668 019	0	136 668 019

Table 4.5: Parameters for each Faster RCNN implementation.

4.5.2 RetinaNet

For RetinaNet we used an implementation available in GitHub [8]. When testing RetinaNet, like we did in F-RCNN, we used ResNet-50 and VGG as feature extractors. In this architecture there are not many hyperparameters that need to be set and we use the original paper hyperparameters. For example, regarding anchor boxes in FPN, we used anchors of areas 32^2, 64^2, 128^2, 256^2 and 512^2 with aspect ratios of $1:2$, $1:1$ and $2:1$. The strides are 8, 16, 32, 64 and 128 for each size, respectively. There are also three defined scales applied in the bounding boxes which are 2^0, $2^{(1/3)}$ and $2^{(2/3)}$.

Backbone	Trainable params	Non-trainable params	Total
ResNet-50	36 276 717	106 240	36 382 957
VGG16	23 407 725	0	23 407 725

Table 4.6: Parameters for each RetinaNet implementation.

A total of thirty three models will be trained and evaluated. From this total, twenty five are object segmentation models and eight are object detection models. Regarding object detection, the models will be trained with a single generic class, damage, and with the specific classes of the different damages.

Chapter 5

Tests and Results

In this Chapter we present all the tests made and the results we obtained. We also discuss the results and choose the best model for the object segmentation and for the object detection tasks.

5.1 Object Segmentation Model

In this section we present the results for the object segmentation models that we tested.

5.1.1 U-Net Models

In table 5.1 we show the results for the different input sizes implementations of U-Net using ReLU as activation function. In order to correctly calculate de Dice coefficient, we also threshold all the predictions at 0.2. This means that all the pixels with probability lower than 0.2 turn into 0 and all the pixels with probability higher than 0.2 turn into 1.

Input	**Loss**	**Dice Coefficient**	**Step time (ms)**
1024^2	0.128	0.340	52
512^2	0.212	**0.907**	15
256^2	0.147	0.874	6
128^2	0.167	0.828	5
64^2	**0.126**	0.793	**4**

Table 5.1: U-Net results with ReLU as activation function and threshold at 0.2.

From table 5.1 we can tell that the model with 512×512 input size was the most accurate, between the architectures of U-Net with ReLU as activation function, with 90.7% Dice coefficient. On the other hand the 1024×1024 input size achieved the lowest Dice coefficient with only 34.0%. This model might have such a low Dice coefficient because the damages are big compared to the size of the kernels thus not being possible for the kernels to have information about the context of that set of pixels. Therefore, the network might assume that that area is not part of the blade since it is not aware of the overall context of that same set of pixels. Differently, we see that lower input size image models have a much higher accuracy. In

these models, the input images were the same but they were resized. By resizing the images some of the damages turn into very small areas or might even disappear becoming possible to the kernels to have a different perception of the context. Resizing to lower dimensions, in this particular case, can be understood as a low-pass filter since only the bigger objects and features remain in the image while the noise, in this case is the damage, is eliminated. Yet, since the architectures for smaller input sizes are shallower than for bigger input sizes, the segmentation for more complex scenarios of windmill blade is faulty.

In Figure 5.1 some examples of segmentation are shown where all the images are the result of merging the tiles back together again. We can confirm that, overall, the 512×512 model has the best performance. However it struggles to perform a correct segmentation in some examples. In the first and third rows some blobs are present in the blade which could delete some damages from the original image when this mask would be applied. In the second row the blade is totally segmented, however on the left side of the image some white areas appear. This white areas are not the windmill blade but the windmill pole. Although we do not take this part of the windmill into account in the ground truth, the characteristics, such as colour and shape, of the pole are very similar to the characteristics of the blades causing this partial segmentation of the pole to happen.

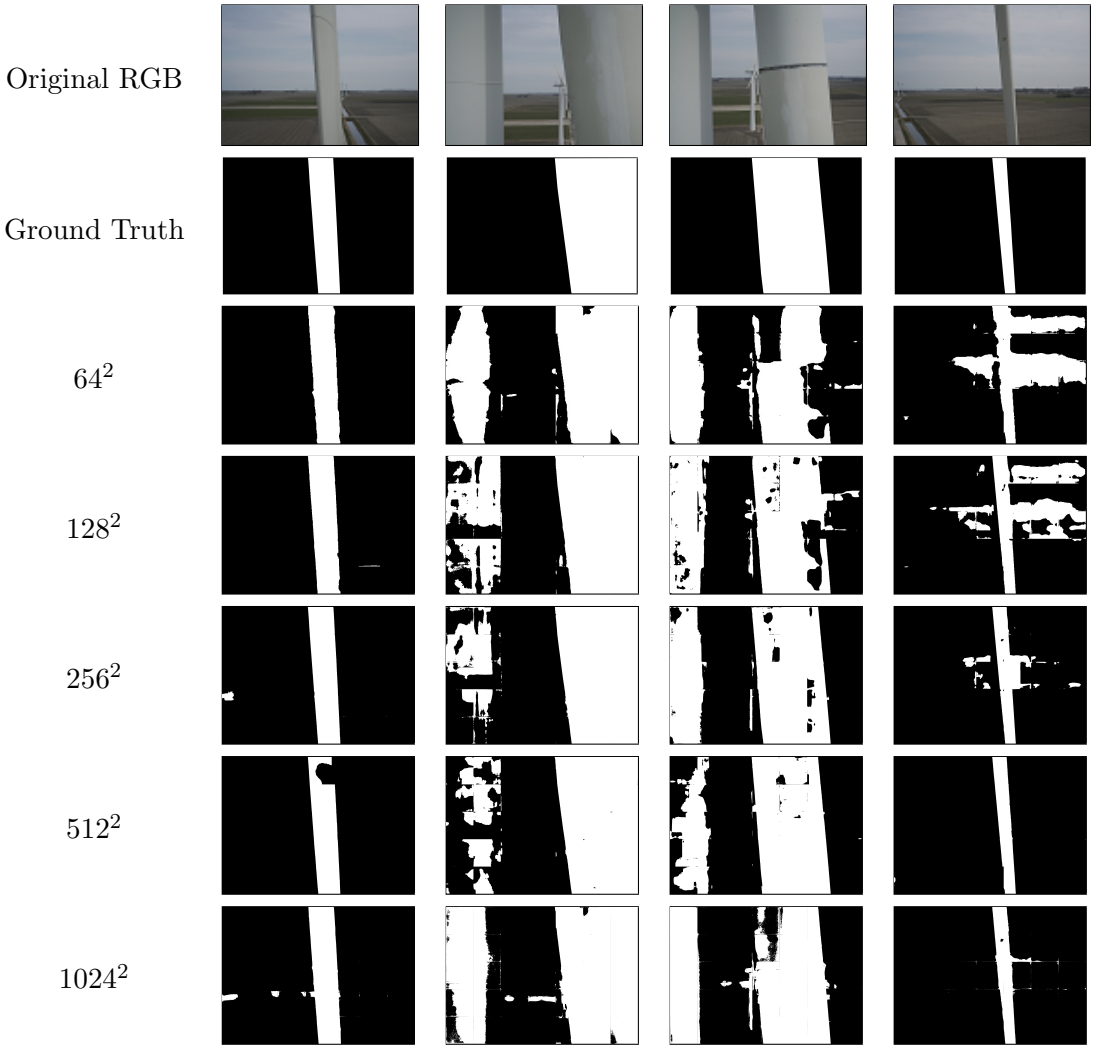


Figure 5.1: Examples of U-Net segmentation with ReLU activation.

Using ELU, instead of ReLU as activation function, the results obtained are shown in table 5.2. In this case, the approach with the best Dice coefficient was, by a big difference, the one with 64×64 input size with 82.5% Dice coefficient. In this case, even though the network is the shallower of all due to its input size, it performs better than the other models. This might be explained, as said before, by the fact that small defects vanish once images are resized to lower dimensions.

It would be expected that the models that use ELU instead of ReLU would have a better performance since the vanishing problem does not occur so easily when using ELU. This way, all neurons contribute for the output. However, we can withdraw from Tables 5.1 and 5.2 that, with ReLU, greater Dice coefficients are obtained, which, generally, means better segmentations.

In Figure 5.2, a few examples of segmentation produced by this network, using ELU as activation function, are shown. Although the 64×64 model was the one that achieved the highest Dice coefficient, in practical uses we can see that the segmentation is very faulty with several blobs in the blade. In practice, the 512×512 input size model seems to perform the

Input	Loss	Dice Coefficient	Step time (ms)
1024^2	0.108	0.622	52
512^2	0.101	0.612	16
256^2	0.125	0.765	6
128^2	**0.09**	0.686	7
64^2	0.140	**0.825**	**4**

Table 5.2: U-Net results with ELU activation.

best segmentation among the U-Net models that use the ELU activation function.

It is visible that, in both ReLU and ELU implementations, all implementations are very sensitive to noise in the background and struggle to do a correct segmentation of the blades.

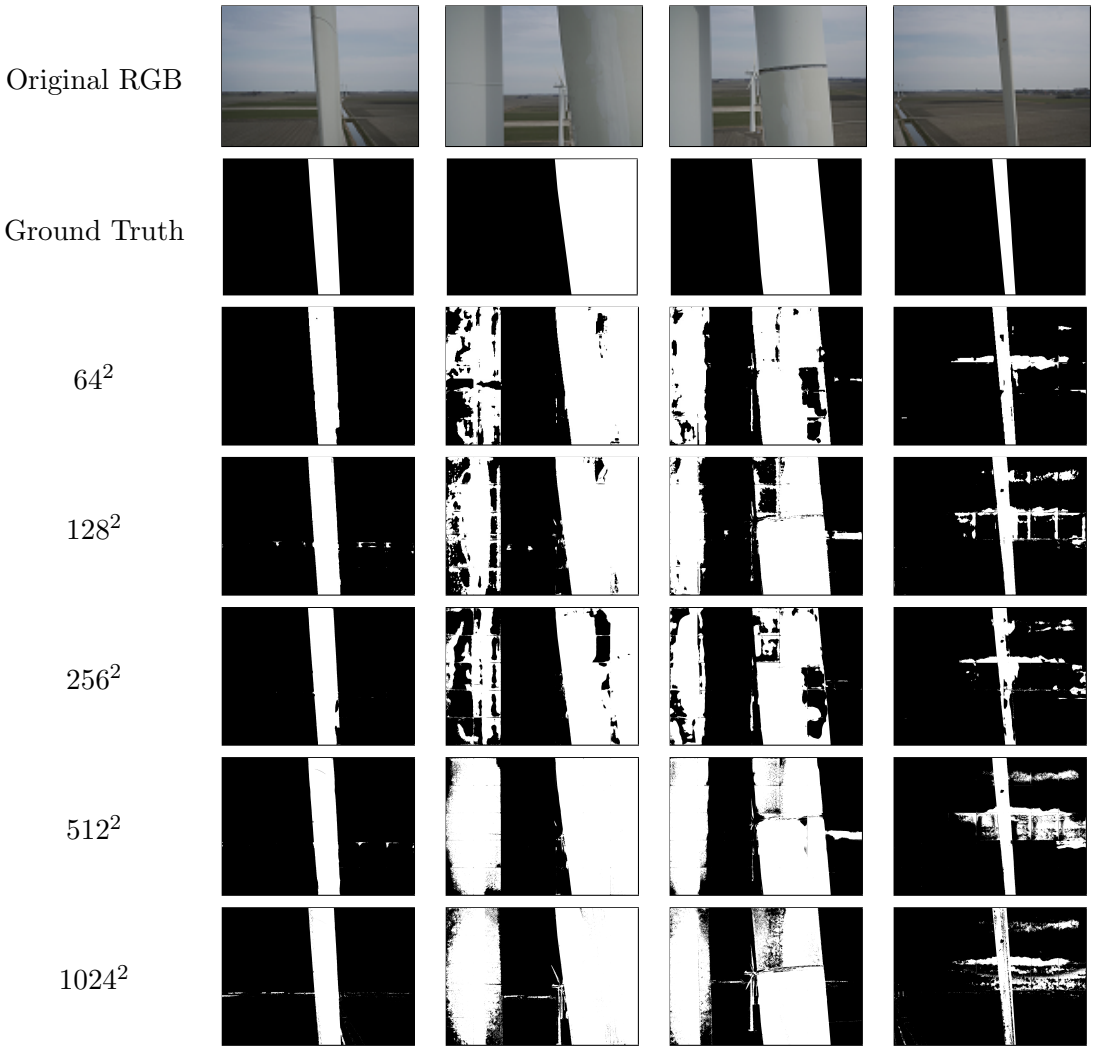
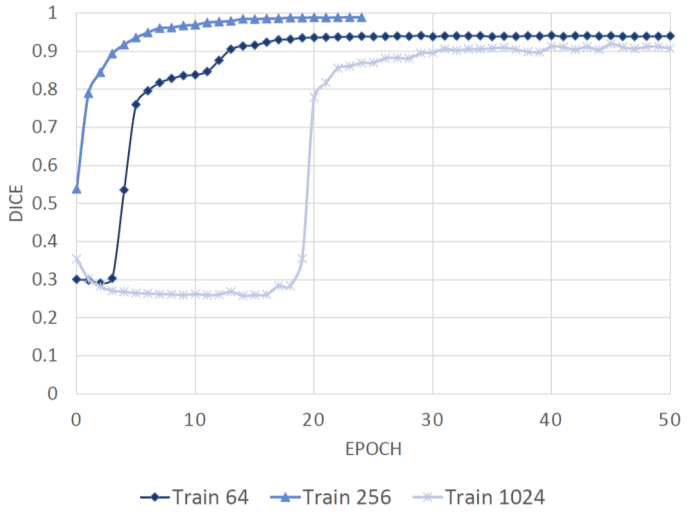
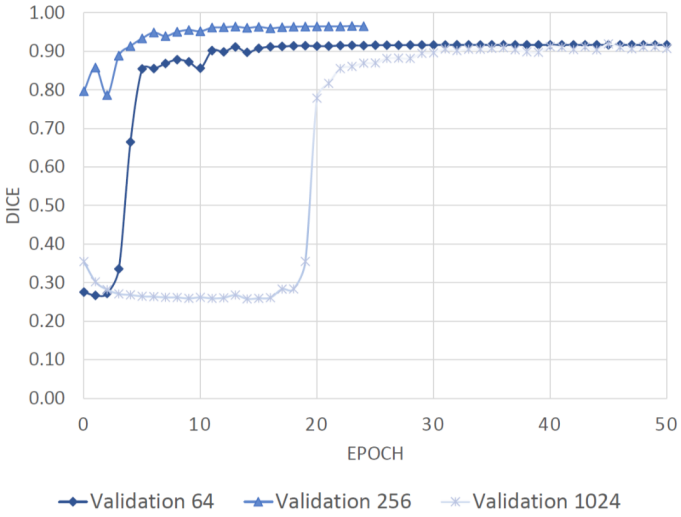


Figure 5.2: Examples of U-Net segmentation with ELU activation.

In Figure 5.3 the evolution of the Dice coefficient for the 64×64, 256×256, 1024×1024 models, using ReLU as activation function, during training stage and validation stage is shown. Here, the Dice coefficient of the 1024×1024 is much higher than when the model was evaluated with test images. This shows that the model adjusted too well to the images in the training and validation set failing to generalise, thus performing poorly with new images. This is a good example of overfitting. Also, the number of epochs needed for the models to converge to the final value of the Dice coefficient seems to be directly related to the size of the input. This behaviour may come from the fact that small images do not have so many details that can act as noise and therefore it is easier for the network to learn the right features.



(a) Training stage.



(b) Validation stage.

Figure 5.3: Evolution of the Dice coefficient of 64×64 (diamond), 256×256 (triangle) and 1024×1024 (cross) U-Net architecture models using ReLU as activation function.

5.1.2 DeepLabv3 Models

Regarding the models tested with the DeepLabv3, the results are shown next. Table 5.3 shows the results of the evalutation of the models with different input sizes, using DeepLabv3 with bilinear interpolation as upsampling method. The loss of the model with 1024×1024 input sized is very high, and the Dice coeffiecent very low when compared to the other models in the table. Therefore, no conclusions can be drawn from here. Although the Dice coefficient of the 128×128 and 256×256 models is very approximate, the smaller input size model shows a faster processing time. In real-time applications this is an important aspect to be taken into account.

Input	**Loss**	**Dice Coefficient**	**Step time (ms)**
1024^2	12.9	0.220	183
512^2	0.111	0.923	65
256^2	**0.027**	**0.95**	36
128^2	0.038	0.945	**28**
64^2	0.085	0.912	**28**

Table 5.3: DeepLabv3 results with using bilinear interpolation as upsampling method and threshold at 0.2.

When using bilinear interpolation as upsampling method, the difference in segmentation, for the different input sizes, is noticeable. In Figure 5.4 we can see coarse edges for 64×64 input size unlike the fine edges for 512×512 input size. This is the effect of the output stride. Given that the output stride is 16, the output image of the network will have 16 times lower resolution than the input. Therefore, for a 64×64 input image, the output will be an image of size 64×64 upsampled, by bilinear interpolation, from a 4×4 feature map. This means that each tile will have size 64×64 but with 4×4 resolution. For the 512×512 input size, applying the same principle, the output resolution will be 32×32. This allows finer edges and, consequently, leads to more resemblance between the ground truth and the prediction.

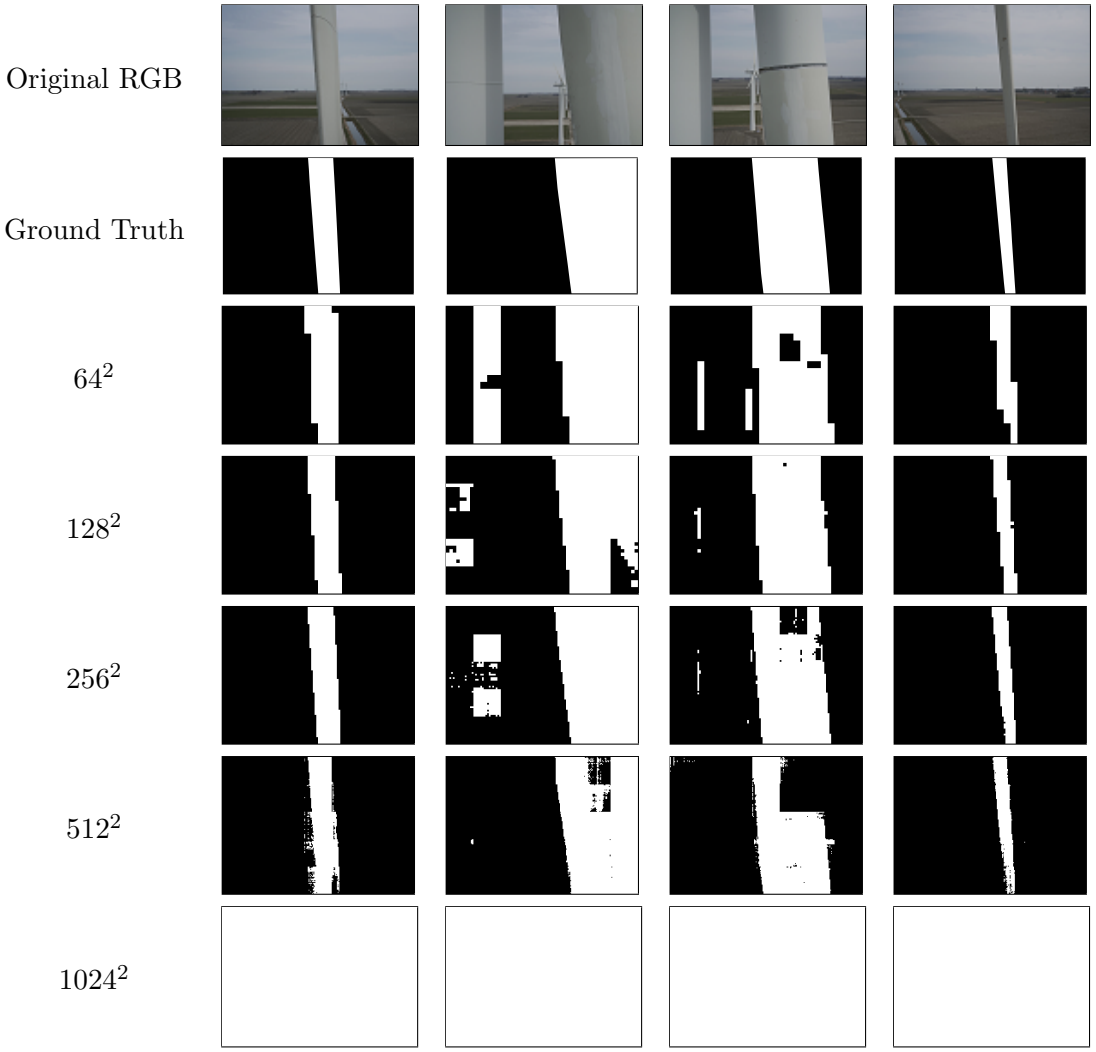


Figure 5.4: Examples of DeepLabv3 segmentation using bilinear interpolation for upsampling.

Regarding the use of transposed convolution, the results of the 1024×1024 are, again, inconclusive. The best model is also the 256×256 input size model with a Dice coefficient of 96.4%, 5% higher Dice coefficient than the second best model.

Input	**Loss**	**Dice Coefficient**	**Step time (ms)**
1024^2	12.9	0.220	174
512^2	0.039	0.391	66
256^2	**0.027**	**0.964**	35
128^2	0.072	0.915	29
64^2	0.063	0.883	**28**

Table 5.4: DeepLabv3 results with using transposed convolution as upsampling method and threshold at 0.2.

In Figure 5.5 we can see that the input size of the image seems not to affect the segmentation output. All the results are very similar and are very close to the ground truth. Models with larger input size are more prone to be affected by the background as we can see in the last two rows. Even though the 256×256 has a higher Dice coefficient than any other model, it still struggles to perform a good segmentation as is the case of the figure in the third column.

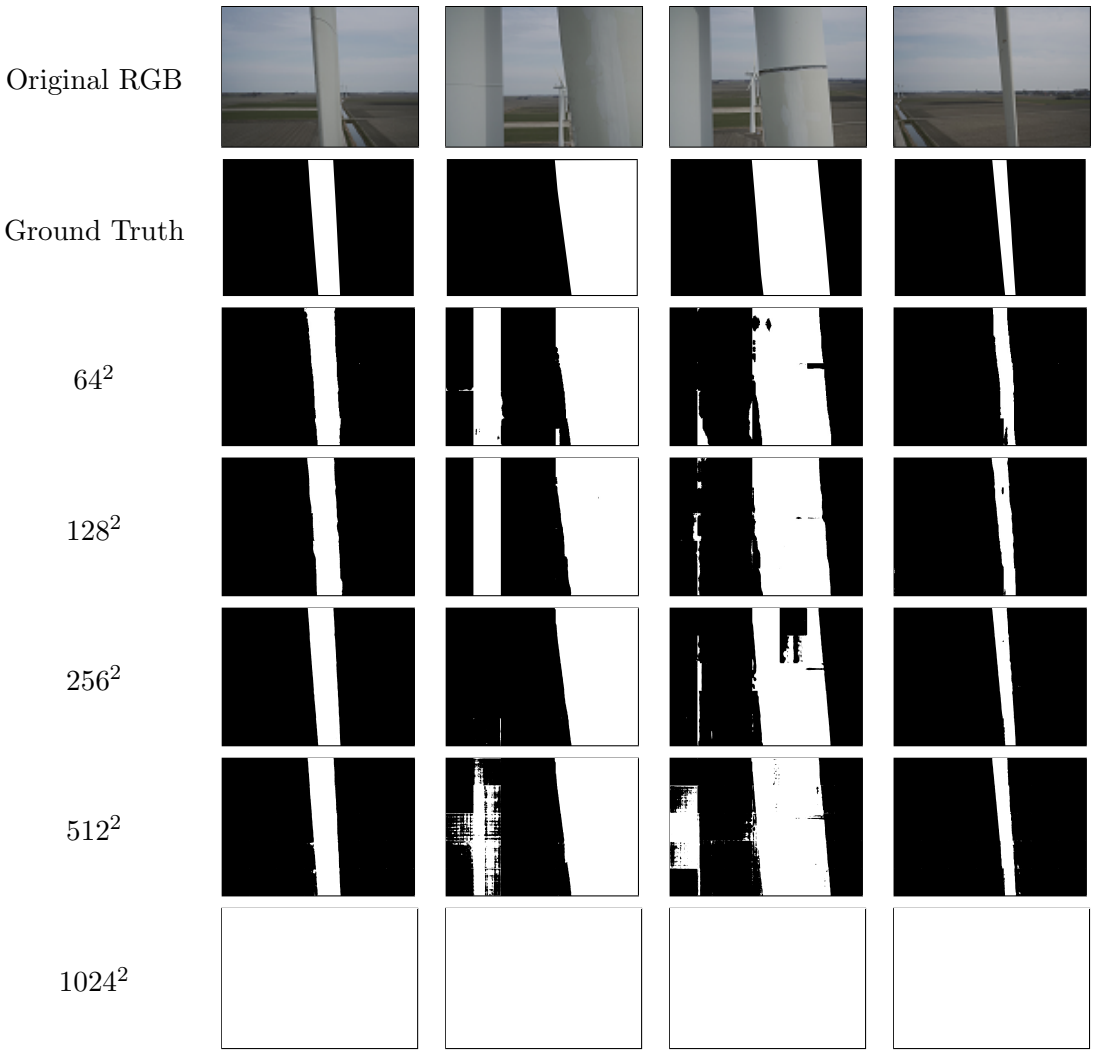


Figure 5.5: Examples of DeepLabv3 segmentation using transposed convolution for upsampling.

When performing upsampling with DUC, we got the results shown in Table 5.5. The best performing model was, yet again, the model with the input size 256×256 with a Dice coefficient of 96.4% when tested with the testing set. Again, the network performs better with smaller input sizes.

Input	**Loss**	**Dice Coefficient**	**Step time (ms)**
1024^2	0.686	0.246	182
512^2	0.165	0.429	68
256^2	**0.066**	**0.964**	39
128^2	0.197	0.862	31
64^2	0.109	0.915	**28**

Table 5.5: DeepLabv3 results with using DUC convolution as upsampling method and threshold at 0.2.

Figure 5.6 shows the output of DeepLabv3, for different sizes, when using DUC as upsampling method. Contrary to what was expected, this method did not produced the best results. It is visible that, in this case, this method revealed to be very sensitive to the background and performed poorly overall in practice.

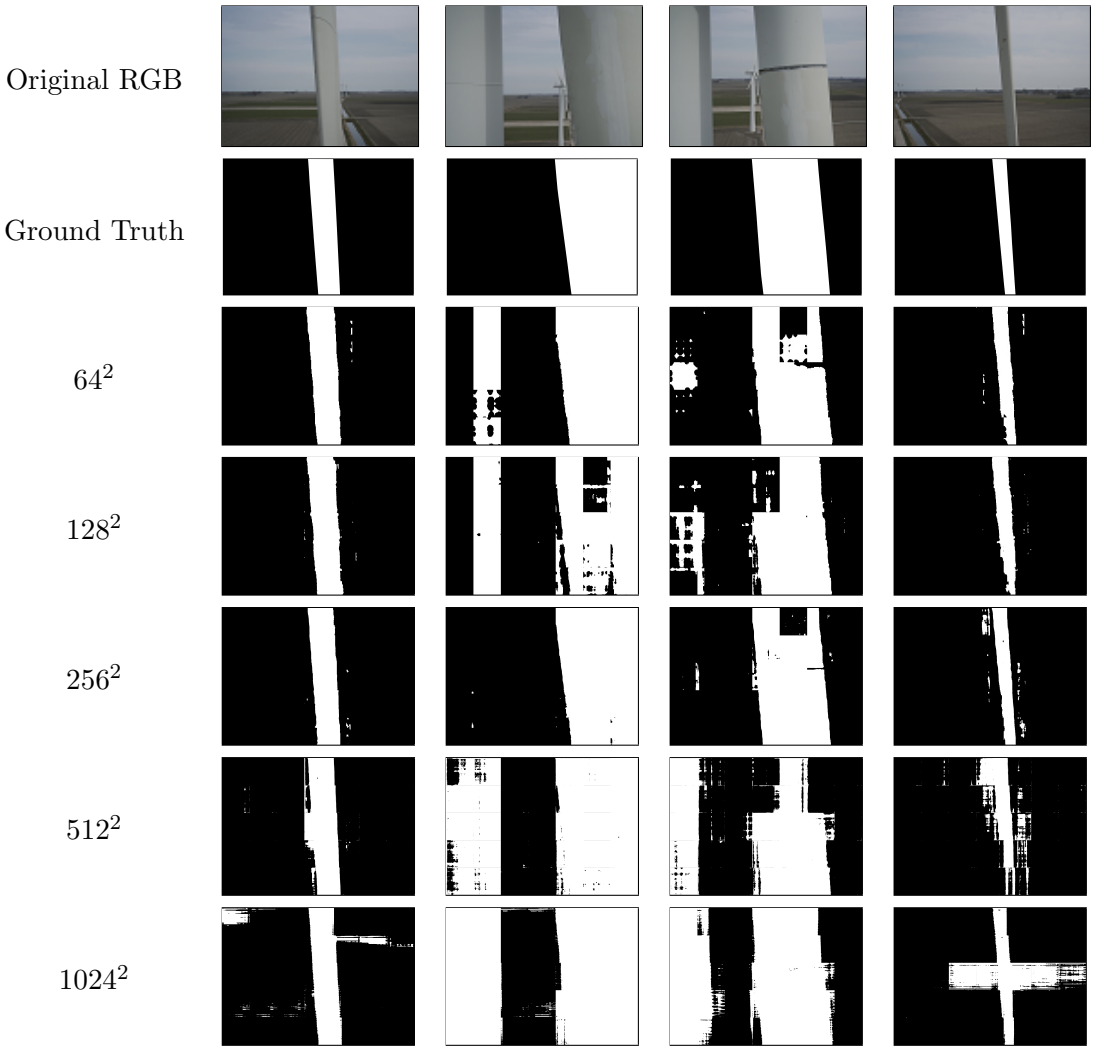


Figure 5.6: Examples of DeepLabv3 segmentation using DUC for upsampling.

DeepLabv3 showed to perform better with the 128×128 input size model with transposed convolution upsampling method. However, due to tiling and the fact that each tile is analysed separately, there are still a lot of errors in the segmentantion. The effect of tiling is reflected in the segmentation by the appearance of black squares in the region of interest, which is the blade, and small blobs that may appear on top of damages. When this happens, the object detection network will not be able to detect the damage since it was removed when applying the mask to the original RGB image, thus decreasing the accuracy of that network.

In every model of both architectures the effect of analysing the images tile by tile is evident. Since the networks predict on each tile individually, and not on the whole image, there is no way to know the full context of an object in the image. This as a visible and negative effect on the segmentation outputs. In some of the previous images there tiles that have pixels that belong to the pole of the wind turbine and are classified as being part of the wind turbine blade. On the other hand, there are pixels that belong to the blade that are labelled as background. Wind turbines from the background are also often segmented. Although these objects should not be segmented, since they are not part of the ground truth, they have the same features, such as shape and colours, as the features of the blades. In order to achieve a better segmentation other approaches should be tested. For example, when creating the ground truth we could consider all the windm turbines present in the image, including poles and background wind turbines. Another approach could be using the full original images. It could be also possible to use some post processing using classical computer vision techniques like dilation in order to remove the blobs. However, this is not recommendend since the goal of deep learning is to achieve a model capable of accurately predict what is supposed to predict whitout any kind of pre or post image processing.

Since none of the models achieved a satisfactory segmentation, without, at least, blobs in the blade, we decided not to use segmentation. These blobs, no matter how small, could eliminate damages of the image before the damage detection task. This would lead to a faulty detection of the damages which is the main goal os this work.

However, these models can be used with different approaches. For exemple, one or more segmentation networks could be used in parallel with an object detection network. This is called an ensemble method in which several models are combined in order to produce a better predictive model. This method gives the same input to all the models aggregating the different outputs afterwards. This result can then be fed to yet another model or simply processed in order to have a final result.

5.2 Object Detection Model

In this section we present the results for the object detection models that we tested.

5.2.1 Faster R-CNN Model

First, we show the results for the Faster R-CNN models trained with a single class, 'damage', in which we grouped together the three annotated classes. Table 5.6 and Table 5.7 show, respectively, the results for FRCNN with VGG as backcbone and with ResNet-50 as backbone. Both models failed to do positive predictions with a score threshold above 0.6. With the score threshold at 0.5, few predictions were accurately predicted. From both tables we can conclude that overfitting occured. The models failed to generalize the patterns in the damaged areas. On the contrary, the models memorized the patters and all the details of the examples in the training set and were not able to predict bounding boxes in the testing set.

Score Threshold	**TP**	**FP**	**GT**	**Precision**	**Recall**	**F_1score**
0.5	2	13	57	0.133	0.035	0.056
0.6	0	3	57	0	0	0
0.7	0	1	57	0	0	0
0.8	0	0	57	0	0	0
0.9	0	0	57	0	0	0

Table 5.6: FRCNN model with VGG as backbone. $IOU = 0.5$ and 'damage' as the only class.

Score Threshold	**TP**	**FP**	**GT**	**Precision**	**Recall**	**F_1score**
0.5	3	19	57	0.136	0.053	0.076
0.6	0	2	57	0	0	0
0.7	0	0	57	0	0	0
0.8	0	0	57	0	0	0
0.9	0	0	57	0	0	0

Table 5.7: FRCNN model with ResNet-50 as backbone. $IOU = 0.5$ and 'damage' as the only class.

Tables 5.8 and 5.9 show the results with the same settings than before but now considering the three classes annotated by the expert rather than only one class. We can see the performance improved. This improvement might be explained by the fact that each kind of class has different patterns depending on the kind of damage. This way the network was able to generalize for each class rather than generalize for all three classes at the same time. As it would be expected, for lower confidence values the number of false negatives is greater than for higher confidence values. The maximum $F_1 score$ was obtained using VGG as feature extractor and score threshold at 0.8. However this model failed to do a single prediction of any 'Appendage' objects unlike ResNet-50.

Score Threshold	**Class**	**TP**	**FP**	**GT**	**Precision**	**Recall**	**F_1score**
0.5	SD	13	106	44	0.109	0.295	0.160
	Erosion	6	447	10	0.013	0.6	0.026
	Appendage	0	0	3	0	0	0
	Total	19	553	57	0.033	0.333	0.060
0.6	SD	11	34	44	0.244	0.25	0.247
	Erosion	5	170	10	0.029	0.5	0.054
	Appendage	0	0	3	0	0.	0
	Total	16	204	57	0.073	0.281	0.1152
0.7	SD	9	20	44	0.310	0.205	0.247
	Erosion	5	63	10	0.074	0.5	0.128
	Appendage	0	0	3	0	0	0
	Total	14	83	57	0.144	0.246	0.182
0.8	SD	8	13	44	0.381	0.182	0.246
	Erosion	4	27	10	0.129	0.4	0.195
	Appendage	0	0	3	0	0	0
	Total	12	40	57	0.231	0.211	0.220
0.9	SD	6	4	44	0.6	0.136	0.222
	Erosion	2	8	10	0.2	0.2	0.3
	Appendage	0	0	3	0	0	0
	Total	8	12	57	0.4	0.140	0.208

Table 5.8: FRCNN model with VGG as backbone. $IOU = 0.5$ and three classes labelled.

Score Threshold	Class	TP	FP	GT	Precision	Recall	F_1score
0.5	SD	20	120	44	0.143	0.455	0.217
	Erosion	6	358	10	0.016	0.6	0.032
	Appendage	1	121	3	0.01	0.333	0.016
	Total	27	599	57	0.043	0.474	0.079
0.6	SD	17	66	44	0.205	0.386	0.268
	Erosion	6	99	10	0.057	0.6	0.104
	Appendage	1	105	3	0.01	0.333	0.018
	Total	24	207	57	0.104	0.421	0.167
0.7	SD	17	41	44	0.293	0.386	0.333
	Erosion	4	35	10	0.103	0.4	0.163
	Appendage	1	90	3	0.011	0.333	0.021
	Total	22	166	57	0.117	0.386	0.180
0.8	SD	11	16	44	0.407	0.25	0.310
	Erosion	4	26	10	0.133	0.4	0.2
	Appendage	1	72	3	0.014	0.333	0.026
	Total	16	114	57	0.123	0.281	0.171
0.9	SD	8	12	44	0.4	0.182	0.25
	Erosion	3	7	10	0.3	0.3	0.3
	Appendage	1	54	3	0.018	0.333	0.034
	Total	12	73	57	0.141	0.211	0.169

Table 5.9: FRCNN model with ResNet-50 as backbone. $IOU = 0.5$ and three classes labelled.

In general, FRCNN performed poorly and these models revealed unfit for a real-world application. It is also clear that both models struggle to detect areas of 'Appendage' damage. Using VGG backbone no predictions are made. On the other hand, when using ResNet-50 backbone, there is high occurrence of false positive predictions. This is a result of possible underfitting. The network might have failed to generalize the patterns present in this class thus the high number of false positives.

5.2.2 RetinaNet Model

With RetinaNet we proceeded the same way. We tested RetinaNet with VGG ans ResNet-50 as backbones and with 'damage' class. Table 5.11 shows the results of RetinaNet, with VGG as backbone and $IOU = 0.5$, for different score thresholds or confidence. The confidence that results in the best compromise between precision and recall is 0.7 with an $F_1score = 0.728$. When using ResNet-50 as backbone, the best F_1score is lower, however, the correspondent confidence is 0.9. We can say this last one is a better model since the difference between both $F_1scores$ is insignificant but the score threshold is much higher thus offering a more confident and accurate prediction.

Score Threshold	**TP**	**FP**	**GT**	**Precision**	**Recall**	**F_1score**
0.5	35	7	57	0.833	0.614	0.709
0.6	35	6	57	0.854	0.614	0.714
0.7	35	4	57	0.897	0.614	**0.728**
0.8	34	3	57	0.919	0.596	0.723
0.9	30	2	57	0.938	0.526	0.674

Table 5.10: RetinaNet model with VGG as backbone. $IOU = 0.5$ and 'damage' as the only class.

Score Threshold	**TP**	**FP**	**GT**	**Precision**	**Recall**	**F_1score**
0.5	37	12	57	0.755	0.649	0.698
0.6	36	11	57	0.766	0.631	0.692
0.7	34	8	57	0.810	0.596	0.687
0.8	34	7	57	0.829	0.596	0.693
0.9	33	4	57	0.892	0.579	**0.702**

Table 5.11: RetinaNet model with ResNet-50 as backbone. $IOU = 0.5$ and 'damage' as the only class.

Table 5.13 shows the results for RetinaNet with ResNet-50 as backbone and considering each class separately. It is very clear that, when comparing to the corresponding table with the only class being 'damage', the $F_1 scores$ decrease by much. The best $F_1 score$, with score threshold at 0.8, is around half of the best in Table 5.11. The network reveals difficulty when detecting damages that belong to 'Appendage' class.

Score Threshold	Class	TP	FP	GT	Precision	Recall	F_1score
0.5	SD	14	14	44	0.5	0.318	0.389
	Erosion	3	9	10	0.25	0.3	0.273
	Appendage	0	1	3	0	0	0
	Total	17	24	57	0.415	0.298	0.347
0.6	SD	14	13	44	0.519	0.318	0.394
	Erosion	3	8	10	0.273	0.3	0.286
	Appendage	0	1	3	0	0	0
	Total	17	25	57	0.405	0.298	0.343
0.7	SD	14	10	44	0.583	0.318	0.412
	Erosion	3	7	10	0.3	0.3	0.3
	Appendage	0	1	3	0	0	0
	Total	17	18	57	0.486	0.298	0.370
0.8	SD	11	8	44	0.579	0.25	0.349
	Erosion	0	6	10	0	0	0
	Appendage	0	0	3	0	0	0
	Total	11	14	57	0.44	0.193	0.268
0.9	SD	11	7	44	0.611	0.25	0.355
	Erosion	0	6	10	0	0	0
	Appendage	0	0	3	0	0	0
	Total	11	13	57	0.458	0.193	0.272

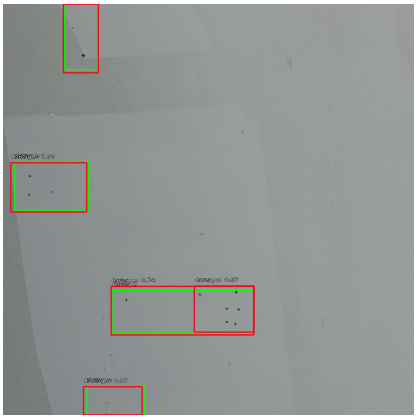
Table 5.12: RetinaNet model with VGG as backbone. $IOU = 0.5$ and three classes labelled.

Score Threshold	Class	TP	FP	GT	Precision	Recall	F_1score
0.5	SD	14	23	44	0.378	0.318	0.345
	Erosion	4	7	10	0.364	0.4	0.381
	Appendage	0	1	3	0	0	0
	Total	18	31	57	0.367	0.316	0.340
0.6	SD	14	18	44	0.438	0.318	0.368
	Erosion	4	6	10	0.4	0.4	0.4
	Appendage	0	1	3	0	0	0
	Total	18	25	57	0.419	0.316	0.36
0.7	SD	14	16	44	0.467	0.318	0.378
	Erosion	4	6	10	0.4	0.4	0.4
	Appendage	0	1	3	0	0	0
	Total	18	23	57	0.440	0.316	0.368
0.8	SD	13	12	44	0.52	0.295	0.376
	Erosion	4	6	10	0.4	0.4	0.4
	Appendage	0	1	3	0	0	0
	Total	17	18	57	0.486	0.298	0.369
0.9	SD	7	4	44	0.636	0.159	0.255
	Erosion	0	3	10	0	0	0
	Appendage	0	0	3	0	0	0
	Total	7	7	57	0.5	0.123	0.197

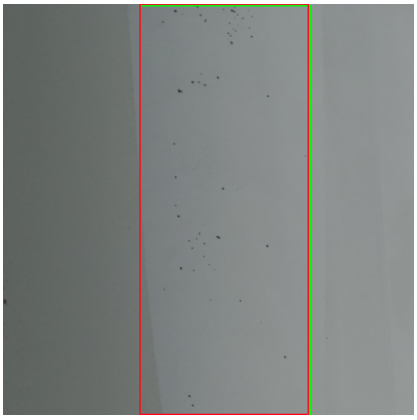
Table 5.13: RetinaNet model with ResNet-50 as backbone. $IOU = 0.5$ and three classes labelled.

From the four approaches tested with RetinaNet, the one using a single class and VGG as backbone achieved the best overall results. The best $F_1 score$ achieved was with score threshold at 0.7 with a value of 0.728. It was, by far, the best model of all the object detection models although it does not distinguish which damage it is detecting. This model can be used in a cascade ensemble were the output is fed to another network which tells apart each type of damage.

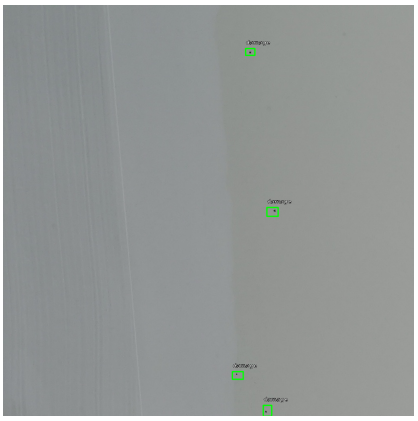
Figure 5.7 shows the output of the model with the highest $F_1 score$. We can see that the model can predict damaged areas with hight accuracy. However, when the ground truth bounding boxes are very small, the network cannot do a correct prediction. It might be possible that, since the ground truth bounding boxes are so small, the IOU computed between them and the predicted bounding boxes is smaller than the IOU we used and, therefore, this predictions were not considered. To overcome this two possible solutions are change the ground truth bounding boxes or, to make the network more sensitive to the different sizes of damages, change the settings of the anchor boxes in the FPN. This way, the network would better fit the predicted bounding boxes to the ground truth bounding boxes.



(a) True Positives.



(b) True Positive.



(c) False Negatives.

(d) False Positive.

Figure 5.7: Output of RetinaNet with VGG as backbone, score thresold at 0.7, IOU at 0.5 and 'damage' as the only class annotated. Green boxes represent ground truth red boxes represent the predictions.

Chapter 6

Conclusions

This thesis addresses pixel-wise segmentation of wind turbine blades, using U-Net [19] and DeepLabv3 [5], and detection of damaged areas in wind turbine blades, using Faster RCNN [17] and RetinaNet [13]. The different networks are explored in order to achieve the best possible damage detection model assited by the blade segmentation network.

The dataset available for this work was made of 80 images in total, very similar to each other. Therefore, the light and weather conditions that the images are subject to have a very low variance. Different light and weather conditions affect the color of the objects and other aspects of the images that, without variability on the dataset, might make the models to have a low performance in real world scenarios. The ground truth, required for training a neural network, for the segmentation networks was developed using GIMP. We annotated each image individually with the region of interest being only the closest blade. All the other objects, including wind turbines in the background, were considered as background. Regarding the ground truth for the object detection networks, it was made by a wind turbine inspection expert. In order to reduce the computational load, all the images were cropped in smaller patches since the original images were extremely large.

U-Net was tested with ReLU and ELU as activation functions. The best models obtained for each were, respectively, with input size 512×512 and 64×64 achieving Dice coefficients of 90.7% and 82.5%, considering a threshold at 0.2. We think these models would benefit from a ground truth where all that is windmill would be labelled. Most of the noise present on the results are background windmills that are not annotated. The effects of cropping the original images into smaller tiles contributed negatively to the performance of the segmentation models, so an approach that would use all the original images instead of cropped patches could also improve the segmentation. Although these problems also show up in DeepLabv3, with this architecture we achieved better results. We tested three different upsampling methods which were bilinear interpolation, transposed convolution and DUC. The best models achieved Dice coeffiecents of, respectively, 95%, 96.4% and 96.4% for input sizes of 256×256 for all. In spite of the high Dice coefficients obtained, the segmentation was not fit for this purpose since it could leave damages out of the segmentation affecting, in this way, the performance and accuracy of the object detection models. Therefore, we did not used segmentation in the final model. Yet, these models can be used, for example, in a ensemble method were this three best models are compared with each other resulting in a better segmentation output.

Regarding object detection, RetinaNet achieved a greather $F_1 scores$ when compared to Faster RCNN. Apart from this, it is also faster to train and to predict since it is a one-stage

detector unlike Faster RCNN. In future works, RetinaNet is definitely a better approach than Faster RCNN. We achieved better results when using one single class that grouped all the three annotated classes than when using three different classes for each type of damage. Therefore, a possible future approach would be to use an improved version of this model to get the damaged regions and another network, placed in cascade, that would be responsible for classifying each region according to the type of damage.

There is still a lot of room for improvement. Adding new images with different conditions to the dataset, increasing the number of examples and variance, might help to achieve better object detection performances. Another important aspect is class balance. Unbalanced classes result in inaccurate models. By balancing classes a better performance might be achieved.

Artificial intelligence, and deep learning in particular, is a field in constant growth and new technologies come up regularly. In future works it would also be interesting to experiment different architectures.

References

[1] A. Arnab, S. Zheng, S. Jayasumana, B. Romera-Paredes, M. Larsson, A. Kirillov, B. Savchynskyy, C. Rother, F. Kahl, and P. H. S. Torr. Conditional random fields meet deep neural networks for semantic segmentation: Combining probabilistic graphical models with deep learning for structured prediction. *IEEE Signal Processing Magazine*, 35(1):37–52, 2018.

[2] Maxime Bucher, Stéphane Herbin, and Frédéric Jurie. Hard negative mining for metric learning based zero-shot classification. In Gang Hua and Hervé Jégou, editors, *Computer Vision – ECCV 2016 Workshops*, pages 524–531, Cham, 2016. Springer International Publishing.

[3] André Cabrita-Mendes. Central eólica offshore ligada à rede em 2019. *Jornal de Negócios*, April 2018. URL: `https://www.jornaldenegocios.pt/empresas/detalhe/central-eolica-offshore-ligada-a-rede-em-2019`.

[4] André Cabrita-Mendes. Potência eólica em portugal deve crescer 8,3% até 2027. *Jornal de Negócios*, April 2018. URL: `https://www.jornaldenegocios.pt/empresas/energia/detalhe/potencia-eolica-em-portugal-deve-crescer-83-ate-2027`.

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

[6] Arden Dertat. Applied deep learning - part 4: Convolutional neural networks. *Towards Data Science*, November 2017. URL: `https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2`.

[7] Oxford Dictionaries. Overfitting. URL: `https://en.oxforddictionaries.com/definition/overfitting`.

[8] fizyr. keras-retinanet. *GitHub*, April 2018. URL: `https://github.com/fizyr/keras-retinanet`.

[9] R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[11] jinfagang. keras_frcnn. *GitHub*, November 2017. URL: `https://github.com/jinfagang/keras_frcnn/`.

[12] T. Y. Lin, P. Dollr, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, 2017.

[13] T. Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollr. Focal loss for dense object detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007, 2017.

[14] LUSA. Produção de renováveis excedeu consumo em portugal pela primeira vez. *Público*, April 2018. URL: `https://www.publico.pt/2018/04/03/economia/noticia/producao-de-renovaveis-excedeu-consumo-em-portugal-pela-primeira-vez-1808894`.

[15] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[16] qqgeogor. keras-segmentation-networks. *GitHub*, October 2017. URL: `https://github.com/qqgeogor/keras-segmentation-networks/blob/master/deeplabv3.py`.

[17] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, June 2017. `doi:10.1109/TPAMI.2016.2577031`.

[18] Jason Roell. From fiction to reality: A beginners guide to artificial neural networks. *Towards Data Science*, June 2017. URL: `https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-0411777571b`.

[19] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[21] A. Shrivastava, A. Gupta, and R. Girshick. Training region-based object detectors with online hard example mining. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 761–769, June 2016. `doi:10.1109/CVPR.2016.89`.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[24] tzutalin. Labelimg. *GitHub*, March 2018. URL: `https://github.com/tzutalin/labelImg`.

[25] Petar Velicković. Deep learning for complete beginners: convolutional neural networks with keras. *Cambridge Spark*, March 2017. URL: https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html.

[26] P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou, and G. Cottrell. Understanding convolution for semantic segmentation. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1451–1460, 2018.

[27] Wikipedia. Artificial neural network. URL: https://en.wikipedia.org/wiki/Artificial_neural_network.

Appendix A

U-Net Python Code

```
input_layer = Input((1024,1024,3))
c1 = Conv2D(filters=16, kernel_size=(3, 3), activation='elu', padding='same')(
    input_layer)
c1 = Conv2D(16, (3,3), activation='elu', padding='same')(c1)


l = MaxPool2D(strides=(2, 2))(c1)
c2 = Conv2D(filters=32, kernel_size=(3, 3), activation='elu', padding='same')(l
    )
c2 = Conv2D(filters=32, kernel_size=(3, 3), activation='elu', padding='same')(
    c2)

l = MaxPool2D(strides=(2, 2))(c2)
c3 = Conv2D(filters=64, kernel_size=(3, 3), activation='elu', padding='same')(l
    )
c3 = Conv2D(filters=64, kernel_size=(3, 3), activation='elu', padding='same')(
    c3)

l = MaxPool2D(strides=(2, 2))(c3)
c4 = Conv2D(filters=128, kernel_size=(3, 3), activation='elu', padding='same')(
    l)
c4 = Conv2D(filters=128, kernel_size=(3, 3), activation='elu', padding='same')(
    c4)

l = MaxPool2D(strides=(2, 2))(c4)
c5= Conv2D(filters=256, kernel_size=(3, 3), activation='elu', padding='same')(l
    )
c5= Conv2D(filters=256, kernel_size=(3, 3), activation='elu', padding='same')(
    c5)

l = MaxPool2D(strides=(2, 2))(c5)
c6= Conv2D(filters=512, kernel_size=(3, 3), activation='elu', padding='same')(l
    )
c6= Conv2D(filters=512, kernel_size=(3, 3), activation='elu', padding='same')(
    c6)

l = MaxPool2D(strides=(2,2))(c6)
c7= Conv2D(filters=1024, kernel_size=(3,3), activation='elu', padding='same')(l
    )
c7= Conv2D(filters=1024, kernel_size=(3,3), activation='elu', padding='same')(
    c7)
```

```
l = concatenate([UpSampling2D(size=(2,2))(c7),c6],axis=-1)
l = Conv2D(filters=512, kernel_size=(3,3), activation='elu', padding='same')(l)
l = Conv2D(filters=512, kernel_size=(3,3), activation='elu', padding='same')(l)

l = concatenate([UpSampling2D(size=(2, 2))(l), c5], axis=-1)
l = Conv2D(filters=256, kernel_size=(3, 3), activation='elu', padding='same')(l
    )
l = Conv2D(filters=256, kernel_size=(3, 3), activation='elu', padding='same')(l
    )

l = concatenate([UpSampling2D(size=(2, 2))(l), c4], axis=-1)
l = Conv2D(filters=128, kernel_size=(3, 3), activation='elu', padding='same')(l
    )
l = Conv2D(filters=128, kernel_size=(3, 3), activation='elu', padding='same')(l
    )

l = concatenate([UpSampling2D(size=(2, 2))(l), c3], axis=-1)
l = Conv2D(filters=64, kernel_size=(3, 3), activation='elu', padding='same')(l)
l = Conv2D(filters=64, kernel_size=(3, 3), activation='elu', padding='same')(l)

l = concatenate([UpSampling2D(size=(2, 2))(l), c2], axis=-1)
l = Conv2D(filters=32, kernel_size=(3, 3), activation='elu', padding='same')(l)
l = Conv2D(filters=32, kernel_size=(3, 3), activation='elu', padding='same')(l)

l = concatenate([UpSampling2D(size=(2, 2))(l), c1], axis=-1)
l = Conv2D(filters=16, kernel_size=(3, 3), activation='elu', padding='same')(l)
l = Conv2D(filters=16, kernel_size=(3, 3), activation='elu', padding='same')(l)

output\_layer = Conv2D(filters=1, kernel_size=(1, 1), activation='sigmoid')(l)

model = Model(input_layer, output_layer)
```

Code A.1: Python code of U-Net implementation using ELU as activation function.

Appendix B

Code to Generate Ground Truth

```
import os
import cv2

# Get bounding boxes of image
def get_bb(xml_name):
# Check if xml file with labels exists
try:
#If it exists, read
xml = xml_name[:-3] + "xml"
file = open(xml, "r")
contents = file.read()
file.close()
except:
return 0

# Search for tags
bbs = contents.count("<xmin>")
contents = contents.split("<name>")
bb = contents[1]
class_name = bb.split("</name>")[0]
bb = bb.split("<xmin>")[1]
# Get coordinates of first bounding box
x_min = bb.split("</xmin>")[0]
bb = bb.split("<ymin>")[1]
y_min = bb.split("</ymin>")[0]
bb = bb.split("<xmax>")[1]
x_max = bb.split("</xmax>")[0]
bb = bb.split("<ymax>")[1]
y_max = bb.split("</ymax>")[0]



bbs_list = [[None for _ in range(4)] for _ in range(20)]
bbs_list[0] = [int(x_min), int(y_min), int(x_max), int(y_max), class_name]

# If there's more bounding boxes, get its values
if bbs > 1:
for n in range(1,bbs):
bb = contents[n+1]
class_name = bb.split("</name>")[0]
bb = bb.split("<xmin>")[1]
```

```
x_min = bb.split("</xmin>")[0]
bb = bb.split("<ymin>")[1]
y_min = bb.split("</ymin>")[0]
bb = bb.split("<xmax>")[1]
x_max = bb.split("</xmax>")[0]
bb = bb.split("<ymax>")[1]
y_max = bb.split("</ymax>")[0]
bbs_list[n] = [int(x_min), int(y_min), int(x_max), int(y_max), class_name]

return  bbs_list

#Check if cropped image is inside bounding box
def check_bbs(x_min, x_max, y_min, y_max, bbs):
label = 0
for bb in range(0,len(bbs)):
if bbs[bb][0] is not None:
#Bounding box values
bbx_min = bbs[bb][0]
bby_min = bbs[bb][1]
bbx_max = bbs[bb][2]
bby_max = bbs[bb][3]

#Check if boundig box is in image
if bbx_min >= x_max or bbx_max <= x_min or bby_min >= y_max or bby_max <= y_min
    :
continue
label = 1

return label

######### main ############

images = []
labels = []

#Go through all files in folder and save .jpg files into list
for image in os.listdir("./Annotations/test/"):
if image.endswith(".JPG"):
images.append((os.path.join("./Annotations/test/", image)))

#Count is the number of image. e.g. 1, 2, 3 and so on
count = 0

file = open("./data/mixed_test/annotations.csv", "w")
for image in images:
#load both original and ground truth image (for segmentation)
img = cv2.imread(image)
img_seg = image.split("test")
img_seg = img_seg[0] + "test_GT" + img_seg[1].split(".")[0] + "_mask.JPG"
img_seg = cv2.imread(img_seg)
count = count + 1
print("Cropping image %d of %d : %s" %(count, len(images),image))
#Get bounding boxes of image if there is any
bbs = get_bb(image)
for row in range(0, 5): #9 if overlaping
for col in range(0, 7: #14 if overlaping
```

```
x_min = col * 1024   #use half tile size for overlaping
x_max = x_min + 1024   #size of the tile
y_min = row * 1024    #use half tile size for overlaping
y_max = y_min + 1024   #size of the tile

cropped = img[y_min:y_max, x_min:x_max]  #crop the tile out of image
seg_cropped = img_seg[y_min:y_max, x_min:x_max] #crop segmentation ground truth

if bbs is 0:
label = 0
else:
#Check if cropped image is inside bounding box
label = check_bbs(x_min, x_max, y_min, y_max, bbs)

if label is 0:
#if there is no bounding box save tiles
img_name = "./data/mixed_test/img/" + str(count) + "_" + str(row) + str(col) +
    ".jpg"
cv2.imwrite(img_name, cropped)

img_name = "./data/mixed_test/mask/" + str(count) + "_" + str(row) + str(col) +
     "_mask.jpg"
cv2.imwrite(img_name, seg_cropped)

#save annotations into csv file
annotation = "./data/mixed_test/img/" + str(count) + "_" + str(row) + str(col)
    + ".jpg" + ",,,,,\n"
file.write(annotation)
else:
#if there is at least one bounding box
for i in range(0,len(bbs)):
if bbs[i][0] is not None:
x1 = 0 if (bbs[i][0]) < (x_min) else (bbs[i][0]) - (x_min)
y1 = 0 if (bbs[i][1]) < (y_min) else (bbs[i][1]) - (y_min)
x2 = 1024 if (bbs[i][2]) > (x_max) else 1024 - ((x_max) - (bbs[i][2]))
y2 = 1024 if (bbs[i][3]) > (y_max) else 1024 - ((y_max) - (bbs[i][3]))

#If bounding box is not in crop, check the next bounding box
if x1>=x2 or y1>=y2:
continue

#if cropping without overlpaing. else, comment this block
cv2.rectangle(cropped, (x1, y1), (x2, y2), (0, 255, 0), 3) #debug: draws
    bounding box in image to check if it right
img_name = "./data/mixed_test/img/" + str(count) + "_" + str(row) + str(col) +
    ".jpg"
cv2.imwrite(img_name, cropped)
img_name = "./data/mixed_test/mask/" + str(count) + "_" + str(row) + str(col) +
     "_mask.jpg"
cv2.imwrite(img_name, seg_cropped)
annotation = "./data/mixed_test/img/" + str(count) + "_" + str(row) + str(col)
    + ".jpg" + "," + str(x1) + "," + str(y1) + "," + str(x2) + "," + str(y2) + "
    ," + bbs[i][4] + "\n"
file.write(annotation)

#if cropping with overlaping uncomment below
#for angle in range(0,4):
```

```
#     num_rows, num_cols = cropped.shape[:2]
#     rotation_matrix = cv2.getRotationMatrix2D((num_cols / 2, num_rows / 2),
    angle*90, 1)
#     rotated = cv2.warpAffine(cropped, rotation_matrix, (num_cols, num_rows))
#     rotated_seg = cv2.warpAffine(seg_cropped, rotation_matrix, (num_cols,
    num_rows))
#     #cv2.imshow('Rotation', rotated)
#     #cv2.waitKey()
#
#     #adjust coordinates depending on the rotation
#     if angle == 0:
#         xr1 = x1
#         yr1 = y1
#         xr2 = x2
#         yr2 = y2
#     if angle == 1:
#         xr1 = y1
#         yr1 = 1024 - x2
#         xr2 = y2
#         yr2 = 1024 - x1
#     if angle == 2:
#         xr1 = 1024 - x2
#         yr1 = 1024 - y2
#         xr2 = 1024 - x1
#         yr2 = 1024 - y1
#     if angle == 3:
#         xr1 = 1024 - y2
#         yr1 = x1
#         xr2 = 1024 - y1
#         yr2 = x2
#
#     img_name = "./data/mixed/img/" + str(count) + "_" + str(row) + str(col) +
    "_" + str(angle) + ".jpg"
#     cv2.imwrite(img_name, rotated)
#     img_name = "./data/mixed/mask/" + str(count) + "_" + str(row) + str(col) +
     "_" + str(angle) + "_mask.jpg"
#     cv2.imwrite(img_name, rotated_seg)
#
#     annotation = "./data/mixed/img/" + str(count) + "_" + str(row) + str(col)
    + "_" +str(angle) + ".jpg" + "," + str(xr1) + "," + str(yr1) + "," + str(xr2
    ) + "," + str(yr2) + "," + bbs[i][4] + "\n"
#     file.write(annotation)
file.close()
```

Code B.1: Python code to generate segmentation and object detection ground truth