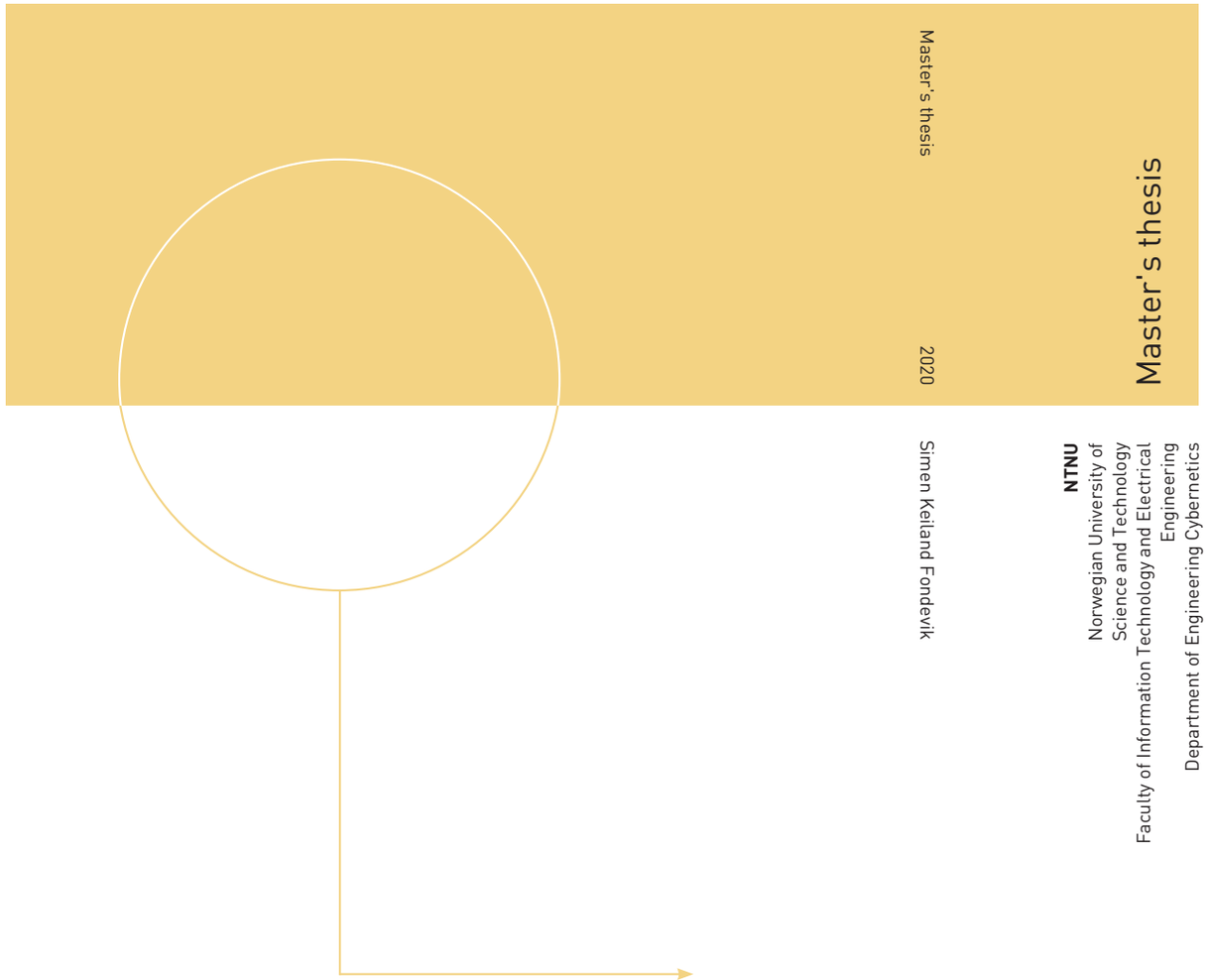


Master's thesis

2020

Simen Keiland Fondevik

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

Simen Keiland Fondevik

Image Segmentation of Corrosion Damages in Industrial Inspections using State-of-the-Art Neural Networks

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Image Segmentation of Corrosion Damages in Industrial Inspections using State-of-the-Art Neural Networks

Simen Keiland Fondevik

Cybernetics and Robotics
Submission date: June 2020
Supervisor: Annette Stahl
Co-supervisor: Aksel Andreas Transeth, Ole Øystein Knudsen

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

This master thesis reviews state-of-the-art image segmentation algorithms for the purposes of automatic corrosion damage segmentation and classification. Automatic image analysis is needed in order to process all data retrieved from drone-driven industrial inspections. Image classification can alert an inspector that a damage is present, whereas image segmentation can further specify its shape and location. This could be the first step towards estimation of total damaged surface area of a construction, an often-used metric to initiate maintenance. To this end, 608 images with corrosion damages are instance-wise annotated with binary segmentation masks. Additionally, a general, two-stage data augmentation scheme empirically shown to significantly reduce overfitting is developed. With this scheme, the neural networks PSPNet and Mask R-CNN obtains, respectively, $86.6\,\%$ and $84.1\,\%$ frequency weighted IoU on a 50-image test set. It is concluded that image segmentation can aid automating industrial inspections of steel constructions in the future, and that instance segmentation algorithms are likely more useful than semantic segmentation algorithms, due to its applications to a wider range of use-cases. However, current performance with the rather small dataset used is not good enough to construct a reliable autonomous system yet.

For further work, the dataset should be extended with more classes, e.g. crack in concrete, paint flaking, intact steel construction and corrosion on rebar. This could also include classes for different degrees of severity of damages. Furthermore, ways to estimate total percentage of damaged construction coating, and applying time series prediction methods to forecast damage development, can be researched.

Keywords: image segmentation, machine learning, industrial inspections, corrosion.

Sammendrag

Denne masteroppgaven studerer avanserte bildesegmenteringsalgoritmer for automatisk korrosjonsskade-segmentering. Automatisk bildeanalyse er nødvendig for å prosessere all data som samles inn med dronebaserte industrielle inspeksjoner. Bilde-klassifisering kan informere en inspektør om at en skade er til stede, mens bildesegmentering kan videre spesifisere dens form og lokasjon. Dette er et første steg mot å kunne estimere totalt areal skadet overflate på en konstruksjon, et ofte brukt mål for å sette i gang vedlikehold. For dette formålet er 608 bilder annotert med binære instans-segmenteringsmasker. Dessuten er det utviklet en to-stegs metodikk for å kunstig øke antall treningsbilder. Metodikken er empirisk vist å betydelig redusere såkalt overfitting. Med denne metodikken oppnår de nevrale nettverkene PSPNet og Mask R-CNN henholdvis $86.6\,\%$ og $84.1\,\%$ frekvensvekted IoU på et 50-bilder testsett. Det konkluderes at bildesegmentering kan bistå i å automatisere industrielle inspeksjoner av stålkonstruksjoner i fremtiden, og at instans-segmenteringsalgoritmer trolig er mer nyttige enn såkalte semantic-segmenteringsalgoritmer, grunnet mulighet for flere anvendelser. Nåværende oppnådd ytelse er imidlertid ikke god nok til å konstruere et autonomt system enda.

I videre arbeid burde datasettet utvides med flere klasser, for eksempel sprekk i betong, malingsavflaking, intakt stålkonstruksjon og korrosjon på armeringsjern. Nye klasser kan også skille mellom ulike grader av alvorlighet på skader. Dessuten kan det studeres nærmere hvordan arealestimering av skadet konstruksjonsoverflate kan gjøres, og om en kan bruke tidsserieprediksjonsmetoder til å forutsi hvordan en skade vil utvikle seg i fremtiden.

Nøkkelord: bildesegmentering, maskinlæring, industrielle inspeksjoner, korrosjon.

Preface

This master thesis is the final assignment of my five-year master program in engineering cybernetics, and the result of a collaboration between NTNU and SINTEF. The thesis problem description was proposed January 21st, 2020, and the final report was delivered June 1st, 2020.

The thesis is written entirely and independently by me. My supervisors gave me very few restrictions, allowing me to propose a problem description myself and pursue paths I have found interesting. All contributions, text and figures are original work by me.

In collaboration with my supervisors I have also written a scientific paper based on the thesis submitted to the *32nd International Conference on Tools with Artificial Intelligence*. The paper is attached in the appendix and also delivered as a separate file along with this thesis.

The first weeks of the project were spent constructing a dataset with annotated corrosion damages and researching image segmentation algorithms. It was decided to implement and evaluate PSPNet and Mask R-CNN for the purposes of corrosion damage segmentation. The main implementation is based on publicly available code but modified to fit the needs of this project. Following weeks were devoted to studying various data augmentation methods and how to properly evaluate image segmentation algorithms. Some augmentation schemes are specifically designed for the corrosion dataset, whereas others are based on examples in the documentation of the framework used. The final month and a half were spent training network models, analyzing results, writing a scientific paper and finishing this report.

Raw images to construct the corrosion dataset was originally provided by Norwegian Public Roads Administration and later shared with me by SINTEF. The Department of Engineering Cybernetics provided all necessary hardware and an office to work in. Due to the circumstances caused by covid-19, however, most of the work had to be done from home with a remote connection to hardware at NTNU. This was, of course, not optimal, but did not cause any delay with regards to final delivery.

A few sections in the thesis are loosely based on sections from my project thesis last semester. Citations are provided in the specific sections, and the project thesis is delivered along with this thesis.

I would like to thank my three supervisors Aksel Andreas Transeth (SINTEF), Annette Stahl (NTNU) and Ole Øystein Knudsen (SINTEF). Transeth has guided me superbly on how to structure, write and convey a good master thesis. Stahl has provided good suggestions on research topics and valuable discussions of machine learning methods. With his

knowledge and industry insight, Knudsen has helped me make the master thesis relevant beyond academia. I also want to thank my supervisors for giving me freedom and faith, allowing me to work independently and research topics I have found interesting. Finally, I would like to thank my beautiful girlfriend Elsie Margrethe Staff Mestl for all the support, love and care throughout this work and for diligent proofreading. The thesis would not be as good as it is without your help.

Simen Keiland Fondevik
NTNU, Trondheim
June 1st, 2020

Table of Contents

Chapter 1

Introduction

Parts of Section 1.1 and Section 1.3 are based on the authors project thesis from last semester [14].

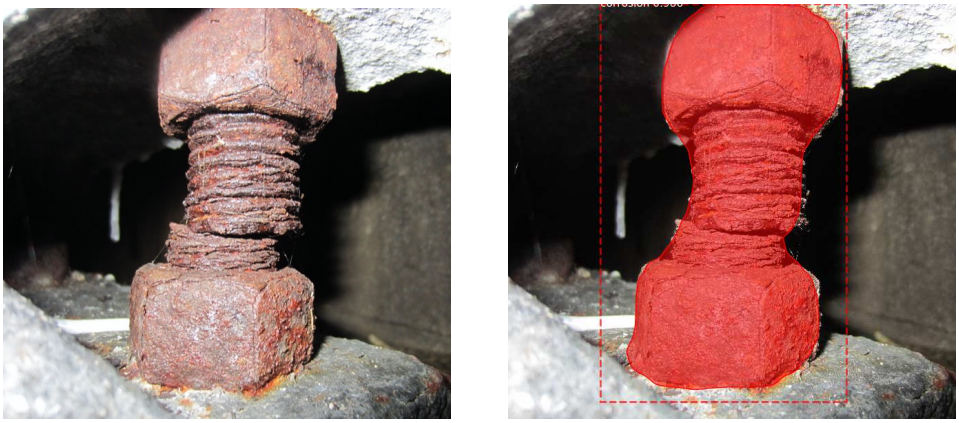
1.1 Motivation and Background

The National Association of Engineers (NACE) International estimated in 2016 corrosion damages to have an annual cost of 2.5 trillion USD [30], equivalent to $3.4\,\%$ of global GDP. Corrosion is a major problem that wear down the steel constructions and can severely reduce their strength, as shown in Figure 1.1a. Constructions in humid and salty environments are particularly exposed, and regular inspections and quality controls are therefore necessary. NACE International estimates savings between 375 – 875 billion USD annually by using already available corrosion control practices.

Inspection of corrosion damages is usually performed through visual inspections onsite. However, there are many areas not easily accessible to inspectors, for instance due to hazardous conditions or simply because it is out of reach. Additionally, manual inspections can be time consuming, expensive and subjective. An interesting approach is therefore to use unmanned aerial vehicles (UAVs) with a mounted camera to take photos of potentially damaged areas. Multiple companies are already specializing in drone-driven industrial inspections, e.g. AirSens, Scout and Orbiton [2, 27, 43]. The UAVs could take photos of constructions without the need for human assistance. Images can then be sent to a cloud service for storage, logging or further analysis on a remote computer. The concept is illustrated in Figure 1.2.

Inspecting images manually is tedious and time consuming, particularly if the UAVs are not well aware of what is worth reporting. With an increasing number of images to inspect, the error rate is also likely to increase. Even more so, manual inspection of images is subject to human subjectivity. One person may conclude the image contains a damage in need of maintenance, while a colleague could say no maintenance is necessary.

The vast amount of data automatically collected using UAVs can simply render manual processing of images infeasible. An automatic image analysis framework is therefore needed, both in terms of efficiency and objectivity. This again allows for more frequent



(a) Corrosion Damage. (b) Segmented Image.

Figure 1.1: (a) Corrosion damages are a major problem in humid and salty environments. Inspections must be performed regularly to verify intactness of metal constructions, such as bridges and oil platforms. (b) Using machine learning, corrosion damages can be detected, localized and its shape be outlined automatically. This process is referred to as image segmentation and is the topic of this thesis.

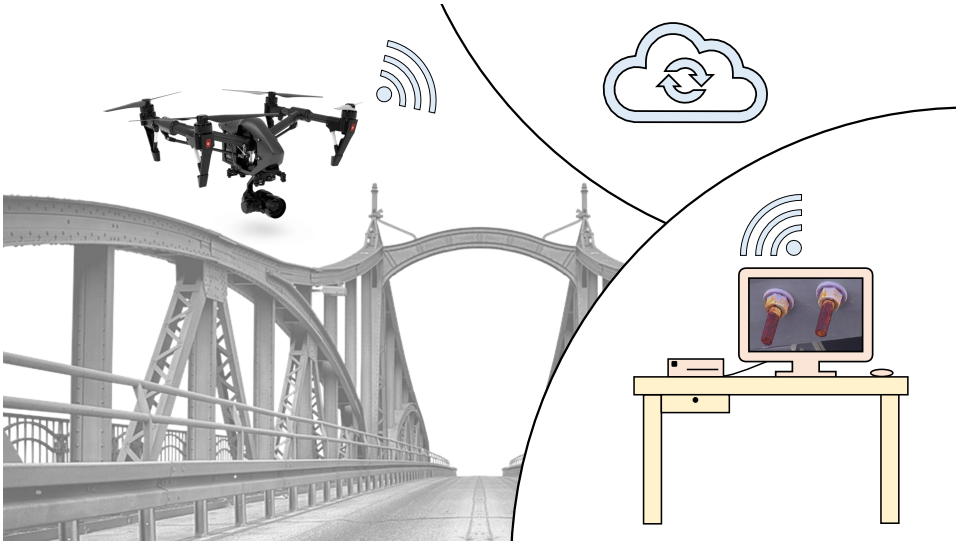


Figure 1.2: UAV inspecting a bridge. Images taken are wirelessly send, to a cloud service and a remote computer for further analysis. This removes the need for inspectors on-site, increasing efficiency and reducing costs and danger.

inspections and thereby detection of damages at an earlier stage.

This thesis studies how automatic image segmentation of corrosion damages can be performed using machine learning. Image segmentation is the process of detecting and outlining objects of interest in images, as shown in Figure 1.1b. There are multiple reasons why this is useful, and why it is superior to simple image classification, i.e. sorting images as corrosion vs. not corrosion. First, successful segmentation of corrosion damages allows an autonomous system (e.g. a UAV) to better understand its surroundings and thereby be able to, for instance, focus the camera in the right direction. Second, estimation of total damaged area of a construction is an often-used criterion to initiate maintenance. Image segmentation is a necessary first step towards this goal.

1.2 Aim and Scope of the Thesis

There are a number of different algorithms for image segmentation, ranging from simple thresholding methods to sophisticated neural networks. Machine learning has become increasingly popular in recent years and is now the de-facto standard for a wide range of visual tasks. For image segmentation there are mainly two types of problems; *semantic* segmentation and *instance* segmentation. The former is concerned with classifying each pixel in images as belonging to a class, not separating instances from each other. Instance segmentation, on the other hand, also distinguishes between instances of the same class. Both methods have shown to be successful in a variety of different fields of study, such as medical and satellite image analysis.

Although promising, existing research on image segmentation to detect corrosion is limited. The aim of this thesis is therefore to assess the applicability of state-of-the-art image segmentation methods to detect corrosion damages, for the purposes of automating industrial inspections. To this end, a dataset for image segmentation of corrosion damages is constructed and used to evaluate two different neural networks. It is a goal for this corrosion dataset to also facilitate further research beyond the scope of this thesis. Furthermore, extensive data augmentation methods are evaluated, with the aim of constructing a general augmentation scheme suitable for any dataset similar to the corrosion dataset.

Due to the nature of the problem and limited time frame of the master thesis, not every aspect of corrosion damage segmentation is covered. First, only a small selection of machine learning algorithms are implemented and tested. The selection is done based on a literature review of publicly available methods. Second, the dataset only has binary segmentation masks, i.e. the machine learning models are trained with the two classes *corrosion* and *background*. An extension of this master thesis can study the addition of more classes, such as crack in concrete and paint flaking. Furthermore, segmentation masks will treat all corrosion damages alike, not differentiating between light and severe corrosion. Third, the dataset contains images of corrosion damages taken in daylight above water only. Additionally, the images are biased towards bridge constructions and resulting dataset may not represent general corrosion damages accurately. Finally, highly optimizing all hyperparameters is not opted for in this thesis as it is time consuming, network dependent and not needed for a general assessment of the image segmentation methods.

1.3 Previous Work

This section presents an overview of previous work on automatic damage detection. Both traditional computer vision and machine learning methods for image segmentation are referenced.

1.3.1 Damage Detection using Traditional Computer Vision

Damage detection has been studied using different methods over the past decades. Gunatilake et al. [19] used a remote controlled directional light source mounted on an aircraft to simulate lighting produced by inspectors' flashlights. Edge detection algorithms on the resulting live imagery was then used to detect cracks and corrosion on the aircraft surface.

Siegel and Gunatilake [51] further studied detection of cracks and corrosion on aircraft surfaces. A pipeline consisting of pre-processing/enhancement of the image and wavelet[1] based feature extraction was found to perform the best for corrosion detection, whereas a fuzzy logic algorithm worked best for cracks.

Livens et al. [38] studied general classification of corrosion images by performing a wavelet decomposition of the images and computing its energy signatures. An LVQ-network[2] was then used for classification.

Lee, Chang, and Skibniewski [34] observed that the above methods work well for damages with long linear shapes and distinct edges, such as for cracks. To better handle small scattered spots of corrosion, the authors performed per-channel statistical analysis on RGB-images utilizing the fact that rust has a characteristic red/brown color.

Generally, traditional computer vision approaches for corrosion damage detection is performed by a color and/or texture analysis. Such algorithms suffer if optimal features are not identified.

1.3.2 Damage Detection using Machine Learning

Using machine learning, the need to manually identify common domain features is removed. Machine learning used for damage detection dates back to (at least) 1999 when Moselhi and Shebab-Eldeen [42] used a three-layer neural network to detect defects in underground sewer pipes.

Petricca et al. [45] compared a traditional computer vision approach based on the number of red pixels in an image with a convolutional deep learning approach for detecting rust. The deep learning network was based on AlexNet [31], and the authors found this to be superior to traditional computer vision.

Atha and Jahanshahi [4] further studied the use of convolutional neural networks for corrosion assessment on metallic surfaces. They used a sliding window approach and tested the effects of different window sizes and color spaces for a few different network architectures. Their findings suggest using 128×128 windows along with VGG16 [52], while both the RGB and YCbCr color space performed well.

[1] A mathematical function recovering weak signals from noise.

[2] Algorithm similar to nearest neighbor.

Direct end-to-end image classification of corrosion damages and paint flaking is also studied by Holm et al. [24, 23]. Holm et al. tested a number of classic, well-known convolutional neural networks and found VGG19 [52] to perform the best.

Fondevik [14] extended the research by Holm et al. and evaluated state-of-the-art classification networks and optimizers. The results show that newer and smarter network architectures outperform often used networks such as VGG [52] and ResNet [21], for the purposes of corrosion damage image classification.

Less research exists on corrosion damage detection, localization and segmentation using artificial intelligence. State-of-the-art methods such as Mask R-CNN [22] have, however, been applied to other types of damages: Moisture marks of shield tunnel linings were segmented by Zhao, Zhang, and Huang [59] using Mask R-CNN. They compared Mask R-CNN with a previously proposed fully convolutional network by the same authors, a region growing algorithm and a thresholding algorithm. First, machine learning-based methods were found to significantly outperform region growing and thresholding. Second, Mask R-CNN outperformed the previously proposed FCN, suggesting the former should be used to detect damages similar to moisture marks.

Zhang, Chang, and Bian [57] used Mask R-CNN to detect and segment damages on vehicles. With a 2000 image dataset, the authors achieved promising results.

Detection of road damages has been studied by numerous people due to a road damage detection and classification challenge held as part of the 2018 IEEE International Conference [53, 56, 3]. However, as submissions were evaluated based on intersection over union scores for bounding boxes, no found produced segmentation masks for the road damages.

Cha et al. [5] also applied object detection to a dataset with construction damages. Using Faster R-CNN, the model was successfully trained to detect corrosion, cracks in concrete and steel delamination using a dataset with 2366 images. The authors conclude that Faster R-CNN can be used with unmanned aerial vehicles to replace human-oriented industrial inspections in the future. Note, however, that Cha et al. performed object detection only, i.e. segmentation masks were not predicted.

To summarize, previous work is usually either focused on the simpler task of object detection, or image segmentation of "simple" damages, i.e. damages clearly standing out from the surroundings (e.g. both tunnel linings and car bodyworks have smooth, monochrome surrounding surfaces). Research on true image segmentation of corrosion damages is lacking.

1.4 Contributions

The main contributions of this master thesis is three-fold:

1.4.1 Dataset Construction

Prior to this work, no dataset with annotated segmentation masks for corrosion damages was found available. A dataset is therefore constructed with binary segmentation masks. 608 images containing corrosion damages are annotated with the software LabelBox. The dataset is constructed with instance-wise segmentation masks allowing for direct use of state-of-the-art instance segmentation algorithms. The dataset can also easily be converted

to semantic segmentation masks with a simple script merging instances. A statistical analysis and summary of the dataset is also provided. The dataset was primarily constructed for usage in this thesis but is highly relevant for further work as well.

1.4.2 Data Augmentation

Annotating images is time-consuming, resulting in only 608 annotated images within the time frame of this master thesis. Fortunately, images of corrosion damages are highly augmentable, i.e. we can generate additional, artificial images by randomly altering existing images. Finding useful augmentation techniques with approximately optimal hyperparameters, however, is challenging due to the huge search space and time requirements of trial and error. It is experimented with a wide range of data augmentation techniques including a proposed general data augmentation scheme requiring very little trial and error. This method is two-staged; first train with heavy data augmentation for a relatively large number of epochs, followed by fine tuning with little to no data augmentation for a few epochs. The hyperparameters for the heavy augmentations can be chosen very liberally as fine tuning is performed with more correct images afterwards. The data augmentation scheme is shown to significantly reduces overfitting, and also improve performance for instance segmentation using Mask R-CNN.

1.4.3 Assessment of Methods for Corrosion Damage Segmentation

State-of-the-art methods for semantic segmentation and instance segmentation are evaluated on the corrosion damage dataset. Models implemented are restricted to PSPNet [58] and Mask R-CNN [22]. Their performance is compared and analyzed, and their applicability with regards to defined use-cases are discussed. Furthermore, the assessment highlights research areas in need of further work.

1.5 Outline

The thesis is structured as follows. Chapter 2 presents preliminary background theory for the purpose of introducing corrosion and machine learning to readers unfamiliar with these topics. It begins with a short introduction to the basics of corrosion damages in Section 2.1. Machine learning and neural networks are then defined, and their inner mechanics explained in Section 2.2.

Chapter 3 follows with a literature study on image segmentation. Its purpose is to discuss use-cases of image segmentation for industrial inspections and investigate available image segmentation methods. The chapter begins with Section 3.1 discussing use-cases of image segmentation, and particularly how segmentation of corrosion damages can aid automating industrial inspections. Next, necessary requirements for an image segmentation framework to be used in industrial inspections are defined in Section 3.2. Finally, Section 3.3 reviews numerous algorithms and machine learning networks with potential for corrosion damage segmentation.

Chapter 4 details methods and implementation. How the dataset was constructed, as well as a statistical analysis of it, is covered in Section 4.1. This is followed by a discussion

of data augmentation methods in Section 4.2. Metrics used to evaluate the performance of different image segmentation algorithms are then discussed in Section 4.3. Implementation, including hardware, software and hyperparameter configurations, are specified in Section 4.5. The chapter ends with an outline of the conducted experiments in Section 4.4

Results and corresponding discussions are presented in Chapter 5. Evaluation of PSP-Net and Mask R-CNN with varying training schemes are given in Section 5.1 and Section 5.2, respectively. A summary and overall discussion is provided in Section 5.3.

Finally, conclusions and further work are found in Section 6.1 and Section 6.2 of Chapter 6.

Chapter 2

Preliminaries

This chapter presents preliminary theory on corrosion damages and machine learning in Section 2.1 and Section 2.2, respectively. Corrosion damages are only briefly reviewed as it is not the main focus of the thesis. Please also note that Section 2.2.1, Section 2.2.3 and partly Section 2.2.5 are loosely based on the authors project thesis written Autumn 2019 [14].

2.1 Controlling Corrosion on Steel Constructions

Corrosion is the process in which metals oxidize, i.e. react with the environment into a more chemically stable form with lower energy. This typically happens in humid and salty environments, such as on oil platforms and bridges. If the metal corroding is steel, the term rust is often used. This is by far the most common type of corrosion and is easily recognized by its red/brown color. Corroded copper and zinc, on the other hand, turns turquoise and white, respectively.

Corrosion is essentially an attack on the metal. It degrades properties such as thickness of cross section, strength, appearance and magnetic permeability. In worst case, metal constructions collapse due to undetected corrosion damages or lack of necessary maintenance, potentially leading to fatalities. Extensive inspections are therefore essential in order to detect damages and initiate maintenance as soon as possible.

Uniform corrosion, i.e. corrosion evenly distributed on the exposed surface of the metal, is often easily detected, repaired and future behavior predicted. An example is shown in Figure 2.1a. Corrosion pits, on the other hand, is much more challenging. As shown in Figure 2.1b, corrosion pits are local holes penetrating deep into the metal creating seemingly small damages, while causing potentially severe damages to the integrity of the construction.

To avoid development of corrosion, metal constructions are often protected with layers of paint. This protects the metal from the aggressive environment as long as the paint is intact. The paint will, however, degrade over time, as shown in Figure 2.1c. Early detection and repair of paint flaking can therefore avoid corrosion of the actual construction.

Constructions with very long design lifetimes, such as bridges, are additionally often

coated with a layer of metallic zinc in between the metal and paint. Thus, if the paint degrades or is damaged, the layer of zinc starts corroding instead of the steel construction itself. The zinc layer essentially works both as a sacrificial anode and a barrier, preventing corrosion of the construction.

(a) Uniform corrosion. (b) Corrosion pits. (c) Degraded paint.

Figure 2.1: Different types of damages. (a) Uniform corrosion, in which the entire metal surface is evenly corroded. (b) Corrosion pits, a type of corrosion damage in which holes penetrate into the metal. (c) Degraded paint, or paint flaking, where the protected metal is exposed to the environment enabling corrosion.

As construction damages are continuously forming, every little damage cannot be repaired immediately. The Norwegian Public Roads Administration (NPRA) therefore uses estimated area of damaged coating as a criterion to initiate maintenance. Today the area fraction of damaged coating is estimated manually, which introduces a large level of uncertainty, due to the complexity of the task and the fact that many people are involved in the process. Automatic calculation of total damaged area from a set of images would increase the accuracy of the estimate by eliminating the "human factor". However, this is a very challenging task. One necessary component in such a system is the ability to detect and outline corrosion within images, i.e. performing image segmentation of corrosion damages. Current state-of-the-art methods for image segmentation are based on artificial intelligence and machine learning, topics of which are discussed in the next section.

2.2 Artificial Intelligence and Machine Learning

This section gives an introduction to artificial intelligence (AI) and machine learning in order to give readers unfamiliar with key concepts the necessary foundation to understand this thesis. Readers familiar with these topics may skip to Chapter 3.

2.2.1 Artificial Intelligence

There are many definitions of AI. Russell and Norvig categorizes AI definitions into four categories [48]; *thinking humanly*, *thinking rationally*, *acting humanly* and *acting rationally*. Perhaps the most wide and general definition is proposed by Kurzweil and can be classified as acting humanly: "*AI is the art of creating machines that perform functions*

that require intelligence when performed by people." [32]. An elaborate discussion of the abilities of AI is subordinated in this thesis. Kurzweil's definition is therefore sufficient.

Classical AI is highly focused on logic, i.e. designing rational agents. Being rational means to do what has the best expected outcome, given the current knowledge [48]. This can be obtained through carefully designed condition-action pairs. A very simple example is a thermostat maintaining a specific temperature t_0. Whenever the temperature is below t_0 it is increased. Otherwise, if the temperature is above t_0, it is decreased. In many cases, however, covering all relevant pre-conditions is impractical and maybe even impossible. Image recognition is one such example. Hard coding rules for what a cat in an image looks like would require thousands of lines of code. However, even small children can easily distinguish cats from dogs and other objects. This is the motivation for *machine learning* (ML); a subset of AI focused on learning from experience rather than by a set of rules.

2.2.2 Machine Learning

The idea of ML is that if an algorithm is provided lots of sample data, it can build a mathematical model predicting underlying information on new data. Samuel, who coined the term in 1959, broadly defined ML as a "*field of study that gives computers the ability to learn without being explicitly programmed*". Typical examples are image and speech recognition.

ML is only recently adapted to a wide range of fields of study for two main reasons. First, a vast amount of data is needed for the algorithm to learn underlying patterns. Fortunately, more data is available than ever before, with cameras and sensors collecting a variety of data at a rate not possible for humans to process. Enough data is, however, not enough for ML to work. We also need a learning algorithm, a method to train the program on sample data. The backpropagation algorithm, explained in Section 2.2.3, does exactly this, but requires a lot of computational resources. Although enough training data and computing power is still an obstacle for making ML available to everyone, the situation is constantly improving.

2.2.3 Neural Networks

ML is usually based on *artificial neural networks*. It is used to learn a mapping from input data to some desired output. The inspiration comes from the human brain and its interconnected biological neural network.

Intuition

At its core, neural networks are very simple. The *network* is a computational graph and its nodes are called *neurons*. The neurons are structured in layers as shown in Figure 2.2 and perform a mathematical function on a given input. The output value is fed forward to all neurons connected by an edge. These edges between neurons contain adjustable weights, one weight for each edge in the network. The weights are multiplied with whatever value is passed along the edge. Thus, when data enters the network, its data points are multiplied with some weights and transformed by some neuron functions. The goal of ML is to adjust

these weights such that the output of the network describes the input in a sensible way. An example is the input being pixel values of an image, and the output being a single value between 0 and 1 representing the probability of the image containing a cat.

The neural network shown in Figure 2.2 is called a *fully connected neural network* since all neurons in one layer are connected to all neurons in the neighboring layers. In this particular network, the input consists of three data points labeled x_1, x_2 and x_3. The output is two values, y_1 and y_2. Neurons and layers in between are often referred to as *hidden neurons* and *hidden layers*, respectively. The size of the input layer is determined by the shape of the input data (e.g. number of pixels in an image), and the size of the output layer is determined by how many values we want to describe the input data by. The number of hidden layers and neurons to use depend on the task to be solved. Generally, for big datasets and complex tasks we want more neurons than for simple tasks or small datasets. The number of hidden layers and neurons define the *architecture* of the network.

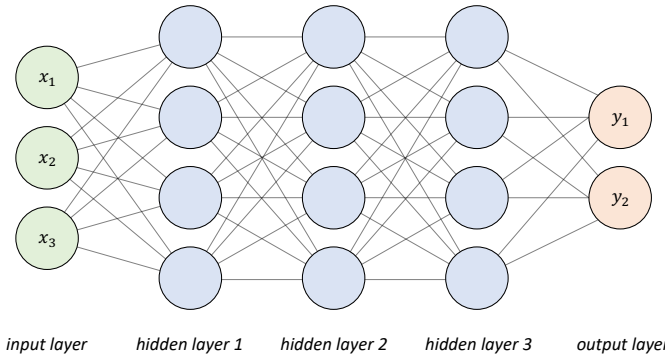


Figure 2.2: A fully connected neural network. Every node, or neuron, in every layer is connected to all neurons in both the previous and next layer. Each edge in the graph corresponds to an adjustable weight between two neurons. The first layer (green) is the input layer and its size is determined by the shape of the input data (e.g. number of pixels in an image). Following layers (blue) are referred to as *hidden layers* containing *hidden neurons*. The number of hidden layers and how many neurons each layer should contain, define the model architecture and is dependent on the task to be solved. The output layer (red) takes as input the output of the final hidden layer and computes a vector of values representing the input data in some way.

The Perceptron

To understand the inner mechanics of neural networks it is easier to start with a simpler architecture. The simplest neural network is the *perceptron*; a single neuron as illustrated in Figure 2.3. It takes as input a weighted sum of input data, $z = \sum_i w_i x_i = \mathbf{wx}$. Again, the vector $\mathbf{x}$ can for instance be pixel values in an image, and $\mathbf{w}$ the vector of adjustable weights on the edges between the input data and the single neuron. The sum is passed to an *activation function* f producing the output $y = f(z)$. f can be any mathematical function mapping the weighted sum to an activation of the neuron. If y is small, it means the neuron is not activated by its input, whereas if y is large the neuron expresses excitement of the input.

The weights are initialized with random numbers, meaning the first input data passed through the perceptron will yield a random output y. If we know what the output should

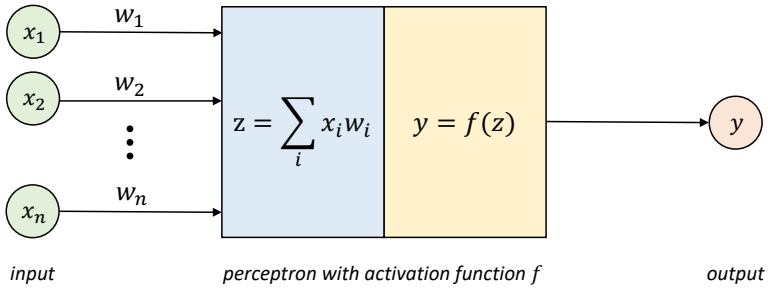


Figure 2.3: A single neuron in a neural network called a perceptron. It takes as input the data points $x_1, ..., x_n$ multiplied by corresponding weights $w_1, ..., w_n$, passes its sum to an activation function f and predicts the output $y = f(\mathbf{wx})$ indicating how activated, or excited, the neuron is.

be, e.g. 1.0 if we know there is a cat in the image, we can adjust the weights so as to get a better output value for this particular image. The next image is then still likely to predict an incorrect output, but perhaps slightly better than for the previous image. We adjust the weights to better classify this image as well and repeat. For each image we pass through the perceptron, the weights are, hopefully, improving and becoming more general, i.e. start understanding the general concept cats, rather than just remembering what images it has seen.

The structure of the perception is, however, very simple and thus limiting. In practice it cannot be used to detect cats in images, as a single perceptron can only classify *linearly separable* data. Linearly separable data, in the two-dimensional case, is data that can be divided into two classes by a single straight line. An example of such is shown in Figure 2.4a. Here the plus-data points (belonging to one class) can be distinguished from the minus-data points (belonging to a different class) by a straight line. In the case of non-linearly separable data, as shown in Figure 2.4b, more than one, and often curved, lines are necessary. In such cases, multiple neurons are needed in the network.

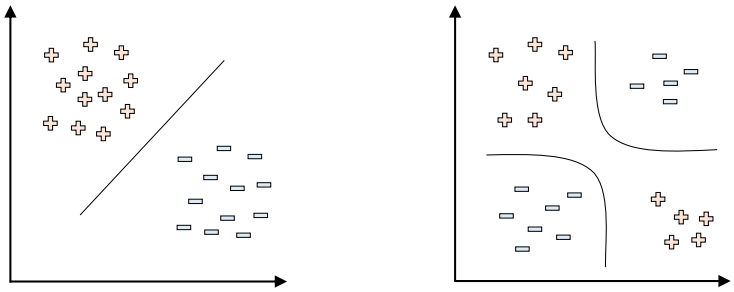
Adjusting the weights of a neural network essentially corresponds to adjusting the shape, slope and positions of the separating lines in Figure 2.4. The next question, then, is how exactly is this performed?

Adjusting the Weights

Constructing a neural network to predict an output $\hat{\mathbf{y}}$[1] for some input data $\mathbf{x}$ is easy. The challenging part is knowing how to properly adjust the weights. First, we need a way for the network to evaluate the quality of the output, as this will affect how much the weights are to be adjusted. This is done using an *objective function*. A simple objective function is mean square error (MSE) which squares the difference between the desired output $\mathbf{y}$ and network-predicted output $\hat{\mathbf{y}}$,

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \,.$$

[1] A vector in the general case, and a hat to indicate it being an estimator/prediction.



(a) Linearly separable data.

(b) Non-linearly separable data.

Figure 2.4: (a) Linearly separable data can (in the case of two dimensions) be separated by a single line. A single perceptron has the expressive power to do this. (b) Non-linearly separable data is not separable by a single, straight line. A single perceptron does not have the expressive power to tackle such data, meaning more neurons are needed.

We want to minimize this function, as this will minimize the error in the predictions. By taking the gradient of the objective function (essentially the chain rule) with respect to a specific weight in the network, we know in what direction to adjust this specific weight. This is exactly what the *backpropagation* algorithm does; it computes the negative gradient of the objective function with respect to each weight, multiplies this with a constant called the *learning rate*[2], and changes the weights accordingly. We use the negative gradient as we want to minimize the objective function, and the learning rate decides (scale-wise) how much to adjust the weights each time (usually a value within the range 10^{-1} – 10^{-6}). If backpropagation is performed for multiple images, the weights will (hopefully) converge to values where they no longer change much for new images. At this point, when new input data enters the network, the predicted output is expected to be close to the desired output. This means the network can, for instance, predict whether there is a cat in the image or not.

Loss Functions and Activation Functions

Objective functions are in the field of machine learning usually called *loss functions*, and the error correspondingly called *loss*. For classification problems, i.e. predicting a discrete class label such as *corrosion* for the input data, the loss function *cross entropy* is more often used than MSE. In the simple case of binary classification (two classes), cross entropy loss takes the form

$$-(y \log p + (1-p) \log (1-p))$$

where the predicted output $p = \hat{y}$ is assumed to be a probability distribution (i.e. value between 0 and 1).

As for activation functions in each neuron we have multiple options. Traditionally, the sigmoid function

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

[2]In practice, during training, the learning rate is usually scheduled to change over time for better convergence.

was used as it squeezes the weighted sum z between 0 and 1 like a probability distribution. In recent years, however, the rectified linear unit $f(z) = \text{ReLU}(z) = \max(0, z)$ (and variations of it) is almost always used as its derivative (1 if $z > 0$ else 0) is more suitable for backpropagation.

If we are to predict, say, one out of K possible labels for some input, the network should have K neurons in its output layer. Using sigmoid or ReLU we could then see what neuron y_j in the output layer is most activated, i.e. has the largest value, and the corresponding class j would be the predicted label. More often, however, the softmax activation function is used in the final layer as it produces a probability distribution over all K classes:

$$\text{Softmax}(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} = \hat{y}_j$$

If the output is, for instance, $\hat{y}_j = 0.8$ for the jth neuron in the final layer, it means the network is $80\,\%$ certain the input belongs to class j. As the final layer now represents a probability distribution, the sum of all its activations equals 1.

Convolutional Neural Networks

Neural networks discussed above consists of fully connected layers, also known as linear layers. In terms of image tasks this has the drawback of being one-dimensional. If an image is input to such a neural network, it must first be flattened, i.e. concatenating all rows of pixels to construct a one-dimensional array. In addition, color images have a third dimension for the RGB-channels also in need of flattening. The result is that neighboring pixels are no longer necessarily next to each other.

The idea of convolutional neural networks is therefore to preserve the 3D structure of color images (width W_i, height H_i and channel depth D_i with $D_i = 3$ for RGB images). Instead of having input be a one-dimensional array of elements, we use a three-dimensional matrix referred to as a feature map. The weights previously attached to edges in linear layers are now exchanged for *filters*.

A filter is a three-dimensional structure of adjustable weights with spatial dimensions $F \times F$ (often 3×3), and depth equal to the depth of the feature map to which it is applied. We move the filter from left to right and top to bottom over the feature map and compute the element-wise product at each location. Taking the sum, we obtain a single scalar value for each location, as shown in Figure 2.5. Applying the filter over the entire input feature map at regular spacing, referred to as *stride* S, thus produces a new 2D output feature map. To make the size of the filter and stride fit the input feature map dimensions, *zero padding*, i.e. neutral pixels, are added if necessary.

The idea is that a filter looks for specific features such as lines and corners in an image, and outputs where in the feature map this is present. If K such filters are applied, we can look for multiple different features in the same feature map. That is, each filter generates a new 2D feature map, and together they construct a 3D output of depth K. More precisely, a convolutional layer accepts input dimensions $W_i \times H_i \times D_i$ and requires the hyperparameters[3] $K =$ number of filters, $F =$ filter size, $S =$ stride and $P =$ amount of zero padding.

[3] A hyperparameter is a parameter describing the network architecture or training procedure, e.g. learning rate, number of epochs (number of training iterations), filter size and how many layers to freeze.

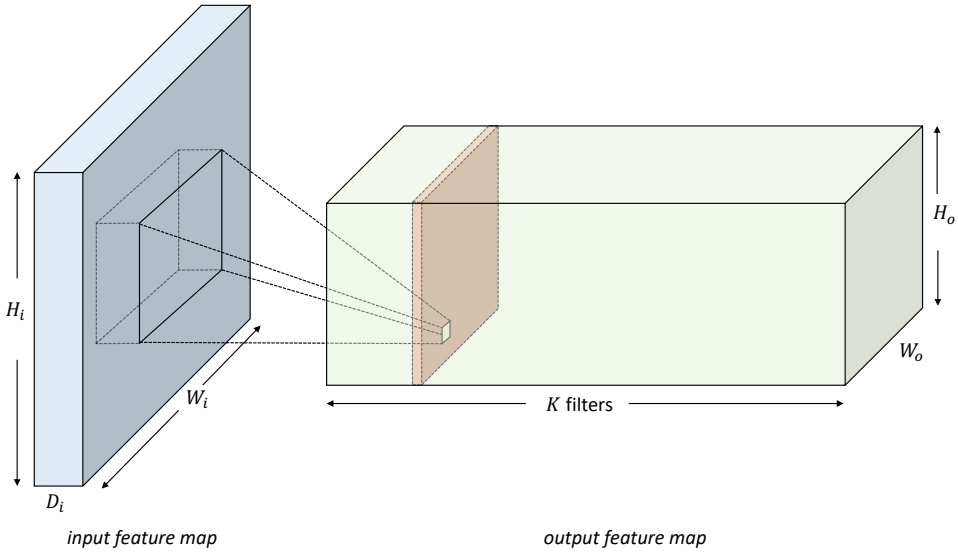


Figure 2.5: Convolutional neural layers. A filter (dashed lines inside blue box) applied to a feature map (blue box) producing a single scalar value (small, light square). A filter detects features in the input and is applied over the entire input map, thus producing an output feature map (red plane). Furthermore, multiple filters are applied over the same input feature map, resulting in a 3D output of depth equal to the number of filters (green box). For input dimensions $W_i \times H_i \times D_i$ the convolutional layer produces a new output feature map with dimensions $W_o = (W_i - F + 2P)/S + 1$, $H_o = (H_i - F + 2P)/S + 1$ and $D_o = K$ where $K =$ number of filters, $F =$ filter size, $S =$ stride and $P =$ amount of zero padding.

The layer produces an output feature map with dimensions $W_o = (W_i - F + 2P)/S + 1$, $H_o = (H_i - F + 2P)/S + 1$ and $D_o = K$. See Figure 2.5.

Multiple convolutional layers are stacked to construct a convolutional neural network. The first layer in a convolutional neural network for image tasks has spatial dimensions equal to the input image dimensions (e.g. 256×256 pixels) and depth 3 if the image is an RGB color image (depth 1 for grayscale). Multiple filters are then run over this feature map to create a new feature map, usually of smaller spatial size and larger depth. This is the input to the next convolutional layer. All weights to be adjusted are found in the filters. The final layer(s) in a convolutional neural network for image classification are usually standard linear layers as these can constitute a probability distribution over the different classes.

Deep Learning

A single convolutional layer can only detect simple features. These features are, however, used in the next layer to detect more complex, composite features.

The term deep learning comes from using many layers in a neural network. In early days of machine learning, networks tended to be shallow as training deeper networks is difficult. Particularly two problems of training deep neural networks have given researchers a challenge over the years. First, during backpropagation two similar, yet opposite, major problems can occur. If the gradient is small, when multiplied repeatedly for each layer dur-

ing the chain rule, the product converges towards zero. Backpropagation then works fine for the deep layers close to the output, while the early layers are practically unchanged. This is called the *vanishing gradient problem*. On the other hand, when the gradient is too large it can explode, called the *exploding gradient problem*. These problems are mainly an issue for deep networks and not that relevant for shallow architectures. Fortunately, development of new techniques, such as ReLU activation function and batch normalization[4], has allowed for really deep networks with hundreds of layers.

Second, deep networks have lots of trainable weights, or parameters, increasing the chance of *overfitting*. When a mathematical model is fit so closely to the training data that it does not generalize well to new data, it is said to overfit. In terms of image classification, we can think of overfitting as if the algorithm starts remembering what images it has seen rather than what characterizes them. If the same few images are used for training over and over again this will easily occur. Two common solutions are usually applied: Either decrease the number of parameters in the model using a smaller architecture with fewer weights or increase the amount of training data. Researchers have put a great effort into minimizing overfitting through smart network architectures [14] as large amounts of training data can be infeasible to collect. The opposite problem, i.e. the network not being able to capture underlying information, is called *underfitting*. This is usually easily solved by increasing the capacity of the network, i.e. using a bigger network.

2.2.4 Classification, Object Detection and Segmentation

Machine learning using labeled data is referred to as *supervised learning*. Supervised learning for vision tasks is often divided into three categories; image classification, object detection and image segmentation. Figure 2.6 illustrates the different types of supervised learning using an image of cats as an example.

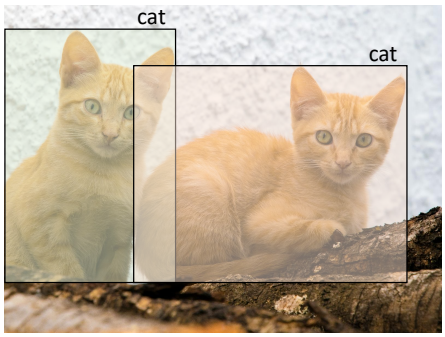
Image classification (Figure 2.6a) is concerned with assigning a class label to images as described earlier, e.g. finding all images in a dataset containing one or more cats. The image is classified as a whole, even if the visual scene is comprised of many different elements. Image classification using neural networks became increasingly popular after the network AlexNet [31] significantly improved state-of-the-art performance back in 2012. Since then, numerous improved architectures have been proposed, including ResNet [21], MobileNet [25] and EfficientNet [55].

Object detection (Figure 2.6b) takes classification a step further by localizing where in the image the objects are located. The output is rectangular bounding boxes around the different objects in the image and a corresponding class label, as shown in Figure 2.6b. Training data must have a similar structure, meaning a dataset for object detection is more complex than for image classification and requires more work to construct. In addition to localizing the objects in an image, object detection also has the benefit of handling objects of different classes in the same image. R-CNN [16] and its improved descendants Fast R-CNN [15] and Faster R-CNN [46] have shown compelling performance on a variety of object detection tasks.

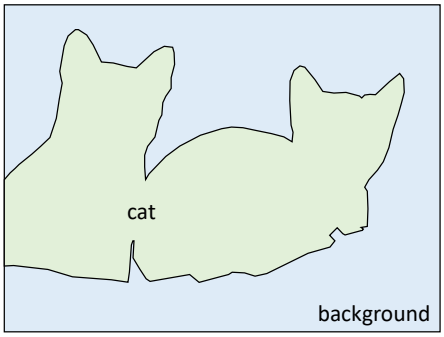
The third type of supervised learning, *image segmentation*, classifies all pixels in an

[4]Batch normalization is a technique to normalizes the layers by adjusting and scaling the activations for better stability.

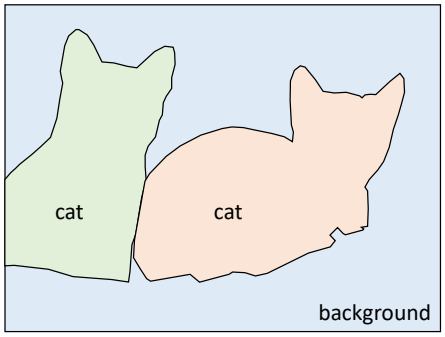
(a) Classification.



(b) Object Detection.



(c) Semantic Segmentation.



(d) Instance Segmentation.

Figure 2.6: Different types of supervised learning for vision tasks. (a) In classification problems, the whole image is assigned a class label, e.g. cats. (b) In object detection, each individual object of interest is both localized and classified, thereby allowing for objects of different classes. (c) Semantic segmentation algorithms assign a class label to each and every pixel in an image. Note that semantic segmentation does not distinguish between multiple instances of the same class. (d) Instance segmentation can be seen as a combination of object detection and semantic segmentation, in which all instances are separately outlined and classified.

image as belonging to a specific class. Matching each class to a color, the output is a color map of the original image outlining object shapes. *Semantic segmentation* (Figure 2.6c) treats all pixels of a class the same, whereas *instance segmentation* (Figure 2.6d) also distinguishes between different instances of the same class. Instance segmentation can thus be seen as a combination of object detection and semantic segmentation and is considered one of the most difficult visual supervised learning tasks.

To predict segmentation masks, convolutional neural networks for image segmentation usually have a stack of *deconvolution* layers, rather than the linear layers used for simple image classification. A deconvolution layer increases the spatial dimensions to any desired size, allowing us to adjust weights based on pixel-wise cross entropy loss between predicted and true segmentation masks of the same size.

A dataset for image segmentation has one or more segmentation masks for each image, i.e. a representation of the shape of all instances in the image. These are referred to as the *ground truths*. Such a dataset is even more challenging and time consuming to

construct than datasets for object detection since every object must be outlined in detail. For semantic segmentation, FCN [39], U-net [47] and PSPNet [58] are popular architectures/algorithms, whereas for instance segmentation the common choice is Mask R-CNN [22].

2.2.5 Transfer Learning

Training deep learning models requires lots of training data and computational resources. Fortunately, many computer vision tasks can still be solved with deep learning through *transfer learning*. When a network is trained from scratch, its weights are initialized to random numbers. There are different theories on what probability distribution gives the best starting point [33], but either way the weights will be far from the result after many iterations of training.

If a network is pre-trained on a big, general dataset, however, the resulting weights probably form a fairly good starting point for other tasks as well. In terms of image classification and segmentation, it makes sense to use pre-trained weights since detection of line segments, corners etc. is similar for all types of images. Thus, weights for early layers can be unchangingly transferred to new tasks, whereas deeper layers detecting more complex features can be retrained, or fine-tuned, on a new target-specific dataset. Using pre-trained models designed to solve one type of task to solve a different task by fine-tuning its layers, is called transfer learning.

Transfer learning will be used in this thesis to solve the problem of corrosion damage segmentation. This will be necessary as the dataset to be used is rather small by today's standards. In addition, designing a new network architecture with proper hyperparameters is challenging. Success is much more likely when using a network architecture proven to work well on different tasks.

ImageNet [26] is the most commonly used dataset for transfer learning for image classification models. It can also be used for image segmentation to more easily detect relevant high-level features. Datasets specifically designed for image segmentation, however, are usually a better option as these contain both images and complete segmentation masks. Pascal VOC 2012 [12], Cityscapes [8] and ADE20K [61] are popular datasets for semantic segmentation, whereas Microsoft COCO [37] is the de facto standard for instance segmentation.

Chapter 3

Image Segmentation of Construction Damages

Image classification networks have been very popular over the last decade. Image segmentation, however, is a much more challenging task and is still in the early phase of being adapted to a wide range of application areas. An increase in number of published papers on image segmentation occurred in the years following the release of Mask R-CNN [22] in 2017, which is state of the art for instance segmentation. Image segmentation of construction damages, however, is still lacking.

This chapter starts by defining use-cases of image segmentation for industrial inspections in Section 3.1. This is followed by Section 3.2 discussing requirements for an image segmentation framework to be used for industrial inspections. Finally, a number of different image segmentation algorithms are reviewed in Section 3.3. Methods based on both traditional computer vision and machine learning are presented.

3.1 Use-Cases of Construction Damage Segmentation for Industrial Inspections

For construction damages in images, there are numerous potential applications for image segmentation. Its real usefulness emerges when automatic inspection systems are used to collect a vast amount of image data. The best such solution is likely camera-enabled UAVs, as is already in the early stage of being adapted to industrial inspections [27, 2, 43]. UAVs have the great benefits of being more efficient than human inspectors, can access areas otherwise difficult to reach and allows for more frequent inspections – all at lower costs. Combined with intelligent damage detection software, this can eliminate the need for most human inspectors as well as being more accurate and objective.

One straight forward use-case of UAVs with image segmentation software is remote controlling the UAV to take images of a construction. Images containing a damage can then be saved along with sensor data locating where and at what altitude the images were taken. This allows an inspector to study the damages remotely and initiate maintenance

where needed.

Furthermore, image segmentation can aid a UAV in becoming autonomous, i.e. removing the need for an operator to control the UAV. For this to work optimally, the system should be combined with 3D models of the construction and SLAM[1] to help keep track of what parts of the construction is covered and to avoid many duplicate reports of the same damage.

Pure image segmentation has no concept of actual sizes, only the size of segmented objects relative to the image frame. Combined with 3D models and SLAM, however, the real size of damages can be computed. Furthermore, the total area of damaged surface can be tracked. This is particularly useful as it is an often-used metric to initiate maintenance. If the total damaged area is monitored for all constructions of relevance, maintenance can be automatically and efficiently scheduled.

With autonomous UAVs able to recognize damages, we can efficiently monitor damages over time, e.g. weekly take an image of all known damages. Time series predictions can then potentially be used to forecast how a damage will continue to develop. If this is successful, it can help estimate future expenses related to maintenance and prevent dangerous damages from developing.

To summarize, the benefits of image segmentation of construction damages are vast and many. There are, however, many challenging requirements needed to be overcome for the image segmentation software to be useful in the above use-cases. These are discussed next.

3.2 Requirements of Image Segmentation for Industrial Inspections

If a UAV is remote controlled by an operator, damages can be outlined and presented for an inspector to decide if maintenance is needed. The main requirement is for the software to have good recall, i.e. avoid missing any damages.

If we are to remove the human involvement altogether, the task becomes much harder. First, to be able to estimate total damaged surface and navigate autonomously, the software must not only be able to recognize damages, but also the constructions themselves. A dataset with annotated constructions as well as damages is then needed. Furthermore, prediction performance needs to be accurate if estimated damaged area is to be used as a reliable criterion to initiate maintenance.

Second, some damages need immediate maintenance and cannot wait for the remaining construction to degrade and trigger maintenance. An ideal image segmentation system should therefore also be able to identify the severity of damages. This is likely a very difficult task for multiple reasons. First, an image may not reveal whether the damage is present on the surface only or if the whole cross section is damaged. Second, determining the potential consequences of a damage requires lots of domain and construction specific knowledge. Two images of seemingly identical corroded bolts, for instance, can have dramatically different importance to the intactness of a construction.

[1] Simultaneous localization and mapping (SLAM) is the problem of constructing a model of the environment through investigation, while simultaneously keeping track of where in the model the agent is present.

It is difficult to define a definitive threshold for accepted prediction performance. First, as will be discussed in Section 4.3, there are many ways to evaluate segmentation performance. Second, the degree of support by human inspectors greatly affects the requirements. Needless to say, the software should not fail to notice important damages, and avoid reporting too many false positives. For binary segmentation, this will likely require at least $90\,\%$ IoU (see Section 4.3 for definition). Furthermore, instance segmentation is considered more versatile for the above discussed use-cases, meaning good distinction of instances is important.

Additionally, there are requirements for the segmentation framework in terms of computational speed and memory consumption. For a UAV to operate *autonomously*, it needs to process images at a rate close to video standards, i.e. 24 images per second. Else, the lag would be too big for the UAV to fly and navigate properly. Lower frame rates can be acceptable for an *automatic* system, i.e. a system partly supervised by an operator or specifically designed to work in a pre-specified area. Furthermore, peak memory usage should not exceed available memory. To process sufficiently many images per second, a GPU is likely needed, meaning its video memory is the constraining parameter, not the CPU's RAM.

Summarizing, the challenges are many and difficult. Partly autonomous industrial inspections are only a realistic goal in the future if accurate damage segmentation proves successful. The following section therefore reviews different algorithms that can be evaluated for construction damage segmentation.

3.3 Algorithms for Image Segmentation

This section presents various methods for image segmentation. First reviewed are thresholding, clustering and region growing; traditional computer vision algorithms suitable for simple tasks. Next, fully convolutional networks for image segmentation are reviewed, particularly FCN, U-Net and PSPNet, the latter considered state of the art. This is followed by a presentation of Mask R-CNN, which is state of the art for instance segmentation.

Presentation of traditional computer vision methods for image segmentation is based on various courses at NTNU, particularly lectures from TTK25 Computer Vision for Control [40]. Literature search for machine learning-based methods is largely based on the website Papers Width Code [6]. It has ranked entries for most machine learning tasks, with links to corresponding published papers and code.

3.3.1 Thresholding

The simplest algorithm for image segmentation is *thresholding*. Thresholding segments an image purely based on pixel values in the image. For binary segmentation, a pixel is considered a positive class if the pixel value is above a specified threshold. For multi class segmentation, ranges of pixel values must be specified. The method is very efficient and easy to implement. The main challenge lies in selecting the threshold value, for which several methods exist: Constant thresholding values are very inflexible and requires all images to have a very similar color distribution. Histogram methods use the pixel intensity distributions to localize interesting values for each image, such as peaks and valleys.

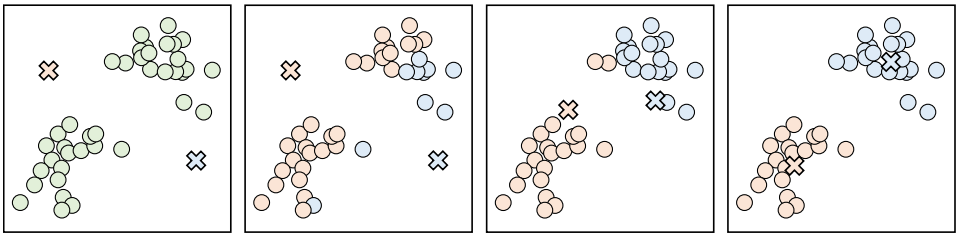
Examples of such methods are balanced histogram thresholding and Otsu's thresholding algorithm [44].

The very characteristic red/brown color of corrosion damages makes thresholding an interesting option. However, the algorithm is very sensitive to different lighting conditions, and anything with colors similar to corrosion damages would be falsely reported as a damage. Classification performance is therefore expected to be poor, and hence we need something more versatile.

3.3.2 Clustering

Whereas thresholding only uses individual pixel values to determine the class, clustering methods also incorporate positional information. An area densely covered with red/brown pixels, for instance, is much more likely to be actual corrosion than random single red/brown pixels.

The easiest clustering algorithm is perhaps K-means clustering: K pixels are randomly chosen as initial cluster centers, and all pixels are classified as the nearest cluster center. The distance is usually measured as a weighted combination of different features, such as pixel color, pixel intensity and Euclidean distance. The image is now segmented into K different clusters, each representing an instance of a class. The cluster centers are, however, randomly chosen, meaning the segmentation is likely very poor. New cluster centers are therefore computed as the center of each cluster, before the algorithm iterates. This effectively moves some pixels from one class to another. We terminate when the cluster centers no longer move between iterations. Figure 3.1 illustrates three iterations of K-means clustering for general data points in two-dimensional space.



(a) Iteration 0. (b) Iteration 1. (c) Iteration 2. (d) Iteration 3.

Figure 3.1: K-means clustering based on Euclidean distance. (a) Initially the data points (green) are not assigned any cluster. We randomly sample K initial cluster centers (X's, here $K = 2$). (b) All data points are then assigned to the closest cluster center, separating the data into K clusters. (c) New cluster centers are selected as the center of each cluster. Data points now closer to a different cluster center changes label. (d) We continue until the cluster centers no longer move around.

K-means clustering is simple to implement and is guaranteed to converge, although it may converge to a local optimum only. Other drawbacks include that the number of clusters, K, must be known in advance, the clusters are assumed to have a circular shape (as defined by the distance measure) and no noise modeling is included (i.e. outliers can badly affect the result). It is worth noting that modifications to K-means clustering exist,

overcoming some of these cons, for instance an adaptive K-means algorithm eliminating the need to specify the hyperparameter K [60].

An alternative clustering algorithm is DBSCAN [10] (density-based spatial clustering of applications with noise). DBSCAN creates clusters based on pixels densely packed together. It starts with a random pixel and adds to this all unlabeled pixels nearby if the number of such neighbors is above a specified threshold (i.e. if the density is big enough). This process is then recursively applied to all newly added pixels.

The main benefit of DBSCAN is that no assumptions on the number and shape of segmentation areas are needed and that noise/outliers are automatically taken care of. However, now the cluster density (i.e. the threshold for adding a pixel) must be known in advance and assumed uniform.

For the purposes of construction damage segmentation and industrial inspections, clustering methods fall short for numerous reasons. First, we do not know in advance how many damage instances there are in an image, rendering K-means clustering infeasible. Similarly, requiring uniform clusters renders DBSCAN infeasible. Second, clustering methods divide the entire image into instances, but do not actually classify them as specific classes. Additionally, the background is unnecessarily divided into instances for all sorts of irrelevant objects.

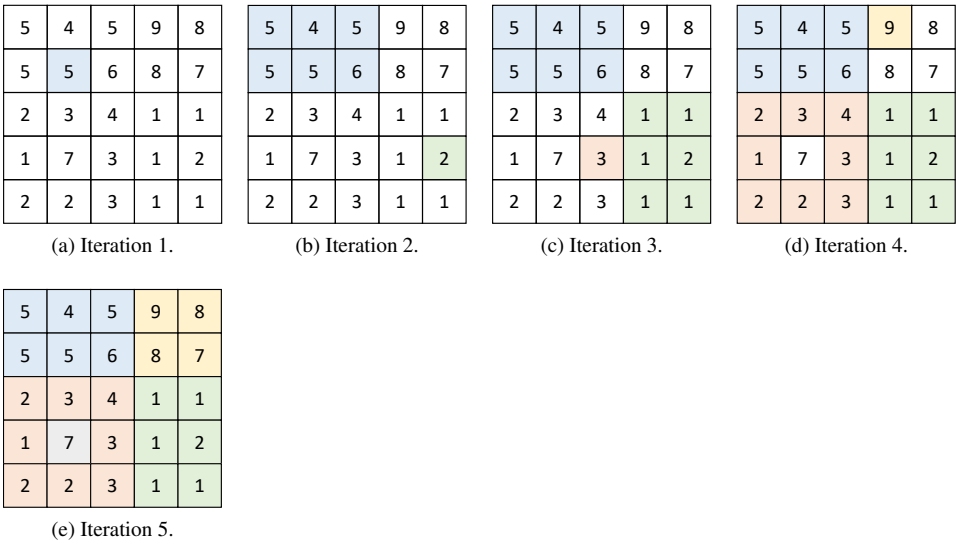
3.3.3 Region Growing

With region growing, pixels are selected as seed points and the local surrounding area is analyzed to determine which pixels should belong to the same class. Surrounding pixels of similarity, usually in terms of pixel value, are thereby added, iteratively increasing a cluster region. The algorithm is similar to clustering methods but has fewer assumptions on the input data and can create more precise segmentation masks. The only parameter needed specified is the allowed difference D between pixels for them to be added to the cluster, with the assumption that neighboring pixels within a region has similar values.

The main drawback of region growing is that the result is highly dependent on selecting good seed points. In fact, the order in which we grow the seeds can affect the resulting masks. The algorithm is illustrated in Figure 3.2.

Region growing is not suitable for construction damage segmentation for multiple reasons. First, a general value for D suitable for all images is practically non-existent. A too small value results in too many instances, whereas too large value generates too big instances. Second, region growing requires relatively high contrast between different regions, whereas corrosion damages often have smooth edges. Finally, similar to clustering methods, the image is segmented without labeling each instance a specific class.

Generally, the above reviewed image segmentation methods are mostly suitable for very specific, simple tasks, e.g. analysis of conveyor belt images taken by a stationary camera. Images taken during industrial inspections are more general, comprising a wide range of different scenes. The remainder of this chapter therefore studies image segmentation methods based on machine learning, which better handle complex data.



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3. (d) Iteration 4.

(e) Iteration 5.

Figure 3.2: Region growing with allowed neighbor difference $D = 1$. (a) A random pixel is selected as the first seed (blue). (b) All neighboring pixels are recursively added to the region if its value differs by at most D. When no more pixel can be added, we select a new seed (green). (c)–(e) We continue selecting new seeds and grow the corresponding region until all pixels are classified.

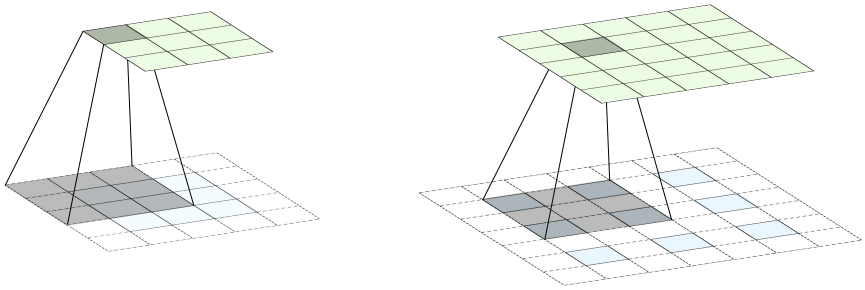
3.3.4 FCN

Classification networks usually consists of a stack of convolutional layers followed by one or more linear layers. The final linear layer creates a probability distribution over the predicted classes, i.e. a one-dimensional vector not suitable for pixel-wise predictions.

An FCN is a network architecture using only convolutional layers, i.e. no need for linear layers, sliding windows or sub-networks for region proposals. Long, Shelhamer, and Darrell [39] introduced in 2014 FCNs for semantic segmentation trained end-to-end with supervised pre-training for pixel-to-pixel predictions. Their work is based on the success of deep classification networks such as VGG [52] and GoogLeNet [54]. When referring to FCN in the following, the architectures by Long, Shelhamer, and Darrell is to be understood, even though FCN is also used as a generic term describing all convolutional-based architectures without linear layers.

In order to produce a segmentation mask as output, the final layer(s) of existing classification networks must be altered to transform coarse outputs to dense pixel maps. The authors of FCN describe multiple solutions and found *deconvolution* to perform the best. Deconvolution adds padding around and/or in between pixels of a feature map before a regular filter is applied as in usual convolution, see Figure 3.3. The result is a spatially larger feature map in which the up-sampling is learned during training. Note that deconvolution also performs the convolution operation, and hence the name *transposed convolution* is perhaps better.

Transforming pre-trained classification networks performing well on ImageNet [26]



(a) Regular Convolution. (b) Deconvolution.

Figure 3.3: (a) Regular convolution as described in Section 2.2. Using zero padding (white tiles) the spatial dimensions remain the same. (b) Deconvolution, in which zero padding is added in between pixels (blue tiles) before regular convolution is applied over the resulting feature map. The result is an up-sampled feature map with increased spatial dimensions. The weights in the filters of a deconvolution layer thus learns how to properly up-sample a feature map.

to an FCN for semantic segmentation is done as follows: The final classification layer is removed, and all remaining linear layers replaced by convolutional layers with K 1×1 filters. The filters will predict one of the K classes for each point in the feature map. Finally, a deconvolution layer is appended to up-sample the outputs to pixel maps of spatial size similar to the input image. The resulting network is illustrated (as FCN-32s) in Figure 3.4. Fine-tuning with per-pixel cross entropy loss on the ground truth and predicted segmentation masks then gives reasonable results. Applied to VGG16 [52], the authors achieved state-of-the-art performance at the time. For GoogLeNet, however, the performance was not compelling.

The above FCN architectures produce coarse segmentation masks with low level of details. The authors therefore added links from early layers to the prediction layer. This enables the network to use the coarse outputs to make local classification (*what*) and the finer, earlier layers to better put the shape of the output mask into context (*where*). Multiple placements and combinations of skip connections were tested as shown in Figure 3.4. The result is significantly improved performance of $62.7\,\%$ mIoU (mean intersection over union, see Section 4.3 for definition) for the best network named FCN-8s, as measured on the PASCAL VOC 2011 [11] test data set. Additional links had diminishing improvements and therefore only introduces unnecessary complexity.

Long, Shelhamer, and Darrell found increasing performance when using a bigger dataset than the original VOC dataset of 1112 images. Interestingly, however, no gain was found using artificial data augmentation such as flipping and translation.

FCN has potential for construction damage detection. Being a machine learning algorithm, it overcomes most problems with traditional computer vision algorithms discussed earlier. On the other hand, FCNs are just modifications of plain classification networks, i.e. they are not specifically designed from scratch for image segmentation. Additionally, as the field of AI is developing fast, FCNs are now rather old and outperformed by newer network models.

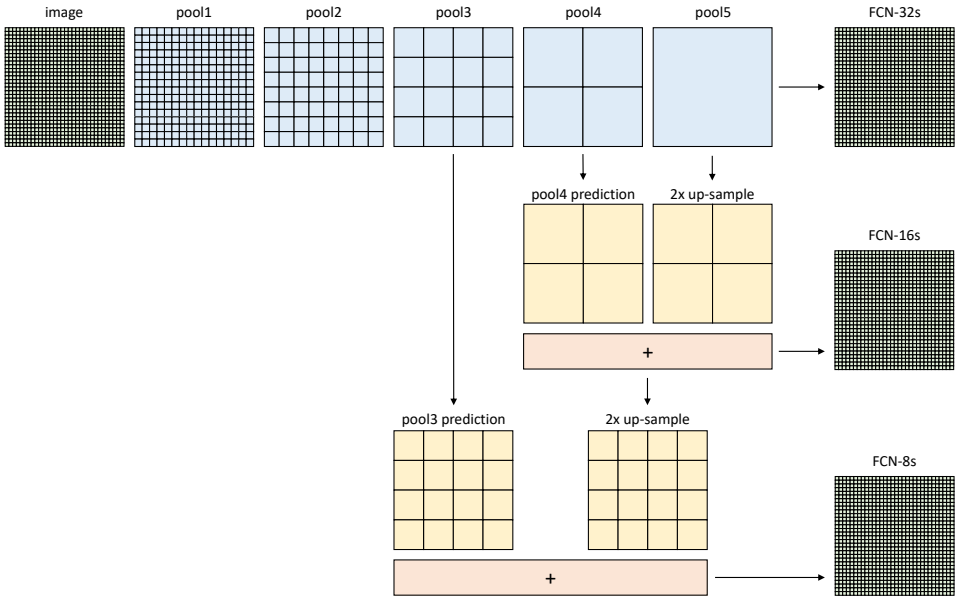


Figure 3.4: FCN architectures for semantic segmentation based on VGG16. Grids shown represent relative spatial sizes after pooling and prediction layers (that is, intermediate convolutional layers are not shown). FCN-32s is the naive network directly up-sampling 32x yielding coarse segmentation masks. Fusing predictions from pool4 with 2x up-sampled predictions from pool5 yield FCN-16s, which produces better and more detailed segmentation masks. Even better performance is obtained with FCN-8s which fuses pool3 predictions with 2x up-sampled predictions from the fused feature map used in FCN-16s.

3.3.5 U-Net

Introduced in 2015, U-Net [47] improved on the original FCN architecture. Ronneberger, Fischer, and Brox created a U-shaped architecture and a training strategy requiring fewer training images yet also providing higher resolution segmentation masks.

The new architecture consists of two parts: First is a regular convolutional network typically used for image classification. Like many popular network architectures, it uses 3×3 convolutional filters, ReLU activation functions and max pooling[2]. This forms a contracting path from the input image down to a wide feature map with low spatial dimensions, see Figure 3.5. The second part of the network is an almost symmetrical expanding path using deconvolutions to obtain a feature map (i.e. a segmentation mask) of approximately same spatial size as the input image. Finally, links between the contracting and expanding paths, on corresponding heights/levels, are added in order to propagate information forward from early layers. This is somewhat similar to the links in the original FCN, but a key difference is the increased number of feature channels in the links and the up-sampling part of the network, allowing for more flow of contextual information resulting in better classification. Furthermore, the links are somewhat simplified, and the number used increased, providing better feature propagation at different scales.

[2]Max pooling returns the maximum value within a grid of values, usually used to reduce spatial dimensions of feature maps in convolutional neural networks

The original U-Net architecture is illustrated in Figure 3.5 and was originally developed for biomedical image segmentation. Later the model has been adapted to a wide range of vision tasks, and varieties of the architecture have been developed. An example is ResNet-U-Net adding residual connections in addition to the longer copy-resize connections.

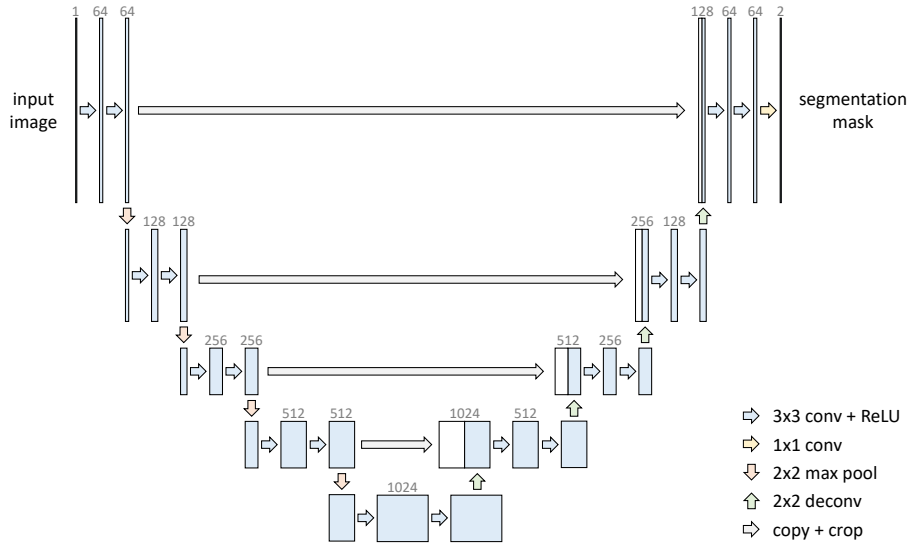


Figure 3.5: The original U-Net architecture. Forming a U-shape, the left-hand side is a contracting path and the right hand side an expanding path. Links between the two paths allow for contextual flow, similar to the links in FCN. The number above each layer denotes the number of feature maps/channels. White boxes indicate copied feature maps. In this very architecture, the input image is gray scale and the output a binary segmentation mask.

With limited training data, the authors extensively used artificial data augmentation. This was essential in order for the network to be invariant to translation and rotation, as well as contrast variations and shape deformations of biological structures. In particular they found that by adding random elastic deformations, performance was significantly improved.

There are multiple aspects making U-Net interesting for construction damage segmentation. First, it was originally designed for biomedical imaging. In biomedical images, very much like for construction damages, there are not as many rich features defining an object as for common everyday objects. This is further discussed in Section 4.1 detailing the corrosion dataset constructed for this thesis. Second, unlike FCN, the authors found improved performance with U-Net through data augmentation. This was crucial since their dataset was small. With 608 images, the corrosion dataset too is considered small by modern standards. However, much like FCN, U-Net is rather old and is now outperformed by newer, more sophisticated networks.

3.3.6 PSPNet

Humans have a tremendous ability to distinguish and classify seen objects, even from a young age. We could say our inference is two-stage: First we recognize the shape, color and size of an object. From this alone we can often conclude accurately what we are looking at. Next, we use surrounding visual clues to further increase our certainty. For instance, if a vehicle is seen on a river, it is likely a boat even if it looks like a car. The previously reviewed segmentation algorithms struggle to utilize this kind of global information; if it looks like a car, it certainly must be a car.

PSPNet [58], short for pyramid scene parsing network, aims at solving this discrepancy. It was developed by Zhao et al. in 2016 and is still one of the very best-performing networks on multiple popular segmentation datasets, e.g. Cityscapes [8], PASCAL VOC 2012 [12] and ADE20K [61].

The links from early layers in both FCN and U-Net can be seen as an attempt to incorporate global contextual information in the predictions. Zhao et al. found, however, that these descriptors are very generic. Their solution is a pyramid pooling module: Input images are first passed through a standard semantic backbone network, e.g. ResNet [21], to produce a feature map. See Figure 3.6. We then perform *sub-region average pooling* at $N = 4$ different spatial sizes: The first level is just global average pooling, i.e. each channel in the feature map is summed and divided by the spatial size. This constructs the green $1 \times 1 \times C$ feature map in Figure 3.6. In parallel, the feature map is divided in a 2×2 sub-grid to compute local average pooling in each quadrant. This produces the yellow $2 \times 2 \times C$ feature map in Figure 3.6. Similarly, $3 \times 3 \times C$ (blue) and $6 \times 6 \times C$ (red) feature maps are computed by sub-region average pooling. The number of levels in this "pyramid", N, can be altered to construct different architectures, with $N = 4$ in the original implementation.

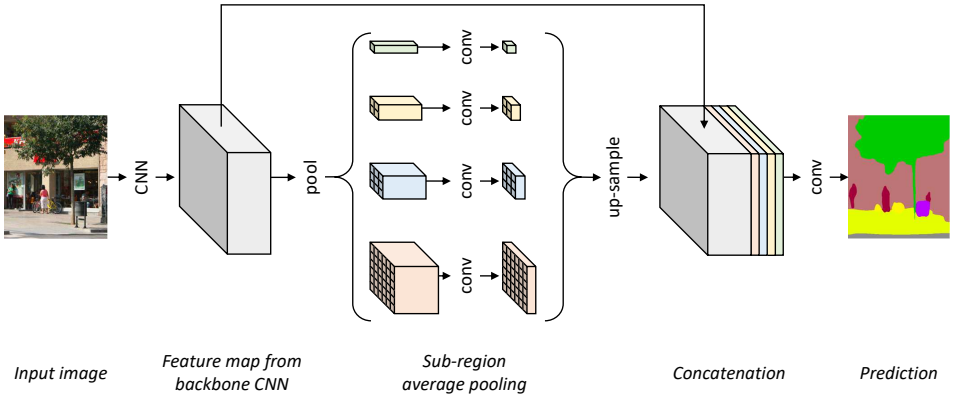


Figure 3.6: PSPNet architecture. The key contribution of PSPNet is a pyramid pooling module performing sub-region average pooling at different scales. This better incorporates contextual information, resulting in increased performance.

In total we now have $N \cdot C$ channels, so to maintain the weights of global features, 1×1 convolution is applied to each feature map, decreasing the channel dimensions from C to C/N. The feature maps are then immediately up-sampled to the size of the original

feature map. Next is concatenation of all feature maps, i.e. both the original and up-sampled feature maps. Finally, a convolutional layer is added to produce the prediction with segmentation masks.

The idea behind the pyramid with sub-region average pooling is that we obtain local information from all areas within an image and at different scales. Intuitively, a small sub-region layer could recognize a vehicle and a bigger sub-region layer could recognize a surrounding river, implying the vehicle is likely a boat.

On the Cityscapes [8] and Pascal VOC 2012 [12] datasets, PSPNet obtains $78.4\,\%$ and $82.6\,\%$ mIoU, respectively. Comparing this to $65.3\,\%$ and $62.2\,\%$ for FCN, shows a huge improvement. For the very challenging ADE20K dataset [61], PSPNet achieves a test score of $55.4\,\%$ mIoU. Four years after its release, this is still the second ever highest reported test score. To obtain these results, two additional key methods were adapted in addition to the pyramid pooling module. First is an auxiliary loss added in the backbone network to help optimize the learning process, similar to ideas in GoogLeNet [54]. Second is data augmentation, including random horizontal flipping, image resizing, slight rotations and Gaussian blur.

PSPNet is likely a great option for construction damage segmentation. First and foremost, PSPNet is state of the art without bells and whistles. Second, PSPNet exists in both a 50- and 100layer version. A reduced network size is desirable for the small dataset used in order to avoid overfitting. Finally, good understanding of contextual information is highly relevant for construction damages: Corrosion is only present on metals, cracks in concrete only appear on, well, concrete, and so forth. That said, the dataset used in this thesis only contains segmentation masks for corrosion damages, and hence knowledge about surrounding metal is not explicitly taught. Full potential of PSPNet is therefore likely not obtained in this thesis.

3.3.7 Mask R-CNN

Mask R-CNN [22] was introduced in 2017 by He et al. and has since been state-of-the-art for *instance* segmentation. The algorithm extends Faster R-CNN [46], which again builds upon the papers Fast R-CNN [15] and R-CNN [16]. This section reviews Mask R-CNN without explicitly introducing its predecessors first. Keep in mind, therefore, that most of the foundations for Mask R-CNN should be credited to the authors of R-CNN (Girshick et al. [16]) and Fast/Faster R-CNN (Girshick [15] and Ren et al. [46]).

Mask R-CNN is a two-stage network: In the first stage the image is scanned for region proposals, i.e. areas likely to contain an object. In the second stage, these proposals are classified, and corresponding bounding boxes and segmentation masks are generated. In order to explain Mask R-CNN in detail, as it is very complex, it is convenient to break it down into four parts; a backbone network, a region proposal network, classification and bounding box regression of region proposals, and finally a segmentation mask prediction branch. An overview of the entire network is shown Figure 3.7 putting each of these four parts into context.

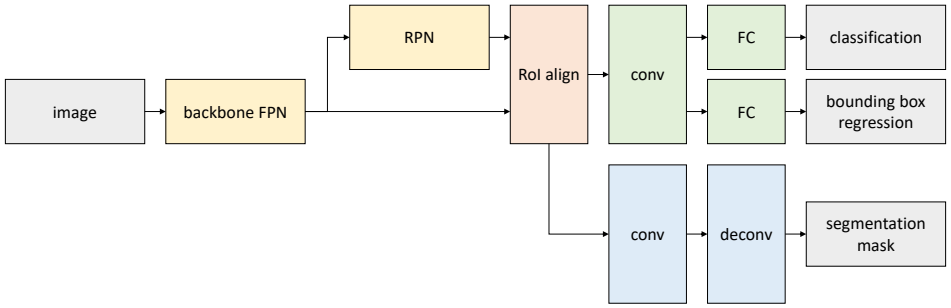


Figure 3.7: Complete Mask R-CNN pipeline. The backbone network first detects features in an input image. To better capture features at different scales, a feature pyramid network (FPN) is often used as the backbone network. Next, a region proposal network (RPN) suggests object candidates, known as regions of interest (RoI). Before classifying and constructing a segmentation mask for the RoIs, Mask R-CNN performs RoI align, a method of aligning the RoIs to properly work with the prediction sub-networks. The final steps of Mask R-CNN are class prediction, refinement of bounding box location and generation of segmentation masks, all of which are performed in parallel.

Backbone Feature Pyramid Network

Mask R-CNN starts with a backbone network whose purpose is to serve as a feature extractor. Any convolutional neural network designed for image classification can be used, e.g. ResNet [21]. The input images are through this network transformed to a feature map with smaller spatial size and increased number of channels detecting high-level features. This forms a contracting path with increasing semantic value and decreasing resolution.

Ideally, however, we want both high-resolution feature maps and big semantic value. We can obtain both using a *feature pyramid network* (FPN) [36]: Given a regular network, referred to as bottom-up, we add an additional "opposite" network, referred to as top-down. See Figure 3.8. The first layer in the top-down network (M5) is constructed using a 1×1 convolutional filter from the final layer in the bottom-up network. The second layer (M4) is a summation of the previous layer (M5) up-sampled and the corresponding layer in the bottom-up network (C4) with a 1×1 filter. This continues for as many layers as desired, but usually the top-down network is chosen somewhat smaller than the bottom-up network for computational complexity reasons. From the top-down network we can apply a filter to each layer (typically 3×3) to produce the final feature maps (P5–P2). The key point, now, is that these feature maps have strong semantic value and are available at different scales, the latter allowing the backbone network to more easily detect objects at different scales.

Region Proposal Network

The next step in Mask R-CNN is a *region proposal network* (RPN). RPNs are neural networks scanning an image for *regions of interest* (RoI), i.e. areas likely to contain a relevant object. To do this, boxes, or *anchors*, are created over the entire image in a sliding window-like fashion, see Figure 3.9a. To cover the entire image and capture all types of objects, the boxes have varying size and aspect ratios. High recall (percentage of relevant

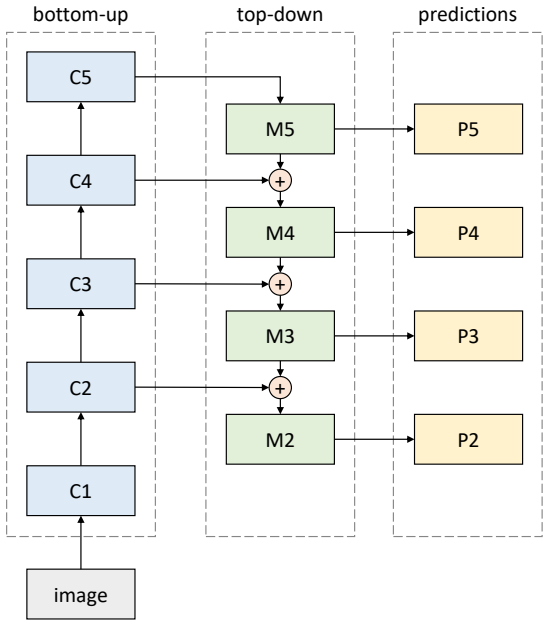
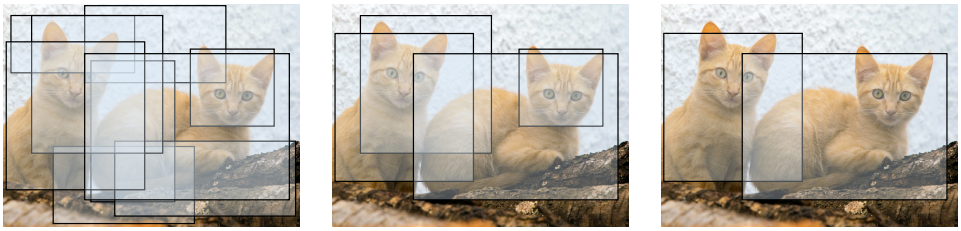


Figure 3.8: Feature pyramid network (FPN). A bottom-up network (C1–C5) detects features in an image like any other classification network. ResNet is often used. A top-down network (M5–M2) working in the opposite "direction" is constructed by adding its previous layer up-sampled and corresponding layers in the bottom-up network convolved. Each of these new layers are then used to produce prediction layers (P2–P5) containing high semantic value at multiple different scales.

instances reported) is crucial in this stage, as non-detected objects cannot be detected in later phases. Approximately 200 000 anchors are therefore instantiated, meaning many will overlap and cover the same object.

In Mask R-CNN, we do not apply RPN directly to the image itself, but rather on the feature maps produced by the backbone network. This significantly speeds up the algorithm as the backbone network and RPN share computation.

Two outputs are computed for each anchor. First is a binary classification score indicating whether the anchor likely covers a relevant object (a positive anchor) or just covers parts of the background (a negative anchor). Second, for each anchor, the RPN outputs a bounding box refinement, i.e. a prediction of how much the anchor box should be moved and resized to better cover the object. All negative anchors can be discarded immediately, as shown in Figure 3.9b. Positive anchors with high probability of containing an object are RoI candidates. Many such anchor boxes will, however, likely overlap much. We therefore remove all sub-optimal anchors overlapping too much with a better, different anchor, a process referred to as non-max suppression. See Figure 3.9c. Remaining anchors are the final RoIs. Note that in the case of multi-class object detection/segmentation, the anchors have no knowledge about which class it represents, just that it is an area worth investigating. Furthermore, any anchor can be discarded in later stages if it is found to not fit any class.



(a) All Generated Anchors. (b) Negative Anchors Discarded. (c) Non-Max Suppression.

Figure 3.9: Proposal and suppression of anchors. (a) Anchors with all sorts of sizes and aspect ratios, i.e. potentially interesting areas, are generated over the entire image. (b) All negative anchors, i.e. anchors classified as covering background, are discarded immediately. (c) As multiple positive anchors may overlap, we remove all sub-optimal anchors, a process referred to as non-max suppression. In practice, anchors are not generated over the actual images directly, but rather over the feature maps from the backbone network.

Classification and Bounding Box Regression of RoIs

The third stage of Mask R-CNN takes as input the RoIs produced by the RPN in the previous stage and computes two new outputs. This is not as straight forward as it might seem. Whereas neural networks for classification typically assume fixed sized inputs, the RoIs can take on different aspect ratios and sizes. A solution to this problem is RoI pooling [15]. Assume we have an $h \times w$ RoI which we need scaled to $H \times W$ by max pooling. Since h/H and w/W in the general case are non-integers, the dimensions do not match up perfectly. In RoI pooling we simply round to the nearest integer and divides the RoI into a sub-grid of this size. We then take the maximum value within each cell to construct the desired $H \times W$ feature map. What effectively happens is a quantization altering the RoI slightly. This method is widely used in object detection algorithms as classification is robust to small translations. Simply put, a rectangle outlining an object is still decently outlining the object if it is moved slightly.

For segmentation masks, however, this is problematic as we want good pixel-to-pixel accuracy. The authors of Mask R-CNN therefore proposed *RoI align*, which uses bi-linear interpolation to avoid any quantization. First, instead of rounding the fractions h/H and w/W to integers, the floating numbers are kept. We then divide the RoI into bins as before, but now allowing decimal-sized sub-grids not perfectly fitting the RoI pixel grid. See Figure 3.10. We want to extract one value from each sub-grid to constitute the $W \times H$ output feature map. Each sub-grid will, however, cover parts of multiple RoI pixel values. To make sure all these pixel values are accounted for, we apply bi-linear interpolation. That is, within each sub-grid we first sample four evenly distributed points, shown as red dots in Figure 3.10. For each point, we find the four closest pixel centers, depicted as green dots in Figure 3.10. Interpolating these values, we obtain a combined value for the sampled point. This is repeated for all sampled points. We can then apply max pooling to the four values within each sub-grid. Effectively we use the entire RoI when computing subsequent feature maps.

Back to Mask R-CNN, RoI align is used to properly align and scale the RoIs. Two values are then computed for each RoI. While the RPN only distinguishes between back-

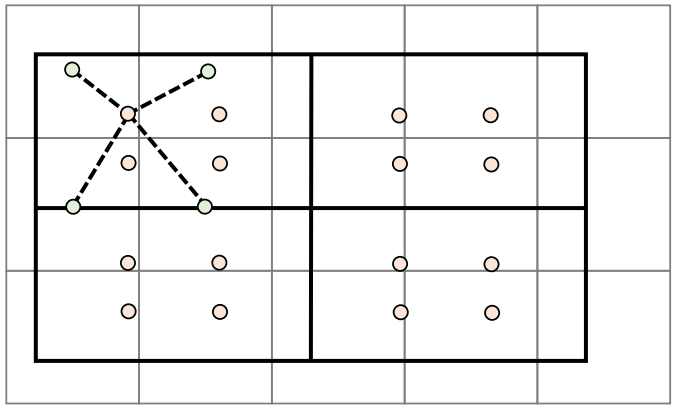


Figure 3.10: RoI align. Since the 2×2 output grid does not fit the RoI pixel values (background grid) perfectly, each cell in the grid covers parts of multiple feature map pixels. To account for all of them, four points (red dots) are sampled in each cell. For each point, we locate the four neighboring pixel centers and interpolate the values. With four computed values in each cell we can now perform max pooling to obtain the desired 2×2 output.

ground and potential objects, the first output of the current stage further classifies the objects as a specific class (e.g. dog, cat, etc.). A deeper sub-network is needed for this finer classification. Note also that this network can still predict an RoI as background, in which case it will be discarded.

The second output for each RoI is a further refinement of its bounding box. This works similar to the RPN but should produce more accurate bounding boxes due to better abstracted features, i.e. knowledge about the correct class.

Segmentation Masks

The network constructed this far is essentially Faster R-CNN [46] (with the addition of RoI align), an algorithm suitable for object detection. The main contribution in Mask R-CNN (besides RoI align) is an additional network branch to compute segmentation masks. The branch is a fully convolutional network running in parallel with the prediction of class label and bounding box offset. To keep the computational complexity low, small masks of size 28×28 are generated. The masks are, however, soft masks (i.e. $0 - 255$ rather than binary $0/1$), meaning they incorporate more information and can be scaled up to full sized binary masks during inference.

In many networks the class label is dependent on the generated masks, meaning different classes compete in forming the segmentation mask. This can lead to segmentation masks "averaged" over multiple classes. Mask R-CNN, on the other hand, generates masks individually optimized for each and every class, and uses the result from the independent class prediction branch to keep the correct mask only. The authors found this decoupling to be important for good instance segmentation results.

Complete Network

The complete Mask R-CNN network is schematically illustrated in Figure 3.7. The pipeline can be summarized as follows: A backbone network abstracts useful features and reduces the spatial size. This network can be any typical classification network, but usually a feature pyramid network (FPN) is used for better scale invariance. A region proposal network (RPN) takes these features as input and generates regions of interest (RoIs). As the RoI bounding boxes have different size and aspect ratios, we use a method called RoI align to properly scale and align them. From here we have two separate branches. The first branch puts a class label to each RoI, as well as refines the precise location of the bounding boxes. The second branch uses a fully convolutional network to, for each RoI, predict a segmentation mask for each class. The two branches are in other words decoupled, allowing the second branch to suggest masks for all classes, instead of them competing against each other to create one "averaged" mask. Finally, the classification result from the first branch is used to discard all but the correct class mask.

Prior to Mask R-CNN, MNC [9] and FCIS [35] were considered state of the art for instance segmentation, winning the COCO 2015 and 2016 challenges, respectively. Mask R-CNN outperform both by a significant margin using all common evaluation metrics. Furthermore, Mask R-CNN is very streamlined compared to most alternatives.

Mask R-CNN is an interesting candidate for construction damage segmentation. First, as each image in the dataset contains information about the shape of every instance, the network is provided a lot more information to learn characteristic features. In a way, the dataset is increased from 608 examples (# images) to 4631 examples (# instances). Performance of Mask R-CNN may therefore exceed PSPNet by a great margin. On the other hand, Mask R-CNN also has to learn how to separate corrosion into individual instances. As will be discussed in Section 4.1, this was found difficult to even do manually, and hence the performance may also be significantly worse than for semantic segmentation algorithms.

Second, many of the use-cases outlined in Section 3.1 requires the inspection system to have a concept of instances, e.g. to ensure a damage is only counted once when estimating area. In other words, instance segmentation is likely needed to reach the full potential of image segmentation for the purposes of automating industrial inspections.

Chapter 4

Methods and Implementation

This chapter details methods and implementation used in this master project. Section 4.1 presents how a dataset for image segmentation was collected, annotated and structured. Also given is a statistical analysis of the dataset. Section 4.2 then explains the methods of data augmentation used to compensate for the relatively small dataset size. Section 4.3 reviews commonly used evaluation metrics for image segmentation and gives reasons for what was used in this project. This is followed by an outline of the experiments conducted in Section 4.4, whose results are presented in Chapter 5. Finally, in Section 4.5, implementation details for each tested network is given.

4.1 Dataset

A total of 608 images were used in this project. This section explains how the images were collected, annotated and structured.

4.1.1 Data Acquisition

Holm [23] was provided $240\ 000$ images from The Norwegian Public Roads Administration for his master thesis work on image classification using machine learning. The images were taken during real inspections and contained a wide variety of different scenes. Holm manually evaluated and categorized 7523 of these images into five folders; *paint flaking*, *white corrosion*, *red corrosion on steel constructions*, *corrosion on reinforcement bars* and *no corrosion*. Due to very few images found for the class *white corrosion*, it was discarded and only the four remaining classes were used for training. This dataset is the starting point for this master project as well but cannot be used directly as it does not contain any segmentation masks.

4.1.2 Annotation Software and Number of Classes

To produce segmentation masks for a set of images, dedicated annotation software is highly recommended. Labelbox [50] offers a complete solution for image annotation and

was chosen for a number of reasons: Up to 2500 images it is free to use, has rich annotation tools and is web based with all data stored in the cloud, allowing working from any computer at any time. Additionally, multiple users can work on the same dataset (feature not used) and it provides statistics of the annotation process, such as total time spent.

Ideally, we would want detailed segmentation masks for all objects in the images, e.g. corrosion, reinforcement bars, metal constructions, concrete, cracks in concrete, and so forth. Annotating a small portion of images with this level of detail was quickly revealed too time consuming: First, metal constructions tend to have very complex shapes (wires, beams, bolts, fences, etc.) tedious to outline. Second, as metal and concrete are often painted, the surfaces may look very similar. Consequently, it was very difficult to annotate all areas in the images with certainty. If even a human annotator finds it difficult, it for sure will be difficult to teach a neural network the difference.

Fortunately, general corrosion on steel has a characteristic red color and is easier to detect. Furthermore, corrosion is, by far, the most important type of damage to detect during inspections. It was therefore chosen to continue only with this one class. 2557 images were collected by Holm for this class and hence uploaded to Labelbox. The Labelbox annotation interface is shown in Figure 4.1.

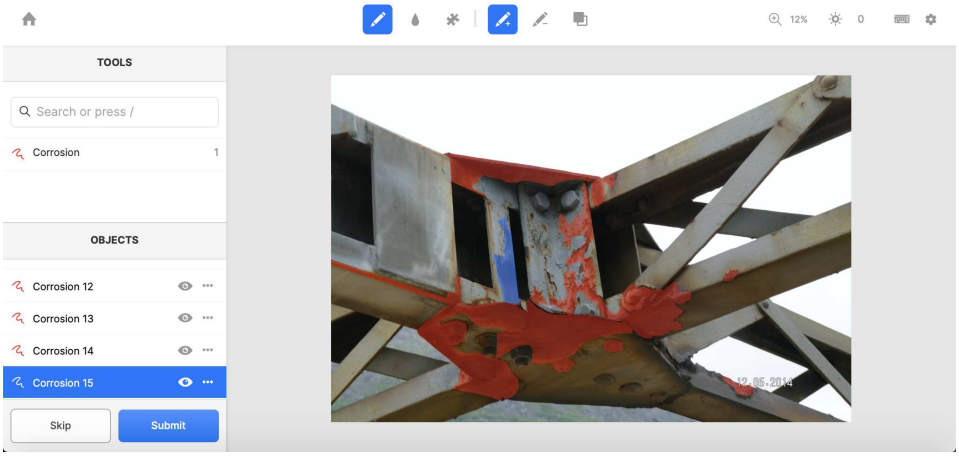


Figure 4.1: Interface of Labelbox [50] used for image annotation.

4.1.3 Image Annotation

With just one class the segmentation masks are binary; all pixels surrounded by a drawn polygon correspond to corrosion, whereas all remaining pixels are to be interpreted as a neutral class, i.e. background. For implementation reasons it is useful to also count the background class as a distinct class. It will in the following therefore be referred to two classes, but keep in mind we are just considering corrosion versus not corrosion.

For semantic segmentation, all corroded pixels can be segmented as one instance, i.e. as the union of a set of polygons. For instance segmentation, however, we need to divide the damages into multiple instances. In typical segmentation problems this is straight

forward. A person, for instance, is very well defined; the annotated instance should contain the apparent parts of a head, a body, two arms and two legs. Similarly, a ball should have a circular shape and a door is an (often skewed) rectangle with well-defined edges. Corrosion damages, on the other hand, is much more challenging to annotate.

Challenges Related to Annotating Corrosion Damages

First and foremost, it can be challenging to decide what is really corrosion and what is dirty brown/red-like surfaces. An example is shown in Figure 4.2a. Furthermore, corrosion has a tendency to smudge onto surrounding areas, for instance covering concrete with corrosion. The smudged areas are not damaged themselves but are still visually corroded. An example is shown in Figure 4.2b. Similarly, the boundaries between corroded and uncorroded areas are often smooth/gradually changing. An example is shown in Figure 4.2c. With no sharp edges, it is very difficult to decide where the annotation should start and end.

Next, corrosion damages are not "objects" as in the typical interpretation of the word. Consequently, there are few to none shape characteristics defining a corrosion damage. The damages can take on all sorts of shapes and sizes, making it difficult to distinguish individual instances. In fact, a general definition of an instance of corrosion is practically impossible to make. Figure 4.2g shows an example in which it is obvious to consider three distinct instances, whereas in the other examples there are many ways to separate instances.

Finally, a corroded area is often not completely and evenly corroded. It may contain scattered corrosion, as shown in Figure 4.2h, and creating detailed masks only containing true corrosion is tedious and impractical. Similarly, some images have areas heavily corroded, with lighter corrosion surrounding it. An example is shown in Figure 4.2i. This will either lead to instances with varying degree of intensity or instances surrounding other instances, depending on how annotation is performed.

Annotation Solutions Opted for

There is no one correct solution to the above issues. It is important, however, to be consistent and to choose segmentation masks useful for machine learning algorithms.

Dirty/brown/red surfaces were excluded as best as possible. Since missing a damage is worse than reporting a few excess damages, a fairly liberal approach was chosen in which cases of doubt were annotated as corrosion. For the same reason, mask boundaries were drawn to contain as much corrosion as possible. This was also chosen to ensure light corrosion was detected.

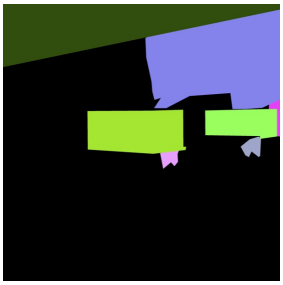
Areas with smudged corrosion were annotated as corrosion for two reasons: First, it was difficult to determine where the damage ends, and the smudge begins. Second, as smudged areas are visually similar to true corrosion, it would likely be confusing for machine learning algorithms to learn the difference.

Separating corroded areas into instances was perhaps the most challenging task. Due to the wide range of different shapes and sizes, it was difficult to be consistent. In fact, consistency is not well defined in this case. Corroded areas with lots of background in

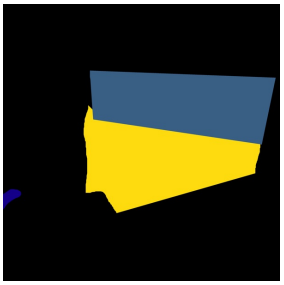
(a) Red/Brown Surfaces.

(b) Smudged Corrosion.

(c) Smooth Boundaries.



(d) Corresponding Mask to (a).



(e) Corresponding Mask to (b).

(f) Corresponding Mask to (c).

(g) Well-Defined instances.

(h) Scattered Corrosion and Undefined Number of Instances.

(i) Heavy Corrosion Surrounded by Light Corrosion.

(j) Corresponding Mask to (g).

(k) Corresponding Mask to (h).

(l) Corresponding Mask to (i).

Figure 4.2: A selection of different types of images annotated and corresponding instance segmentation masks. Colors are randomly assigned to instances for visibility, and have no further interpretation.

between were separated as distinct instances. Corroded areas were also considered separate instances if their sizes were similar and boundaries sharp. Continuous corroded areas were considered one instance, even if the resulting shape was very wide or tall (e.g. metal beams), unless it was evident from the image that the damage is spread over different objects (e.g. across two neighboring beams).

Regarding scattered corrosion, small uncorroded areas within larger-region segmentation masks were accepted. This was necessary to maintain a reasonable annotation time and was considered not harming training of machine learning models. Very small corrosion dots not part of a cluster or scattered region were ignored for the same reason and to reduce training time for instance segmentation.

Finally, heavy corrosion was annotated as one instance, with surrounding light corrosion as a separate instance, but only when the difference/boundary was strong/obvious. This decision was made as it was considered the best for machine learning algorithms (e.g. edges are very important features).

Figure 4.2 shows examples of the above-mentioned challenges with corresponding segmentation masks indicating the solutions chosen. Note that for semantic segmentation, all instance masks were merged to one, eliminating some of the aforementioned problems.

4.1.4 Quality Control

During annotation, produced segmentation masks may be slightly inaccurate, inconsistent or corroded areas missed. For better quality control, all images were therefore reviewed four weeks after the final image was annotated (12 weeks after the first image was annotated).

Roughly one third of the images were altered before approval. Although seemingly a lot, most changes just fixed inconsistency regarding what is considered worth annotating and not, usually in the direction of including more smudged corrosion. No major annotation mistakes were revealed in the process. Whether the quality control increased actual performance is uncertain as the non-reviewed dataset was never systematically tested. It is, however, reasonable to assume that (1) training becomes slightly easier with the updated dataset, and (2) that evaluation metrics yield more accurate scores due to greater consistency.

4.1.5 Downloading and Splitting Dataset

Labelbox stores all annotations as a json file, i.e. a structured text file, according to the format in the below (simplified) listing: The file is a list with structured information for each annotated image. The first entry, for each image, is a unique ID followed by a link to the original image. The field "Label" then contains a list named "objects" containing all instances for that image. For each instance, a unique ID, its class value (in our case always "corrosion") and a URL to the instance mask is provided.

```
[{
"ID": "label_id",
"Labeled Data": "url to original image",
"Label": {
```

```
    "objects": [{"featureId": "ID1",
                 "instanceURI": "url to instance mask 1"
                 "value": "class for instance 1"}
                {"featureId": "ID2",
                 "instanceURI": "url to instance mask 2"
                 "value": "class for instance 2"}],
        }
}]
```

A python script was used to download the annotated dataset using the json file, see Appendix B. A folder was created for each image, with names corresponding to the IDs in the json file. For instance segmentation, all instance masks were saved as separate PNG files in a sub-folder below the original image. For semantic segmentation, all instance masks for an image were combined and saved as one image.

Dividing the dataset into training, validation and test sets was done using command line operations. 50 images were randomly selected for testing, 50 images for validation and the remaining 508 images were used for training. The entire pipeline from data acquisition to a ready-to-use dataset is illustrated in Figure 4.3.

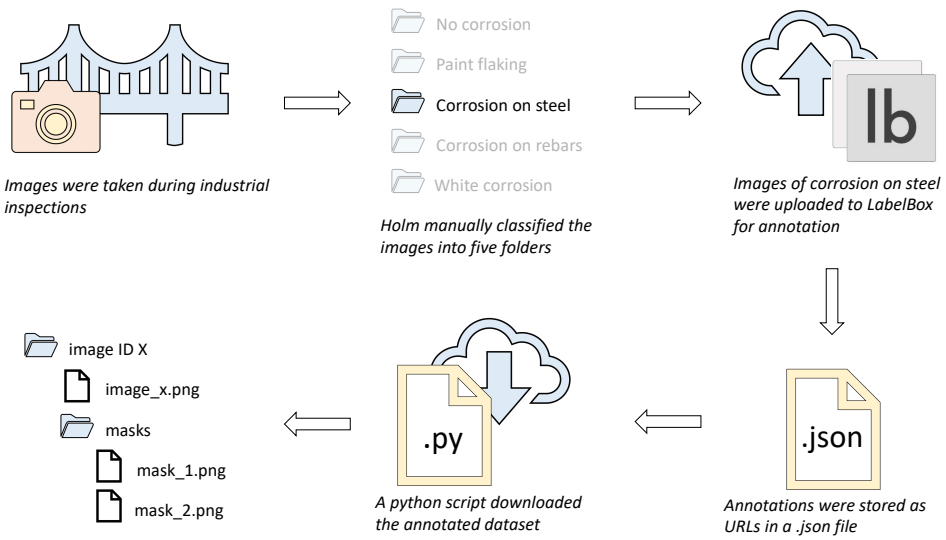


Figure 4.3: Pipeline from data acquisition to dataset with annotated images ready for segmentation algorithms. The resulting file structure shown is for instance segmentation. For semantic segmentation, the python script merged all masks to one PNG file.

4.1.6 Dataset Summary and Analysis

The final dataset consists of 608 images with a total of 4631 instances. The entire dataset took 43.3 hours to construct (effective time, excluding breaks), averaging to 4.3 minutes

per image. An additional 8 hours were spent during the quality control review. Table 4.1 summarizes statistics of the annotated dataset.

Table 4.1: Metadata statistics of the annotated dataset.

# Images	# Instances	Corroded Pixels	Avg. Image Size	Avg. Aspect Ratio
608	4631	$19.1\,\%$	4.46 MP	$1.32 \approx 4:3$

Analyzing the distribution of percentage corroded area in the images yields the histogram shown in Figure 4.4. On average $19.1\,\%$ of the pixels in the dataset were annotated as corroded. This imbalance allows a network to predict each and every pixel as background and still obtain an unreasonably high accuracy of $80.9\,\%$. Section 4.3 discusses other possible evaluation metrics accounting for this.

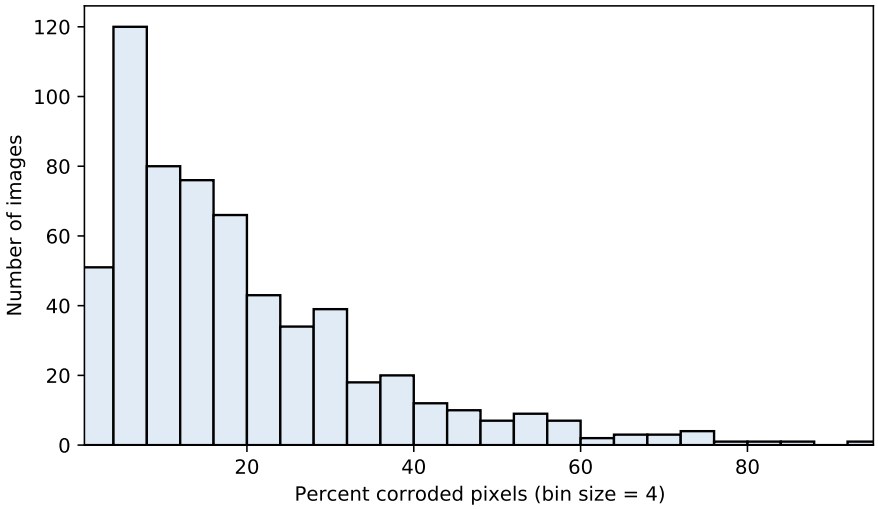


Figure 4.4: Distribution of percentage corroded pixels in an image in the dataset. On average, each image in the dataset has $19.1\,\%$ corroded pixels.

The distribution of the instance sizes, relative to the size of their images, is shown in Figure 4.5. The instances average to no more than $2.53\,\%$ relative the image frame (note the logarithmic y-axis), i.e. most instances are very small. The heavy concentration of small damages means instance segmentation algorithms might learn this as a characteristic feature for corrosion damages. That is, larger damages may perform worse than small damages. On the other hand, larger instances are easier to "see" due to larger number of features (edges, red/brown surface, etc.). Additionally, some small damages might implicitly get removed if images are scaled down during training. In conclusion, the size imbalance is not expected to cause any major problems. After all, Mask R-CNN uses a feature pyramid network for the very purpose of detecting instances of different scales.

608 images are rather few by modern machine learning standards. For instance, 5000 images were used by Zhao, Zhang, and Huang to detect moisture marks using Mask R-

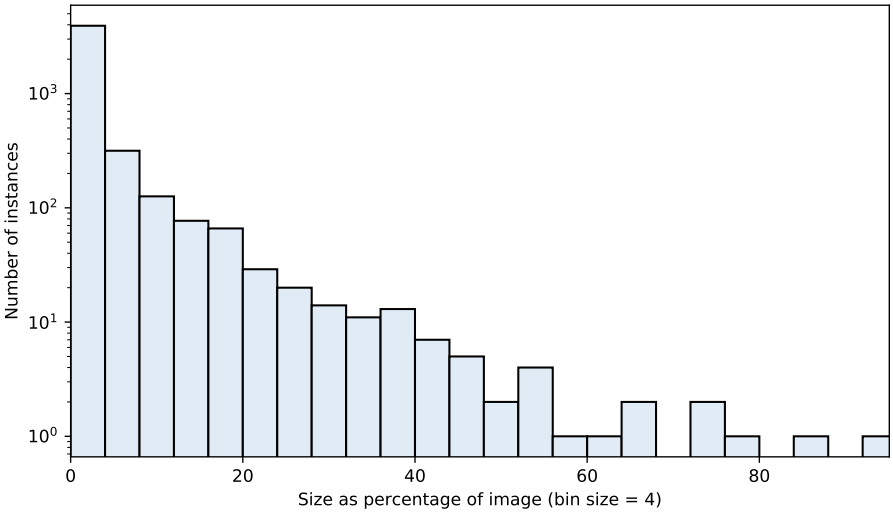


Figure 4.5: Distribution of instance sizes as percentage of the image size. On average, each instance in the dataset is $2.53\,\%$ of the size of its image. Please note the logarithmic y-axis.

CNN [59] and 9000 images were used in the 2018 IEEE road damage detection competition. Commonly used datasets with everyday objects, such as PASCAL VOC 2012 [12], Cityscapes [8], ADE20K [61] and COCO [37] contain up to $120\,000$ annotated images.

On the other hand, binary image segmentation should allow for much smaller datasets. ADE20K, for instance, contains 3196 classes which is a much more challenging task. Additionally, the corrosion dataset is highly augmentable, as will be discussed in Section 4.2. Extensive data augmentation can vastly and effectively increase the dataset size. 608 images might therefore be feasible.

As annotating images is very time consuming, it was there chosen to settle with a 608-image dataset and rather focus on good data augmentation methods. Two key points should be noted, however. First, if in further work the dataset is extended with more classes, more images are likely needed. Second, to avoid overfitting on the small dataset, small network models should be considered. See Section 4.5 for details.

4.2 Data Augmentation

Artificial data augmentation can increase the effective number of images used for training. Keras has support for some basic data augmentation out of the box, but a better option is the Python library imgaug [28]. It supports more than 100 different image transformations, each of which has one or more parameters to tune its mode and intensity. Additionally, imgaug has support for combining transformations in smart ways, e.g. sometimes performing horizontal flipping and always performing either random rotation or brightness adjustments or both.

4.2.1 Standard Method

Finding an optimal augmentation scheme is impractical as the search space is huge. A natural starting point is researching what others have done with success. However, little research is found on image augmentation for the type of images used in this project. Most research use very restrictive data augmentation, and the actual performance gain is very rarely reported. The most commonly used augmentations on datasets somewhat similar to the one used in this project is horizontal flipping [59, 5, 53] and brightness, contrast and grey scale adjustments [56, 3]. Furthermore, most experiments in the literature (not limited to damage-like datasets), use only a one-stage data augmentation method, i.e. constant data augmentation performed to all images for the entirety of the training process.

Only performing flipping and slight color adjustments, as described above, is very conservative, and the corrosion dataset may benefit from heavier augmentation approaches. Finding the right balance between heavy augmentations and light augmentations, however, is difficult. Too strong augmentations can "break" the images, whereas too weak augmentations may have no effect and can increase the chances of overfitting. Extensive and time-consuming experiments, with lots of hyperparameter tuning, are needed to find the one augmentation process performing best. Additionally, different network models may have different optimal augmentation configurations, meaning the search process must be performed individually for each network.

4.2.2 Proposed Method

A different solution is therefore proposed in this thesis: First train the neural network with heavy augmentations, followed by fine-tuning with little or no augmentation. The hypothesis is that a two-stage data augmentation scheme will reduce overfitting and increase performance. Similar methods may have been adapted before, but no references are found in the literature.

The heavy augmentations, although perhaps breaking some of the images, should produce a useful starting point for the weights of neural networks. In fact, training on artificially augmented images, followed by fine-tuning without augmentations, can be viewed as a method of transfer learning. This comparison was very briefly drawn by Mikołajczyk and Grochowski [41].

An objective with a heavy/light two-stage augmentation scheme is for it to be general and applicable to any highly augmentable datasets. This would remove the need for individually tuning the augmentation configurations for different networks and datasets. In this thesis, only different networks are tested. Investigating how the scheme generalizes to other datasets is suggested as further work.

Heavy data augmentation will not replace standard transfer learning. Rather it is an intermediate step between general images and corrosion images. The majority of training epochs (e.g. 30 – 100) should be performed with heavy augmentations as this dataset is more difficult. Training is then continued with little or no augmentation for relatively few epochs (e.g. 1 – 30).

There are a number of ways to construct a generic, heavy augmentation scheme. The transformations used in this project was based on examples and default values in the imgaug documentation [28]. The resulting full, detailed augmentation scheme used in this

thesis can be found in Appendix C. An overview of applied transformations is given below.

Geometric Transformations

Geometric transformations are applied identically to both the image and the corresponding segmentation mask. The following geometric transformations were used in the heavy augmentation scheme.

- **Horizontal/Vertical Flipping:** As corrosion damages have no directional orientation, we can randomly flip images both horizontally and vertically. This was applied to each image $50\,\%$ of the time.
- **Rotation:** Correspondingly, damages can be rotated. With random flipping both horizontally and vertically, we need only randomly rotate images $\pm 45°$ to effectively obtain full rotational freedom. Rotation was applied $50\,\%$ of the time.
- **Scale:** Damages can be of any size. To increase scale invariance, images were randomly scaled to $80\,\%$–$120\,\%$ of their original size $50\,\%$ of the time. Scaling was applied individually per axis (horizontally and vertically).
- **Translation:** Translating a damage, either horizontally, vertically or both, can increase positional invariance. Images were therefore randomly translated $\pm 20\,\%$ relative to the height/width of the image, individually per axis. This, too, was performed $50\,\%$ of the time for each image.
- **Shear Mapping:** Stretching rectangular images to a parallelogram alters the shape of the damage and can increase robustness of the model. Shearing images randomly in the range $\pm 16°$ was applied $50\,\%$ of the time.

Some of the above transformations generate new pixels, e.g. to fill the empty space in corners when rotating an image. New pixels were assigned a solid random color.

Non-Geometric Transformations

Non-geometric transformations alter the pixel values and can increase robustness to different light conditions, contrast levels, color variations, etc. Such augmentations are only applied to the images and not the segmentation masks. The following non-geometric transformations were used in the heavy augmentation scheme. In the following, "sometimes" means that a random selection of up to 5 of the below listed transformations were applied to each image.

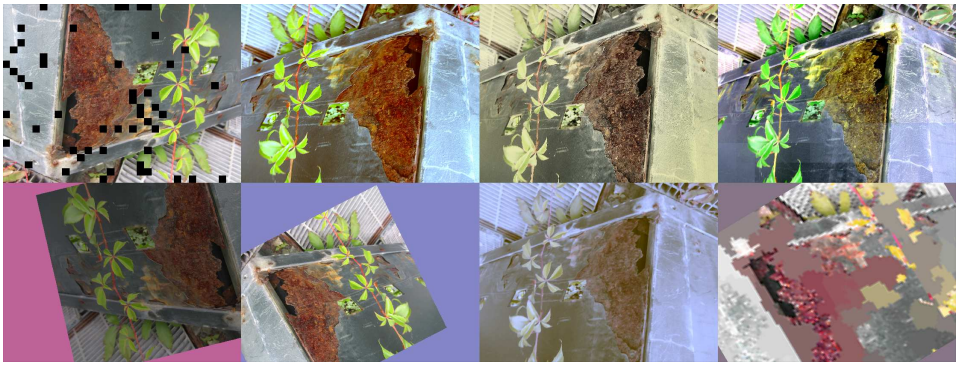
- **Brightness:** As images during inspection can be taken under vastly different lighting conditions, it is beneficial to alter the brightness of images. This was obtained by sometimes multiplying either all pixel values or random subareas by a value in the range 0.5–1.5.
- **Add:** Somewhat similar to brightness, sometimes a random value in the range -10 to 10 was added to all pixels. Half the time, the value was sampled individually per channel (RGB).

- **Contrast:** For similar reasons, adjusting the contrast in images is beneficial. Linear contrast was therefore sometimes used, half the time individually per channel.
- **Hue and Saturation:** Transforming images from the RGB color space to HSV (hue, saturation, value) allows for easy change of color (hue) and its intensity (saturation). Adjusting these values and transforming back to RGB can increase robustness to color variations. Changes of $\pm 20\,\%$ were sometimes applied.
- **Grayscale:** Similar to hue/saturation, further robustness to colors can be achieved by sometimes randomly converting images partly to grayscale.
- **Blur:** To simulate areas out of camera focus, blur can be added. Sometimes either Gaussian, average or median blur was applied with moderate, random intensity.
- **Invert:** Sometimes image channels were inverted with a probability of $5\,\%$ (sampled individually per channel), i.e. pixel value v was changed to $255 - v$.
- **Gaussian Noise:** Robustness to noise can be achieved by sometimes randomly adding noise. Half the time, the noise was sampled individually per channel and pixel, the other half of the time it was only sampled once per pixel.
- **Edge Detection:** Edges are highly relevant features for neural networks. Sometimes a method of edge detection was applied, highlighting the edges in images.
- **Sharpen:** Sometimes images were sharpened, i.e. enhancing the definition of edges. This is somewhat similar to edge detection above but is a more natural approach as the edges are not artificially highlighted.
- **Emboss:** Emboss is yet another method of enhancing the appearance of edges. This was sometimes applied.
- **Superpixel:** Sometimes converting images to their superpixel representation (i.e. grouping pixels based on similarity, effectively reducing the resolution of the color space) was applied $50\,\%$ of the time. This effectively creates a cartoon-like effect, increasing robustness to image resolution.
- **Dropout:** Sometimes adding artificially small black boxes may remove common features of a damage, forcing neural networks to also pay attention to less important features.

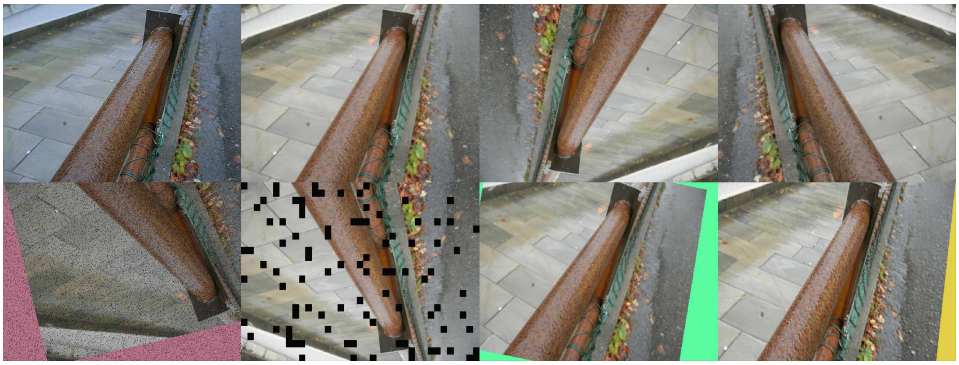
Figure 4.6 shows a few results of the heavy data augmentation scheme on three example images.

Tested Augmentation Schemes

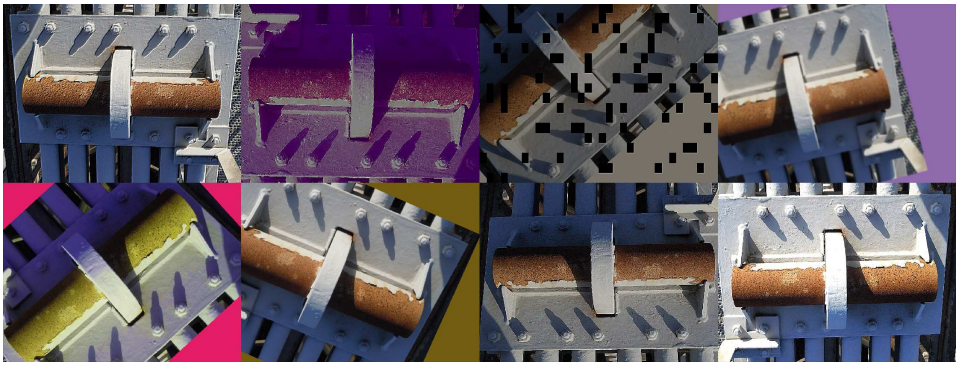
The strength of the heavy data augmentation is easily adjusted by two simple parameters; number of non-geometric transformations performed, and the probability of performing a geometric transformation. Adjusting individual transformation parameters/intensities is of course also possible but contradicts the purpose of this being a general scheme.



(a) Example 1.



(b) Example 2.



(c) Example 3.

Figure 4.6: Heavy data augmentation scheme on three example images. Many of these results are too heavily augmented for regular use in neural networks. However, training on such images lays a foundation for further fine-tuning with little or no data augmentation.

Chapter 5 shows results of using the above described heavy data augmentation followed by light augmentation in the form of horizontal and vertical flipping. A semi-heavy version in which the strength was reduced was also tested. Finally, training with flipping only and various augmentations considered realistic at low intensities, both for the entire training process, was tested. See Appendix C for details.

4.3 Evaluation Metrics

In order to evaluate the performance and usefulness of different methods, we need good evaluation metrics. A number of options are available for image segmentation. This section first discusses the importance of execution time and memory consumption. The metrics accuracy, recall, precision, specificity, F_β-score and IoU are then defined, and reasons for what was used in this project is given.

4.3.1 Execution Time

An essential part of evaluating the usefulness of a machine learning model is its run-time. For real life applications of neural networks, this translates to inference time, i.e. average or maximum time used to predict a segmentation mask for an image. A road line detection network for autonomous vehicles, for instance, is useless if inference time exceeds the time before an action is needed.

Inference time will vary depending on hardware, back-end and implementation, and is therefore irrelevant on its own. Given the specific configuration at which the inference time was measured, however, gives an idea of how it will run on different hardware. Furthermore, comparisons between models using the same configurations is a good indication of relative speed differences. Inference time is therefore reported for each model in Chapter 5.

Training neural networks is a one-time offline process, and corresponding training time is therefore usually less important. Furthermore, training times vary with dataset size and complexity of data augmentation methods used. Is therefore not used to *evaluate* any model, but still reported for reproducibility reasons and to guide fellow researchers on what to expect.

4.3.2 Memory Consumption

Similar to execution time, memory constraints can render a neural network model useless for a specific target hardware. However, actions can be taken to reduce the memory footprint with only minor loss in performance, e.g. decrease the batch size, use lower resolution images or train a smaller network model. Additionally, installing more memory is usually easier than to increase the computational capabilities of a system. Memory footprint is therefore arguably less constraining than inference time but should still be reported and taken into consideration when evaluating a model. In this regard, peak memory usage is the limiting factor and is reported in Chapter 5.

4.3.3 Prediction Performance

TP, TN, FP and FN

To evaluate the performance of predictions, we must first identify whether a prediction is correct or not. This is usually done in terms of true positives, true negatives, false positives and false negatives, defined as follows.

- **True positives (TP):** Total number of pixels predicted as positive (i.e. corrosion) that are, in fact, positive. These are correct predictions.
- **True negatives (TN):** Total number of pixels predicted as negative (i.e. background) that are, in fact, negative. These are correct predictions.
- **False positives (FP):** Total number of pixels predicted as positive that are, in fact, negative. These are incorrect predictions.
- **False negatives (FN):** Total number of pixels predicted as negative that are, in fact, positive. These are incorrect predictions.

Positive and negative here refers to whether a pixel belongs to a specific class (positive) or not (negative). See Figure 4.7 for a visualization of each term in relation to segmentation masks.

We can now define useful evaluation metrics for classification models. In the following, when referring to TP, TN, FP and FN, it is to be interpreted as the total number of such pixels in the entire dataset.

Accuracy

Accuracy is an intuitively simple metric measuring the proportion of pixels correctly classified,

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{\text{\# correct pixels}}{\text{\# total pixels}}.$$

Accuracy is a widely used metric. There is, however, a big drawback to accuracy for practical use. The corrosion dataset, for instance, has $19\,\%$ positive pixels and $81\,\%$ negative pixels. We can therefore easily obtain a seemingly good model with $81\,\%$ accuracy by predicting all pixels as negative. We should therefore only use accuracy as a definitive metric when the target classes are well balanced, thus rendering accuracy unavailing for the corrosion dataset.

Precision

Precision measures how many of the positively predicted pixels really are positive, relative to the total number of positive predictions,

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

As can be seen from the equation, maximizing precision corresponds to minimizing FPs. Precision is a good metric when it is important not to report too many FPs. An example

of such is medical treatment, which can be harmful if the patient really does not have the assumed disease. Security clearance approval is another example where high precision is needed. It is, however, trivial to obtain $100\,\%$ precision by simply predicting any input as a negative, e.g. decline any security clearance application or predict no pixel as corrosion. This is, of course, impractical.

Recall

Recall (also called sensitivity) measures the proportion of positive pixels that are predicted as positive pixels,

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

As can be seen from the equation, maximizing recall corresponds to minimizing FNs. Recall is a good metric when it is important not to miss any FNs, e.g. when diagnosing patients. However, similar (but opposite) to precision, a big drawback of recall is that $100\,\%$ recall is easily obtained by predicting any example or pixel as a positive.

Specificity

Specificity is the exact opposite of recall, measuring the proportion of truly negative pixels that are predicted as negative pixels,

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}.$$

Similar to precision, maximizing specificity corresponds to minimizing FPs. Specificity is a less used metric in the context of machine learning compared to precision and recall, partly because of the F_β-score discussed below. $100\,\%$ specificity is obtained by classifying any pixel as a negative pixel, thus too being a rather useless metric when optimized alone.

F_β-score

As seen for precision, recall and specificity above, we should not exclusively optimize either without taking other metrics into consideration. An attempt to solve this could be taking the arithmetic mean of precision and recall; $(\text{precision} + \text{recall})/2$. This, however, often gives unreasonably good scores to models with a big difference in precision and recall values. As an example, consider the case where a model is to predict the presence of corrosion in images, using a dataset in which $2\,\%$ of the pixels really are corroded. If the model predicts all pixels as positives (i.e. as corrosion), precision and recall values are, respectively, $2\,\%$ and $100\,\%$, thus giving a mean value of $(2 + 100)/2\,\% = 51\,\%$.

A better solution, the F_1-score, is taking the harmonic mean of precision and recall,

$$\text{F}_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}}.$$

Applied to the previous example, we get $\text{F}_1 = 2 \cdot 2 \cdot 100/(2 + 100)\,\% = 3.92\,\%$, which is a much better evaluation of the actual performance. A criticism, however, of the F_1-score

is the fact that it weights precision and recall equally. In many practical scenarios one is more important than the other. A generalization in which recall is considered β times as important as precision therefore exists,

$$\mathrm{F}_\beta = \left(1 + \beta^2\right) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}.$$

The most commonly used F_β-scores are F_1 and F_2. In the field of image segmentation, the term *dice coefficient* is sometimes used instead of F_1.

IoU

Intersection over union, IoU, is a very popular metric for object detection and segmentation. It is particularly useful for instance segmentation as not all pixels contribute to every instance. As the name suggests, it is defined as the intersection between the ground truth mask and the predicted mask, divided by the union of the two masks. It can also be defined in terms of TP, FP and FN as

$$\mathrm{IoU} = \frac{\text{Intersection}}{\text{Union}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}.$$

See Figure 4.7 for a visual explanation.

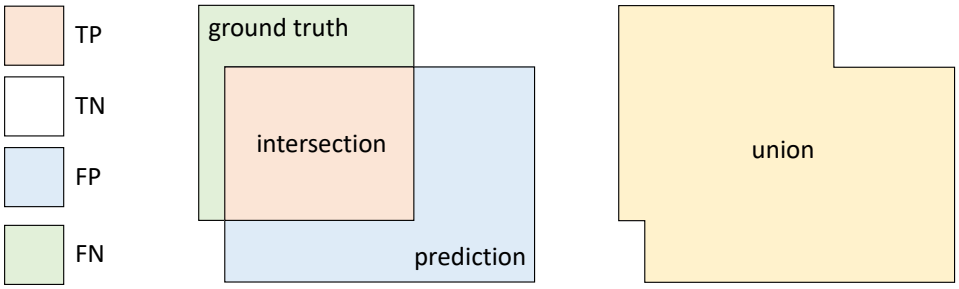


Figure 4.7: Visualization of TP, TN, FP, FN, intersection and union between a predicted segmentation mask and the corresponding ground truth.

Comparing the formulas for IoU and F_1, reveals a big similarity. In fact, their values are always within a factor of two of each other, $\mathrm{F}_1/2 \leq \mathrm{IoU} \leq \mathrm{F}_1$, and their ratio can be explicitly stated in terms of IoU,

$$\frac{IoU}{\mathrm{F}_1} = \frac{1}{2} + \frac{\mathrm{IoU}}{2}.$$

Additionally, for machine learning we can state something even stronger: IoU and F_1 are always positively correlated, meaning if model A is better than model B as evaluated by either metric, model A is also better than model B under the other metric. That is not to say, however, that the two metrics are equivalent in the sense that whichever to use is irrelevant. Their difference is apparent when evaluating *how much* better one model is compared to another. When averaging the scores over a dataset, IoU tend to penalize bad predictions more than F_1 (since the latter weights TPs more).

Mean and Frequency Weighted Metrics

The above discussion of evaluation metrics implicitly assumed a specific class was under consideration. All classes should be accounted for, however. In the case of the corrosion dataset, we need to calculate IoU for both corrosion predictions and background predictions. These scores can be reported individually, or we can take the mean, referred to as mean intersection over union (mIoU).

If, however, one or more classes have a relatively low number of ground truth pixels present in the dataset, mIoU can be somewhat misleading. Assume a model is to segment white squares with a black border, in addition to corrosion and background. This is a very simple task, likely yielding $100\,\%$ IoU for white box predictions. Even if such boxes were present in just one image, the score would be $100\,\%$ for that class. Effectively, this would increase the mIoU score by an unreasonable amount.

A solution is frequency IoU (fIoU), in which each class is weighted by their relative frequency in the dataset when computing the mean. In case of the corrosion dataset, the two classes are imbalanced ($19.1\,\%$ vs. $80.9\,\%$), suggesting fIoU is likely a better metric than mIoU.

Choice of Prediction Evaluation Metrics

In the literature, IoU, and in particular mIoU, is the most common metric for image segmentation. Class-wise IoU and mIoU will therefore be reported for each tested model. fIoU will also be reported. Since the dataset is imbalanced, fIoU will be emphasized.

When evaluating instance segmentation predictions, all instances were merged to one and evaluated against the correspondingly merged ground truth mask. This was done for two reasons. First, although Mask R-CNN predicts many reasonable instance masks, they do not necessarily match the way instances are separated in the ground truth masks. This result in unreasonably bad IoU scores. Second, merging instances makes performance of semantic and instance segmentation models more comparable as they are evaluated on the same terms. A consequence is that separation of instances is not fully assessed for Mask R-CNN.

Training, Validation and Test Sets

Models were trained on the training set and performance monitored on the validation set. The test set, however, was not used until all training configurations were decided, e.g. number of epochs, augmentation schemes, learning rate, etc. This is important since altering training parameters to increase validation performance essentially means we indirectly train on the validation images. Performance on the test set is therefore a better representation of true performance, and hence weighted the most in Chapter 5. Gaps between performance on training data and test data can then be used to see if models are overfitting.

4.4 Experiments

It is interesting to compare state-of-the-art methods for semantic segmentation and instance segmentation for the purposes of corrosion damage detection. From the literature

review, PSPNet was found a great candidate for semantic segmentation. It has superb performance without bells and whistles and is relatively easy to understand. Also, open source implementations are available. Benchmarking other reviewed network models for semantic segmentation is considered redundant as their performance is expected to be sub-par relative PSPNet. Mask R-CNN was chosen for much the same reasons as for PSPNet. Additionally, it is by far the most well-known instance segmentation algorithm.

The following experiments were conducted, whose results are presented and discussed in Chapter 5.

- **Transfer learning:** The advantages of transfer learning for the corrosion dataset was established by training PSPNet for 70 epochs both with and without transfer learning from ADE20K [61].

- **Common data augmentation:** Common data augmentation techniques were tested by training both PSPNet and Mask R-CNN for 70 epochs with flipping as the only data augmentation.

- **Advanced data augmentations:** The proposed two-stage augmentation scheme was evaluated by training both networks for 50 epochs with heavy data augmentation, followed by training for 20 epochs with only horizontal and vertical flipping as data augmentation. Also tested for PSPNet is the performance of a semi-heavy version of the same scheme and a composite augmentation scheme with "realistic" transformations only. The latter was applied constantly to all epochs. See Appendix C for details on each augmentation scheme.

- **Non-corroded images:** Finally, prediction masks for selected, difficult images not part of the corrosion dataset were analyzed. The images contain no corrosion but has areas with red/brown-like colors easily misinterpreted as corrosion.

4.5 Implementation

This section specifies the implementation of PSPNet and Mask R-CNN, as well as the system on which the experiments were conducted.

4.5.1 Implementation of PSPNet

The implementation of PSPNet is based on the GitHub repository by Divam Gupta [20]. Parts of the code base had to be updated to work properly with the imgaug library used for image augmentation. All relevant configuration parameters used are specified in Table 4.2. PSPNet-50 was chosen over PSPNet-101 as the corrosion dataset is small. ADE20K [61] was used for transfer learning as it is the most versatile and challenging dataset available for semantic segmentation. Default values for optimizer, image size and batch size were used.

Table 4.2: PSPNet Configuration.

Backbone	ResNet-50
Transfer learning dataset	ADE20K [61]
Optimizer	Adadelta (default, adaptive learning rate)
Image size	473×473 (default)
Batch size	2 (default)

4.5.2 Implementation of Mask R-CNN

The excellent implementation by Abdulla [1] available on GitHub was used for Mask R-CNN. The most relevant network parameters used are listed in Table 4.3. All other parameters use default values.

Table 4.3: Mask R-CNN Configuration.

Backbone	ResNet-50
Transfer learning dataset	COCO [37]
Optimizer	SGD with momentum (default)
Learning rate	0.001 (default)
Learning momentum	0.9 (default)
Image size	512×512
Batch size	1
Max instances per image	50
Detection confidence threshold	$90\,\%$

The 50-layer ResNet backbone was chosen for the same reason as for PSPNet. Additionally, Mask R-CNN has a vast memory consumption, meaning a decreased model size was beneficial. The COCO [37] dataset was used for transfer learning, as it is de facto standard for instance segmentation and the only option available. All default parameters regarding optimizer were used, as initial testing with more advanced optimizers such as Adam [29] was unsuccessful. The implementation requires the image size to be six times divisible by 2. The closest such number to 473 (the image size used for PSPNet) is 512 and therefore used for Mask R-CNN. A batch size of 1 was needed in order to not exceed the memory capacity of the GPU. Limiting the maximum number of instances to predict per image also decreased the memory footprint. It was opted for a maximum value 50, half of the default value 100. Finally, a confidence score of at least $90\,\%$ was chosen as the limit for accepting an instance as a positive RoI/prediction. This value was chosen relatively high since we have only two classes.

Compared to the original Mask R-CNN, which was implemented in Pytorch [13] rather than Tensorflow [18], the implementation by Abdulla has two main differences. First, bounding box ground truths are not explicitly provided, but determined on the go as the smallest box encapsulating the entire instance mask. Second, the Pytorch implementation used a learning rate of 0.02, which was found too large for Tensorflow as the weights tended to explode. A learning rate of 0.001 is therefore used instead. Additionally, for

this project, it was opted for a learning rate schedule halving the learning rate every 10th epoch to better ensure convergence of the weights. This was not needed for PSPNet as it uses an adaptive optimizer/learning rate.

4.5.3 System

All experiments were run on the hardware and software versions listed in Table 4.4 and Table 4.5, respectively. The newest version of Python, v. 3.8, was not compatible with Tensorflow at the beginning of this master thesis, and hence the older version 3.7 was used. Similarly, the Mask R-CNN implementation by Abdulla [1] does not support Tensorflow 2.0 (or above), and hence it was opted for version 1.14.0 instead. All packages, drivers and software were installed and managed in an Anaconda environment.

Table 4.4: Hardware Configuration.

CPU	Intel Core i7-6850K (6 cores @ 3.6 GHz)
RAM	Corsair 32 GB DDR4 @ 2666 MHz (2×16 GB, CL16)
GPU	NVIDIA Titan X (Pascal) (12 GB GDDR5X)
Storage (dataset)	Seagate Barracuda 1TB SATA HDD @ 7200 RPM

Table 4.5: Software Configurations.

OS	Microsoft Windows 10 Home Edition
Python	v. 3.7.6
Keras	v. 2.3.1
Tensorflow	tensorflow-gpu v. 1.14.0
CUDA	v. 10.0
CuDNN	v. 7.6.5

Chapter 5

Results and Discussion

This chapter presents and discusses results from conducted experiments. PSPNet is first reviewed in Section 5.1, followed by Mask R-CNN in Section 5.2. Readers only interested in the main results may skip to the summary in Section 5.3.

5.1 Results for Semantic Segmentation using PSPNet

This section evaluates PSPNet on the corrosion dataset with and without transfer learning, and with various data augmentation schemes. Predicted segmentation masks are visualized and compared in Section 5.3.

5.1.1 Transfer Learning

PSPNet was trained both with and without transfer learning from ADE20K [61]. Resulting test frequency IoU values are compared in Figure 5.1. Final class-wise IoU, mean IoU and frequency IoU without and with transfer learning are listed in Table 5.1 and Table 5.2, respectively.

Table 5.1: IoU for PSPNet trained without transfer learning.

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	94.2 %	76.8 %	85.5 %	90.9 %
Validation	86.8 %	61.2 %	74.0 %	81.2 %
Test	83.0 %	48.9 %	65.9 %	77.1 %

Discussion

Even though ADE20K is very dissimilar to the corrosion dataset, it is reasonable to expect performance gain from transfer learning. This has to do with basic features, such as edges,

Table 5.2: IoU for PSPNet trained with transfer learning from ADE20K [61].

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	96.8 %	86.7 %	91.8 %	94.9 %
Validation	90.2 %	65.1 %	77.7 %	84.7 %
Test	92.3 %	65.0 %	78.6 %	87.5 %

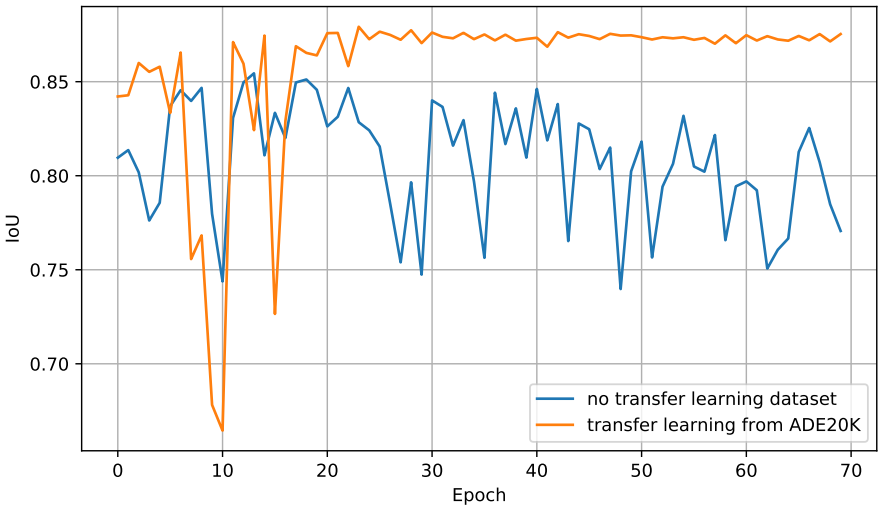


Figure 5.1: Frequency IoU on test set for PSPNet trained with no transfer learning vs. trained with transfer learning from ADE20K.

being similar across all types of images. For similar reasons, training should also converge faster.

Both of these advantages are observed for PSPNet. First, final performance is improved by a significant margin, both on validation and test data. Second, whereas transfer learning from ADE20K results in a flat curve after roughly 20 epochs, training without transfer learning never really converges within the 70 epochs. Lack of convergence is also evident in Table 5.1 revealing a big discrepancy between performance on validation and test data.

Without transfer learning, weights may at one point be decent by chance, but in the next epoch be really bad since the weights are not stable. The weight from ADE20K, however, are already in local minima to begin with, and are therefore more stable. We may conclude that transfer learning from ADE20K unquestionably increases performance. Transfer learning is therefore used in all following experiments.

5.1.2 Random Flipping Data Augmentation

PSPNet was trained both using horizontal flipping only, and using both horizontal and vertical flipping. Resulting test frequency IoU values are plotted in Figure 5.2. Comparison

between performance on training, validation and test sets (using flipping both horizontally and vertically) are plotted in Figure 5.3. Final class-wise IoU, mean IoU and frequency IoU are listed in Table 5.3 and Table 5.4.

Table 5.3: IoU for PSPNet trained with horizontal flipping as data augmentation.

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	96.7 %	85.9 %	91.3 %	94.6 %
Validation	90.6 %	66.6 %	78.6 %	85.3 %
Test	91.7 %	62.5 %	77.1 %	86.7 %

Table 5.4: IoU for PSPNet trained with horizontal and vertical flipping as data augmentation.

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	95.3 %	80.1 %	87.7 %	92.4 %
Validation	90.9 %	68.0 %	79.4 %	85.9 %
Test	92.2 %	64.7 %	78.5 %	87.5 %

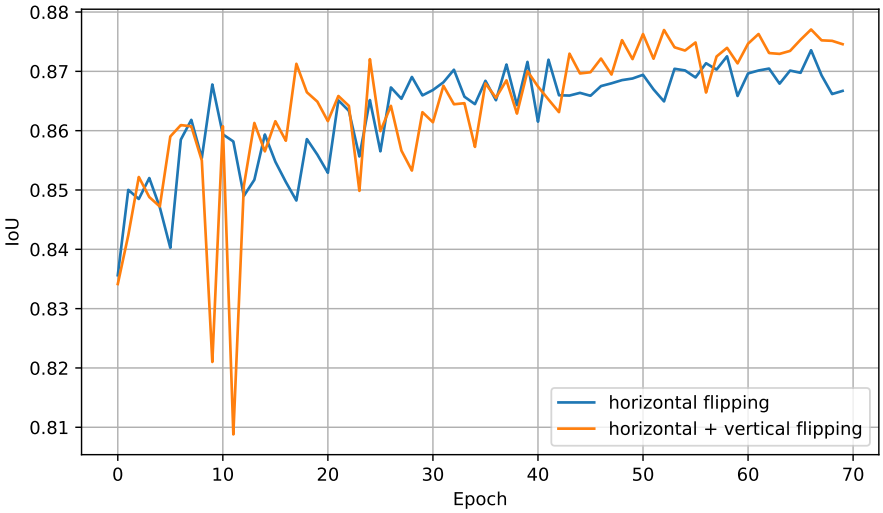


Figure 5.2: Frequency IoU on test set for PSPNet trained with horizontal flipping only as data augmentation vs. both horizontal and vertical flipping as data augmentation.

Discussion

Image flipping is the most commonly used type of data augmentation. Most datasets can be flipped horizontally without altering the essence of the images or segmentation masks.

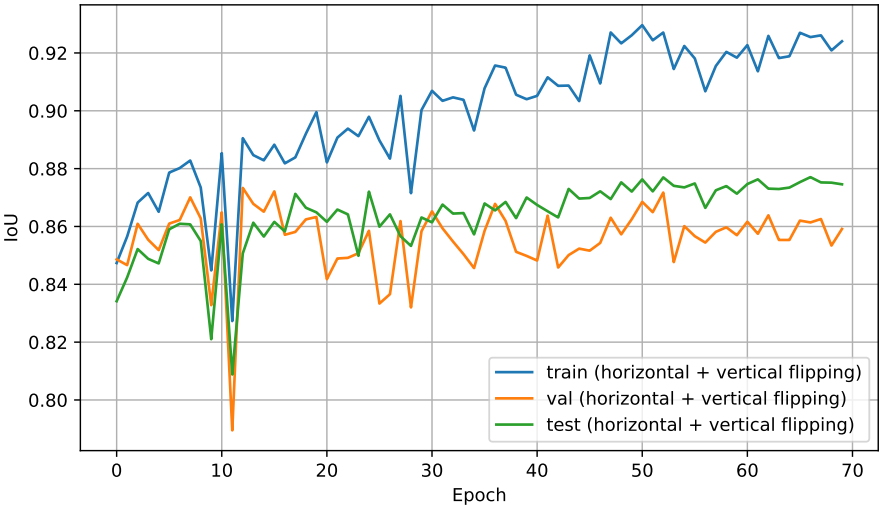


Figure 5.3: Frequency IoU on training, validation and test set for PSPNet trained with horizontal and vertical flipping as data augmentation.

Vertical flipping is less used as most real-life objects are never present upside down in normal images. In our case, however, flipping should be applicable both horizontally and vertically due to the non-directional properties of corrosion damages.

A first observation from Figure 5.2 is that flipping both horizontally and vertically seems to have a minor positive effect compared to only flipping horizontally. However, compared to no augmentation in the previous section, no drastic performance gain is obtained with either flipping scheme. This is a somewhat surprising result as the dataset is increased up to four times its original size.

Lack of improved performance could be caused by three different reasons; (1) the data augmentation method essentially has no effect on images, (2) we are underfitting or (3) we are overfitting.

Flipping only produces images with new damage locations, i.e. the shape and size of damages themselves do not change. The effective dataset size is therefore nowhere near up to four times increased if the damages in the dataset are already well distributed location-wise. However, we should still see somewhat improved performance, rendering this cause unlikely.

Underfitting is also highly unlikely since the dataset used is rather small and PSPNet is designed for much bigger datasets with many more classes, i.e. PSPNet has more than enough capacity.

However, this discrepancy between small dataset size and large network capacity has great potential for overfitting. In fact, this is exactly what is observed in Figure 5.3 as well as in Table 5.3 and Table 5.4: Whereas performance on validation and test data stagnate already around epoch 15, performance on training data continuously improve for all 70 epochs. The resulting trained models have tremendous performance on training images and see no reason to do things differently as these are the only images used to give the models feedback on performance. The true performance on never-before-seen images is,

however, much worse.

The best way to handle overfitting is increasing the dataset. This is not possible within the time frame of this master thesis, and hence left as further work. A different solution is reducing the capacity of the trained model. However, this was already taken into consideration by using PSPNet-50 rather than the standard, larger PSPNet-101. Different models with even smaller capacity might be viable but is left as further work as well. A third option is artificially augmenting the dataset even more. This is tested next.

5.1.3 Composite Data Augmentation

PSPNet was trained using the heavy data augmentation scheme detailed in Section 4.2. Resulting test frequency IoU values are plotted in Figure 5.4, with corresponding final class-wise IoU, mean IoU and frequency IoU listed in Table 5.5.

Figure 5.5 further shows a comparison of a wide range of different augmentation schemes. "Semi heavy" augmentation refers to the same augmentation scheme as heavy, but with reduced probability of applying each transformation. "Light/real" augmentations refer to a combination of affine geometric transformations and light non-geometric color changes. See Appendix C for further details of each scheme.

Table 5.5: IoU for PSPNet trained for 50 epochs with heavy data augmentation followed by 20 epochs with light data augmentation (horizontal and vertical flipping).

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	93.8 %	75.6 %	84.7 %	90.3 %
Validation	89.7 %	66.7 %	78.2 %	84.7 %
Test	91.3 %	63.7 %	77.5 %	86.6 %

Discussion

As was discussed in Section 4.2, the heavy data augmentation scheme alters images a lot, and maybe so much so that the contents of some images are broken. This could result in relatively bad classification performance in early epochs and longer convergence times. This is not a problem, however, as the objective in this stage is to facilitate a better and more robust starting point for the weights of the network. Good IoU scores are then later pursued during the light augmentation part of the scheme. This should, hopefully, reduce overfitting.

Several interesting observations can be made from Figure 5.4. First, convergence is substantially delayed compared to no augmentation (Figure 5.1) and flipping (Figure 5.3). This is reasonable as the images are more difficult and varying.

Second, as expected, a performance gain is observed at the 50-epoch mark for both training and test data. At this point images are easier to segment. As the validation data only consists of 50 images, the random nature of machine learning might be the reason why performance on validation data does not experience the same performance boost after 50 epochs.

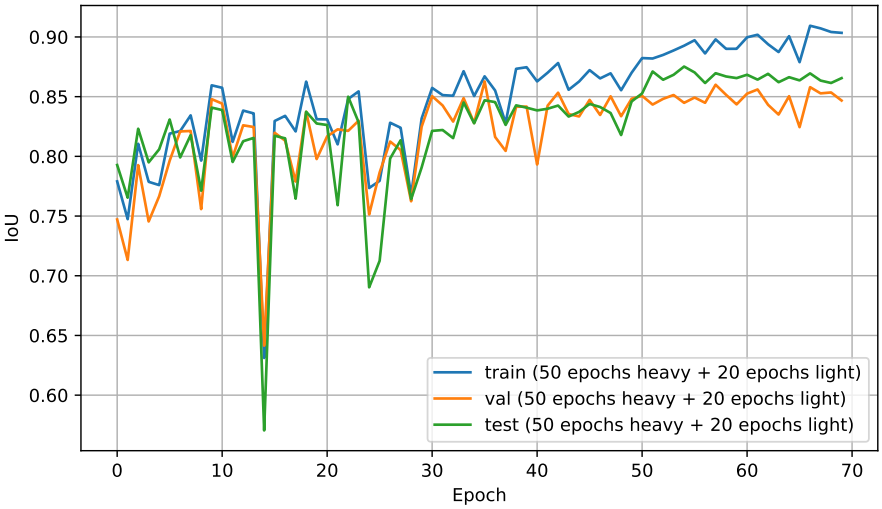


Figure 5.4: Frequency IoU on training, validation and test set for PSPNet trained with heavy data augmentation for 50 epochs followed by 20 epochs with light data augmentation (horizontal and vertical flipping).

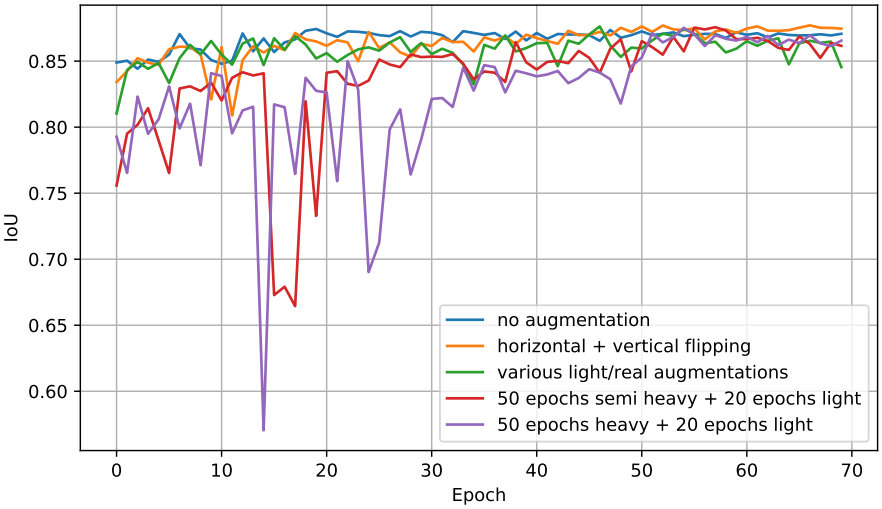


Figure 5.5: Frequency IoU on test set for PSPNet trained with various data augmentation schemes.

Third, and most interestingly, the discrepancy between performance on training and validation/test data is significantly reduced compared to previous results. In other words, the heavy augmentation scheme reduces overfitting. This is not to say, however, that actual segmentation performance on the current dataset is improved. Instead it means that if we are able to levitate performance on training data, we can expect test data performance to improve accordingly. Simply put, non-overfitted models are more reliable.

Figure 5.5 compares frequency IoU performance using a number of different aug-

mentation schemes. As shown, no augmentation scheme improves performance over no augmentation. As the dataset used is rather small, this is somewhat surprising. On the other hand, lack of performance gain using data augmentation coincide with multiple references in the literature. For instance, no gain using data augmentation was found by the authors of FCN. Additionally, none of the reviewed papers in Chapter 3 used extensive data augmentation beyond flipping, slight rotations and minor color/contrast adjustments.

We can conclude that data augmentation can be used to reduce overfitting on the corrosion dataset. For better performance on test data, however, we need a bigger dataset or an improved algorithm. Introducing more classes for different types of construction damages will likely better utilize the capacity of PSPNet. This is suggested as further work

5.1.4 Inference Time and Memory Footprint

Peak memory usage, average inference time per image and average training time per epoch for various data augmentation schemes are listed in Table 5.6.

Table 5.6: Peak memory usage [GB], inference frame rate [img/s] and average time per epoch [s] for training PSPNet on the corrosion dataset using various data augmentation schemes. Inference and training were performed on a Titan X (Pascal) GPU with 2 image batch size, see Section 4.5 for details.

Memory	Inference	No Augmentation	Flipping	Light/Real	Heavy (stage 1)
10 GB	3.7 img/s	106 s	107 s	410 s	411 s

Discussion

Training times of PSPNet on the corrosion dataset using various data augmentation schemes are reported for reproducibility reasons and to guide fellow researchers on what to expect. No further discussion of training times is considered relevant as this is a one-time offline process dependent on dataset size, data augmentation scheme, etc.

As for memory usage, 10 GB fits on most modern, high-end GPUs (Titan X (Pascal), for instance, has 12 GB video memory). Such graphics cards, however, are usually rather large and heavy, meaning a UAV might not be able to carry one. A solution could be remote computation at a central computer or using an even smaller network. The former solution will require superb band-with and response time of the wireless connection, whereas the latter might decrease performance.

Inference time (time to predict segmentation masks), although highly hardware-dependent, is of great importance. When averaged over the entire dataset, PSPNet predicts 3.7 segmentation masks per second on average. This is quite far from standard video frame rate of 24 images per second, even though inference is run on a high-end GPU. It should be noted, however, that PSPNet was never designed for real-time predictions. A larger frame rate is achievable using images of lower resolution, but likely at the cost of lower classification performance. Another solution is using networks specifically designed for real-time video segmentation. Related work has obtained more than 100 frames per second while maintaining a decent base level of accuracy [7].

5.2 Results for Instance Segmentation using Mask R-CNN

This section evaluates Mask R-CNN on the corrosion dataset using various data augmentation schemes. Results for Mask R-CNN follow the same structure as for PSPNet. Parts of the discussions in this section are therefore brief. Predicted segmentation masks are visualized and compared in Section 5.3. Transfer learning from MS COCO [37] was used in all experiments.

5.2.1 No Data Augmentation

Mask R-CNN was first trained without any data augmentation to define a performance base line. Resulting test frequency IoU values are plotted in Figure 5.6. Final class-wise IoU, mean IoU and frequency IoU are listed in Table 5.7.

Table 5.7: IoU for Mask R-CNN when trained without data augmentation.

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	94.7 %	78.9 %	86.8 %	91.7 %
Validation	88.0 %	57.1 %	72.5 %	81.3 %
Test	89.5 %	54.4 %	72.0 %	83.4 %

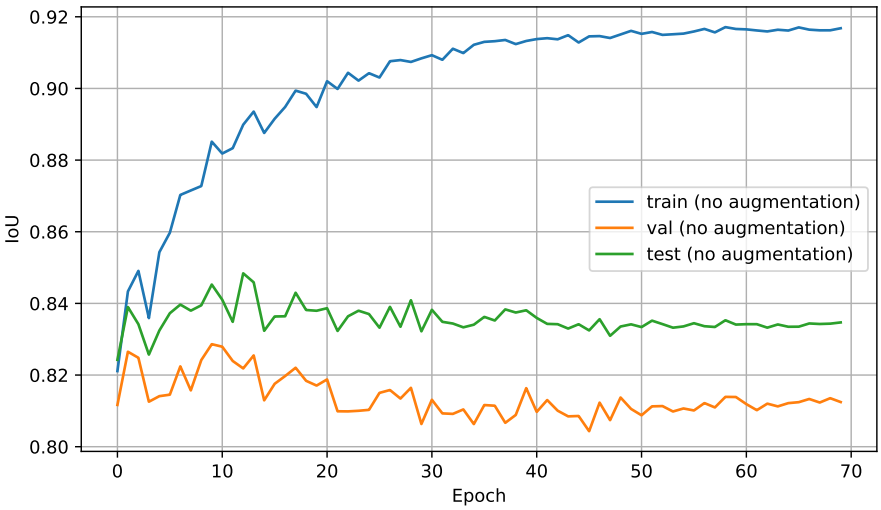


Figure 5.6: Frequency IoU on training, validation and test set for Mask R-CNN trained with no data augmentation.

Discussion

It is evident from Figure 5.6 that the network is severely overfitting. Whereas performance on training data continuously improves for the first 60 epochs before stagnating, perfor-

mance on validation and test data never really increase after the first few epochs. After 50 epochs, the weights in the network seem to have stabilized to a degree where validation and test data performance no longer oscillate between epochs. At this points, frequency IoU on the training set is approximately $13\,\%$ higher than for validation data, a huge gap indicating the network struggles to generalize.

The difference between frequency IoU on validation and test data in Figure 5.6 may be explained by the fact that more pixels contain background in the test set compared to the validation set, and that background IoU on the test set is higher than for the validation set. Additionally, small variations are to be expected when the dataset size, and particularly the size of the test and validation set, are small. The discrepancy should therefore not be emphasized.

As discussed for PSPNet, we are dependent on more images, better networks or better data augmentation to reduce overfitting and hopefully increase test performance. The next section therefore tests random flipping as data augmentation.

5.2.2 Random Flipping Data Augmentation

Mask R-CNN was trained with horizontal and vertical flipping as data augmentation. Resulting frequency IoU values are plotted in Figure 5.7. Final class-wise IoU, mean IoU and frequency IoU are listed in Table 5.8.

Table 5.8: IoU for Mask R-CNN when trained with horizontal and vertical flipping as data augmentation.

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	$92.7\,\%$	$70.4\,\%$	$81.5\,\%$	$88.5\,\%$
Validation	$87.6\,\%$	$58.6\,\%$	$73.1\,\%$	$81.3\,\%$
Test	$90.2\,\%$	$58.7\,\%$	$74.4\,\%$	$84.7\,\%$

Discussion

We see from Table 5.8 that using flipping as data augmentation decreases overfitting compared to no augmentation (Table 5.7). The difference between validation and training frequency IoU is reduced from 10.4 to 7.2, and corresponding numbers for the test set is a reduction from 8.3 to 3.8.

Similar to the previous section, when training without any data augmentation, there is a rather big discrepancy between validation and test performance. A similar explanation applies to Figure 5.7, but it is somewhat surprising that flipping does not seem to decrease the gap. In fact, the gap is enlarged by 1.3 percentage points. This could indicate that flipping enhances the networks certainty in cases where it is actually incorrect. On the other hand, as discussed earlier, varieties are to be expected when the validation and test set each contain only 50 images.

The main cause of the overfitting reduction is training IoU being lower. Essentially, the artificially increased dataset size makes it harder for Mask R-CNN to "remember" seen images, and hence the resulting performance is more reliable. What is particularly

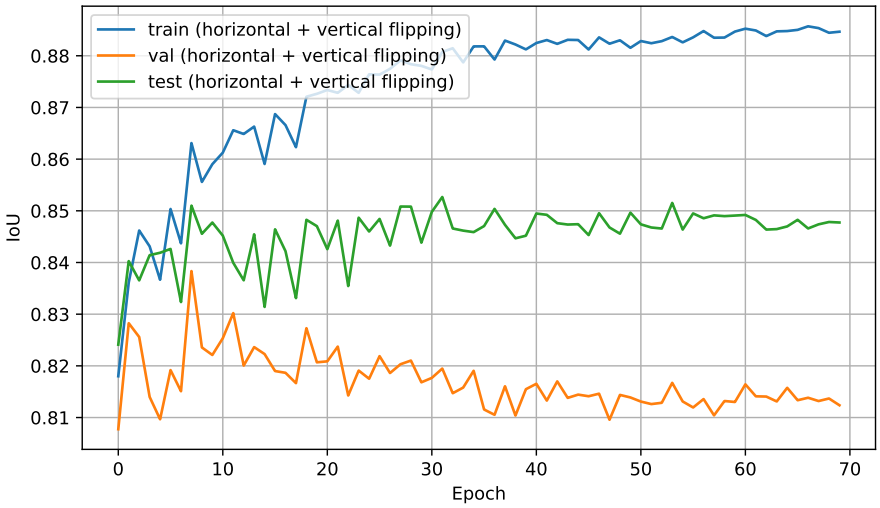


Figure 5.7: Frequency IoU on training, validation and test set for Mask R-CNN trained with horizontal and vertical flipping as data augmentation.

interesting, however, is that actual performance is also increased with flipping as data augmentation. That is, for PSPNet data augmentation was only found to reduce overfitting, but for Mask R-CNN both validation and test performance is increased using flipping. A possible explanation is that the real benefits of data augmentation are more prominent for larger datasets with more varying examples. In terms of number of images, the dataset used for Mask R-CNN is equally large as the dataset used for PSPNet, but in terms of training examples the number is vastly increased if we consider each instance a separate image.

The model still overfits more than desired, however. Further data augmentation is therefore tested in the next section.

5.2.3 Composite Data Augmentation

Mask R-CNN was trained with the heavy data augmentation scheme detailed in Section 4.2. Resulting frequency IoU values are plotted in Figure 5.8. Final class-wise IoU, mean IoU and frequency IoU are listed in Table 5.9.

Table 5.9: IoU for Mask R-CNN trained for 50 epochs with heavy data augmentation followed by 20 epochs with light data augmentation (horizontal and vertical flipping).

Dataset	Background IoU	Corrosion IoU	Mean IoU	Frequency IoU
Training	92.0 %	68.0 %	80.0 %	87.4 %
Validation	87.9 %	60.9 %	74.4 %	82.0 %
Test	89.9 %	56.5 %	73.2 %	84.1 %

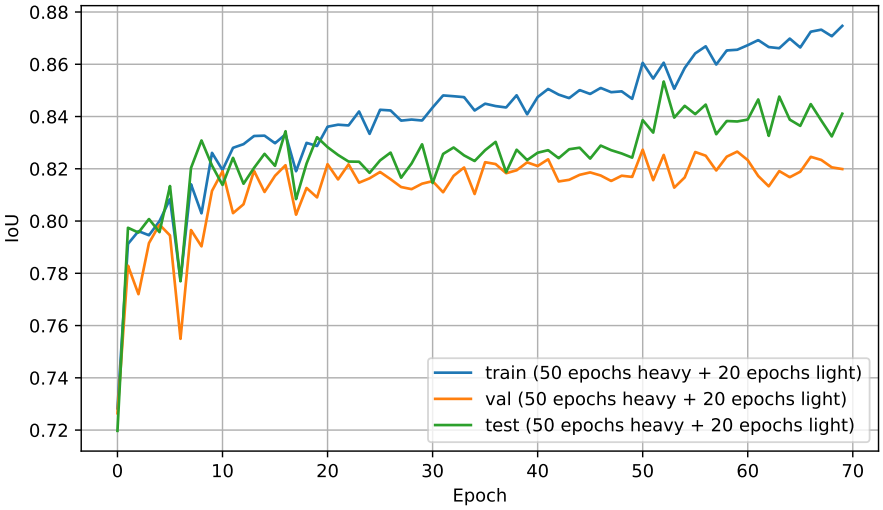


Figure 5.8: Frequency IoU on training, validation and test set for Mask R-CNN trained with heavy data augmentation for 50 epochs followed by 20 epochs with light data augmentation (horizontal and vertical flipping).

Discussion

Yet again the heavy data augmentation scheme is shown to reduce overfitting. The differences between training performance and validation/test performance in Figure 5.8 are significantly smaller compared to no augmentation (Figure 5.6) and random flipping (Figure 5.7). This corresponds well with results for PSPNet using the same data augmentation scheme. Furthermore, a performance boost similar to for PSPNet occurs at the 50 epochs mark on training and test data. Validation performance remains roughly constant even after switching to light augmentation, also corresponding well with the somewhat odd validation behavior discussed earlier.

Similar to flipping, although not as prominent, the heavy data augmentation scheme increases test performance compared to no augmentation. Combined with significantly less overfitting, it is easy to argue that the heavy data augmentation scheme is very useful for Mask R-CNN on the corrosion dataset.

It would be interesting to further investigate how the heavy data augmentation scheme would facilitate even larger or more complex dataset with more classes. Due to limited time of the master thesis, this is suggested as further work.

5.2.4 Inference Time and Memory Footprint

Peak memory usage, average inference time per image and average training time per epoch for various data augmentation schemes are listed in Table 5.10.

Table 5.10: Peak memory usage [GB], inference frame rate [img/s] and average time per epoch [s] for training Mask R-CNN on the corrosion dataset using various data augmentation schemes. Inference and training were performed on a Titan X (Pascal) GPU with 1 image batch size, see Section 4.5 for details.

Memory	Inference	No Augmentation	Flipping	Heavy (stage 1)
10 GB	1.4 img/s	1750 s	1780 s	1720 s

Discussion

As seen in Table 5.10, Mask R-CNN is a computationally demanding network. After all, it consists of four sub-networks, each of which is relatively large. The main reason training Mask R-CNN is considerably slower than PSPNet, however, is that each instance is provided as a separate png file. That is, an image with spatial size $H \times W$ annotated with K instances constitutes a $H \times W \times K$ input matrix, i.e. K times as large as a corresponding image for semantic segmentation.

Since PSPNet is much more efficient than Mask R-CNN, yet still fairly slow, it is obvious that more research is needed to obtain good classification performance with reasonable inference speed.

5.3 Comparison and Summary of Results

5.3.1 Comparison of Predicted Segmentation Masks

IoU is a number representation of performance. More interesting, perhaps, is the actual predicted segmentation masks. Prediction masks for both Mask R-CNN and PSPNet when trained with the heavy data augmentation scheme are shown in Figure 5.9 along with corresponding raw images and ground truth masks. Furthermore, Figure 5.10 compares predicted segmentation masks for difficult images containing no corrosion.

Discussion

It is evident from Figure 5.9 that PSPNet is more aggressive than Mask R-CNN, i.e. it has greater recall. For PSPNet, any output feature map pixel greater than 0.5 is considered corrosion, whereas for Mask R-CNN a damaged area must first be considered a region of interest with sufficiently large probability. In this thesis, a confidence threshold of $90\,\%$ was chosen, meaning higher recall for corrosion is easily obtained for Mask R-CNN by reducing the required RoI acceptance probability. Doing so, however, might increase the number of false positives. Furthermore, accepting more RoIs can result in an increased number of instances rather than a more aggressive approach on existing instances. This is undesirable as Mask R-CNN already has a tendency to construct unnecessary many and complicated instances. The first three examples in Figure 5.9, for example, have embedded instances where the ground truth only has one instance.

As discussed in Chapter 4, defining and distinguishing corrosion instances is difficult. Still, Mask R-CNN seems to correspond well with the ground truth instances. Although

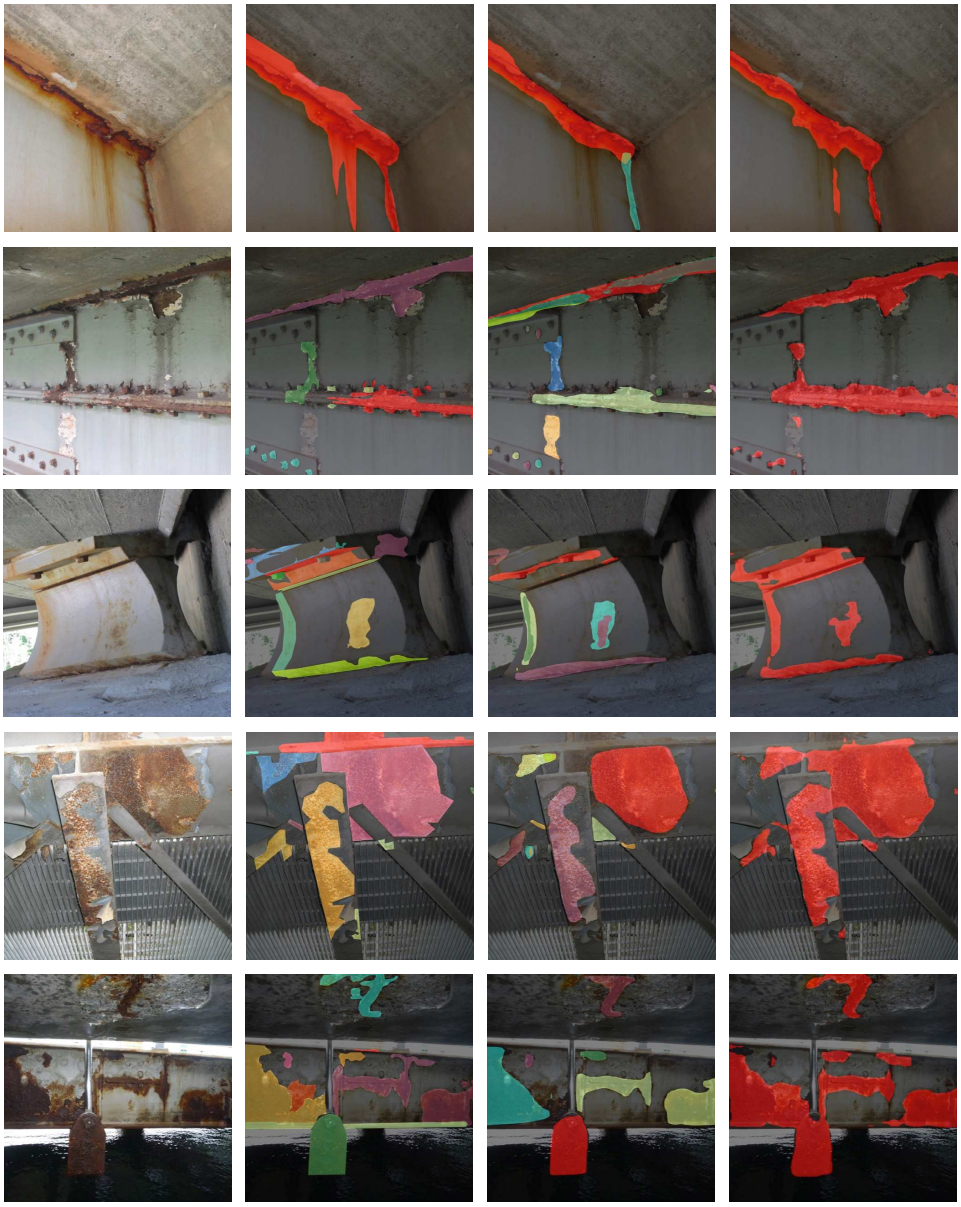


Figure 5.9: Predicted segmentation masks for Mask R-CNN and PSPNet, trained with the heavy data augmentation scheme, compared with corresponding raw image and ground truth. Instances are assigned random colors with no further interpretation.

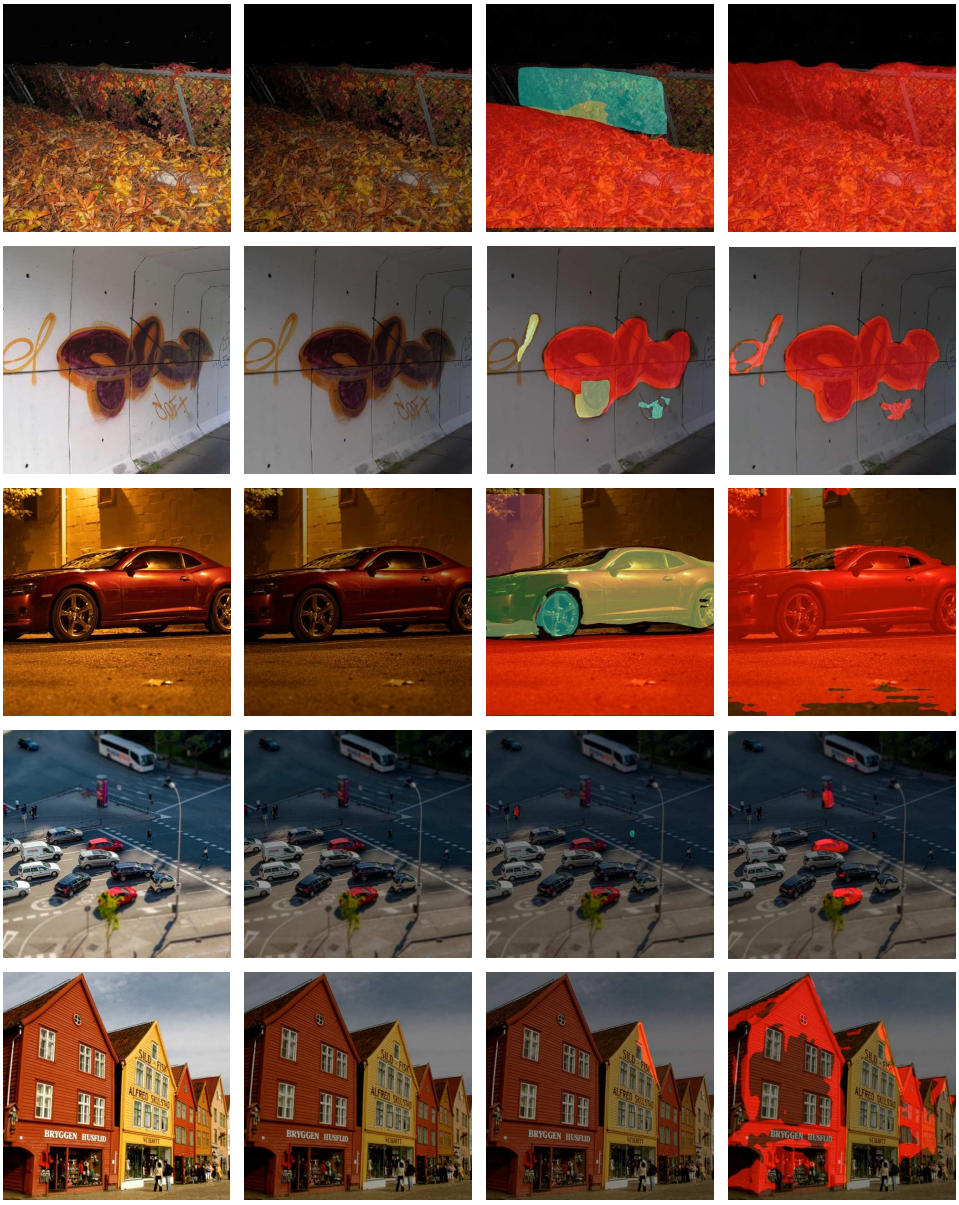


Figure 5.10: Predicted segmentation masks for Mask R-CNN and PSPNet, trained with the heavy data augmentation scheme, on images *not* containing any corrosion. Instances are assigned random colors with no further interpretation.

some instances by Mask R-CNN are divided into multiple small instances as seen in Figure 5.9, the algorithm very rarely merges multiple ground truth instances into one. It can be questioned, however, whether instance correspondences between ground truths and Mask R-CNN predictions are good enough for the use-cases discussed in Chapter 3. For general corrosion detection it is of little relevance. However, for damage area estimation and to know if a damage is severe or not, good correspondence could be of great importance. Similarly, if monitoring a damage over time is to be performed autonomously, good correspondence is needed.

Overall, the predicted segmentation masks seem promising. When predicted for non-corroded images, however, the situation changes vastly. Figure 5.10 compares predicted masks for hard images especially selected to contain red/brown colored areas. These images are not part of the corrosion dataset and were therefore neither used for training nor during calculation of IoU. It is immediately evident from Figure 5.10 that both Mask R-CNN and PSPNet suffer from false positives. Simply put, the algorithms predictions, and PSPNet in particular, are too heavily based on the color properties of corrosion. In terms of the previously discussed use-cases, the networks will consequently report too many damages, over-estimate areas and generally be too pessimistic regarding the integrity of a construction.

Two solutions are proposed. First, it is important to remember that the corrosion dataset contains no images similar to those shown in Figure 5.10. As every training example contains at least one corroded area, a predicted segmentation mask without any corroded pixels would in fact be peculiar. Train on images without corrosion in addition to the current corrosion dataset might therefore improve the prediction masks in Figure 5.10. This is not performed in this thesis since images containing no corrosion are not supported by the used implementation of Mask R-CNN. Incorporating background only images would require changes to the framework. Furthermore, background only images would introduce more imbalance to the dataset, enabling decent IoU by predicting all pixels as background.

A second solution is constructing a two-stage pipeline: First classify images in two categories, *corrosion* vs. *no corrosion*, and then apply image segmentation to all corroded images. This might be a viable approach as previous work has found such classification to perform very well on corrosion and background images [14]. Extending upon this idea, it should be possible to train the first sub-networks of Mask-RCNN to better learn the distinction between background only images and corrosion images using both types of images. We could then freeze early layers and continue training the latter sub-networks on corrosion images only. This would be beneficial as the classification and segmentation networks share computation. Again, this would require major changes to the Mask R-CNN framework. A similar strategy should be applicable to the backbone network of PSPNet as well.

5.3.2 Summary and Assessment of Image Segmentation for Industrial Inspections

Table 5.11 summarizes IoU for the most important results. It is evident from the last column that the heavy data augmentation scheme significantly reduces overfitting for both

PSPNet and Mask-RCNN. Additionally, Mask R-CNN obtains higher test performance using this scheme. It is therefore concluded that the heavy data augmentation scheme has great potential and should be investigated further.

Table 5.11: Summary of training and test frequency IoU for PSPNet and Mask R-CNN trained without data augmentation and with the heavy data augmentation scheme.

Network	Data Augmentation	Training fIoU	Test fIoU	Difference
PSPNet	None	94.9 %	87.5 %	7.4
PSPNet	Heavy + light	90.3 %	86.6 %	3.7
Mask R-CNN	None	91.7 %	83.4 %	8.3
Mask R-CNN	Heavy + light	87.4 %	84.1 %	3.3

Regarding general performance of image segmentation, and its applicability to industrial inspections, it is more complex to draw a conclusion. First, it is difficult to compare obatined IoU values with previous results on other datasets. PSPNet, for instance, obtains 85.4 % mIoU on Pascal VOC 2012 [12], but only 44.9 % mIoU on the more challenging dataset ADE20K [61]. Nevertheless, obtained IoU values and predicted masks are not good enough for an autonomous system to be relied on, especially in terms of false positives on non-corroded areas of red/brown-like colors. A larger dataset would of course likely increase performance, but it is difficult to estimate by how much. Training on background only images in addition to corrosion images might also be a viable option but requires modifications to the existing frameworks. Either way, only segmenting corrosion damages has limited potential compared to a model able to predict all sorts of construction damages as well as intact constructions. Constructing a larger and more complex dataset should therefore be a priority in further work.

Second, although PSPNet obtains slightly better IoU on the test set, Mask R-CNN, and instance segmentation in general, is considered a more versatile approach with more application areas. Additionally, the performance difference is likely caused by the more aggressive predictions produced by PSPNet, which can be achieved for Mask R-CNN as well by accepting more RoIs.

Regarding time and space complexity, both networks requires a high-end GPU with decent amount of video memory in order to run somewhat efficiently. The frame rate is still nowhere close to video standards of 24 img/s, achieving 3.7 img/s and 1.4 img/s for PSPNet and Mask R-CNN, respectively. Smaller, more efficient networks or better hardware are therefore needed before the full potential of image segmentation can be applied to industrial inspections.

Chapter 6

Conclusion and Further Work

6.1 Conclusion

This master thesis contains three main contributions. Each contribution is concluded below.

6.1.1 Dataset

No dataset of construction damages was available prior to this work. A 608-image dataset of annotated corrosion damages was therefore constructed. The images were taken during real inspections of steel bridges and comprise a wide range of different scenes. Due to it being a time-consuming task, only binary segmentation masks were created, indicating where in images corrosion is present. The segmentation masks consist of a total of 4631 instances, most of which are very small, averaging to $2.53\,\%$ relative the image frame. $19.1\,\%$ of the pixels in the dataset contain corrosion.

Annotating images was a challenging task. First, it can be difficult to know precisely where corrosion begins and ends, for instance due to corrosion smudging or dirty surfaces looking similar to corrosion. Second, unlike most everyday objects, there are few characteristics properly defining a corrosion instance. Consequently, there are many equally advantageous ways to divide a corroded construction into instances.

The corrosion dataset is a first step towards creating a big, general dataset with many different classes, such as crack in concrete, paint flaking and intact steel. The current version of the dataset, however, is rather small and best serves as a utility for proof of concepts. This could, and hopefully will, benefit other researchers and further work beyond the scope of this thesis.

6.1.2 Data Augmentation

Modern deep learning methods can obtain great results but are highly dependent on high-quality training data. 608 images are in this regard rather few, and data augmentation was therefore utilized to increase the effective dataset size. A variety of different data

augmentation schemes were tested, ranging from simple flipping to a sophisticated, two-stage scheme consisting of both heavy and light augmentations.

The results of this thesis find only minor performance gain using data augmentation. Overfitting, however, is significantly reduced for both PSPNet and Mask R-CNN, especially using the two-stage heavy data augmentation scheme. This means the trained networks become more reliable, and increased performance is hopefully more easily obtained with more annotated images.

The proposed heavy data augmentation scheme might be useful for other highly augmentable datasets as well. Evaluating the data augmentation scheme on other datasets is suggested as further work.

6.1.3 Assessment of Image Segmenting for Construction Damage Detection

State-of-the-art methods for semantic segmentation and instance segmentation were evaluated for the purpose of automating industrial inspections. Specifically, quality of predicted masks, memory consumption, inference speed and possible use-cases were discussed with regards to the neural networks PSPNet and Mask R-CNN.

It is concluded that image segmentation can aid automating industrial inspections of steel constructions in the future, and that instance segmentation algorithms are likely more useful than semantic segmentation algorithms, even though Mask R-CNN is outperformed by PSPNet in terms of IoU on the corrosion dataset. The obtained results in this thesis is, however, not good enough to construct a reliable, fully autonomous system as of yet.

Three major tasks remain. First and foremost, the dataset must be vastly enlarged. Increased recall for corrosion, fewer false positives and more classes are some areas in need of improvement and hopefully obtained with more training data. Second, more efficient networks and better hardware are needed to produce predictions at a faster frame rate. With the current implementation, image segmentation is only possible for post-analysis or real-time segmentation with lag. Finally, ways to estimate total damaged area, forecast damage development and classification of damage severity need more research in order to reach the full potential of image segmentation.

6.2 Further Work

Image segmentation of corrosion damages is by no means fully studied. The following are some suggestions for further work on this topic.

6.2.1 More Classes

Trained models in this project only has a concept of corrosion. However, there are many other damages of relevance in industrial inspections as well, such as paint flaking, crack in concrete and white corrosion (i.e. corroded zinc layers). Additionally, it is beneficial to detect different parts of a construction, such as intact steel, corrosion and wires.

Implementation-wise, minimal changes are needed to incorporate new classes. Furthermore, data acquisition is not needed as the original raw dataset provided by Norwe-

gian Public Roads Administration has thousands of unused images. The challenge lies in annotating more images. First, all current images must be reviewed and masks for the new classes must be added. Second, as the dataset becomes more complex and the current images are biased towards corrosion damages, more images must be annotated.

6.2.2 Test the Heavy Data Augmentation Scheme on Other Datasets

The heavy data augmentation scheme is shown to greatly reduce overfitting for both PSPNet and Mask R-CNN on the corrosion dataset. To investigate its general applicability, further research should be done to test it on other highly augmentable datasets as well. Examples of such datasets may for instance be found in the area of medical imaging.

6.2.3 Train using Adversarial Images

As red/brown colored areas within images containing no corrosion are easily predicted as corrosion, it would be interesting to provide the network models with increased number of such difficult examples. Note that the implementation used for Mask R-CNN does not support pure background images, but it should be possible to overcome this with some modifications to the framework used.

A generalization of the above idea is using *adversarial images*, as explained by Goodfellow, Shlens, and Szegedy [17] to increase performance and reduce overfitting for image classification. Adversarial images are "*formed by applying small but intentionally worst-case perturbations to examples from the dataset, such that the perturbed input results in the model outputting an incorrect answer with high confidence*" [17]. That is, the authors took regular images, computed in what direction each pixel value should be altered for the prediction to change the most[1], and then used this image for training. The original and modified image may look identical to the naked eye, but all small differences in total constitutes a wrong prediction and hence valuable training data for the network.

The authors used adversarial images for image classification. Further work can research if this is applicable to image segmentation of the corrosion dataset as well.

6.2.4 Efficient Inference on Mobile Processors

Both PSPNet and Mask R-CNN are dependent on high-end hardware to run somewhat efficiently. Full-sized GPUs are very expensive and might not fit on a UAV, however, meaning ways to obtain more lightweight predictions run on a mobile processor would be interesting further work.

6.2.5 Estimation of Damaged Surface Area

As discussed in Section 3.1, successful image segmentation of construction damages can enable estimation of damaged surface area using a UAV. This is useful as not all damages can be repaired immediately. Instead, maintenance is initiated when total damaged coating area exceeds a given threshold.

[1] The perturbations needed are computed as the gradient of the loss function with respect to the input, multiplied by a small scaling factor.

Lots of work still remains before this is a reality, however. First, the segmentation algorithm must accurately segment intact constructions as well as damages themselves. Second, SLAM or similar methods should be adapted for the UAV to be aware of its surroundings. Finally, a 3D model of the construction to be inspected should be incorporated in order to ensure the whole construction is inspected and to store exactly where an image was taken.

6.2.6 Classification of Construction Damage Severity

Some damages need immediate maintenance and we cannot wait for the total area of the construction to degrade before taking an action. Further work could therefore study how to also classify the severity of damages. One straight forward solution is considering different levels of severity as different classes, e.g. one class for light corrosion, one class for medium corrosion and one class for severe corrosion.

What makes up a severe damage, however, is not necessarily obvious from 2D image only. First, one part of a construction might be more important for the integrity of the construction than others. Second, a damage might be hardly visually present on the surface of a construction, yet still cause major problems to its integrity, e.g. corrosion pits discussed in Section 2.1. Using 3D images and other sensor data would therefore be interesting to incorporate into a machine learning model.

6.2.7 Forecasting Damage Development

An interesting topic of research is applying time series prediction methods to corrosion damages. The idea is that given a series of images taken over time of the same damage, an algorithm might be able to forecast how the damage will develop in the future. This would help inspectors better schedule when maintenance is needed, what damages to prioritize and estimate future costs related to maintenance.

It is uncertain whether currently available methods can forecast damage development sufficiently well purely based on time series images. Corrosion development models and domain knowledge might need to be incorporated in addition to a machine learning algorithm. Besides, a dataset must first be constructed, which will take time as damages must be monitored over time. For proof of concept, a viable approach might be to artificially accelerate corrosion development in a laboratory. This would also make it easier to take accurate images. An artificial dataset might perhaps also be constructed based on corrosion development models.

Bibliography

[1] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. https://github.com/matterport/Mask_RCNN. 2017.

[2] Airsens. *Homepage Airsens*. URL: https://www.airsens.no.

[3] Abdullah Alfarrarjeh et al. "A Deep Learning Approach for Road Damage Detection from Smartphone Images". In: *Proceeding of the 2018 IEEE International Conference on Big Data (Big Data)*. IEEE. Seattle, WA, USA, 2018, pp. 5201–5204.

[4] Deegan J Atha and Mohammad R Jahanshahi. "Evaluation of deep learning approaches based on convolutional neural networks for corrosion detection". In: *Structural Health Monitoring* 17.5 (2018), pp. 1110–1128. DOI: 10.1177/1475921717737051. URL: https://doi.org/10.1177/1475921717737051.

[5] Young-Jin Cha et al. "Autonomous Structural Visual Inspection Using Region-Based Deep Learning for Detecting Multiple Damage Types". In: *Computer-Aided Civil and Infrastructure Engineering* 00 (Nov. 2017), pp. 1–17. DOI: 10.1111/mice.12334.

[6] Papers with Code. *State-of-the-Art Machine Learning Methods*. 2019. URL: https://paperswithcode.com/ (visited on 03/01/2020).

[7] Papers with Code. *State-of-the-Art Real-Time Semantic Segmentation*. 2019. URL: https://paperswithcode.com/task/real-time-semantic-segmentatio (visited on 05/23/2020).

[8] Marius Cordts et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[9] Jifeng Dai, Kaiming He, and Jian Sun. "Instance-aware Semantic Segmentation via Multi-task Network Cascades". In: *CoRR* abs/1512.04412 (2015). arXiv: 1512.04412. URL: http://arxiv.org/abs/1512.04412.

[10] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[11] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2011 (VOC2011) Results*. http://www.pascal-network.org/challenges/VOC/voc2011/workshop/index.html.

[12] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[13] Facebook and Various. *Pytorch Framework*. URL: https://pytorch.org/ (visited on 10/14/2019).

[14] Simen Keiland Fondevik. "Evaluation of Modern Deep Learning Methods for Automatic Image Classification of Corrosion Damages". Specialization Thesis. Department of Engineering Cybernetics, Dec. 2019.

[15] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083.

[16] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[17] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2014. arXiv: 1412.6572 [stat.ML].

[18] Google and Various. *Tensorflow Framework*. URL: https://pytorch.org/ (visited on 10/14/2019).

[19] Priyan Gunatilake et al. "Image understanding algorithms for remote visual inspection of aircraft surfaces". In: *Machine Vision Applications in Industrial Inspection V*. Ed. by A. Ravishankar Rao and Ning S. Chang. Vol. 3029. International Society for Optics and Photonics. SPIE, 1997, pp. 2–13. DOI: 10.1117/12.271231. URL: https://doi.org/10.1117/12.271231.

[20] Divam Gupta. *Image Segmentation Keras*. https://github.com/divamgupta/image-segmentation-keras. 2017.

[21] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[22] Kaiming He et al. "Mask R-CNN". In: *CoRR* abs/1703.06870 (2017). arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870.

[23] Egil Holm. "Classification of Corrosion and Surface Damage on Bridge Constructions using Deep Learning". Master Thesis. Department of Engineering Cybernetics, June 2019.

[24] Egil Holm et al. "Classificationof corrosion and coating damages on bridge constructions from images using convolutional neural networks". In: vol. 11433. 2019.

[25] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[26] ImageNet. *ImageNet: Summary and Statistics*. 2010. URL: http://www.image-net.org/about-stats (visited on 10/14/2019).

[27] Scout Drone Inspection. *Homepage Scout*. URL: https://www.scoutdi.com.

[28] Alexander B. Jung. *imgaug*. https://github.com/aleju/imgaug. 2018.

[29] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). URL: http://arxiv.org/abs/1412.6980.

[30] Gerhardus Koch et al. "International Measures of Prevention, Application, and Economics of Corrosion Technologies Study". In: ed. by Gretchen Jacobson. NACE International. 2016. URL: http://impact.nace.org/documents/Nace-International-Report.pdf.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[32] Raymond Kurzweil. *The Age of Intelligent Machines*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-11121-7.

[33] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. London, UK, UK: Springer-Verlag, 1998, pp. 9–50. ISBN: 3-540-65311-2. URL: http://dl.acm.org/citation.cfm?id=645754.668382.

[34] Sangwook Lee, Luh-Maan Chang, and Miroslaw Skibniewski. "Automated recognition of surface defects using digital color image processing". In: *Automation in Construction* 15.4 (2006), pp. 540–549. URL: https://www.sciencedirect.com/science/article/pii/S0926580505000981.

[35] Yi Li et al. "Fully Convolutional Instance-aware Semantic Segmentation". In: *CoRR* abs/1611.07709 (2016). arXiv: 1611.07709. URL: http://arxiv.org/abs/1611.07709.

[36] Tsung-Yi Lin et al. "Feature Pyramid Networks for Object Detection". In: *CoRR* abs/1612.03144 (2016). arXiv: 1612.03144. URL: http://arxiv.org/abs/1612.03144.

[37] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[38] Stefan Livens et al. "Classification of corrosion images by wavelet signatures and LVQ networks". In: *Computer Analysis of Images and Patterns*. Ed. by Václav Hlaváč and Radim Šára. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 538–543. ISBN: 978-3-540-44781-8.

[39] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *CoRR* abs/1411.4038 (2014). arXiv: 1411.4038. URL: http://arxiv.org/abs/1411.4038.

[40] Leonardi Marco. “TTK25 Computer Vision for Control, Lecture Notes Autumn 2019”. University Lecture. 2019.

[41] A. Mikołajczyk and M. Grochowski. “Data augmentation for improving deep learning in image classification problem”. In: *2018 International Interdisciplinary PhD Workshop (IIPhDW)*. 2018, pp. 117–122.

[42] Osama Moselhi and Tariq Shebab-Eldeen. “Classification of Defects in Sewer Pipes Using Neural Network”. In: *Journal of Infrastructure Systems* 6 (1999). DOI: 10.1007/978-3-642-31439-1_19. URL: https://doi.org/10.1007/978-3-642-31439-1_19.

[43] Orbiton. *Homepage Orbiton*. URL: https://orbiton.no.

[44] N. Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66.

[45] Luca Petricca et al. “Corrosion Detection Using A.I : A Comparison of Standard Computer Vision Techniques and Deep Learning Model”. In: vol. 6. May 2016, pp. 91–99. DOI: 10.5121/csit.2016.60608.

[46] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[47] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[48] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 9781292153964.

[49] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229.

[50] Manu Sharma and Brian Rieger. *LabelBox*. https://labelbox.com. 2018.

[51] Mel Siegel and Priyan Gunatilake. “Remote Enhanced Visual Inspection of Aircraft by a Mobile Robot”. In: *IEEE Workshop on Emerging Technologies, Intelligent Measurement and Virtual Systems for Instrumentation and Measurement* (Aug. 1999).

[52] Karen Simonyan and Andrew Zissermann. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: (Apr. 2015). URL: https://arxiv.org/pdf/1409.1556.pdf.

[53] Janpreet Singh and Shashank Shekhar. “Road Damage Detection And Classification In Smartphone Captured Images Using Mask R-CNN”. In: *CoRR* abs/1811.04535 (2018). arXiv: 1811.04535. URL: http://arxiv.org/abs/1811.04535.

[54] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[55] Mingxing Tan and Quoc V. Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”. In: *CoRR* abs/1905.11946 (2019). arXiv: 1905.11946. URL: http://arxiv.org/abs/1905.11946.

[56] W. Wang et al. “Road Damage Detection and Classification with Faster R-CNN”. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 5220–5223.

[57] Quinghui Zhang, Xianing Chang, and Shanfeng Bian Bian. “Vehicle-Damage-Detection Segmentation Algorithm Based on Improved Mask RCNN”. In: *IEEE Access* 8 (2020). URL: https://ieeexplore.ieee.org/document/8950115.

[58] Hengshuang Zhao et al. “Pyramid Scene Parsing Network”. In: *CoRR* abs/1612.01105 (2016). arXiv: 1612.01105. URL: http://arxiv.org/abs/1612.01105.

[59] Shuai Zhao, Dong Ming Zhang, and Hong Wei Huang. “Deep learning–based image instance segmentation for moisture marks of shield tunnel lining”. In: *Tunnelling and Underground Space Technology* 95 (2020), pp. 103–156. ISSN: 0886-7798. DOI: https://doi.org/10.1016/j.tust.2019.103156. URL: http://www.sciencedirect.com/science/article/pii/S0886779819301452.

[60] Xin Zheng et al. “Image segmentation based on adaptive K-means algorithm”. In: *EURASIP Journal on Image and Video Processing* 2018 (Dec. 2018). DOI: 10.1186/s13640-018-0309-3.

[61] Bolei Zhou et al. “Scene Parsing through ADE20K Dataset”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017.

Appendices

Appendix A

Scientific Paper Submitted to ICTAO 2020

This master thesis is considered highly relevant for the industry and further work. It was therefore decided to write a scientific paper based on the obtained results. As of delivering this thesis, the paper is submitted to the 32nd International Conference on Tools with Artificial Intelligence. The conference was supposed to take place in Maryland, USA in the beginning of November 2020. However, due to the circumstances concerning covid-19, it is changed to an all-digital event.

The scientific paper is written by the author of this thesis and reviewed by his supervisors before submission. The paper is attached below.

Image Segmentation of Corrosion Damages in Industrial Inspections

Simen Keiland Fondevik*, Annette Stahl†, Aksel Andreas Transeth‡ and Ole Øystein Knudsen§
*dept. of Engineering Cybernetics, Norwegian University of Science and Technology, simenkf@stud.ntnu.no
†dept. of Engineering Cybernetics, Norwegian University of Science and Technology, annette.stahl@ntnu.no
‡dept. of Mathematics and Cybernetics, SINTEF Digital, aksel.a.transeth@sintef.no
§dept. of Materials and Nanotechnology, SINTEF Digital, ole.knudsen@sintef.no

***Abstract*—In this paper we assess image segmentation algorithms for the purposes of automatic corrosion damage segmentation. Automatic image analysis is needed in order to process all data retrieved from drone-driven industrial inspections. This could be the first step towards estimating total damaged area of a construction, an often used metric to initiate maintenance. To this end we provide three main contributions. First, 608 images with corrosion damages are instance-wise annotated with binary segmentation masks to construct a dataset. Second, an advanced, two-stage data augmentation scheme is developed and empirically shown to significantly reduce overfitting. Finally, Mask R-CNN and PSPNet are evaluated on the corrosion dataset using this and other data augmentation methods. With $77.5\,\%$ and $73.2\,\%$ mean IoU for Mask R-CNN and PSPNet, respectively, the results are very promising. It is concluded that image segmentation can aid automating industrial inspections of steel constructions in the future, and that instance segmentation is likely more useful than semantic segmentation due to its applications to a wider range of use-cases. However, current performance given the rather small dataset used, is not considered good enough to construct a reliable, fully autonomous inspection system as of yet.**

***Index Terms*—semantic segmentation, instance segmentation, PSPNet, Mask R-CNN, industrial inspection, corrosion.**

I. Introduction

NACE[1] International estimated in 2016 corrosion damages to have an annual cost of 2.5 trillions USD [1], equivalent to $3.4\,\%$ of global GDP. Corrosion is a major problem that wear down steel constructions and can severely reduce their strength. Constructions in humid and salty environments are particularly exposed. Regular inspections and quality controls are therefore necessary.

Inspection of constructions is usually performed visually on-site. However, there are many areas not easily accessible to inspectors, for instance due to hazardous conditions or simply because it is out of reach. Additionally, manual inspections can be time consuming, expensive and subjective. An interesting approach is therefore to use unmanned aerial vehicles (UAVs) to take photos of potentially damaged areas. Images are sent to a cloud service or remote computer for further analysis. The concept is illustrated in Fig. 1.

Inspecting images manually, however, is also tedious, time consuming, and subject to human subjectivity. With an increasing number of images to inspect, the error rate is also likely to increase. The vast amount of data automatically collected using UAVs can simply render manual processing of images infeasible. An automatic image analysis framework is therefore needed, both in terms of efficiency and objectivity. This again allows for more frequent inspections and thereby detection of damages at an earlier stage.

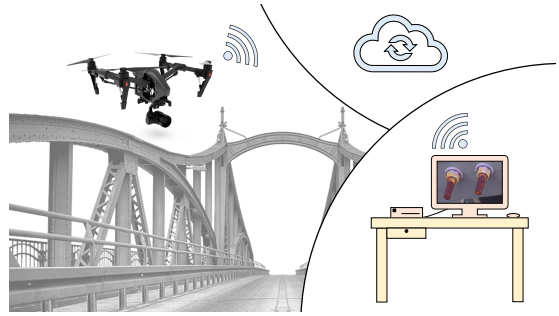


Fig. 1: Concept of a UAV inspecting a bridge. Images are sent to a cloud service/remote computer for further analysis.

In this paper we study how automatic image segmentation of corrosion damages can be performed using machine learning. There are multiple reasons why this is useful, and why it is superior to simple image classification, i.e. sorting images as corrosion vs. not corrosion. First, successful segmentation of corrosion damages allows an autonomous system (e.g. a UAV) to better understand its surroundings and thereby be able to, for instance, focus the camera in the right direction and aid navigation. Second, estimation of total damaged area of a construction is an often-used criterion to initiate maintenance. Image segmentation is a necessary first step towards this goal.

Although promising, existing research on image segmentation to detect corrosion damages is limited. The aim of this paper is therefore to assess and improve the applicability of state-of-the-art image segmentation for the purposes of automating industrial inspections. To this end, we provide three main contributions: First, since no other dataset was found available, a proof-of-concept dataset containing 608 annotated images with corrosion damages is constructed. Second, an advanced, two-stage data augmentation scheme is developed to artificially increase the dataset size. The scheme

[1]National Association of Engineers.

is empirically shown to significantly reduce overfitting and improve performance for Mask R-CNN. Finally, state-of-the-art algorithms for semantic and instance segmentation are evaluated on the corrosion dataset using this and other data augmentation methods.

II. Related Work

Traditional computer vision methods for corrosion damage detection are generally based on a color and/or texture analysis [2]–[5]. Due to the characteristic red/brown color of corrosion, very simple algorithms such as thresholding can obtain decent results on certain images. Traditional computer vision suffers, however, if optimal features are not easily identified.

Using machine learning, the need to manually identify common domain features is removed. Machine learning used for damage detection dates back to (at least) 1999 when a three-layer neural network was used to detect defects in underground sewer pipes [6].

In [7], a traditional computer vision approach based on the number of red pixels in an image was compared to a convolutional deep learning approach for detecting rust. The deep learning network was based on AlexNet [8], and the authors found this to be superior to traditional computer vision.

[9] studied convolutional neural networks for corrosion assessment on metallic surfaces using sliding windows. Their findings suggest 128×128 windows along with the image classification network VGG16 [10].

Direct end-to-end image classification of corrosion damages and paint flaking is also studied in [11] and [12]. The results show that newer and smarter network architectures outperform common networks such as VGG [10] and ResNet [13], with EfficientNet [14] obtaining the best performance.

Object detection was applied to a dataset with various construction damages in [15]. Using Faster R-CNN [16], the model was successfully trained to detect corrosion, cracks in concrete and steel delamination. The authors conclude that Faster R-CNN can be used with UAVs to replace human-oriented industrial inspections in the future. Note, however, that authors performed object detection only, i.e. segmentation masks were not predicted.

A concrete crack detection method was proposed in [17] based on the FCN architecture [18]. The authors found that cracks are reasonably detected and accurately evaluated.

In [19], moisture marks of shield tunnel linings in images were segmented. Mask R-CNN [20] was compared with a previously proposed fully convolutional network by the same authors, a region growing algorithm and a thresholding algorithm. Mask R-CNN was found superior.

To summarize, previous work is usually either focused on the simpler task of image classification and object detection, or image segmentation of "simple" damages, i.e. damages clearly standing out from the surroundings (e.g. tunnel linings have a smooth, monochrome surface). Research on true image segmentation of corrosion damages is lacking.

III. State of the Art

Image segmentation is the problem of outlining relevant objects in images. We typically distinguish between *semantic* segmentation and *instance* segmentation. Semantic segmentation classifies every pixel in an image as one of multiple classes, whereas instance segmentation also distinguishes between different instances of the same class, e.g. outlining each individual person within an image rather than the whole group as one. Thus, instance segmentation can be considered a combination of object detection and semantic segmentation.

A. Semantic Segmentation

FCN [18], introduced in 2014, was the first fully convolutional network trained end-to-end with supervised pre-training for pixel-to-pixel predictions. The base network is a modification of the classification network VGG [10] in which linear layers are exchanged for deconvolution layers to up-sample the feature maps. Increased performance was then obtained by fusing the output layer with intermediate layers through element-wise addition.

U-Net [21] build upon the success of FCN to further improve performance in 2015. The network forms a U-shape with the first part being a contracting convolutional path similar to any image classification network (e.g. ResNet [13]). This is followed by an almost symmetrical expanding path increasing the spatial dimensions of the feature maps to construct a segmentation map. To better incorporate location information of features, links between corresponding layers in the contracting and expanding path were added. U-Net obtained impressive results at the time using very few training images.

PSPNet [22] was introduced in 2016 with the aim of improving utilization of contextual information. Humans have a tremendous ability to distinguish and classify seen objects, even from a young age. First, we recognize the shape, color and size of an object, from which we can often conclude accurately what we are looking at. Next, we use surrounding visual clues to further increase our certainty. For instance, if a vehicle is seen on a river, it is likely a boat even if it looks like a car. FCN and U-Net struggle to utilize this kind of global information; if it looks like a car, it certainly must be a car. The main contribution of PSPNet is a pyramid pooling module performing sub-region average pooling. In short, the algorithm collects contextual information at different scales through average pooling, up-scales the produced feature maps and finally uses convolutional layers to produce a segmentation mask. When published, PSPNet improved state of the art on numerous datasets, and is still one best-performing semantic segmentation networks to date.

B. Instance Segmentation

Since published in 2017, *Mask R-CNN* [20] has become the de facto standard for instance segmentation. The algorithm can be divided into four parts. First is a *backbone network* serving as a feature extractor. This can be any image classification network, but usually a *feature pyramid network* [23] is used

(a) Easy image to annotate.

(b) Difficult image to annotate.

Fig. 2: Instance segmentation masks for two images. (a) Three instances are immediately recognized, and their boundaries are sharp, making it easy to annotate the image. (b) With a large number of potential instances, some areas more severely corroded than others, boundaries being smooth, etc., this is a very challenging image to annotate. Black indicate no corrosion, i.e. background. Instances are assigned random colors and have no further interpretation.

to better detect features at different scales. Second is a *region proposal network* (RPN). RPNs are neural networks scanning an image for *regions of interest* (RoI), i.e. areas likely to contain a relevant object. The RoIs are then used as input to the third stage of Mask R-CNN; *classification and bounding box regression of RoIs*. A sub-network is used to assign a class label to each RoI and compute refinements to their locations and sizes. Finally, and in parallel, a fully convolutional network predicts a segmentation mask for each RoI. In fact, one mask for each possible class label is generated, before the result from the independent class prediction network is used to keep the correct mask only.

Mask R-CNN has been adapted to a wide range of different datasets with great results. Compared to many alternatives, the algorithm is very streamlined, yet still state of the art in terms of prediction performance.

IV. Methods and Implementation

A. Dataset Construction

As no dataset for corrosion damage segmentation was available prior to this work, a new dataset was constructed from scratch. The images used were taken during real inspections of steel bridges by NPRA[2]. A large number of images known to contain corrosion damages were uploaded to the online software Labelbox [24] for instance-wise labeling. Ideally, we would want segmentation masks for all sorts of damages, such as cracks in concrete and paint flaking, as well as intact construction. This, however, is very time consuming and it was therefore opted for binary segmentation masks only indicating where *corrosion* is present. A total of 4631 instances were labeled across 608 images. In total, $19.1\,\%$ of the pixels contained corrosion with an average instance size of $2.53\,\%$ relative to the image frame.

In addition to time consuming, annotating corrosion damages is also challenging. First, the boundaries of corrosion are often smooth, making it difficult to accurately outline the damage. Second, unlike most everyday object, there are few rich characteristics defining a corrosion damage instance. Hence, separating instances can often be done in multiple equally good ways. Third, a surface can look corroded, but in reality, be only dirty or contain smudged corrosion from nearby corroded steel. Fig. 2 shows two example images with corresponding segmentation mask, one very easy to annotate and one very difficult.

[2]The Norwegian Public Roads Administration.

B. Data Augmentation

608 images are rather few by modern standards. Extensive data augmentation was therefore tested as it increases the effective number of training images and acts as a method of regularization. The most commonly used data augmentation scheme found in the literature is horizontal flipping. Slight rotations, minor color/contrast/brightness perturbations and cropping/translation are also often-used. However, corrosion damages have no direction/orientation, nor any defined size, shape or absolute color. Only using the previously mentioned data augmentations is therefore fairly conservative with respect to the corrosion dataset. Applying too heavy data augmentation, on the other hand, can break the contents of images and consequently render bad training examples.

A more sophisticated, two-stage scheme is therefore proposed in this paper: First train using heavy data augmentation, followed by fine-tuning using none or light data augmentation (e.g. flipping). Heavy data augmentation could for instance use a combination of various affine transformations, edge enhancements, added noise or blur and dropout. The idea is that during the first stage, weights in neural networks will obtain a better starting point than those obtained directly from transfer learning using general datasets. Optimal weights are then searched for in the second stage as usual. Since training examples of the first stage are more difficult, we train the networks for a relatively large number of epochs (e.g. 30–100) before fine-tuning for relatively few epochs (e.g. 1–30). This data augmentation scheme will in the following be referred to as the two-stage data augmentation scheme.

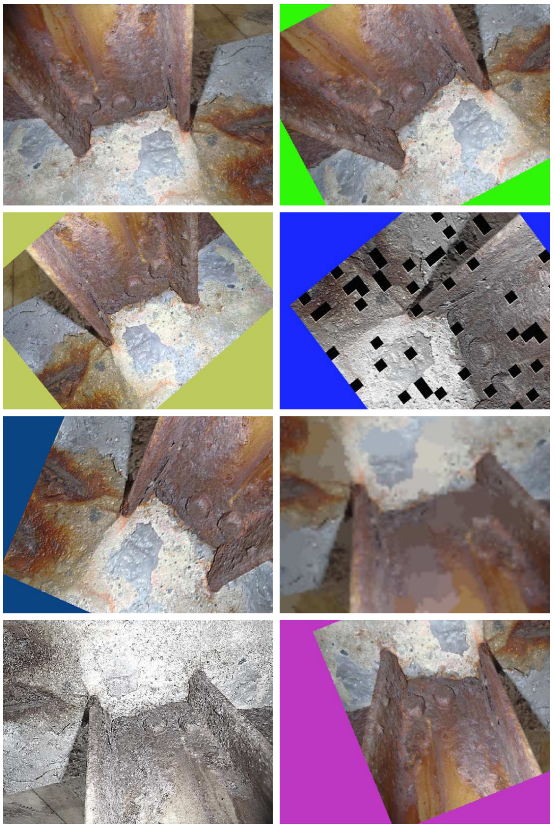


Fig. 3: First stage of the two-stage data augmentation scheme applied to an example image (top left).

Fig. 3 shows seven possible outputs from heavy data augmentation of an example image.

C. Evaluation Metrics

Evaluation of trained models are based on inference speed, peak memory usage and intersection over union (IoU) for predicted segmentation masks. Ideally, video standard frame rate of 24 images per second is desirable for inference as this would enable real-time image segmentation. Peak memory usage has no further requirements than not to exceed available memory. IoU measures how well a predicted mask corresponds to the ground truth segmentation mask, and is defined as follows:

$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}} \tag{1}$$

TP, FP and FN refer to, respectively, true positives, false positives and false negatives. IoU is calculated for both corrosion (cIoU) and background (bIoU). For simple comparison of models, however, mean intersection over union (mIoU) is mainly used:

$$\text{mIoU} = \frac{\text{cIoU} + \text{bIoU}}{2} \tag{2}$$

50 images were set aside as a test set and not used until final evaluation of trained models. An equally large validation set was used to evaluate the models while experimenting with different network configurations and data augmentation schemes. The remaining 508 images were used for training. The test set is emphasized when evaluating the models.

As discussed in Section IV-A, there are potentially many ways to divide a corroded image into instances. Thus, even if Mask R-CNN predicts good instance masks, performance might be reported as poor since instances are separated differently compared to the ground truth. It was therefore decided to merge all predicted instances from Mask R-CNN when computing mIoU. This also makes it easier to compare the network with PSPNet. The true, instance-wise predictions are manually analyzed by visually comparing them against the ground truth masks.

D. Implementation

All experiments and implementation were done with Python, Keras, Tensorflow and the comprehensive data augmentation library imgaug. Configurations for the first stage of the two-stage data augmentation scheme is based on example values in the library documentation [25]. Hardware and software configurations are detailed in Table I and Table II, respectively.

The implementation of PSPNet is based on the GitHub repository by Divam Gupta [26]. Parts of the code base was modified to work properly with imgaug. All relevant configuration parameters used are specified in Table III. PSPNet-50 was chosen over PSPNet-101 in order to decrease overfitting as the corrosion dataset is small. ADE20K [27] was used for transfer learning as it is the most versatile and challenging dataset available for semantic segmentation. Default values for optimizer, image size and batch size were used for simplicity.

The excellent implementation by Waleed Abdulla [28] available on GitHub was used for Mask R-CNN. Relevant hyperparameters are listed in Table IV.

TABLE I: Hardware Configuration.

CPU	Intel Core i7-6850K (6 cores @ 3.6 GHz)
RAM	Corsair 32 GB DDR4 @ 2666 MHz
GPU	NVIDIA Titan X (Pascal) (12 GB GDDR5X)
Storage (dataset)	Seagate Barracuda 1TB SATA HDD @ 7200 RPM

TABLE II: Software Configurations.

Operating System	Microsoft Windows 10 Home
Python	v. 3.7.6
Keras	v. 2.3.1
Tensorflow	tensorflow-gpu v. 1.14.0
CUDA	v. 10.0
CuDNN	v. 7.6.5
imgaug	v. 0.4.0

TABLE III: PSPNet Configuration.

Backbone network	ResNet-50 [13]
Transfer learning dataset	ADE20K [27]
Optimizer	Adadelta (default, adaptive)
Image size	473×473 (default)
Batch size	2 (default)

TABLE IV: Mask R-CNN Configuration.

Backbone network	ResNet-50 [13] (with FPN [23])
Transfer learning dataset	COCO [29]
Optimizer	SGD with momentum (default)
Learning rate	0.001 (default)
Learning momentum	0.9 (default)
Image size	512×512
Batch size	1
Max instances per image	50
Detection confidence threshold	90 %

TABLE V: Training and test mIoU for PSPNet and Mask R-CNN trained with various data augmentation schemes.

Network	Data augm.	Train mIoU	Test mIoU	Difference
PSPNet	None	91.7 %	78.6 %	13.1
PSPNet	Flipping	87.7 %	78.5 %	9.2
PSPNet	Two-stage	84.7 %	77.5 %	7.2
Mask R-CNN	None	86.8 %	72.0 %	14.8
Mask RCNN	Flipping	81.5 %	74.4 %	7.1
Mask R-CNN	Two-stage	80.0 %	73.2 %	6.8

TABLE VI: Inference frame rate and peak memory usage of PSPNet and Mask R-CNN.

Network	Inference frame rate	Peak memory usage
PSPNet	3.7 imgs/s	10 GB
Mask R-CNN	1.4 imgs/s	10 GB

The 50-layer ResNet backbone was chosen for the same reason as for PSPNet. Additionally, Mask R-CNN has a vast memory consumption, meaning a decreased model size is beneficial. The COCO [29] dataset was used for transfer learning, as it is the de facto standard for instance segmentation and the only available option. All default parameters regarding optimizer were used, as initial testing with more advanced optimizers such as Adam [30] was unsuccessful. The algorithms require the image size to be six times divisible by 2. The closest such number to 473 (the image size used for PSPNet) is 512 and therefore used for Mask R-CNN. A batch size of 1 was needed in order not to exceed the memory capacity of the GPU used. Limiting the maximum number of instances to predict per image was also done to decreases the memory footprint. It was opted for a maximum value of 50 (default being 100). Finally, a confidence score of at least 90 % was chosen as the lower limit for accepting an instance as a positive RoI/prediction. This value was chosen relatively high since we have only two classes (corrosion and background).

Compared to the original Mask R-CNN, which was implemented in Pytorch rather than Tensorflow, the implementation used in this paper has two main differences. First, bounding box ground truths are not explicitly provided in the dataset, but determined during run-time as the smallest box encapsulating the entire instance mask. Second, the Pytorch implementation used a learning rate of 0.02, which was found too large for Tensorflow as the weights tended to explode. A learning rate of 0.001 was therefore used instead. Additionally, we opted for a learning rate schedule halving the learning rate every 10th epoch, ensuring better convergence. (No explicit learning rate schedule was used for PSPNet as it uses an adaptive optimizer.)

V. Results

Table V compares training and test mIoU for PSPNet and Mask R-CNN when trained for 70 epochs with and without various data augmentation schemes. "Flipping" refers to random horizontal and vertical flipping, whereas "Two-stage" refers to heavy data augmentation for 50 epochs followed by flipping for 20 epochs. Inference speed and peak memory usage for both networks are listed in Table VI. Predicted segmentation masks for PSPNet and Mask R-CNN compared with corresponding ground truth instance masks are shown in Fig. 4. Finally, Fig. 5 shows prediction masks for images containing no corrosion.

VI. Discussion

A. IoU Scores

We see from Table V that the gap between mIoU on training data and test data is significantly reduced by data augmentation, i.e. overfitting is reduced. Particularly the two-stage data augmentation scheme is shown to produce models generalizing better to new, unseen images. This is not to say, however, that actual segmentation performance on the current dataset is necessarily improved. Instead it means that if we are able to levitate performance on training data, we can expect test data performance to also improve. Trained models are, simply put, more reliable. This is important as overfitted models give a false sense of security.

That said, actual mIoU for Mask R-CNN as measured on the test set *is* improved by data augmentation. Since PSPNet is not experiencing a similar performance boost from data augmentation, it might indicate that data augmentation is more useful for instance masks rather than semantic segmentation masks. This seems plausible as a data augmented damage is still a unique damage, whereas for semantic segmentation the whole image, so to say, is changed. This could perhaps leave us with a segmentation mask no longer representing realistic corrosion in a useful way. On the other hand, the authors of PSPNet did find increased performance using data augmentation, indicating the results of Table V are representative for the corrosion dataset only. The main differences between the corrosion dataset and other datasets found in the literature is its small size and few classes. Further work should investigate if an enlarged and more complex version of the corrosion dataset would benefit more from data augmentation in general, and the two-stage data augmentation scheme in particular.

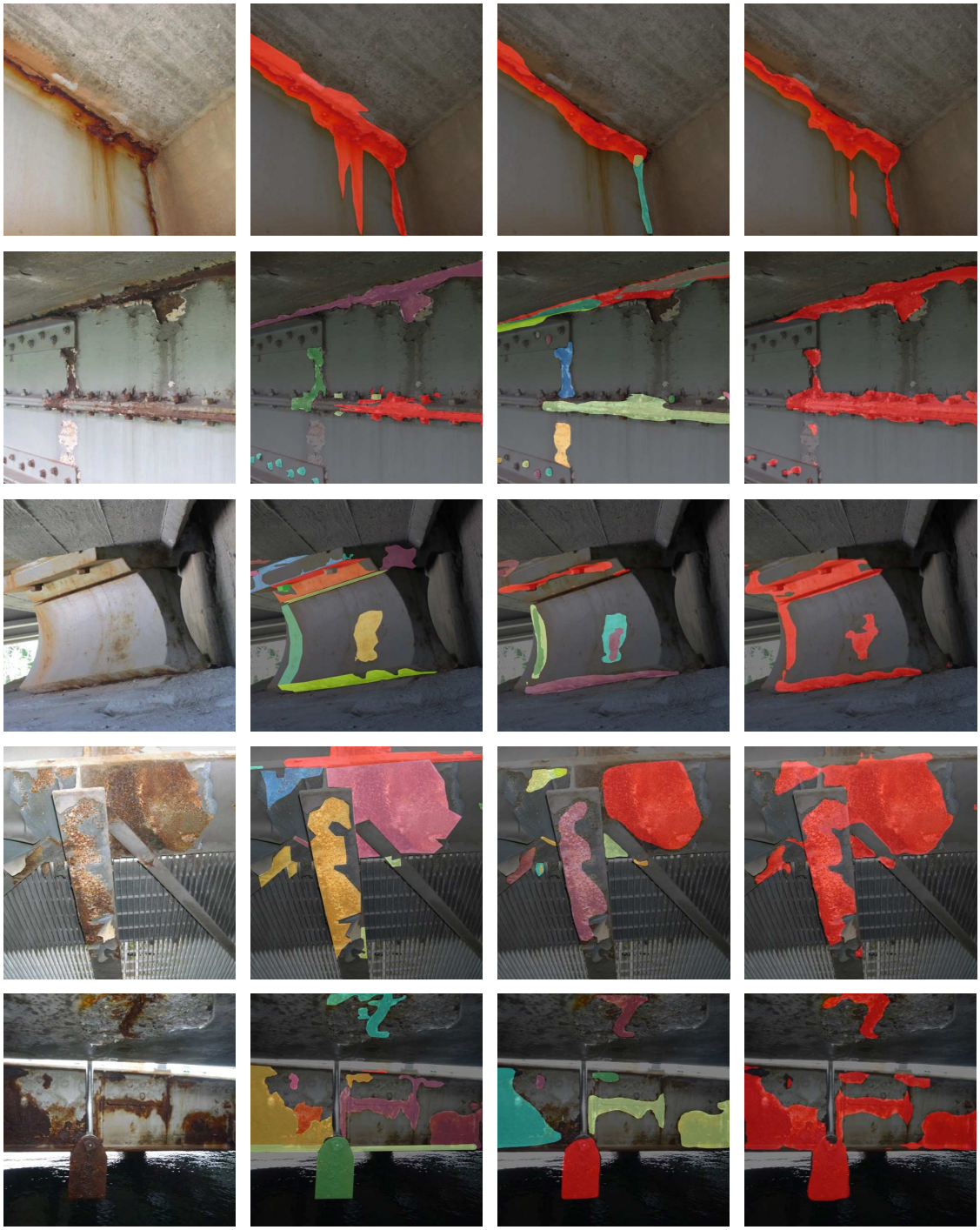


Fig. 4: Predicted segmentation masks for Mask R-CNN and PSPNet compared with corresponding raw images and ground truths. Instances are assigned random colors with no further interpretation.

B. Predicted Segmentation Masks

When studying the predicted segmentation masks shown in Fig. 4, several interesting observations can be made. First, PSPNet is more aggressive than Mask R-CNN, i.e. it has greater recall. High recall is important in industrial inspections in order not to miss any dangerous damages. For PSPNet, any output feature map pixel greater than 0.5 is considered corrosion, whereas for Mask R-CNN a damaged area must first be considered a region of interest with sufficiently large probability. In this paper, we used a confidence threshold of $90\,\%$, meaning higher recall for corrosion is easily obtained for Mask R-CNN by reducing this value. Doing so, however, could increase the number of false positives. Furthermore, accepting more RoIs can result in increased number of instances rather than more aggressive, existing instances. This is undesirable as Mask R-CNN already has a tendency to construct unnecessary many and complicated instances. The first three examples in Figure 4, for instance, have instances inside other instances where the ground truth only has one instance.

As discussed in Section IV-A, defining and distinguishing corrosion instances is difficult. Still Mask R-CNN seems to correspond well with the ground truth. Although some instances are divided into multiple small instances in the examples in Figure 4, Mask R-CNN very rarely merges multiple ground truth instances into one. It can be questioned, however, whether instance correspondences between ground truths and Mask R-CNN predictions are good enough for the use-cases discussed in Section I. For general corrosion detection it is of little relevance. For estimation of damage area and severity, on the other hand, good correspondence is of great importance. Similarly, if monitoring a damage over time is to be performed autonomously, good correspondence is needed.

Overall, predicted segmentation masks seem promising. When predicted for non-corroded images, however, the situation is somewhat different. Fig. 5 compares predicted masks for hard images especially selected to contain red/brown colored areas. These images are not part of the corrosion dataset and were therefore neither used for training nor for calculation of IoU. It is immediately evident that both Mask R-CNN and PSPNet suffer from false positives. Simply put, the algorithms predictions, and PSPNet in particular, are too heavily based on the color properties of corrosion. In terms of the previously discussed use-cases, the networks will consequently report too many damages, over-estimate areas and generally be too pessimistic regarding the integrity of a construction.

Two solutions are proposed. First, it is important to remember that the corrosion dataset contains no images similar to those shown in Figure 5. As every training example contains at least one corroded area, a predicted segmentation mask without any corroded pixels would in fact be peculiar. Training on images without corrosion in addition to the current corrosion dataset might therefore reduce the number of false positives. This is not done in this paper since images containing no corrosion are not supported for by Mask R-CNN. Incorporating background only images would require changes to the framework. Furthermore, background only images would introduce more imbalance to the dataset.

Fig. 5: Predicted segmentation masks for Mask R-CNN and PSPNet on images *not* containing any corrosion. Instances are assigned random colors with no further interpretation.

A second solution is constructing a two-stage pipeline: First classify images in two categories, *corrosion* vs. *no corrosion*, and then apply image segmentation to all corroded images. This might be a viable approach as previous work has found such classification to perform very well on corrosion and background images [11], [12]. Extending upon this idea, it should be possible to train the first sub-networks of Mask-RCNN to better learn the distinction between background only images and corrosion images using both types of images. We could then freeze early layers and continue training the latter sub-networks on corrosion images only. This would be beneficial as the classification and segmentation networks share computation. Again, this would require major changes to the Mask R-CNN framework. A similar strategy should be applicable to the backbone network of PSPNet as well.

C. Image Segmentation for Industrial Inspections

Although promising results are found in this paper, general performance is not good enough for a fully autonomous system yet, especially in terms of false positives on non-corroded images with red/brown-like colors. A larger dataset would of course likely increase performance, but it is difficult to estimate by how much. Training on background only images in addition to corrosion images might also be a viable option but requires modifications to existent frameworks. Either way, only segmenting corrosion damages has limited potential compared to a model able to predict all sorts of damages as well

as intact construction. Collecting a larger and more complex dataset should therefore be a priority in further work.

Although PSPNet obtains slightly better IoU on the test set, Mask R-CNN and instance segmentation in general are considered more versatile for the purposes of industrial inspections using a UAVs. Additionally, the performance difference is likely caused by the more aggressive predictions by PSPNet, which can be achieved for Mask R-CNN as well by accepting more RoIs.

Regarding time and space complexity, both networks require a high-end GPU with descent amount of video memory in order to run somewhat efficiently. The frame rate is still nowhere close to video standards of 24 imgs/s, with 3.7 imgs/s and 1.4 imgs/s for PSPNet and Mask R-CNN, respectively. Smaller, more efficient networks or better hardware are therefore needed before the full potential of image segmentation can be applied to industrial inspections.

VII. Conclusion

In this paper we have evaluated PSPNet and Mask R-CNN on a constructed 608 image dataset for the purposes of segmenting corrosion damages and to automate industrial inspections. A two-stage data augmentation scheme was developed and empirically shown to significantly reduce overfitting, in addition to improve performance for instance segmentation. With $73.2\,\%$ mIoU and decent instance mask correspondence, the obtained results are very promising. However, current performance is not considered good enough to design a reliable, fully autonomous inspection system as of yet. With a larger and more complex dataset, performance is expected to increase, and a wide range of possible use-cases emerge. Further work should therefore collect more data and introduce additional classes, such as crack in concrete, paint flaking and intact steel.

References

[1] G. Koch, J. Varney, N. Thompson, O. Moghissi, M. Gould and J. Payer, "International measures of prevention, application, and economics of corrosion technologies Study," NACE International, 2016, p. 3.

[2] P. Gunatilake, M. Siegel, A. G. Jordan and G. W. Podnar, "Image understanding algorithms for remote visual inspection of aircraft Surfaces," in Machine Vision Applications in Industrial Inspection V, vol. 3029, pp. 2–13, 1997.

[3] M. Siegel, P. Gunatilake, "Remote enhanced visual inspection of aircraft by a mobile robot," in IEEE Workshop on Emerging Technologies, Intelligent Measurement and Virtual Systems for Instrumentation and Measurement, 1998.

[4] S. Livens, P. Scheunders, G. van de Wouwer, D. van Dyck, H. Smets, J. Winkelmans and W. Bogaerta, "Classification of corrosion images by wavelet signatures and LVQ networks," in Computer Analysis of Images and Patterns, pp. 538–543, 1995.

[5] S. Lee, L. Chang and M. Skibniewski, "Automated recognition of surface defects using digital color image processing," in Automation in Construction, vol. 15, pp. 540–549, 2006.

[6] O. Moselhi and T. Shebab-Eldeen, "Classification of defects in sewer pipes using neural network," in Journal of Infrastructure Systems, vol. 6, 1999.

[7] L. Petricca, T. Moss, G. Figueroa and S. Broen, "Corrosion detection using A.I : A comparison of standard computer vision techniques and deep learning model," in Computer Science & Information Technology, vol. 6, pp. 91–99, 2016.

[8] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems, vol. 25, pp. 1097–1105, 2012.

[9] D. J. Atha and M. R. Jahanshahi, "Evaluation of deep learning approaches based on convolutional neural networks for corrosion detection," in Structural Health Monitoring, vol. 17, pp. 1110–1128, 2018.

[10] K. Simonyan and A. Zissermann, "Very deep convolutional networks for large-scale image recognition," in Computer Science, 2015.

[11] E. Holm, A. A. Transeth, O. Ø. Knudsen and A. Stahl, "Classification of corrosion and coating damages on bridge constructions from images using convolutional neural networks," in Twelfth International Conference on Machine Vision (ICMV), vol. 11433, 2019.

[12] S. K. Fondevik, "Evaluation of modern deep learning methods for automatic image classification of corrosion damages," unpublished, 2019.

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

[14] T. Mingxing and V. L. Quoc, "EfficientNet: Rethinking model scaling for convolutional neural networks," in CoRR, 2019.

[15] Y. Cha, W. Choi, G. Suh, S. Mahmoudkhani and O. Buyukozturk, "Autonomous structural visual inspection using region-based deep learning for detecting multiple damage types," in Computer-Aided Civil and Infrastructure Engineering, pp. 1–17, 2017.

[16] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," in 28th International Conference on Neural Information Processing Systems, vol. 1, pp. 91–99, 2015.

[17] V. D. Cao and D. A. Le, "Autonomous concrete crack detection using deep fully convolutional neural network," in Automation in Construction, vol. 99, pp. 52–58, 2019.

[18] L. Jonathan, E. Shelhamer and T. Darrell, "Fully convolutional networks for semantic segmentation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, 2017.

[19] S. Zhao, D. M. Zhang and H. W. Huang, "Deep learning–based image instance segmentation for moisture marks of shield tunnel lining," in Runnelling and Underground Space Technology, vol. 95, pp. 103–156, 2020.

[20] K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN," in IEEE International Conference on Computer Vision (ICCV), pp. 2980-2988, 2017.

[21] R. Olaf, F. Philipp and B. Thomas, "U-net: Convolutional networks for biomedical image segmentation," in International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI), Lecture Notes in Computer Science, vol. 9351, pp. 234–241, 2015.

[22] H. Zhao, J. Shi, X. Qi, X. Wang and J. Jia, "Pyramid scene parsing network," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6230-6239, 2017.

[23] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, "Feature pyramid networks for object Detection," IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 936-944, 2017.

[24] M. Sharma and B. Rieger, "Labelbox," Online annotation software to construct datasets for machine learning, https://labelbox.com, 2018.

[25] B. J. Alexander, "imgaug," Python library for image segmentation, https://imgaug.readthedocs.io/en/latest/source/examples_basics.html, 2018.

[26] D. Gupta, "Image segmentation keras," GitHub repository with implementation of PSPNet, https://github.com/divamgupta/image-segmentation-keras, 2017.

[27] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso and A. Torralba, "Scene parsing through ADE20K dataset," in IEEE Conference on Computer Vision and Pattern Recognition, 2017.

[28] W. Abdullah, "Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow," GitHub repository with implementation of Mask R-CNN, https://github.com/matterport/Mask_RCNN, 2017.

[29] L. Tsung-Yi, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollar and C. L. Zitnick, "Microsoft COCO: Common objects in context," in Computer Vision – ECCV 2014, Lecture Notes in Computer Science, vol. 8693, 2014.

[30] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in 3rd International Conference on Learning Representations, 2014.

Appendix B

Dataset Download

The Python script used to download the dataset from Labelbox given a json file is given below. For semantic segmentation, the downloaded masks are merged into one.

```
from PIL import Image
import requests, urllib, json, os, csv, sys
from urllib.request import urlopen
from pathlib import Path
from collections import Counter
import re


def create_dir_if_missing(path):
    if not os.path.exists(path):
        os.makedirs(path)
        return True
    return False


def clean_string_for_filename(name_str):
    pattern = re.compile('[^a-zA-Z0-9_-]')
    string = re.sub(pattern, "", name_str)
    return string


def save_from_url(image_url, save_path):
    im = Image.open(urlopen(image_url))
    im.save(save_path)


def parseCSV(file_name):
    with open(file_name, "r") as f:
```

```
        csv_info = csv.DictReader(f, delimiter=',')
        export_info = []
        for row in csv_info:
            row["Label"] = json.loads(row['Label'])
            export_info.append(row)
    return export_info


def downloadExportData(export_info, directory, save_masks=True):
    if not isinstance(directory, Path):
        directory = Path(directory)

    save_dir = directory /
                clean_string_for_filename(export_info[0]["Project Name"])

    create_dir_if_missing(save_dir)
    print("Downloading images and masks to:")
    print(save_dir)

    total = len(export_info)
    next = 1
    cur = 0

    print("0\%")
    for row in export_info:
        subdirectory = save_dir / row['ID']
        if create_dir_if_missing(subdirectory):
            save_base_image(row, subdirectory)
            if save_masks:
                save_mask_images(row, subdirectory)

        cur += 1
        perc = (cur / total) * 100
        if perc >= next:
            print("{}\%".format(round(perc)))
            next += 1


def downloadImages(export_info, directory):
    downloadExportData(export_info, directory, False)


def save_base_image(label, directory):
    im_path = label['Labeled Data']
```

```
    external_id_cleaned = clean_string_for_filename(label["External ID"])
    if not external_id_cleaned:
        external_id_cleaned = label["ID"] + "_Image"
    name = external_id_cleaned + ".png"

    save_path = directory / name
    save_from_url(im_path, save_path)


def save_mask_images(label, directory):
    mask_dir = directory / "masks"
    create_dir_if_missing(mask_dir)
    mask_counter = Counter()
    if not label['Label']:
        return

    for obj in label['Label']['objects']:
        o_class = obj['value']
        mask_name = o_class + "_" + str(mask_counter[o_class]) + ".png"
        mask_counter.update([o_class])

        im_path = obj['instanceURI']
        save_path = mask_dir / mask_name
        save_from_url(im_path, save_path)


if __name__ == '__main__':
    # to run: python3 download_export.py <FILENAME>, [SAVE_DIRECTORY]
    filename = sys.argv[1]
    try:
        download_dir = sys.argv[2]
    except IndexError:
        download_dir = os.path.expanduser("~") + "/folder/path/"

    file_type = filename.split('.')[-1]
    if file_type == 'json':
        with open(filename, "r") as fp:
            exported_data = json.load(fp)
    elif file_type == "csv":
        exported_data = parseCSV(filename)
    else:
        raise NameError

    downloadExportData(exported_data, download_dir)
```

Appendix C

Data Augmentation

C.1 Flipping

Horizontal and vertical flipping are applied straight forwardly with $50\,\%$ probability as follows.

```
flipping_augmentation = iaa.Sequential([
    iaa.Fliplr(0.5),  # horizontally flip 50% of all images
    iaa.Flipud(0.5),  # vertically flip 50% of all images
])
```

C.2 Heavy Data Augmentation

The heavy data augmentation sequence below is likely too heavy on its own. It should therefore be followed by a number of epochs with flipping or none data augmentation. Default values are mostly used [28]. The augmentation scheme referred to as semi-heavy is identical to the below listing, except the function *sometimes* is defined with probability $30\,\%$, and a maximum of three non-geometric augmentations were applied at once.

```
# Adjusting the definition of sometimes
# adjusts how heavily images are augmented
sometimes = lambda aug: iaa.Sometimes(0.5, aug)

# Define our sequence of augmentation steps
heavy_augmentation = iaa.Sequential([
    # Apply the following augmenters to most images.

    # Flipping
    iaa.Fliplr(0.5),  # horizontally flip 50% of all images
    iaa.Flipud(0.5),  # vertically flip 50% of all images
```

```
# Apply affine transformations to some of the images
# - scale to 80-120% of image height/width (each axis independently)
# - translate by -20 to +20 relative to height/width (per axis)
# - rotate by -45 to +45 degrees
# - shear by -16 to +16 degrees
# - order: use nearest neighbour or bilinear interpolation (fast)
# - mode: use any available mode to fill newly created pixels
#         see API or scikit-image for which modes are available
# - cval: if the mode is constant, then use a random brightness
#         for the newly created pixels (e.g. sometimes black,
#         sometimes white)
sometimes(iaa.Affine(
    scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
    translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
    rotate=(-45, 45),
    shear=(-16, 16),
    order=[0, 1],
    cval=(0, 255),
    mode='constant'
)),

# Execute 0 to 5 of the following (less important) augmenters per
# image. Don't execute all of them, as that would often be way too
# strong.
iaa.SomeOf((0, 5),
   [
       # Convert some images into their superpixel representation,
       # sample between 20 and 200 superpixels per image, but do
       # not replace all superpixels with their average, only
       # some of them (p_replace).
       sometimes(
           iaa.Superpixels(p_replace=(0, 1), n_segments=(20, 200))
       ),

       # Blur each image with varying strength using
       # gaussian blur (sigma between 0 and 3.0),
       # average/uniform blur (kernel size between 2x2 and 7x7)
       # median blur (kernel size between 3x3 and 11x11).
       iaa.OneOf([
           iaa.GaussianBlur((0, 3.0)),
           iaa.AverageBlur(k=(2, 7)),
           iaa.MedianBlur(k=(3, 11)),
       ]),

       # Sharpen each image, overlay the result with the original
```

```
    # image using an alpha between 0 (no sharpening) and 1
    # (full sharpening effect).
    iaa.Sharpen(alpha=(0, 1.0), lightness=(0.75, 1.5)),

    # Same as sharpen, but for an embossing effect.
    iaa.Emboss(alpha=(0, 1.0), strength=(0, 2.0)),

    # Search in some images either for all edges or for
    # directed edges. These edges are then marked in a black
    # and white image and overlayed with the original image
    # using an alpha of 0 to 0.7.
    sometimes(iaa.OneOf([
        iaa.EdgeDetect(alpha=(0, 0.7)),
        iaa.DirectedEdgeDetect(
            alpha=(0, 0.7), direction=(0.0, 1.0)
        ),
    ])),

    # Add gaussian noise to some images.
    # In 50% of these cases, the noise is randomly sampled per
    # channel and pixel.
    # In the other 50% of all cases it is sampled once per
    # pixel (i.e. brightness change).
    iaa.AdditiveGaussianNoise(
        loc=0, scale=(0.0, 0.05 * 255), per_channel=0.5
    ),

    # Either drop randomly 1 to 10% of all pixels (i.e. set
    # them to black) or drop them on an image with 2–5% percent
    # of the original size, leading to large dropped
    # rectangles.
    iaa.OneOf([
        iaa.Dropout((0.01, 0.1), per_channel=0.5),
        iaa.CoarseDropout(
            (0.03, 0.15), size_percent=(0.02, 0.05),
            per_channel=0.2
        ),
    ]),

    # Invert each image's channel with 5% probability.
    # This sets each pixel value v to 255–v.
    iaa.Invert(0.05, per_channel=True),  # invert color channels

    # Add a value of -10 to 10 to each pixel.
    iaa.Add((-10, 10), per_channel=0.5),
```

```
            # Change hue and saturation
            iaa.AddToHueAndSaturation((-20, 20)),

            # Change brightness of images (50-150% of original value)
            # or change the brightness of subareas.
            iaa.OneOf([
                iaa.Multiply((0.5, 1.5), per_channel=0.5),
                iaa.FrequencyNoiseAlpha(
                    exponent=(-4, 0),
                    first=iaa.Multiply((0.5, 1.5), per_channel=True),
                    second=iaa.ContrastNormalization((0.5, 2.0))
                )
            ]),

            # Improve or worsen the contrast of images.
            iaa.LinearContrast((0.5, 2.0), per_channel=0.5),

            # Convert each image to grayscale and then overlay the
            # result with the original with random alpha. I.e. remove
            # colors with varying strengths.
            iaa.Grayscale(alpha=(0.0, 1.0)),

            # In some images move pixels locally around (with random
            # strengths).
            # This is, and the next, augmentation is very slow and
            # therefore, unfortunately, not used.
            # sometimes(
            #     iaa.ElasticTransformation(alpha=(0.5, 3.5), sigma=0.25)
            # ),

            # In some images distort local areas with varying strength.
            # sometimes(iaa.PiecewiseAffine(scale=(0.01, 0.05)))
        ],
        # Do all of the above augmentations in random order
        random_order=True
        )
    ],
    # Do all of the above augmentations in random order
    random_order=True
)
```

C.3 Realistic Augmentations

The following augmentation sequence only applies transformations that could be considered natural. That is, the produced images should be somewhat realistic, and not too heavily augmented. This augmentation scheme was therefore applied in all training epochs when used.

```
def geometric_augmentations():
    return iaa.Sequential([
        iaa.Fliplr(0.5),
        iaa.Flipud(0.5),
        iaa.Sometimes(0.5, iaa.Affine(
            scale={"x": (0.9, 1.1), "y": (0.9, 1.1)},
            translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
            rotate=(-90, 90),
            shear=(-16, 16),
            cval=(0, 255),
        )),
        iaa.Sometimes(0.5, iaa.Crop(percent=(0, 0.3))),
    ])

def non_geometric_augmentations():
    sometimes = lambda aug: iaa.Sometimes(0.25, aug)
    return iaa.Sequential([
        sometimes(iaa.Multiply((0.5, 1.5))),
        sometimes(iaa.GaussianBlur(sigma=(0, 2.0))),
        sometimes(iaa.MultiplyHue((0.9, 1.1))),
        sometimes(iaa.MultiplySaturation((0.9, 1.1))),
        sometimes(iaa.Grayscale(alpha=(0.0, 0.5))),
        sometimes(iaa.ChangeColorTemperature((4000, 6000))),
        sometimes(iaa.GammaContrast((0.5, 2.0)))
    ], random_order=True)


realistic_augmentation = iaa.Sequential([
        geometric_augmentations(),
        non_geometric_augmentations()
    ], random_order=True)
```

Appendix D

Code for PSPNet

See GitHub implementation by Divam Gupta for full framework [20].

D.1 Load and Train Model

```
import imgaug as ia
import imgaug.augmenters as iaa
import matplotlib.pyplot as plt

import os

from keras_segmentation.predict import model_from_checkpoint_path, evaluate
from keras_segmentation.models.model_utils import transfer_weights
from keras_segmentation.models.pspnet import pspnet_50
from keras_segmentation.models.pspnet import pspnet_101
from keras_segmentation.pretrained import pspnet_50_ADE_20K


train_new_model = True

if train_new_model:
    pretrained_model = pspnet_50_ADE_20K()
    model = pspnet_50(n_classes=2)
    transfer_weights(pretrained_model, model)
else:
    checkpoints_path = os.path.join('checkpoints', 'pspnet')
    model = model_from_checkpoint_path(checkpoints_path)

EPOCHS = 70
BATCH_SIZE = 2
```

```
dataset = 'dataset'

history = model.train(
    verify_dataset=False,
    batch_size=BATCH_SIZE,
    validate=True,
    steps_per_epoch=508//BATCH_SIZE,
    val_steps_per_epoch=50//BATCH_SIZE,
    train_images=os.path.join(dataset, 'train_images'),
    train_annotations=os.path.join(dataset, 'train_segmentation'),
    val_images=os.path.join(dataset, 'val_images'),
    val_annotations=os.path.join(dataset, 'val_segmentation'),
    checkpoints_path=os.path.join('checkpoints', 'pspnet'),
    epochs=EPOCHS,
    do_augment=False,
    optimizer_name='adadelta',
    augmentation_name='aug_heavy'
)
```

D.2 Evaluation, Prediction and Visualization

```
import glob
import random
import json
import os
import six

import cv2
import numpy as np
from tqdm import tqdm

from .train import find_latest_checkpoint
from .data_utils.data_loader import get_image_array,
get_segmentation_array, DATA_LOADER_SEED,
class_colors, get_pairs_from_paths
from .models.config import IMAGE_ORDERING

random.seed(DATA_LOADER_SEED)

local_class_colors = [(0, 0, 0), (0, 0, 255)]

mask_rcnn_colors = local_class_colors + class_colors
```

```
def model_from_checkpoint_path(checkpoints_path):

    from .models.all_models import model_from_name
    assert (os.path.isfile(checkpoints_path+"_config.json")
            ), "Checkpoint not found."
    model_config = json.loads(
        open(checkpoints_path+"_config.json", "r").read())
    latest_weights = find_latest_checkpoint(checkpoints_path)
    assert (latest_weights is not None), "Checkpoint not found."
    model = model_from_name[model_config['model_class']](
        model_config['n_classes'],
        input_height=model_config['input_height'],
        input_width=model_config['input_width'])
    print("loaded weights ", latest_weights)
    model.load_weights(latest_weights)
    return model


def get_colored_segmentation_image(seg_arr, n_classes,
                    colors=class_colors):
    output_height = seg_arr.shape[0]
    output_width = seg_arr.shape[1]

    seg_img = np.zeros((output_height, output_width, 3))

    print("N_CLASSES =", n_classes)

    for c in range(n_classes):
        seg_arr_c = seg_arr[:, :] == c
        seg_img[:, :, 0] += ((seg_arr_c)*(colors[c][0])).astype('uint8')
        seg_img[:, :, 1] += ((seg_arr_c)*(colors[c][1])).astype('uint8')
        seg_img[:, :, 2] += ((seg_arr_c)*(colors[c][2])).astype('uint8')

    return seg_img


def get_legends(class_names, colors=class_colors):

    n_classes = len(class_names)
    legend = np.zeros(((len(class_names) * 25) + 25, 125, 3),
                      dtype="uint8") + 255

    class_names_colors = enumerate(zip(class_names[:n_classes],
                                       colors[:n_classes]))
```

```
    for (i, (class_name, color)) in class_names_colors:
        color = [int(c) for c in color]
        cv2.putText(legend, class_name, (5, (i * 25) + 17),
                    cv2.FONT_HERSHEY_COMPLEX, 0.5, (0, 0, 0), 1)
        cv2.rectangle(legend, (100, (i * 25)), (125, (i * 25) + 25),
                      tuple(color), -1)

    return legend


def overlay_seg_image(inp_img, seg_img):
    orininal_h = inp_img.shape[0]
    orininal_w = inp_img.shape[1]
    seg_img = cv2.resize(seg_img, (orininal_w, orininal_h))

    fused_img = (inp_img/2 + seg_img/2).astype('uint8')
    return fused_img


def concat_lenends(seg_img, legend_img):

    new_h = np.maximum(seg_img.shape[0], legend_img.shape[0])
    new_w = seg_img.shape[1] + legend_img.shape[1]

    out_img = np.zeros((new_h, new_w, 3)).astype('uint8') +
    legend_img[0, 0, 0]

    out_img[:legend_img.shape[0], : legend_img.shape[1]] =
    np.copy(legend_img)
    out_img[:seg_img.shape[0], legend_img.shape[1]:] =
    np.copy(seg_img)

    return out_img


def visualize_segmentation(seg_arr, inp_img=None, n_classes=None,
                           colors=class_colors, class_names=None,
                           overlay_img=False, show_legends=False,
                           prediction_width=None, prediction_height=None):

    if n_classes is None:
        n_classes = np.max(seg_arr)+1


    seg_img = get_colored_segmentation_image(seg_arr, n_classes,
```

```
            colors=colors)

    if inp_img is not None:
        orininal_h = inp_img.shape[0]
        orininal_w = inp_img.shape[1]
        seg_img = cv2.resize(seg_img, (orininal_w, orininal_h))

    if (prediction_height is not None) and (prediction_width is not None):
        seg_img = cv2.resize(seg_img, (prediction_width,
        prediction_height))
        if inp_img is not None:
            inp_img = cv2.resize(inp_img,
                                 (prediction_width, prediction_height))

    if overlay_img:
        assert inp_img is not None
        seg_img = overlay_seg_image(inp_img, seg_img)

    if show_legends:
        assert class_names is not None
        legend_img = get_legends(class_names, colors=colors)

        seg_img = concat_lenends(seg_img, legend_img)

    return seg_img


def predict(model=None, inp=None, out_fname=None,
            checkpoints_path=None, overlay_img=False,
            class_names=None, show_legends=False, colors=class_colors,
            prediction_width=None, prediction_height=None):

    if model is None and (checkpoints_path is not None):
        model = model_from_checkpoint_path(checkpoints_path)

    assert (inp is not None)
    assert ((type(inp) is np.ndarray) or
    isinstance(inp, six.string_types)),\
        "Input should be the CV image or the input file name"

    if isinstance(inp, six.string_types):
        inp = cv2.imread(inp)

    assert len(inp.shape) == 3, "Image should be h,w,3 "
```

```
    output_width = model.output_width
    output_height = model.output_height
    input_width = model.input_width
    input_height = model.input_height
    n_classes = model.n_classes

    x = get_image_array(inp, input_width, input_height,
                        ordering=IMAGE_ORDERING)
    pr = model.predict(np.array([x]))[0]
    pr = pr.reshape((output_height,
    output_width, n_classes)).argmax(axis=2)

    seg_img = visualize_segmentation(pr, inp, n_classes=n_classes,
                colors=colors, overlay_img=overlay_img,
                show_legends=show_legends,
                class_names=class_names,
                prediction_width=prediction_width,
                prediction_height=prediction_height)

    if out_fname is not None:
        cv2.imwrite(out_fname, seg_img)

    return pr


def predict_multiple(model=None, inps=None, inp_dir=None, out_dir=None,
                     checkpoints_path=None, overlay_img=False,
                     class_names=None, show_legends=False,
                     colors=local_class_colors,
                     prediction_width=None, prediction_height=None):

    if model is None and (checkpoints_path is not None):
        model = model_from_checkpoint_path(checkpoints_path)

    if inps is None and (inp_dir is not None):
        inps = glob.glob(os.path.join(inp_dir, "*.jpg")) + glob.glob(
            os.path.join(inp_dir, "*.png")) + \
            glob.glob(os.path.join(inp_dir, "*.jpeg"))
        inps = sorted(inps)

    assert type(inps) is list

    all_prs = []

    for i, inp in enumerate(tqdm(inps)):
```

```
        if out_dir is None:
            out_fname = None
        else:
            if isinstance(inp, six.string_types):
                out_fname = os.path.join(out_dir, os.path.basename(inp))
            else:
                out_fname = os.path.join(out_dir, str(i) + ".jpg")

        pr = predict(model, inp, out_fname,
                     overlay_img=overlay_img, class_names=class_names,
                     show_legends=show_legends, colors=colors,
                     prediction_width=prediction_width,
                     prediction_height=prediction_height)

        all_prs.append(pr)

    return all_prs


def evaluate(model=None, inp_images=None, annotations=None,
             inp_images_dir=None,
             annotations_dir=None, checkpoints_path=None):

    if model is None:
        assert (checkpoints_path is not None),\
                "Please provide the model or the checkpoints_path"
        model = model_from_checkpoint_path(checkpoints_path)

    if inp_images is None:
        assert (inp_images_dir is not None),\
                "Please provide inp_images or inp_images_dir"
        assert (annotations_dir is not None),\
            "Please provide inp_images or inp_images_dir"

        paths = get_pairs_from_paths(inp_images_dir, annotations_dir)
        paths = list(zip(*paths))
        inp_images = list(paths[0])
        annotations = list(paths[1])

    assert type(inp_images) is list
    assert type(annotations) is list

    tp = np.zeros(model.n_classes)
    fp = np.zeros(model.n_classes)
    fn = np.zeros(model.n_classes)
```

```
    n_pixels = np.zeros(model.n_classes)

    for inp, ann in tqdm(zip(inp_images, annotations)):
        pr = predict(model, inp)
        gt = get_segmentation_array(ann, model.n_classes,
                model.output_width, model.output_height,
                no_reshape=True)
        gt = gt.argmax(-1)
        pr = pr.flatten()
        gt = gt.flatten()

        for cl_i in range(model.n_classes):

            tp[cl_i] += np.sum((pr == cl_i) * (gt == cl_i))
            fp[cl_i] += np.sum((pr == cl_i) * ((gt != cl_i)))
            fn[cl_i] += np.sum((pr != cl_i) * ((gt == cl_i)))
            n_pixels[cl_i] += np.sum(gt == cl_i)

    cl_wise_score = tp / (tp + fp + fn + 0.000000000001)
    n_pixels_norm = n_pixels / np.sum(n_pixels)
    frequency_weighted_IU = np.sum(cl_wise_score*n_pixels_norm)
    mean_IU = np.mean(cl_wise_score)

    return {
        "frequency_weighted_IU": frequency_weighted_IU,
        "mean_IU": mean_IU,
        "class_wise_IU": cl_wise_score
    }
```

Appendix E

Code for Mask R-CNN

See GitHub implementation by Waleed Abdulla for full framework [1].

E.1 Load and Train Model

```
"""

Usage: import the module (see Jupyter notebooks for examples), or run from
the command line as such:
    # Train a new model starting from pre-trained COCO weights
    python3 damage_detection.py train
    --dataset=/path/to/dataset --weights=coco
    # Resume training a model that you had trained earlier
    python3 damage_detection.py train
    --dataset=/path/to/dataset --weights=last
    # Train a new model starting from ImageNet weights
    python3 damage_detection.py train
    --dataset=/path/to/dataset --weights=imagenet
    # Apply inference to an image
    python3 damage_detection.py test
    --weights=/path/to/weights/file.h5 --image=<URL or path to file>
"""

import os
import sys
import json
import datetime
import numpy as np
import skimage.draw
from imgaug import augmenters as iaa
```

```
# Root directory of the project
ROOT_DIR = os.path.abspath("../../")

# Import Mask RCNN
sys.path.append(ROOT_DIR)  # To find local version of the library
from mrcnn.config import Config
from mrcnn import model as modellib, utils, visualize

from keras.callbacks import LearningRateScheduler

# Path to trained weights file
COCO_WEIGHTS_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")

# Directory to save logs and model checkpoints, if not provided
# through the command line argument --logs
DEFAULT_LOGS_DIR = os.path.join(ROOT_DIR, "logs")

############################################################
#  Configurations
############################################################


class DamageConfig(Config):
    """Configuration for training on the toy  dataset.
    Derives from the base Config class and overrides some values.
    """
    # Give the configuration a recognizable name
    NAME = "damage"

    # We use a GPU with 12GB memory, which can fit two images.
    # Adjust down if you use a smaller GPU.
    IMAGES_PER_GPU = 1

    # Number of classes (including background)
    NUM_CLASSES = 1 + 1 # +1 for background

    # Number of training steps per epoch
    STEPS_PER_EPOCH = 508
    VALIDATION_STEPS = 50

    # Skip detections with < 90% confidence
    DETECTION_MIN_CONFIDENCE = 0.9

    #LEARNING_RATE = 0.001 # 0.001
    OPTIMIZER = "SGD"  # default is SGD
```

```
    # Reducing memory
    BACKBONE = "resnet50"
    IMAGE_MIN_DIM = 512 # similar to PSPNet
    IMAGE_MAX_DIM = 512
    MAX_GT_INSTANCES = 50

############################################################
#  Dataset
############################################################

class DamageDataset(utils.Dataset):

    def load_damage(self, dataset_dir, subset):
        """Load a subset of the damage dataset.
        dataset_dir: Root directory of the dataset.
        subset: Subset to load: train or val
        """
        # Add classes.

        self.add_class("damage", 1, "corrosion") # +1 since 0=BG

        # Train or validation dataset
        assert subset in ["train", "val"]
        dataset_dir = os.path.join(dataset_dir, subset)
        image_ids = next(os.walk(dataset_dir))[1]

        for image_id in image_ids:
            image_example_dir = os.path.join(dataset_dir, image_id)
            (_, _, file_names) = next(os.walk(image_example_dir))
            file_name = file_names[0]
            image_path = os.path.join(image_example_dir, file_name)
            image = skimage.io.imread(image_path)
            height, width = image.shape[:2]

            self.add_image(
                "damage",
                image_id=image_id,  # use ids from LabeBox
                path=image_path,
                width=width, height=height)


    def load_mask(self, image_id):
        """Generate instance masks for an image.
        Returns:
```

```
        masks: A bool array of shape [height, width, instance count] with
            one mask per instance.
        class_ids: a 1D array of class IDs of the instance masks.
        """
        info = self.image_info[image_id]
        # Get mask directory from image path
        mask_dir = os.path.join(os.path.dirname(info['path']), "masks")

        # Read mask files from .png images
        mask = []
        for f in next(os.walk(mask_dir))[2]:
            if f.endswith(".png"):
                m = skimage.io.imread(os.path.join(mask_dir, f),
                as_gray=True).astype(np.bool)
                mask.append(m)

        mask = np.stack(mask, axis=-1)
        class_ids = np.ones([mask.shape[-1]], dtype=np.int32)

        # Return mask, and array of class IDs of each instance.
        return mask.astype(np.bool), class_ids

    def load_mask_semantic(self, image_id):
        """Generate instance masks for an image.
        Returns:
        masks: A bool array of shape [height, width, instance count] with
            one mask per instance.
        class_ids: a 1D array of class IDs of the instance masks.
        """
        info = self.image_info[image_id]
        # Get mask directory from image path
        mask_dir = os.path.join(os.path.dirname(info['path']), "masks")

        # Read mask files from .png images
        mask = None
        mask_initialized = False
        for f in next(os.walk(mask_dir))[2]:
            if f.endswith(".png"):
                if not mask_initialized:
                    mask = skimage.io.imread(os.path.join(mask_dir, f),
                    as_gray=True).astype(np.bool)
                    mask_initialized = True
                else:
                    m = skimage.io.imread(os.path.join(mask_dir, f),
                    as_gray=True).astype(np.bool)
```

```
                    mask = np.add(mask, m)

        mask = [mask]
        mask = np.stack(mask, axis=-1)
        class_ids = np.ones([mask.shape[-1]], dtype=np.int32)

        # Return mask, and array of class IDs of each instance.
        return mask.astype(np.bool), class_ids


    def image_reference(self, image_id):
        """Return the path of the image."""
        info = self.image_info[image_id]
        if info["source"] == "damage":
            return info["path"]
        else:
            super(self.__class__, self).image_reference(image_id)


def lr_scheduler(epoch, lr):
    decay_rate = 0.5
    decay_step = 10
    if epoch % decay_step == 0 and epoch:
        return lr * decay_rate
    return lr


def train(model):
    """Train the model."""
    # Training dataset.
    dataset_train = DamageDataset()
    dataset_train.load_damage(args.dataset, "train")
    dataset_train.prepare()

    # Validation dataset
    dataset_val = DamageDataset()
    dataset_val.load_damage(args.dataset, "val")
    dataset_val.prepare()

    # Custom callbacks
    change_lr = LearningRateScheduler(lr_scheduler, verbose=1)

    print("Training network heads")
    model.train(dataset_train, dataset_val,
                learning_rate=config.LEARNING_RATE,
```

```
                epochs=60,
                augmentation=aug_flipping(), # see appendix C
                custom_callbacks=[change_lr],
                layers='all')


############################################################
#  Interface
############################################################

if __name__ == '__main__':
    import argparse

    # Parse command line arguments
    parser = argparse.ArgumentParser(
        description='Train Mask R-CNN to detect construction damages.')
    parser.add_argument("command",
                        metavar="<command>",
                        help="'train', 'test' or 'evaluate'")
    parser.add_argument('--dataset', required=False,
                        metavar="/path/to/dataset/",
                        help='Directory of the construction damage dataset')
    parser.add_argument('--weights', required=True,
                        metavar="/path/to/weights.h5",
                        help="Path to weights .h5 file or 'coco'")
    parser.add_argument('--logs', required=False,
                        default=DEFAULT_LOGS_DIR,
                        metavar="/path/to/logs/",
                        help='Logs and checkpoints directory (default=logs/)')
    parser.add_argument('--image', required=False,
                        metavar="path or URL to image",
                        help='Image to apply inference on')
    args = parser.parse_args()

    # Validate arguments
    if args.command == "train":
        assert args.dataset, "Argument --dataset is required for training"
    elif args.command == "test":
        assert args.image, "Provide --image to apply inference on"
    elif args.command == "evaluate":
        assert args.dataset, "Argument --dataset is required for training"
        assert args.weights, "Argument --weights is required for evaluation"

    print("Weights: ", args.weights)
    print("Dataset: ", args.dataset)
```

```
print("Logs: ", args.logs)

# Configurations
if args.command == "train":
    config = DamageConfig()
else:
    class InferenceConfig(DamageConfig):
        # Set batch size to 1 since we'll be running inference on
        # one image at a time. Batch size = GPU_COUNT * IMAGES_PER_GPU
        GPU_COUNT = 1
        IMAGES_PER_GPU = 1
    config = InferenceConfig()
config.display()

# Create model
if args.command == "train":
    model = modellib.MaskRCNN(mode="training", config=config,
                              model_dir=args.logs)
elif args.command == "test":
    model = modellib.MaskRCNN(mode="inference", config=config,
                              model_dir=args.logs)
elif args.command == "evaluate":
    model = modellib.MaskRCNN(mode="inference", config=config,
                              model_dir=args.logs)

# Select weights file to load
if args.weights.lower() == "coco":
    weights_path = COCO_WEIGHTS_PATH
    # Download weights file
    if not os.path.exists(weights_path):
        utils.download_trained_weights(weights_path)
elif args.weights.lower() == "last":
    # Find last trained weights
    weights_path = model.find_last()
elif args.weights.lower() == "imagenet":
    # Start from ImageNet trained weights
    weights_path = model.get_imagenet_weights()
else:
    weights_path = args.weights

# Load weights
print("Loading weights ", weights_path)
if args.weights.lower() == "coco":
    # Exclude the last layers because they require a matching
    # number of classes
```

```
        model.load_weights(weights_path, by_name=True, exclude=[
            "mrcnn_class_logits", "mrcnn_bbox_fc",
            "mrcnn_bbox", "mrcnn_mask"])
    else:
        model.load_weights(weights_path, by_name=True)

    # Train, test or evaluate
    if args.command == "train":
        train(model)
    elif args.command == "test":
        apply_interference(model, image_path=args.image)
    elif args.command == "evaluate":
        evaluate_model(model)
    else:
        print("'{}' is not recognized. "
              "Use 'train' or 'test'".format(args.command))
```

E.2 Evaluation, Prediction and Visualization

```
def apply_interference(model, image_path=None):
    # Load image
    image = skimage.io.imread(image_path)

    # Run detection
    results = model.detect([image], verbose=1)

    # Visualize results
    r = results[0]
    class_names = ["BG", "corrosion"]
    visualize.display_instances(image, r['rois'], r['masks'],
    r['class_ids'], class_names, r['scores'])


def evaluate_model(model):
    dataset_val = DamageDataset()
    dataset_val.load_damage(args.dataset, "val")
    dataset_val.prepare()

    image_ids = dataset_val.image_ids

    iou_corr_list = []
    iou_bg_list = []
```

```
    for image_id in image_ids:

        image = dataset_val.load_image(image_id)
        mask_gt, class_ids = dataset_val.load_mask(image_id)

        mask_gt = combine_masks_to_one(mask_gt)

        result = model.detect([image], verbose=0)[0]

        predicted_masks = result["masks"]
        if predicted_masks.shape[-1] == 0:
            continue
        mask_pred = combine_masks_to_one(predicted_masks)

        iou_corr = compute_overlaps_masks(mask_gt, mask_pred)[0][0]
        iou_bg = compute_overlaps_masks(mask_gt, mask_pred, BG=True)[0][0]

        print(image_id, "IoU =", (iou_corr, iou_bg))
        iou_corr_list.append(iou_corr)
        iou_bg_list.append(iou_bg)

    mean_corr_iou = sum(iou_corr_list) / len(iou_corr_list)
    mean_bg_iou = sum(iou_bg_list) / len(iou_bg_list)
    print("Total mean values")
    print("Corrosion IoU =", mean_corr_iou)
    print("BG IoU=", mean_bg_iou)
    print("Mean IoU =", (mean_corr_iou + mean_bg_iou) / 2)


def combine_masks_to_one(masks):
    combined_mask = masks[:, :, 0]

    for i in range(masks.shape[-1]):
        combined_mask += masks[:, :, i]

    return np.expand_dims(combined_mask, 2)


def compute_overlaps_masks(masks1, masks2, BG = False):
    """Computes IoU overlaps between two sets of masks.
    masks1, masks2: [Height, Width, instances]
    """

    # If either set of masks is empty return empty result
    if masks1.shape[-1] == 0 or masks2.shape[-1] == 0:
```

```
        return np.zeros((masks1.shape[-1], masks2.shape[-1]))
    # flatten masks and compute their areas
    if BG:
        masks1 = np.reshape(masks1 < .5,
        (-1, masks1.shape[-1])).astype(np.float32)
        masks2 = np.reshape(masks2 < .5,
        (-1, masks2.shape[-1])).astype(np.float32)
    else:
        masks1 = np.reshape(masks1 > .5,
        (-1, masks1.shape[-1])).astype(np.float32)
        masks2 = np.reshape(masks2 > .5,
        (-1, masks2.shape[-1])).astype(np.float32)

    area1 = np.sum(masks1, axis=0)
    area2 = np.sum(masks2, axis=0)

    # intersections and union
    intersections = np.dot(masks1.T, masks2)
    union = area1[:, None] + area2[None, :] - intersections
    overlaps = intersections / union

    return overlaps



def overlay_prediction_single_maskrcnn(pr=None, inp=None,
out_dir=None, overlay_img=True, colors=mask_rcnn_colors):

    if isinstance(pr, six.string_types):
        pr = cv2.imread(pr, 0)

    if isinstance(inp, six.string_types):
        out_fname = os.path.join(out_dir, os.path.basename(inp))
        inp = cv2.imread(inp[:-4] + ".jpg")


    assert len(inp.shape) == 3, "Image should be h,w,3 "

    seg_img = visualize_segmentation(pr, inp, colors=colors,
    overlay_img=overlay_img,)

    if out_fname is not None:
        cv2.imwrite(out_fname, seg_img)

    return pr
```

```
def overlay_predictions_all_maskrcnn(pr_dir=None, inp_dir=None,
out_dir=None,
overlay_img=True, colors=mask_rcnn_colors):
    for f in next(os.walk(pr_dir))[2]:
        print("File name =", f)
        if "DS_Store" in f:
            print("Skipping DS_Store file")
            continue
        if f.endswith(".png"):
            pr = os.path.join(pr_dir, f)
            inp = os.path.join(inp_dir, f)
            overlay_prediction_single_maskrcnn(pr, inp,
            out_dir, overlay_img, colors)
```