Norwegian University of Science and Technology

Project report fall 2018

An investigation into using state-of-the-art convolutional neural network object detectors as real-time fish detectors on underwater grayscale images

---

Storm Jaran Bruvoll Westlie

Trondheim, December 18, 2018

SUPERVISOR:

Annette Stahl, NTNU

Problem formulation

The problem formulation is as follows.

1. Create a suitable image set with boundary box data which may be used to train object detection neural networks.
2. Provide one or several object detection neural network which is trained on these images and may be used to start a recording whenever a fish is detected, which operates at "real-time" (say 8 FPS or higher).
3. Provide a detailed project report detailing all steps of the development (results, usage, setup, etc.).

Abstract

Camera vision is a state-of-the-art technique which allows observation and analysis of fish cultures within fish cages. It is needed in order to estimate bio mass and to monitor the general health state of the fish population. A key component in the process is to determine when a fish is present within the cameras field of view and detect it within the time frame it is present in order to perform calculations and estimations on the fish population. This paper presents a comparison of the performance and detections results of three different state-of-the-art convolutional neural networks for object detection. The networks were trained and tested through Tensorflow's object detection API on a dataset consisting of 2658 grayscale images of fish. Result vise, FRCNN Inception v2 claimed the best detection results with a recall of 0.848 and mean average precision of 0.853. The best performing network was SSD Mobilenet v2 with an estimated 4.29 frames per second. None of the tested networks managed to operate in real-time which was defined as operating at 8 or more frames per second.

Challenges, solutions and knowledge gained

The first dataset received from Optoscale was presented in a different format than mentioned later in the report. Only one image per fish was received, where the grayscale image of the fish was encoded into the green channel and the binary image in the blue channel. Before sending this data, it was JPEG-compressed to save space and as a result the once independent data channels became dependent, as information from the binary image spread to the grayscale image and vice versa through compression. As a result, when 5 different convolutional neural networks had been trained in the beginning of November showing promising validation results, the author noticed that only some visible fish would be detected, whereas other more visible fish were given a probability of less than 0.01 percent of being a fish. It turned out that the convolutional neural networks had learned the patterns of JPEG-compression and not those of fish. As neither validation loss nor recall or mAP results showed any signs of overfitting, the author can only assume that the image compression corrupted the data in such a way that it could no longer be used to train convolutional neural networks to detect fish.

The solution to the issue was to start anew by requesting new data from Optoscale in a different format that separates the fish image from the binary images.

From this challenge, the author learned the importance of data integrity and convolutional neural networks sensitivity towards noise. Data needs to be kept as close as possible to its original format from the sensor and various methods used to manipulate the data in later stages may influence detection results. In addition, the trainer of neural networks may be fooled by the numbers that are supposed to measure the networks ability to generalize on new data. As a result, it is important to manually inspect results and not rely completely on automatically generated results.

Another issue that was encountered quite frequently was related to the quality of the Tensorflow Object detection API. As the API is free of charge without generating any revenue to its makers, updates may change key parameters without it being documented. For instance, when trying to set the interval at which the neural network should be evaluated against the validation set, the standard command mentioned in the documentation seemed to have no effect. Another unintentional effect was that some of the configuration files for different convolutional neural networks would not load as the framework had been updated, but not the configuration files themselves. Lastly, when changing the hyperparameters of methods within the CNNs, one needs the correct keyword to do so in the configuration file. This presented an issue as to the author’s knowledge, some of these keywords have not been officially documented.

The solution to these issues where found through thorough googling. As it turned out, several others had faced these issues and a knowledgeable user mentioned the solution of editing the python scripts of the framework, thus circumventing the configuration files. Consequently, the author edited and scoured files python files in the tensorflow-gpu framework in order to solve the issues mentioned above.

From this issue the author learned that free of charge services such as the object detection API may be incomplete to some extent and thus it might have been wiser to use the inbuilt functions of TensorFlow to create a tailor-made solution and not rely on the API itself. Another piece of wisdom one should bring along is that no matter how complex the issue or error message might be, Google may often point the user in the right direction towards a solution.

Contents

List of figures

List of tables

Chapter 1: Introduction

Motivation

Monitoring the state of a fish population in a fish pen in an important aspect within the industry of fish farming. As the fish live under water, human beings struggle to gather vital information regarding the fish' health and environment which are detrimental factors to any company in the aquaculture industry. If not monitored properly, the worst-case scenario would be the mass death and suffering of the fish population, whereas the company most likely will face catastrophic economic backlashes.

There exist technologies which enable the monitoring of fish populations, however many of these either involve high risk with regards to human life or direct intrusion into the fish population, thus affecting the fish's health. One promising technology, which does not involve risk to human life or unwanted side effects is the underwater camera. By submerging a camera into a fish pen, one can extract vital information on the fish population. Although it is not able to monitor the entirety of the fish pen, cameras can over time generate a strong image of the general state of the fish population. However, one problem arises, how would one extract the information gathered by the camera? One solution would be to assign a fish pen worker to manually inspect each image as it is loaded from the camera, however it is both tiresome and time-consuming. Another solution, however, would be to employ state-of-the-art computer vision techniques to automate the process entirely.

This is the background for this study, as a company named Optoscale has created such a solution. However, they would like to investigate further into techniques that may detect a fish when it is present in front of the camera. Consequently, this paper will focus on various traditional and state-of-the-art methods of doing so, in order to improve the estimates on the fish populations state. Hopefully increasing the wealth of the fish, as well as the wealth of the companies that take care of them.

Outline

Chapter	Purpose
2	Introduces the fish data set received from Optoscale and preliminary thoughts on weaknesses in the data.
3	An overview of various techniques for image object detection under water.
4-5	An overview of which libraries, tools and hardware this paper has been dependent on.
6	An introduction into the field of convolutional neural networks which focuses on their inner workings, how they learn and the metrics they are evaluated on.
7	A summary of what had to be done in order to train and evaluate the CNNs.
8	A presentation of the results from each respective CNN.
9	Discusses the results in terms of what they mean, how they came to be and possible solutions.
10-11	Concludes the report as a whole and suggests topics for future work.

Chapter 2: Materials

The dataset received from Optoscale

The image set contained around 38000 usable images of fish. It contained two types of files for each image of fish. The first file was the image of the fish itself with structured light on its surface and the other was one or several binary images depicting where Optoscale's current fish detection algorithm had detected a fish in the image. See the figure 1 and 2 below.

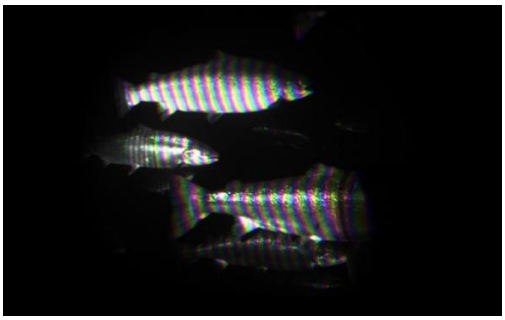

Figure 1: Example of a fish image received from Optoscale.

Figure 2: Example of a binary image denoting the position of a fish in an image.

An aspect of the dataset that will influence later results is that in some images, only one visible image is marked through a binary image, however other candidates that also should have been marked are also present. This may particularly influence the precision measure, as the network most likely will detect these fish, and as they are not marked as fish they will be marked as a false positive although that is not the case.

In addition, many of the images were a result of sequential tracking of one fish from one frame to the next. Since the camera operates at about 200 frames per second, some images would look roughly the same. This may cause issues with regards to overfitting.

Chapter 3: Literature study

Introduction

In order to get an overview of which techniques might be applicable to fish detection through camera vision a literature study on the subject has been conducted. It covers traditional methods and new research on neural networks and finishes of with a comparison of the two with regards to performance, detection results and weaknesses. To the author's knowledge, few papers have been released on the subject of fish detection and as a result the study will also rely on results from fish recognition as this process necessarily also needs a form of fish detection in order to classify at a later stage.

Background

Early developments within the use of camera vision to detect and recognize fish consisted of manually creating shape and color descriptors, deriving first and second moments of the image that describes what is on it and a simple version of template matching (Strachan, Nesvadba and Allen, 1990). The algorithms input was a clean image file containing the silhouette of a certain type of fish and nothing else. Although some of the results are satisfiable, for instance the shape descriptor method reached 90% reliability which in today's terminology translates to 90 % precision, the methods only worked, at that point in time, on images of a fish' silhouette without noise or any background texture.

More advanced methods that are in use today and expand on these concepts manage to isolate the pixels belonging to the fish from more complex scenes containing more objects and noise, whether it is the silhouette or the entirety of the fish itself, using an image segmentation method. The isolated pixels, or features found from the isolated pixels after passing them as input to a feature extractor, are subsequently passed as input to a classifier in order to determine whether the isolated object could be a fish. To exemplify, (Giordano, Palazza and Spampinato, 2016) used an advanced type of background subtraction in combination with a foreground detection method as the image segmentation method. Then features of the blobs found from background subtraction such as luminosity and "motion objectness" were extracted and finally compared to known features of fish in order to determine the probability of the isolated pixels in fact being a fish caught on camera. Another way of doing it was shown by (Boudhane and Nsiri, 2016). They used a Poisson-Gauss mixture filter to eliminate noise created by the transmission of image data from the camera to the computer. In order to segment the different parts of the image, the mean shift clustering algorithm was used. Furthermore, a log likelihood ratio test served as the classifier on the different clusters found. Finally, (Rodríguez *et al.*, 2015) used a dynamic background subtraction method to separate moving objects from the background. Moreover, a version of a canny edge detector was used

to sample features from the foreground objects and the Otsu method was used to eliminate white background pixels. Lastly, manually created reference features of fish were compared to the output of the feature extractor in order to determine whether a fish was present or not.

Newer development within fish detection occurs in the field of research on convolutional neural networks, or CNNs for short. CNNs for object detection work in a vastly different way than previously mentioned methods. In contrast to firstly identifying objects of interest and then finding its features, CNNs either scan the entire image looking for features of interest or make an educated guess as to where the object may be that is later refined. Later this information is used to conclude whether the object of interest was present and if so where in the image (*"Convolutional Neural Networks (CNNs / ConvNets)", cs231n, Stanford University*, 2018a).The default structure of CNNs is firstly a model which is used to detect features and secondly, if one wants to detect where in the image the object may be found, an object detector responsible for proposing where in the image an object of interest is, what it is and how confident it is in the prediction. Further, CNNs can learn by themselves what to look for when distinguishing an object of interest from everything else that may be present in the image (*"Convolutional Neural Networks (CNNs / ConvNets)", cs231n, Stanford University*, 2018a), given enough image data of an object, through the process of training (Soulié, 1991). (Krizhevsky, Sutskever and Hinton, 2012a) proposed one of the first successful CNNs used as an image classifier, firstly competing in the object detection competition known as ImageNet Large Scale Visual Recognition Competition (ILSVRC) and the winner of the 2012 version of the competition (Russakovsky *et al.*, 2015). Further development on the field saw the introduction of various other models for image classification such as Inception (Szegedy *et al.*, 2014), ResNet (He *et al.*, 2015) and MobileNet (Howard *et al.*, 2017) who vastly outperform the original AlexNet model (Canziani, Paszke and Eugenio, 2016, Huang *et al.*, 2016).

Furthermore, said CNNs are only able to classify images with regards to what objects are present, without giving information on the whereabouts of the object. In order to predict where those objects are in the image, one needs to add a region proposal method on top of the CNN (Lu, Du and Chang, 2018). A region proposal method may be as simple as creating a fixed set of boundary boxes on the image and look for objects in them, to training a region-based network that gives probabilities of where an object of interest might be present. Examples of region proposal methods that are in use today are the Single Shot Detector (Liu *et al.*, 2015), which uses a fixed set of boundary boxes to estimate the location of an object in an image, and Faster Region-based convolutional network (Ren *et al.*, 2015), which uses a region proposal network that needs to be trained along with the neural net model, in order to predict the most likely place in an image an object is present.

Method

Search databases

The database Oria by Bibsys has been used in the search of relevant sources. The database covers most of the material present at Norwegian research libraries as well as other open electronically available sources. In addition, Google and Google Scholar was used to broaden the number of sources.

Source criteria

In order to eliminate yet-to-be confirmed or inadequate research the following criteria were essential in deciding whether a source was to be considered or not. Firstly, the article needed either to be peer-reviewed, to have been cited in 10 or more articles, to have been part of a published book or be a part of a course from a renowned university. Secondly, the source had to contain relevant information regarding aspects of fish detection and a result measure which contained information about how well the technique could perform when being used as a fish detector.

The Search

The search was conducted by entering various search words relevant to fish detection into the mentioned sources, in addition to scouring already found sources for new ones, see the table 1 below for search words used and subsequently the sources found from the search query.

Source	Search words	Criteria match	Sources
Oria	Fish detection	Published book, peer-reviewed	(Giordano, Palazza and Spampinato, 2016, Mandal *et al.*, 2018, Ravanbakhsh *et al.*, 2015)
Oria	Fish identification	Peer-reviewed	(Shafait *et al.*, 2016)
Oria	Camera vision fish detection	Peer-reviewed	(Rodríguez *et al.*, 2015)
Oria	Cnn fish detection	Peer-reviewed	(Sun *et al.*, 2018)
Oria	State of the art fish detection	Peer-reviewed	(Boudhane and Nsiri, 2016)
Google Scholar	Fish detection cnn	Citations	(Xiu *et al.*, 2015)

Google Scholar	Underwater fish recognition	Citations	(Hongwei *et al.*, 2015)
Google	Comparison Alexnet	Citations	(Canziani, Paszke and Eugenio, 2016)
Google	Tensorflow Object Detection API	Citations	(Huang *et al.*, 2016)
Google	Alexnet paper	Citations	(Krizhevsky, Sutskever and Hinton, 2012a)
Oria	Neural net training	Published book	(Menéndez de Llano and Bosque, 2010)
(Menéndez de Llano and Bosque, 2010)		Published book	(Soulié, 1991)
Google	Imagenet paper	Citations	(Deng *et al.*, 2009)
Google	Convolutional neural networks	Course at Stanford University	(*"Convolutional Neural Networks (CNNs / ConvNets)", cs231n, Stanford University*, 2018a)
Google	Alexnet imagenet	Description of contest	(ILSVRC)
Google	Alexnet imagenet	Citations	(Russakovsky *et al.*, 2015)
Google	Inception paper	Citations	(Szegedy *et al.*, 2014)
Google	Resnet paper	Citations	(He *et al.*, 2015)
Google	Mobilenet paper	Citations	(Howard *et al.*, 2017)
Oria	Region proposal	Peer-reviewed	(Lu, Du and Chang, 2018)
Google	Ssd paper	Citations	(Liu *et al.*, 2015)
Google	Faster region-based convolutional neural network paper	Citations	(Ren *et al.*, 2015)
Google Scholar	how much data neural network	Citations	(Srivastava *et al.*, 2014)

Oria	transfer learning neural networks	Citations	(Lucena *et al.*, 2017)

Table 1: An overview of the sources found through the literature search.

The sources found where collected in an annotated bibliography made in Word containing the following columns: “link”, “source”, “search words used”, “trustworthiness”, “relevance to project” and “summary”.

Finally, the summary column in addition to extra information extracted directly from the articles where used to create table 1.

Results

Before jumping into the results, a few words used in the result and discussion section must be defined. Firstly, by real-time one means an algorithm able to operate at 8 frames per second or higher. Secondly, when describing the background as simple, one means that the background of the image remains static and barely changes from one image to another. Consequently, a complex background does not remain the same over time and contains major changes from one image to the another.

Table 2 sums up the important information extracted through the literature study. The columns are source, which simply refers to the source of the findings. Data which describes the size of the dataset used, whether the image was colorized or not and a measure of the background complexity. Detection method which describes the methods used in order to perform detection of fish, frames per second which describes how many images the method reportedly managed to process per second. Measure which describes what kind of measure was used to evaluate the method with regards to detection capabilities and results which put a number on the success of the method on detecting fish.

Source	Data	Detection method	Frames per second	Measure used to quantify result	Results
8	30000 images from the Baltic sea Color: Yes Background: Complex	Filtering, mean-shift clustering and Log-Likelihood ratio test	Not Real-time due to denoising	Pixel classification (belonging to fish or not)	Correct classification: ~94 %
(Giordano, Palazza and Spampinato, 2016)	F4K fish videos Color: Yes Background: Complex	Background subtraction algorithm followed by blob features compared to known blob features of fish such as luminosity and size	0.3-1.5 (image size dependant)	Recall and precision	Recall: 97.4 % Precision: 89.4 %
(Ravanbakhsh *et al.*, 2015)	35 fish images from the transfer gate between two cages Color: No Background: Simple	Haar-like classifier	Real-time	Recall (completeness) and precision (correctness)	Recall: 91.4-100 % Precision: 89.6-100 %
(Shafait *et al.*, 2016)	ImageCLEF 2014 fish task (fish manually cropped) Color: Yes	PCA and One-nearest-neighbor	Real-time	Precision	Precision: 71.4-100 % (species dependent)

	Background: Complex				
(Rodríguez *et al.*, 2015)	Live testing on full-scale fishway model Color: No Background: Simple	Background subtraction algorithm, canny edge detection and Otsu region classification	3.2	Recall and precision	Recall: 94 % Precision: 94-95 % (Background dependent)
(Hongwei *et al.*, 2015)	F4K fish videos Color: Yes Background: Complex	Foreground extraction, PCA filter feature extraction and SVM classifier	N/A, however ran on CPU thus indicating not real-time	Classification accuracy	C.a.: 55.56-100 % (species and data size dependent)
(Mandal *et al.*, 2018)	4909 fish images Color: Yes Background: Complex	VGG16, ZF and CNN-M models with FRCNN identifier	Real-time	Mean average precision	mAP: 0.71-0.824
(Sun *et al.*, 2018)	F4K fish videos Color: Yes Background: Complex	Neural networks, SIFT-Fisher, LDA, DeepFish, AlexNet	Real-time	Recall and precision	Recall: 45.84-99.45 % Precision: 48.55-99.68% (Neural net model and species dependent)

(Xiu *et al.*, 2015)	ImageCLEF fish videos Color: Yes Background: Complex	AlexNet and FRCNN	5-8	mAP	0.654-0.892 (species dependant)

Table 2: An overview implementations and results of fish detectors from the literature.

Discussion

Most of the traditional methods seem to excel within their task of detecting fish, however they also show weaknesses within performance. Most recall and precision results are close to or above 90 %. Although each method is tailored towards the environment they operate in, they will be suitable as fish detectors in the environment this task is concerned with due to it being mostly black background with some illumination noise which can be modelled thereafter. However, the performance of each method is questionable. Specifically, it seems that the first filtering step of each algorithm uses too much time and consequently none of the end-to-end methods show results close to real-time performance. The one method, created by (Shafait *et al.*, 2016), is real-time, however the detection was performed on already cropped images of fish. Thus, it lacks perhaps the most vital step which would be to segment the image into its parts or objects if one will.

Further the data set the traditional methods have tackled are in general quite large and comprise of both colorized and grayscale images with either simple or complex backgrounds. Images used later in this report will be in grayscale and thus one knows that the structure behind each method, namely filtering, image segmentation and classification is a potential candidate for the fish detectors tested later in the report.

The neural networks have varying results which can be summarized as either inadequate or great fish detection. Some of the articles, namely (Hongwei *et al.*, 2015, Sun *et al.*, 2018, Xiu *et al.*, 2015), report that the main issue causing inadequate results would be the lack of data and neural net model choice. Most results on species, in which there has been enough data, show recall and precision close to or above 90 %. As for the performance part, all but one neural net implementation ran real-time, that is operating with more than or equal to 8 FPS. (Xiu *et al.*, 2015) comes close. Thus, meeting the requirements set earlier in this report.

Data vise, all datasets contain several thousand images with complex backgrounds and thus must be considered quite large. This is a natural consequence of using neural networks, as they need a lot of data to generalize well on to other data than the training data (Srivastava *et al.*, 2014). The dataset received from Optoscale is large, and as a result this will not pose an issue at later development stages. An interesting aspect regarding the different neural net implementations is that they use datasets containing colorized images, perhaps a result of these images containing more information through having more than 1 color channel. The author tried to find neural net fish detectors taking grayscale images as input, however found none.

When comparing both ways of tackling the issue of fish detection, both show great results when the necessities connected to the size of the image set or environment are present. Thus, there is no clear choice to go for when considering this information. Regarding performance, the neural nets far outperform the traditional methods by a minimum of 4.8 FPS, where not one traditional managed to meet the real-time requirement of 8 FPS or higher. Still, one needs to keep in mind that the performance is hardware and implementation dependent, for instance most neural nets run on the GPU. This might not be the case regarding the traditional methods. To summarize the performance comparison, neural networks clearly come out on top. Next, when comparing data aspects, both ways of detecting fish seem to operate well on large sets of data. The dataset's used in traditional implementations contain both simple and complex backgrounds and background complexity does not seem to affect the results. On the other hand, the articles found on CNN fish detectors all handled complex data and as a result it is unsure how well they will perform on fish images with simple backgrounds. However, there is good reason to believe that this will not be an issue as they perform well on complex backgrounds, and consequently perform well on nearly any background. Moreover, an important difference is the fact that neural nets generally need massive amounts of data in order to avoid overfitting, unless transfer learning is used (Lucena *et al.*, 2017), and later in the test phase. The traditional methods merely need lots of data in the test phase. For this reason, the winner with regards to the data aspect would be the traditional methods.

Finally, an interesting observation is that there seems to be a lack of research on how convolutional neural networks perform as fish detectors on grayscale images.

When making a choice of whether to opt for the traditional method of fish detection versus the newer way of handling the issue through neural networks, all aspects above needs to be considered. Table 3 below summarizes the results for each category.

	Traditional methods	Convolutional neural networks
Results	Good	Good
Performance	Not real-time	Real-time
Data	1. Needs little to none data 2. Handles both simple and complex backgrounds 3. Tested and proven as fish detector on grayscale images	1. Needs a lot of data, especially if transfer learning is not used 2. Handles complex fish image backgrounds and most likely simple ones 3. Not tested as fish detector on grayscale images as far as the scope of this literature study is concerned.

Table 3: An overview of weaknesses and strengths related to traditional methods and convolutional neural networks.

For these reasons, convolutional neural networks seem to be correct choice given the fact that enough data, several thousands of images, is supplied by Optoscale. They seem to be able to operate in real-time and using CNNs as fish detectors on grayscale images is a relatively unexplored area.

Summary

To summarize, a literature study on various methods for fish detection has been conducted in order to get an overview of existing methods for fish detection that would be able to handle the problem presented in this paper. Both traditional algorithms for fish detection and convolutional neural networks have been presented and their respective detection results and performances in terms of frames per second have been compared. In the end, convolutional neural networks picked the longest straw as a result of having similar detection results to the other methods and the best performance, in addition to being a relatively unexplored way of detecting fish on grayscale images.

Chapter 4: Hardware and Drivers

Hardware

- CPU: Intel Core i7 4790 @ 3.6GHz
- GPU: 2047MB NVIDIA GeForce GTX 1060 6GB (EVGA)
- RAM: 16GB Dual-channel DDR3 @ 799MHz
- Motherboard: Alienware 0PGRP5

Drivers

- NVIDIA Graphics Driver 397.64
- CUDA Toolkit 9.0
- cuDNN v7.4

Chapter 5: Libraries, frameworks and tools

Tensorflow-gpu 1.11.0

"Tensorflow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains." (*"https://www.tensorflow.org"*). Throughout this study, the GPU version of Tensorflow (Abadi *et al.*, 2015) is used, thus the library performs most of the numerical operations on the computer's GPU thus leading to greater performance and time efficiency when for example training neural networks.

Tensorflow Object Detection API

The API is an open source framework containing most tools a neural network trainer needs. It ships with state-of-the-art object detection models in the form of pipeline configuration files. Some of which also come with pretrained weights on the COCO-image set thus enabling the usage of transfer learning. Training is initiated by running one of the shipped python scripts after they have been built using protobuf and the process of data formatting has been completed. In order to integrate the API with TensorFlow-gpu, download the following GitHub repository (https://github.com/tensorflow/models) and place the files in the "TensorFlow" folder created by pip.

Pipeline configuration files are files which tell TensorFlow what neural network model to load. It also defines the values of the important hyperparameters such as batch size, learning rate, regularization weight values and so forth, making it easy for the user to tune the network if training results are not satisfactory. The TensorFlow team has published several pipeline files for different state-of-the-art neural network and a list of the available configs may be found here (*Tensorflow detection model zoo*).

Tensorboard 1.11.0

Tensorboard ships with the Tensorflow-gpu package installed through pip. It is a powerful visualization tool aimed at helping users interpret the current state of their trained neural network. Tensorboard reads the output created by tensorflow-gpu when training and visualizes important numbers such as classification and total loss, learning rate and evaluation results in the form of recall and mAP. Thus, it is of great help when deciding whether the neural net is training as expected or needs to be tuned in order to give satisfying results.

Python 3.6.5

TensorFlow is a Python based library and Python is consequently needed in order to initiate training and evaluation sessions of neural networks. Some of the modules shipped with TensorFlow seem to be written in Python2 or earlier and as a result these modules must be changed in order to accommodate the updated syntax of Python3. The author specifically experienced crashes caused by this issue when initially attempting to start the training of neural nets. Luckily a google search of the thrown error message gave solutions to every issue.

Pip 18.1

Pip is a package management system aimed at simplifying the installation of various libraries and frameworks for Python. It is used by the author to install TensorFlow-gpu and protobuf to the machine used for training and evaluation of neural networks.

Protobuf 3.6.1

Protobuf is a language and platform neutral data compressor, extractor and compiler. In this paper it is used to extract and build the protobuf files provided by the Tensorflow Object Detection API to usable ".py" Python files in order to train neural networks.

ShareLatex

Some equations were written in the ShareLatex user interface to increase readability. It is an online Latex-editor with online compilation capabilities.

Chapter 6: Theory

Neural network terminology

Batch

A batch is a subset of data from the training data set. For example, a batch size of 20 images would mean that 20 images have been extracted from the training set.

Global step

A global step is defined as how many times the neural network has processed a batch. Say the batch size is 20 images and the network in total has processed 2000 images. Then the value of global steps will be 2000/20=100.

Epoch

An epoch is defined as how many times the neural network has processed the entire training set. Say the neural network has processed 2000 images, and that the training set consists of 1000 images. Then the neural network has trained for 2 epochs.

Ground truth and boundary box

The boundary box is a rectangle which completely, and as tightly as possible, encapsulates an object in an image. It serves as the ground truth when a convolutional neural network predicts the location of an object in an image.

Confidence score

The confidence score is part the output of a convolutional neural network and is the probability provided by the network on how confident it is in its prediction. For instance, when predicting an object to be a fish, the network would also provide a probability of how certain it is on this prediction.

Annotated image

An annotated image comes with metadata, usually stored in a separate file, which describes certain aspects of the image. In this paper, an annotated image also comes with information regarding the position of fish in the image.

Activation functions

Rectified Linear Units (Krizhevsky, Sutskever and Hinton, 2012b) commonly known under the abbreviation ReLU, is a nonlinear mathematical function which keeps positive numbers as they are and sets negative numbers to zero. The mathematical equation for ReLU is the following

$$f(x)=\max(0, x)$$

Equation 1: The ReLU function

ReLU is commonly used as the activation function within convolutional layers.

The Sigmoid function (*"Sigmoid function"*, *Wikipedia*, 2002) is another type of nonlinear mathematical function normally used in the last layers of a fully connected neural network. Its mathematical definition is the following

$$S(x) = \frac{1}{1 + e^{-x}}$$

Equation 2: The Sigmoid function

Whereas its output is restricted to the interval <-1,1>.

Filters

Filters are simply 2-dimensional arrays of a fixed size FxF, where each element of the array is known as a weight (*"Convolutional Neural Networks (CNNs / ConvNets)"*, *cs231n*, *Stanford University*, 2018b). In the example below, the value of F would be 2. Dependent on which elements in the filter has the greatest values, the filter may encode a shape or information about color composition in an image. For instance, figure 3 could encode a line.

3	0
0	3

Figure 3: A simple 2x2 filter

A simple example of a convolutional layer

The following example is based on (*"Convolutional Neural Networks (CNNs / ConvNets)"*, *cs231n*, *Stanford University*, 2018a). Convolution in neural networks is the operation of moving a filter over a 2-

dimensional input array of size WxH, for example a small grayscale image with an edge on the diagonal. Firstly, the filter is superimposed on the input in for instance the top left corner, and then the dot product is made. The result of the dot product is stored in a new array of smaller size than the input array for later calculations. Consider the 4x4 input in figure 4, which is a small grayscale image depicting a line going across the diagonal of the image.

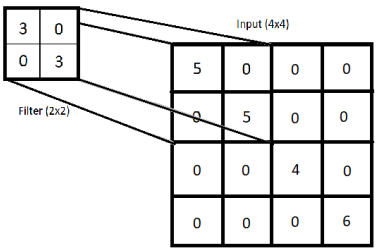

Figure 4: A 2x2 filter is superimposed on the 4x4 input

The filter has been superimposed on the input and the resulting dot product will be:

$$3x5+0x0+0x0+3x5 = 30$$

Next, the filter needs to slide over other values of the input. This is usually done by sliding the window a set number of places along the row dimension, called stride (S), and when the filter reaches the end of the row, it moves down S rows and starts from left to right yet again. Figure 5 illustrates the operation with a stride of 2.

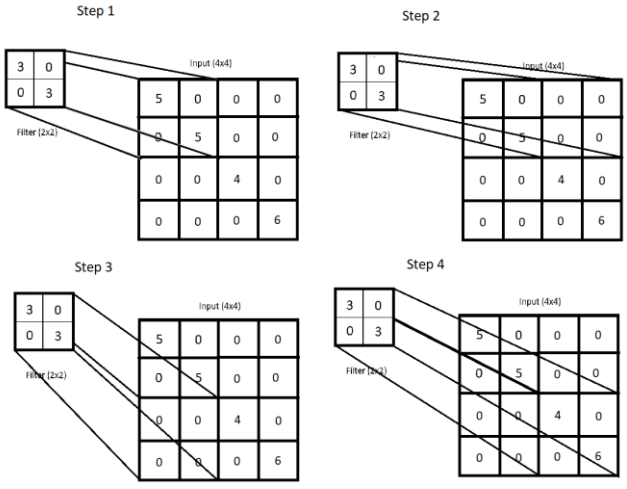

Figure 5: A 2x2 filter sliding over the 4x4 input

The dot product of step 1 is 30, for step 2 it is 0, for step 3 it is 0 and for step 4 it is 30. Thus, we may gather the output of the convolutional operation in the following 2D-array (figure 6).

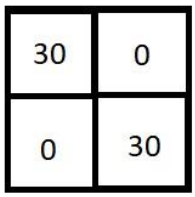

Figure 6: The outputted result from the filter operation.

As a result of the convolution, our 4x4 grayscale image has been downscaled to 2x2 while still containing enough information to tell us that a line was present in the original image.

Moreover, a bias is added to each element of the dot product array. Let's assume all biases are 0. Then we end up with the same array, and this is called the output volume. The output volume is subsequently passed through an activation function, where each element is filtered through a nonlinear function. This step is important, as it makes the convolutional neural network able to not only estimate linear functions, but nonlinear ones as well. Say our activation function is a simple one that only allows positive numbers to flow through, negative numbers are set to zero. This is known as the ReLU activation function and it is non-linear. Thus, when passing our dot product array through the ReLU function, the output array remains the same. Lastly, a non-maximum suppression method is applied to the output of the activation function. The non-maximum suppression method could simply be a window sliding over the output array that only keeps the largest value. Consider a window of size 2x2 which covers the entirety of the output array in our example. The output of the non-maximum suppression method would thus be a single number with value 30. In this example however, the non-maximum suppression method is excluded and the final output of the convolution operation on our 4x4 grayscale image becomes figure 4.

A simple example of a fully connected feed-forward layer

The following example is based on (*"Modeling one neuron", cs23n1, Stanford University*). A fully connected layer is a directed acyclic graph where all input nodes are connected to all output nodes. Consider figure 10, which takes the output of the previous example as input and outputs a single value. Notice that the output of the convolutional layer has been flattened to a 1-dimensional vector.

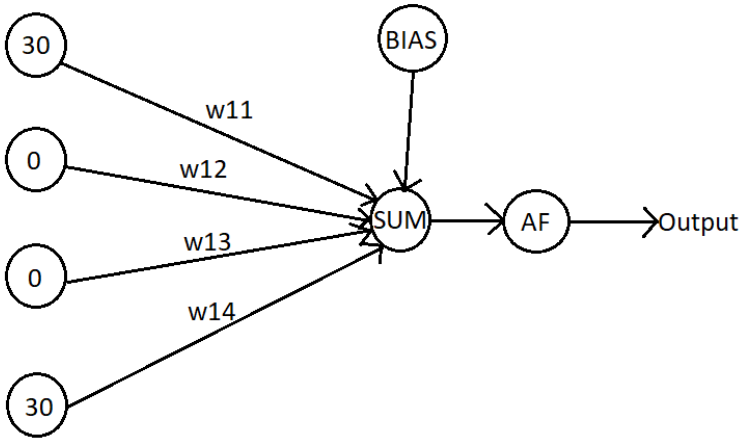

Figure 7: A visual explanation of fully connected feed-forward layer.

Assume that the bias is zero, the activation function (AF) is sigmoid (S(x)) and the weight vector looks as follow

Weight	w11	w12	w13	w14
Value	1	0	0	1

Thus, SUM equals

$$SUM = 1 \cdot 30 + 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 30 + 0 = 60$$

and when entering the SUM as input to equation … the output becomes

$$Output = S(SUM) = 1$$

and based on this number the network can decide if an edge has been detected (sigmoid is restricted in the interval <0,1> and thus a score close to 1 suggests that an edge was present).

Pooling layer

The following explanation is based on (*"Convolutional Neural Networks (CNNs / ConvNets)", cs231n, Stanford University*, 2018b).

A pooling layer works as a down sampler of the output from for example the convolutional layers. The idea is to pass on important values, whereas less important values are discarded. An example of a pooling layer is the max pooling layer. It slides a window of a fixed size of the input it is given and chooses only the largest value within the window. Consider the following example in figure 13 where the window and input size are 4x4, consequently the output becomes 1x1.

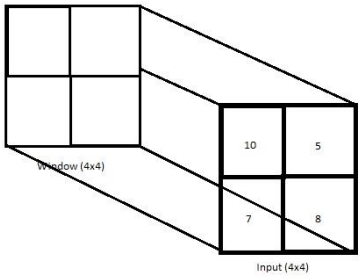

Figure 8: A visual explanation of the pooling layer operation.

The subsequent output of the max pooling layer would thus be max(10,5,7,8)=10

The general structure of a convolutional neural network

The following explanation is based on (*"Convolutional Neural Network Architectures", TUM*, 2017).

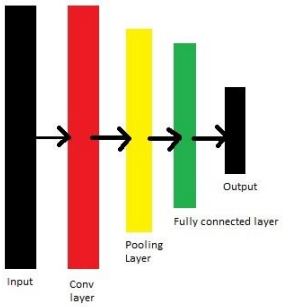

Figure 9: The general structure of a simple convolutional neural network.

Consider figure 14 above, which depicts an excessively simple convolutional neural network. Typically, the input is sent through a large convolutional layer, next the output of the convolutional layer is passed through a pooling layer to keep the prominent values. In the more complex case, several cascades of convolutional and pooling layers are added. Furthermore, the output of the final pooling layer is flattened and passed to the fully connected layer which outputs the final result.

Equations of the cost function and backpropagation algorithm for learning

The following explanations are based on chapter 2 in (Nielsen, 2015). In this explanation, the quadratic cost function is considered, however many others exist with similar properties (*"A list of cost functions..."*, *StackExchange*, 2015) and the solutions regarding this cost function can be adapted to other cost functions as seen in chapter 3 in (Nielsen, 2015) .

In order to train the network, a measure on how well it is performing is needed. This measure is called the loss or cost function. As defined in chapter 2, equation 26 in (Nielsen, 2015) the cost function looks the following

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Equation 3: The quadratic cost function. Source: (Nielsen, 2015), chapter 2, equation 26

Where y(x) is the desired output of the neural network, $a^L(x)$ is the activation output from the last layer of the neural network and x is the input to the neural network which are summed over the total number of training examples in the batch.

Two assumptions are needed on the cost function. Firstly, the cost function needs to be able to be written as an average over cost functions C_x for individual training examples. This assumption is needed in order to train the neural network with batch sizes larger than 1, as the final cost is estimated by averaging over costs for individual training examples. Secondly, the cost function needs to be a function of the outputs of the neural network. This is essential in updating weight values deep down the neural network through partial derivation of the original cost function.

Figure 15-18 and equation 5 outlines the steps taken in order to update the weights and biases at a given layer in the neural network, thus making it able to learn. This technique is known as backpropagation. Large L denotes the last layer of the network.

1. **Input x:** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Figure 10: The general outline of the backpropagation algorithm. Source: (Nielsen, 2015), chapter 2, subchapter "The backpropagation algorithm".

where the equations for the different expressions used in the outline are defined by figure 16-18 in addition to equation 5.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Figure 11: Definition of the equations used in the backpropagation algorithm. Source: (Nielsen, 2015), chapter 2, subchapter "The four fundamental equations behind backpropagation".

Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\partial C/\partial a_j^L$.

Figure 12: Mathematical definition of the vector containing the cost function's partial derivatives with respect to the last layers outputs. Source: (Nielsen, 2015), chapter 2, subchapter "The four fundamental equations behind backpropagation".

$$\partial C/\partial a_j^L = (a_j^L - y_j)$$

Equation 4: Definition of the cost's partial derivative with respect to the last layer's outputs. Source: (Nielsen, 2015), chapter 2, subchapter "Exercises".

Lastly, weights and biases would we updated according to the stochastic gradient descent scheme

3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \to w^l - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T$, and the biases according to the rule $b^l \to b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$.

Of course, to implement stochastic gradient descent in practice you also need an outer loop generating mini-batches of training examples, and an outer loop stepping through multiple epochs of training. I've omitted those for simplicity.

Figure 13: Final equations for updating the weight of each layer l. Source: (Nielsen, 2015), chapter 2, subchapter "Exercises".

Where η denotes the learning rate of the neural network, and thus decides how large of a step the networks weight should take when learning. x denotes which input sample is used from the batch and m the batch size.

Choosing the learning rate

The following explanations are based on the article (Zulkifli, 2018).

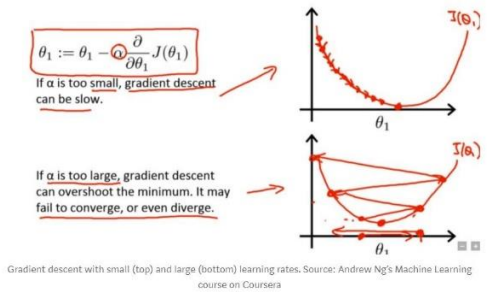

Figure 14: Effect of different learning rates on a 2-dimensional optimization problem. Source (Zulkifli, 2018)

Figure 19 depicts a simple 2-dimensional optimization problem where gradient descent it used to update the weight at each step. α denotes the learning rate, J() is the cost function and θ is the weight that is being updated. As one may observe, when operating with a small learning rate it may time several epochs for the algorithm to converge to the global minima. If one considers a multidimensional space with several local minima, which is the case when training neural networks on images (Choromanska *et al.*, 2014), it might also lead to the network being stuck in a sub-optimal local minimum. When operating with a large learning rate, the global or local minima may never be reached, and the learning algorithm might even diverge from a solution. For these reasons, picking the right learning rate schedule for a neural network is a tricky ordeal.

Fortunately, there exists learning rate optimizers such as Momentum and RMSProp (Ruder, 2016) that attempt to automatically tune the learning rate based on an algorithm. Thus, lightening the time required to tune a network considerable.

Avoiding overfitting

Overfitting, as defined in (*"Overfitting", Oxford Dictionary*), is "The production of an analysis which corresponds too closely or exactly to a particular set of data and may therefore fail to fit additional data or predict future observations reliably.". Translated to neural network language, this means that a neural network model corresponds correctly to the training data, however, does not fit new data such as the test data. Consequently, detection results on new data will be inadequate if not non-existent.

To avoid overfitting, a variety of methods exist. Most notably is the process of monitoring training and validation loss and results by splitting the dataset into training and validation sets (*"Overfitting", Wikipedia*, 2018, Guyon, 1996, Nielsen, 2015) introducing dropout to the classifying layer (Srivastava *et al.*, 2014), introducing more data to the training set, including L1/L2 regularization techniques, using data augmentation or reducing the complexity, e.g. size, of the network (Ruizendaal, 2017).

Transfer learning

The following explanations are based on (*"Transfer Learning", cs231n, Stanford University* 2018). Transfer learning is a technique within the field of artificial neural network. The technique consists of pretraining a given network model on a large set of data, for instance the COCO image dataset. The first layers of the neural network will encode features that generalize to most objects, such as edges and curved shapes, whereas the final classification part will be specialized towards the objects that were included in the pretraining part. By completely resetting the weights in the classification layers of the network after the pretraining was completed, one is left with a network that can detect shapes or color patterns, however

being clueless with regards to what object these data points might represent. Thus, one finetunes the network on new data to mainly train the classification layers. By doing so, one saves a vast amount of time as not all parts of the network need to be trained. In addition, the size of the dataset can be shrunk as not that much data is needed to finetune the network.

Object detection metrics

First off, one needs to provide some definitions on how to categorize the outcome of a prediction, they are based on the following article (*"Precision vs Recall", Wikipedia*, 2007).

- A true positive (TP) is the result when the network correctly predicts the class of an object.
- A false positive (FP) is the result when the network predicts that an object is of a certain class, when in fact it is not.
- A true negative (TN) result is when the network correctly predicts that an object is not the class it is looking for.
- A false negative (FN) is when the neural network incorrectly predicts that an object is not the class it is looking for, when in fact it is.

Recall

Recall (*"Precision vs Recall", Wikipedia*, 2007) is the number of true positives divided by the actual number of positives. For instance, say there are 30 images of fish in a dataset, and the network correctly predicts 25 of them and does not find the others. Then the recall would be 25/30 = 0.83. Equation 6 shows its mathematical definition.

$$Recall = \frac{TP}{TP + FP}$$

Equation 5: Definition of recall.

Precision

Precision (*"Precision vs Recall", Wikipedia*, 2007) is the number of true positives divided by the total number of predictions made. It serves as a measure on how correct the network is when classifying objects. For instance, say there are 30 images of fish and 10 images of otters in a dataset, and the network correctly predicts 25 of them, however predicted otter on the 5 remaining. Equation 7 shows its mathematical definition.

$$Precision = \frac{TP}{TP + FN}$$

Equation 6: Definition of precision.

Intersection over Union

Intersection over Union (IoU) is a measure on how well a convolutional neural network predicts the location of an object. It is calculated by dividing the area of the intersection between the ground truth boundary box and predicted boundary box of an object by the total area of both boundary boxes combined (Rosebrock, 2016). Equation 8 shows its mathematical equation.

$$IoU = \frac{\text{Area of overlap}}{\text{Area of union}}$$

Equation 7: Definition of Intersection over Union.

Precision-recall curve

There are several definitions of mAP and precision-recall curves, and in this report the variant used in the COCO object detection competition will be explained. The explanation is based on (Arlen, 2018).

The precision-recall curve is a mathematical function showing the relationship between precision and recall p(r). It is calculated by varying the confidence score of a neural network at certain IoU thresholds. The recall and precision are recorded when varying the confidence score, and in the end, precision is plotted against recall. Figure 20 shows an example of a precision-recall curve where only detections with an IoU score of above 0.5 are included.

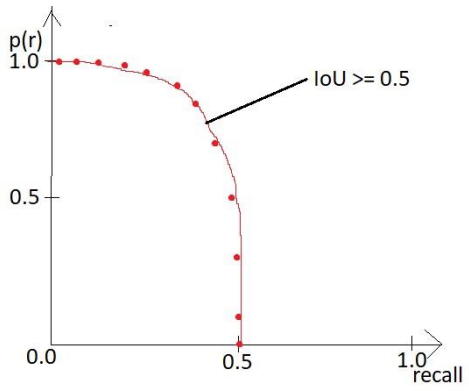

Figure 15: An example of a precision-recall curve.

Where recall is close to zero, one can expect the confidence score to be so high that when making predictions it is almost always correct, thus precision is close to 1, however it finds few of the objects present as the confidence score is too high and as a result recall is close to 0. Where precision is close to zero, one can expect that the confidence score is quite relaxed, and as a result several accepted predictions are made where some are wrong, and others correct. As a result, the precision plummets whereas recall increases since the network finds more of the objects present this time.

Mean average precision at IoU threshold 0.5

The mean average precision is in this case the average precision across all recall values. Consequently, it is the area under the p(r) graph and is a number that describes the graph's shape. As the maximum area under the graph is 1, a perfect mAP@0.5IoU score would be 1. Then the network shows 100 % precision and recall at all confidence score levels when the IoU of the detection's predicted location is above 0.5.

Chapter 7: Method

Preprocessing the dataset

Firstly, each image of fish was matched to its binary images. This operation was somewhat tricky, as the fish images had the caption “13_Oct_2017_00_26_22_178_FullCam1.jpg”, namely “day_month_year_hour_minute_second_image” whereas their binary images had the caption “13_Oct_2017_00_26_22F1_178_Binary”. As one may notice, the binary image’s caption has inserted the letters “FX”, where X is a whole number which defines which detected fish in the fish image the binary image labels, and thus a simple comparison between the captions would not suffice. As a result, a script (Appendix B, B1) was written that manually separated binary and fish images into two different arrays based on their filetype and then extracted the time information from each caption. Finally, the extracted time information from every element of the two arrays where compared to match fish images to their binary images. The binary images were renamed to match the caption of the fish image and then both the fish images and binary images were stored in a new folder. As a result of this operation, one could now easily match binary images to their fish image and sort them in for example Windows Explorer to manually inspect each fish image and its binary images to get a better overview of the dataset.

Furthermore, since some of the fish images would look roughly the same due to the camera operating at 200 FPS and tracking the same fish over several frames, some of these need to be removed as they do not present new information to the convolutional neural network and may result in overfitting. A script (Appendix B, B2) was written to find all images tracking the same fish, and then sort by image number. In the end every fifth image of the sequence was kept, as the author noticed some change between images when applying this interval. The images one decided to keep was saved to a new folder along with their respective binary images. Lastly, as this process of elimination had downsized the dataset to about 2700 images of fish, each binary image was manually inspected to see whether the binary image was deformed and thus would lead to an inaccurate boundary box for the fish. If the binary image seemed misformed, it would be removed, and if this was the last binary image belonging to a fish image, the fish image would also be removed. See the figure 21 and 22 below for an example of binary image misformation.

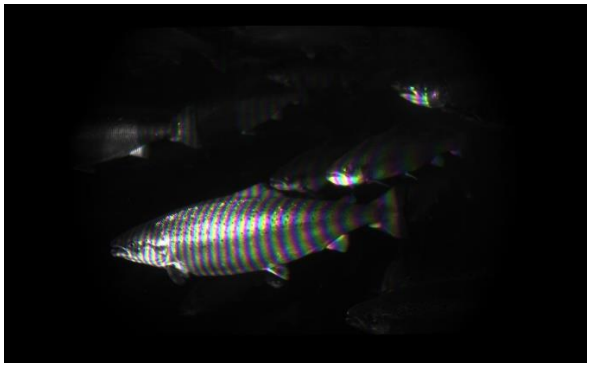

Figure 16: The original fish image.

Figure 17: Misformation in the binary image labeling the fish.

Moreover, the boundary box given by each binary image had to be found in terms of its coordinates and describing values, namely top left corner, width and height. A script (Appendix B, B3) was written that iterates through each pixel value of the binary image from left to right, top to bottom and vice versa in order to identify the image coordinates of the points of the top left and bottom right corner of the fish. This was done through simple thresholding, as the value of a pixel belonging to the fish has the value 1 and the rest 0. From these points, the information that describes the size and position of the boundary box was extracted. See the image below for an illustration. Lastly, binary images in which the script failed to detect a boundary box was removed, and if it was the last binary image relating to a fish image, the fish image would also be removed. Figure 23 illustrates the mentioned concepts.

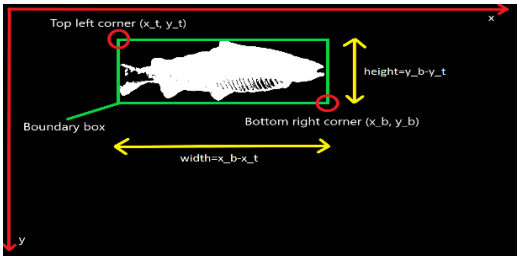

Figure 18: Visual explanation of how the boundary boxes metadata was found.

Next, the RGB stripes of the structured light needed to be removed as they were not a part of this task. Thus, a script (Appendix B, B4) was created which summed each channel's value at each pixel location and divided the sum by the number of channels, namely three. The resulting value was put into a new grayscale image, with only one channel, at the same pixel position. After, the grayscale image was pasted into each channel of an RGB image. The reason behind this decision would be that then one can fully exploit the shape information already encoded in convolutional neural networks when using the technique of transfer learning, as the weights one transfers from another trained convolutional network, usually is trained on a three channel RGB-image. See figure 24 and 25 below for the result of this operation.

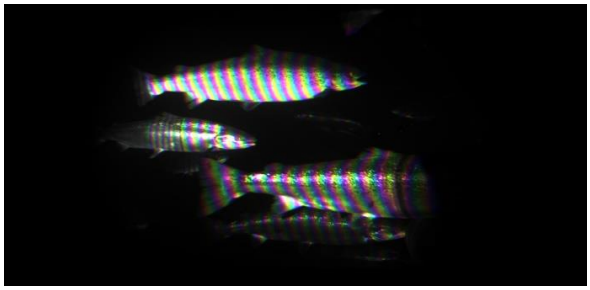

Figure 19: Original image before removal of structured light.

Figure 20: Image after the structured light was removed.

Creating the train, evaluation and test set

The resulting size of the dataset after preprocessing stages was 2658 images of fish. These needed to be divided into three sets, namely training, validation and test set. A script (Appendix B, B5) based on a script created by Daniel Stang (Stang, 2017) was modified to create the TFRecord file which is the file format Tensorflow's object detection API reads data from. In addition, all images related to training, validation and testing were stored in their own respective folders for later fish detection testing. Table 4 summarizes the distribution of images in each set.

Dataset	Number of images
Training	1860
Validation	532
Testing	266

Table 4: The size of each dataset after division.

Choosing which convolutional neural networks to train

In order to get an overview of which convolutional neural networks Tensorflow offers configuration files for, as well as pretrained models, the Tensorflow detection model zoo was visited. At the site, various models and methods for object detection are outlined, as well as their fps and mAP score on the COCO dataset. As mentioned in their article (Huang *et al.*, 2016), the measurements on fps were conducted on a Nvidia GeForce GTX Titan X, which is more powerful than the GPU used in this study and by Optoscale. As a result, when picking which networks to train for fish detection, this needs to be considered. From that knowledge, models reporting an operating fps of 12 or lower were dropped as they most likely do not run in real-time on this study's GPU. In the end, the choice fell on three types of convolutional neural networks and their respective method for proposing where an object may be. Table 5 summarizes the choices and their reported attributes.

Chosen CNN	Chosen region proposal method	Pretrained	Reported FPS
Mobilenet v2	Single-shot detector	Pretrained on COCO	34.48
Mobilenet v1	Single-shot detector and feature pyramid network	Pretrained on COCO	13.15
Inception v2	Faster region-based convolutional neural network	Pretrained on COCO	17.24

Table 5: An overview of the chosen CNNs and their reported attributes.

Preparing configuration files for training

Firstly, the data and models were organized in the folder structure recommended by Tensorflow ("*Running locally", Tensorflow*, 2018).Next, the config file relating to the CNN model which was downloaded from the detection zoo was modified to match file paths to both model and config. In addition, number of classes was set to 1 and number of evaluation examples to 532 and the training data was set to shuffle after each epoch. All models used the data augmentation techniques "random_horizontal_flip" and "random_crop_image" or "ssd_random_crop" which is the SSD variant of

“random_crop_image”. Also, the measure for classification loss function was set to “weighted sigmoid”, which is an abbreviation of binary cross entropy with weighted sigmoid activation function, as this supposedly is the best classification loss function for data with 1 class (*"Softmax Regression", Stanford University Wiki*, 2011). Lastly, the metrics used to evaluate the networks detection results was chosen to be the detection metrics average recall and mAP, which is used in the COCO object detection competition.

Initially all hyperparameters in the config file were kept as they came from the detection zoo as they are shipped ready to fine tune the downloaded CNN on new data. Training was initiated by running the python script “model_main.py” which ships with the tensorflow-gpu package through a command window. In addition, the Tensorboard application was run in a separate command window in order to keep tabs on loss and evaluation results while training through the web interface.

Tuning hyperparameters during training

The classification and localization loss of both training and validation, in addition to evaluation results were closely monitored through Tensorboard. The total training loss was updated in Tensorboard every 100^{th} global step, whereas the validation loss and results every 465^{th} global step, which is 4 times per epoch. These measurements where used to determine if the network was training correctly and when its training was completed.

When tuning the network, a couple of rules were followed with regard loss and tuning of the learning rate.

1. If both training loss and validation loss steady decrease in value and stays roughly at the same value, then the network is training properly. Thus, the learning rate need not be changed.
2. If the training loss and validation loss slowly decrease until they stabilize, and validation detection results are bad, then the learning rate is most likely too low and needs to be increased.
3. If the learning rate skyrockets from the beginning, the learning rate is most likely too high and needs to be decreased.
4. If the distance between training loss and validation loss increases early on, the network is most likely overfitting due to too high learning rate and thus the learning rate needs to be lowered.

5. It is normal for the training loss to oscillate between increasing and decreasing, however if the amplitude of these oscillations is large the learning rate is most likely too high and needs to be lowered.

Furthermore, it was hard to balance the learning rate to achieve optimal training for FRCNN Inception v2. As a result, a dropout value of 0.5 was set in the box predictor layer to increase the networks robustness against overfitting, making it easier to find an appropriate learning rate scheme.

When deciding whether to tune the learning rate itself or for instance the decay factor or steps as part of the learning rate scheme, the following guidelines were followed.

1. If the network shows signs of overfitting early on, the learning rate needs to be adjusted.
2. If the network shows signs of overfitting at a later stage without satisfactory detection results, the decay steps parameter needs to be adjusted. Preferably to a step that comes before the point where the network showed signs of overfitting.
3. If the situation in point 2 occurs again, decay factor needs to be adjusted.

As for the batch size, the GPU struggled with having enough memory when the batch size was larger than 4. To avoid any crashes due to memory issues while training, it was decided that if the original config's batch size was larger than one, the batch size would be set to two.

After countless trials and errors, the following table 6 shows the final values of the most important hyperparameters. For details on the remaining hyperparameters and their values, see appendix A.

Hyperparameters	**CNN**		
	FRCNN Inception v2	**SSD+FPN Mobilenet v1**	**SSD Mobilenet v2**
Batch size	1	2	2
Learning rate (LR)	Manual learning rate Step 0 LR: 0.0002 Step 2000 LR: .00002 Step 5000 LR: .000002 Step 7500 LR: .0000002	Exponential decay Step 0 LR: 0.004 Decay steps: 465 Decay factor: 0.85 Staircase	Exponential decay Step 0 LR: 0.004 Decay steps: 500 Decay factor: 0.85
Dropout	Not used	0.5	0.5

Table 6: An overview of the final hyperparameter values used.

Deciding when to end the training of a network

Two criteria were followed in order to decide if the training of a network should be stopped.

1. If the validation detection results, namely mAP and recall, looked satisfactory and showed little signs of improvement, the network's training would be considered complete.
2. If validation detection results, namely mAP and recall, looked inadequate and showed little signs of improvement, and the validation loss had stalled on a certain level or was oscillating, the training would end, and readjustments should be made to the network's hyperparameters.

Frames per second test

All test images were resized according to the resize scheme defined in each network model and stored in new folders. That is, images for FRCNN Inception v2 were resized to a maximum width of 1024 with the preservation of the image's original aspect ratio, images for Mobilenet v1 were resized to a fixed size of 640x640 pixels and images for Mobilenet v2 were resized to a fixed size of 300x300 pixels.

Furthermore, the graph from each network was extracted through the TensorFlow provided python script called "export_inference_graph.py". The graphs were extracted from the models that showed the best mAP@0.5 and recall validation scores. Lastly, a script (Appendix B, B6) for measuring frames per second was written based on TensorFlow's "off-the-shelf-inference" script for inference on single images. The script excluded the 10 first images in the fps count as it needs some time to warm up before reaching a stable fps. The timer was started after the first 10 images had been inferred. As a result, the final fps value was found through equation 9.

$$FPS = \frac{\text{Number of test images} - 10}{\text{end time - start time}}$$

Equation 8: Equation used to calculate the CNNs FPS.

Chapter 8: Results

Training and validation results

Note that the transparent graphs are the true values measured, whereas the clearly visible graph is a smoothed variant which removes the inherent noise present in the measurement and thus making the graphs easier to interpret when training. In addition, the red dot should be ignored.

The figure 26-29 show the training and validation losses and detection metrics related to FRCNN Inception v2.

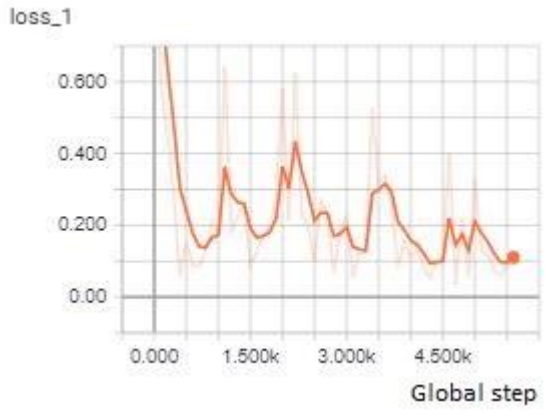

Figure 21: Training loss of FRCNN Inception v2 per global step.

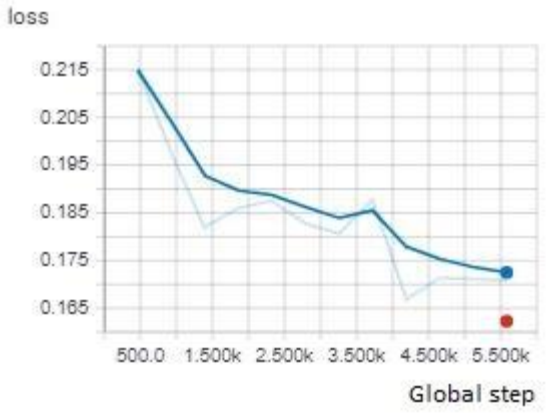

Figure 22: Validation loss of FRCNN Inception v2 per global step.

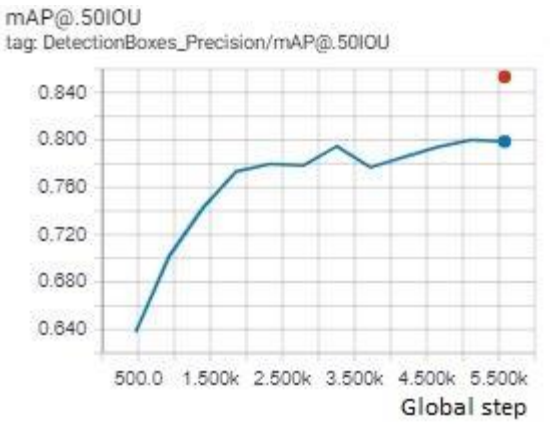

Figure 23: mAP@0.50IoU of FRCNN Inception v2 per global step.

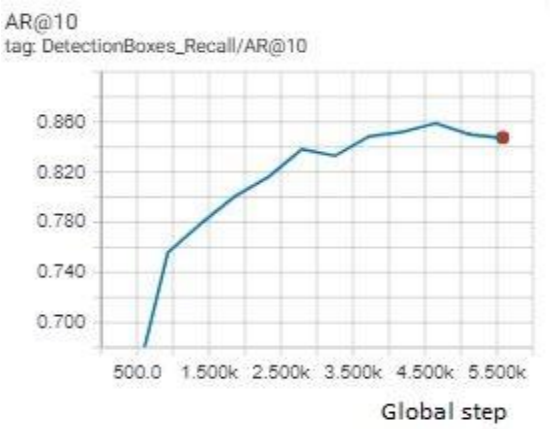

Figure 24: Recall of FRCNN Inception v2 per global step.

Figure 30-33 show the training and validation losses and detection metrics related to SSD+FPN Mobilenet v1.

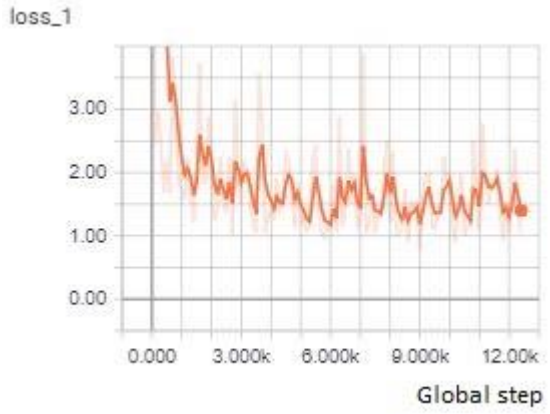

Figure 25: Training loss of SSD+FPN Mobilenet v1 per global step.

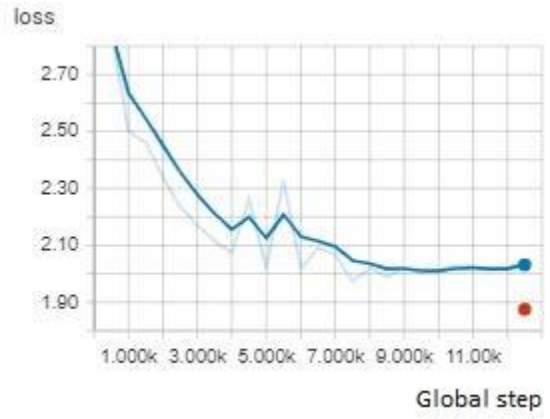

Figure 26: Validation loss of SSD+FPN Mobilenet v1 per global step.

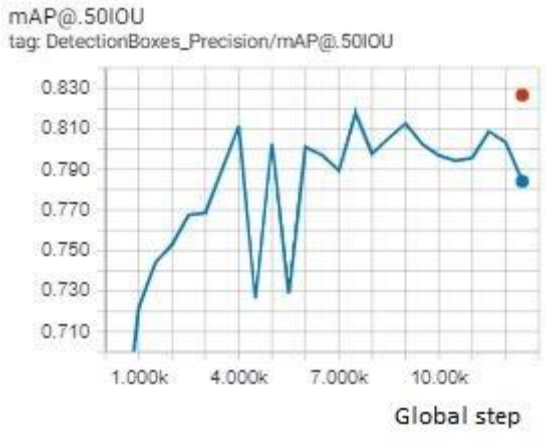

Figure 27: mAP@0.5IOU of SSD+FPN Mobilenet v1 per global step.

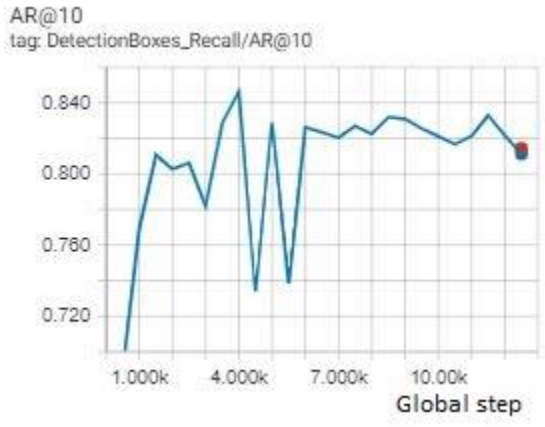

Figure 28: Recall of SSD+FPN Mobilenet v1 per global step.

Figure 34-37 show the training and validation losses and detection metrics related to SSD Mobilenet v2.

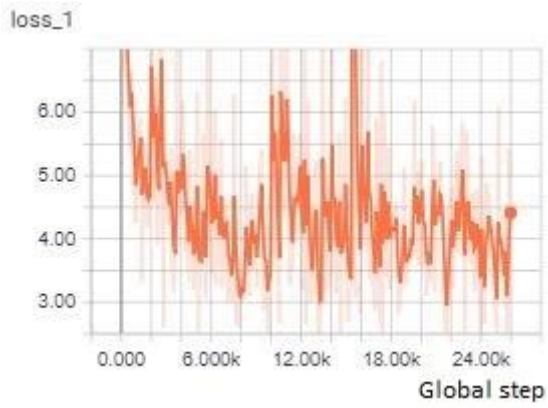

Figure 29: Training loss of SSD Mobilenet v2 per global step.

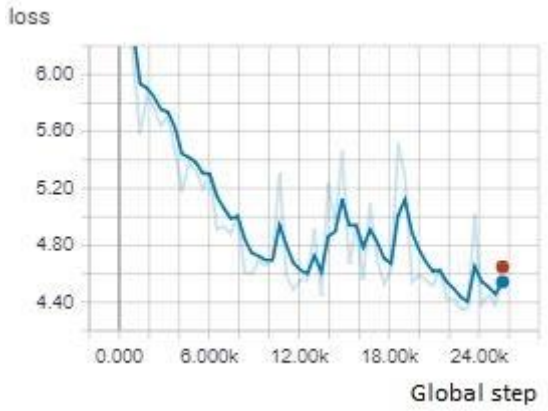

Figure 30: Validation loss of SSD Mobilenet v2 per global step.

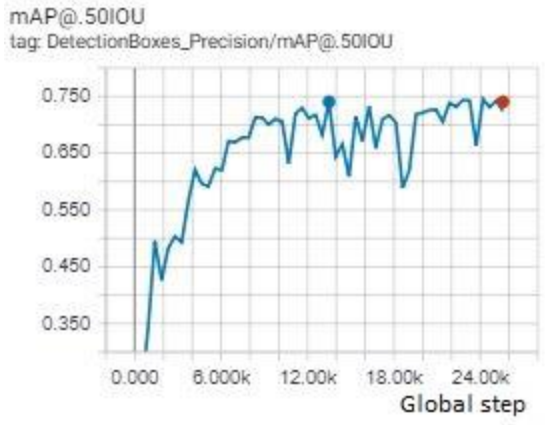

Figure 31: mAP@0.5IOU of SSD Mobilenet v2 per global step.

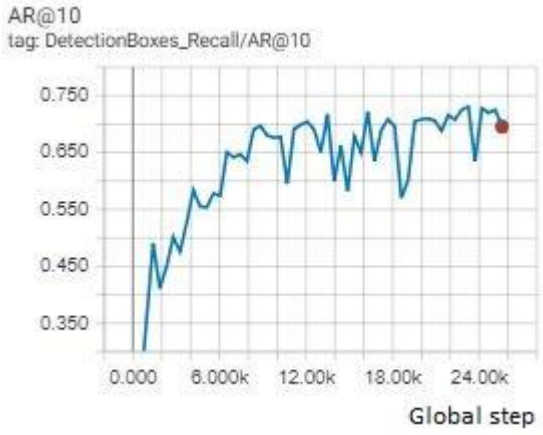

Figure 32: Recall of SSD Mobilenet v2 per global step.

Test results

Table 7 shows the detection results after inferring on 266 test images, which corresponds to the red dots seen on the recall and mAP graphs above.

CNN	Recall	mAP@0.5IoU	FPS
FRCNN Inception v2	0.848	0.853	2.02
SSD+FPN Mobilenet v1	0.814	0.827	3.38
SSD Mobilenet v2	0.694	0.741	4.29

Table 7: Overview of recall and mAP results for each CNN.

Chapter 9: Discussion

Training and validation results

As seen in all training loss graphs (figure 26, 30, 34), the loss tends to fluctuate while slowly decreasing. This may be a property of the data used in training and batch size, as smaller batch sizes are more susceptible to the noise in each image in contrast to using large batch sizes where the noise is averaged out (Keskar *et al.*, 2016). More notably, is the amplitude of the fluctuations in the training and validation loss of SSD Mobilenet v2. Generally, one would assume that this is caused by having to high learning rate, however the author experimented with extremely low values and yet the fluctuations were still present. A possible explanation could be caused by a methods used by the network called hard example miner (Shrivastava, Gupta and Girschick, 2016, Hui, 2018) and the lack of annotation of the fish dataset. The hard example miner stores false positive detections and subsequently trains the network on these examples to learn what the object is not. However, since not all fish are annotated in the dataset, the network may detect an object that in fact is a fish, however it gets labeled as a false positive as the object has not been annotated. These examples are subsequently used to train the network what a fish does not look like and may be the cause of the oscillations.

When looking at the recall and mAP for SSD+FPN Mobilenet v1 (fig ..), the results take a dive around step 4500 before recovering. This may be a sign of overfitting, however later the network recovers. As far as the author can tell, this may be due to a suboptimal learning rate scheme, where the learning rate decreases too slowly or learning rate optimizers. The learning rate optimizers take an educated guess based on the initially set learning rate to update the weights of the network through backpropagation, and this may not be optimal (Wilson *et al.*, 2017). The issue could also have been caused by the network

being too complex, however FRCNN Inception v2 does not show the same tendencies while being a more complex network, and thus is explanation is discarded.

Finally, the results of the respective networks' training seem to be acceptable. Both training loss and validation loss decrease most of the time and stay relatively close. Furthermore, recall and mAP values confirm this observation as they slowly rise to an asymptotic maximum value.

Test results

The most noticeable result is that not one of the convolutional neural networks managed to operate in real-time, although Tensorflow reported that they can do so. The most prominent reason for this paper's difference in fps compared to Tensorflow's own measurements would be the different hardware used. Tensorflow's GPU outshines the GPU used in this study on all performance criteria. Furthermore, the author discovered that the provided numbers on fps from Tensorflow's model zoo used a different input image size when compared to the configuration publicly available for FRCNN Inception v2 and SSD Mobilenet v1. The input image size in the publicly available configurations was larger, and as a result the neural network becomes slower as more data must be processed. This most likely explains why SSD+FPN Mobilenet v1 was measured to be faster than FRCNN Inception v2 although the opposite was reported. Consequently, in order to increase the fps by some degree, one could try to train the same neural networks on smaller images. Out of curiosity, the author halved the width of the images input to the FRCNN Inception v2 network and the fps rose to 5.14 while still showing promising results. However, these results were only visually observed by the author and not confirmed by an objective detection result measure. Finally, the issue could have been avoided early on if the fps of the pretrained networks was tested before they were trained.

When comparing the various networks results to those gathered in the literature study, the networks seem to perform above average. They are not competing with the top results of other implementations, however neither are they at the bottom. A reason as to why they were not able to reach the top may be the number of images in the dataset and the fact that many of the images track the same fish over several frames. Thus, this sequence of images does not represent entirely new information and may lead to a slight increase in overfitting. Furthermore, the other implementations used RGB images, thus additional information regarding the color of fish might have been to their advantage.

Chapter 10: Conclusion

The objective of this study was to create an underwater fish dataset consisting of grayscale images and to compare fish detection methods found in the literature in order to determine which of these could serve as real-time fish detectors. Lastly, the most prominent method was to be implemented and tested on the dataset.

In order to do so, a literature study was conducted that found convolutional neural networks to be the best candidate. The proposed method was subsequently realized by choosing three promising types of CNNs, creating an annotated dataset consisting of fish images received from Optoscale, implementing and testing the solution through TensorFlow's object detection API and finally measuring the methods fps on a test set.

The results of the three different implementations varied and compared to similar results from the literature, one may conclude that they performed above average. However, when looking at their performance, they performed below average and did not operate in real-time.

Chapter 11: Future work

1. It would be informative to investigate whether a smaller input size could have thrusted the fps of the implementations above 8 FPS.
2. A larger dataset could have been gathered to see whether detection results would be improved, that also had annotated all examples of fish in the image.
3. Less complicated convolutional neural networks could have been tested in order to improve FPS and issues with overfitting, perhaps leading to better detection results.

Bibliography

Abadi, M. *et al.* (2015) TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Available at: http://tensorflow.org/.

Arlen, T. C. (2018) *Understanding the mAP Evaluation Metric for Object Detection*. Available at: https://medium.com/@timothycarlen/understanding-the-map-evaluation-metric-for-object-detection-a07fe6962cf3 (Accessed: 11/17/2018).

Boudhane, M. and Nsiri, B. (2016) Underwater image processing method for fish localization and detection in submarine environment, *Journal of Visual Communication and Image Representation, vol 39*, pp. 226-238.

Canziani, A., Paszke, A. and Eugenio, C. (2016) An Analysis of Deep Neural Network Models for Practical Applications, *arXiv:1605.07678 [cs.CV]*.

Choromanska, A. *et al.* (2014) The Loss Surfaces of Multilayer Networks, *arXiv:1412.0233v3 [cs.LG]*.

"Convolutional Neural Network Architectures", TUM (2017). Available at: https://wiki.tum.de/display/lfdv/Convolutional+Neural+Network+Architectures (Accessed: 11/15/2018).

"Convolutional Neural Networks (CNNs / ConvNets)", cs231n, Stanford University (2018a). Available at: https://cs231n.github.io/convolutional-networks/ (Accessed: 09/26/2018).

"Convolutional Neural Networks (CNNs / ConvNets)", cs231n, Stanford University (2018b). Available at: https://cs231n.github.io/convolutional-networks/#case (Accessed: 11/14/2018).

Deng, J. *et al.* (2009) ImageNet: A large-scale hierarchical image database, *2009 IEEE Conference on Computer Vision and Pattern Recognition*.

Giordano, D., Palazza, S. and Spampinato, C. (2016) Fish Detection. In: Fisher R., Chen-Burger YH., Giordano D., Hardman L., Lin FP. (eds) Fish4Knowledge: Collecting and Analyzing Massive Coral Reef Fish Video Data., *Intelligent Systems Reference Library, vol 104. Springer, Cham*.

Guyon, I. (1996) A scaling law for the validation-set training-set size ratio. Available at: https://pdfs.semanticscholar.org/452e/6c05d46e061290fefff8b46d0ff161998677.pdf.

He, K. *et al.* (2015) Deep Residual Learning for Image Recognition, *arXiv:1512.03385 [cs.CV]*.

Hongwei, Q. *et al.* (2015) DeepFish: Accurate underwater live fish recognition with a deep architecture, *Neurocomputing, vol 187*, pp. 49-58.

Howard, A. G. *et al.* (2017) MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, *arXiv:1704.04861 [cs.CV]*.

"*https://www.tensorflow.org*" (Accessed: 10/11/2018).

Huang, J. *et al.* (2016) Speed/accuracy trade-offs for modern convolutional object detectors, *arXiv:1611.10012 [cs.CV]*.

Hui, J. (2018) *What do we learn from single shot object detectors (SSD, YOLOv3), FPN & Focal loss (RetinaNet)?* Available at: https://medium.com/@jonathan_hui/what-do-we-learn-from-single-shot-object-detectors-ssd-yolo-fpn-focal-loss-3888677c5f4d (Accessed: 12/17/2018).

ILSVRC *IMAGENET Large Scale Visual Recognition Challenge*. Available at: http://www.image-net.org/challenges/LSVRC/.

Keskar, N. S. *et al.* (2016) On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, *arXiv:1609.04836v2 [cs.LG]*.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012a) ImageNet classification with deep convolutional neural networks, *In NIPS*, pp. 1106-1114.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012b) ImageNet classification with deep convolutional neural networks, chapter 3.1, *In NIPS*.

"A list of cost functions...", StackExchange (2015). Available at: https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications (Accessed: 11/15/2018).

Liu, W. *et al.* (2015) SSD: Single Shot MultiBox Detector, *arXiv:1512.02325v5 [cs.CV]*.

Lu, H., Du, X. and Chang, P. (2018) Toward Scale-Invariance and Position-Sensitive Region Proposal Networks. In: Ferrari V., Hebert M., Sminchisescu C., Weiss Y. (eds) Computer Vision – ECCV 2018. ECCV 2018., *Lecture Notes in Computer Science, vol 11212. Springer, Cham*.

Lucena, O. *et al.* (2017) Transfer Learning Using Convolutional Neural Networks for Face Anti-spoofing. In: Karray F., Campilho A., Cheriet F. (eds) Image Analysis and Recognition. ICIAR 2017., *Lecture Notes in Computer Science, vol 10317. Springer, Cham*.

Mandal, R. *et al.* (2018) Assessing fish abundance from underwater video using deep neural networks, *arXiv:1807.05838v1 [cs.CV]*.

Menéndez de Llano, R. and Bosque, J. L. (2010) Study of neural net training methods in parallel and distributed architectures, *Future Generation Computer Systems, vol 25, issue 2*, pp. 267-275.

"Modeling one neuron", cs23n1, Stanford University. Available at: http://cs231n.github.io/neural-networks-1/ (Accessed: 11/15/2018).

Nielsen, M. A. (2015) Neural Networks And Deep Learning, Determination Press. Available at: http://neuralnetworksanddeeplearning.com/.

"Overfitting", Oxford Dictionary. Available at: https://en.oxforddictionaries.com/definition/overfitting (Accessed: 11/15/2018).

"Overfitting", Wikipedia (2018). Available at: https://en.wikipedia.org/wiki/Overfitting (Accessed: 11/15/2018).

"Precision vs Recall", Wikipedia (2007). Available at: https://en.wikipedia.org/wiki/Precision_and_recall (Accessed: 11/17/2018).

Ravanbakhsh, M. *et al.* (2015) Automated fish detection in underwater images using shape-based level sets, *Photogrammetric Record, vol 30*, pp. 46-62.

Ren, S. *et al.* (2015) Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, *arXiv:1506.01497 [cs.CV]*.

Rodríguez, Á. *et al.* (2015) Fish tracking in vertical slot fishways using computer vision techniques, *Journal of Hydroinformatics, 17(2)*, pp. 275-292. Available at: http://dx.doi.org/10.2166/hydro.2014.034.

Rosebrock, A. (2016) *Intersection over Union (IoU) for object detection*. Available at: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (Accessed: 11/17/2018).

Ruder, S. (2016) An overview of gradient descent optimization algorithms, *arXiv:1609.04747v2 [cs.LG]*.

Ruizendaal, R. (2017) *Deep Learning #3: More on CNNs & Handling Overfitting*. Available at: https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d (Accessed: 11/15/2018).

"Running locally", Tensorflow (2018) (Accessed: 09/10/2018).

Russakovsky, O. *et al.* (2015) ImageNet Large Scale Visual Recognition Challenge, *International Journal of Computer Vision, vol 115, issue 3*, pp. 211-252.

Shafait, F. *et al.* (2016) Fish identification from videos captured in uncontrolled underwater environments, *ICES Journal of Marine Science, vol 73, Issue 10*, pp. 2737–2746. Available at: https://doi.org/10.1093/icesjms/fsw106.

Shrivastava, A., Gupta, A. and Girschick, R. (2016) Training Region-based Object Detectors with Online Hard Example Mining, *arXiv:1604.03540 [cs.CV]*.

"Sigmoid function", Wikipedia (2002). Available at: https://en.wikipedia.org/wiki/Sigmoid_function (Accessed: 11/14/2018).

"Softmax Regression", Stanford University Wiki (2011). Available at: http://ufldl.stanford.edu/wiki/index.php/Softmax_Regression#Softmax_Regression_vs._k_Binary_Classifiers.

Soulié, F. F. (1991) Neural networks and computing, *Future Generation Computer Systems, vol 7, issue 1*, pp. 69-77.

Srivastava, N. *et al.* (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting, *Journal of Machine Learning Research 15 (2014)*, pp. 1929-1958.

Stang, D. (2017) Step by Step TensorFlow Object Detection API Tutorial — Part 2: Converting Existing Dataset to TFRecord. Available at: https://medium.com/@WuStangDan/step-by-step-tensorflow-object-detection-api-tutorial-part-2-converting-dataset-to-tfrecord-47f24be9248d.

Strachan, N. J. C., Nesvadba, P. and Allen, A. R. (1990) Fish species recognition by shape analysis of images, *Pattern Recognition, vol 23, issue 5*, pp. 539-544.

Sun, X. *et al.* (2018) Transferring deep knowledge for object recognition in Low-quality underwater videos, *Neurocomputing, vol 275*, pp. 897-908.

Szegedy, C. *et al.* (2014) Going Deeper with Convolutions, *arXiv:1409.4842v1 [cs.CV]*.

Tensorflow detection model zoo. Available at: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.

"Transfer Learning", cs231n, Stanford University (2018). Available at: http://cs231n.github.io/transfer-learning/ (Accessed: 11/17/2018).

Wilson, A. C. *et al.* (2017) The Marginal Value of Adaptive Gradient Methods in Machine Learning, *arXiv:1705.08292 [stat.ML]*.

Xiu, L. *et al.* (2015) Fast accurate fish detection and recognition of underwater images with Fast R-CNN, *OCEANS 2015 - MTS/IEEE Washington*.

Zulkifli, H. (2018) *Understanding Learning Rates and How It Improves Performance in Deep Learning*. Available at: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 (Accessed: 11/15/2018).

Appendix

A: Network configuration files

A1: FRCNN Inception v2 config file

```
model {
 faster_rcnn {
  num_classes: 1
  image_resizer {
   keep_aspect_ratio_resizer {
    min_dimension: 600
    max_dimension: 1024
   }
  }
  feature_extractor {
   type: 'faster_rcnn_inception_v2'
   first_stage_features_stride: 16
  }
  first_stage_anchor_generator {
   grid_anchor_generator {
    scales: [0.25, 0.5, 1.0, 2.0]
    aspect_ratios: [0.5, 1.0, 2.0]
    height_stride: 16
    width_stride: 16
   }
  }
  first_stage_box_predictor_conv_hyperparams {
   op: CONV
   regularizer {
    l2_regularizer {
     weight: 0.0
```

```
    }
   }
   initializer {
    truncated_normal_initializer {
     stddev: 0.01
    }
   }
}
first_stage_nms_score_threshold: 0.0
first_stage_nms_iou_threshold: 0.7
first_stage_max_proposals: 300
first_stage_localization_loss_weight: 2.0
first_stage_objectness_loss_weight: 1.0
initial_crop_size: 14
maxpool_kernel_size: 2
maxpool_stride: 2
second_stage_box_predictor {
  mask_rcnn_box_predictor {
    use_dropout: false
    dropout_keep_probability: 1.0
    fc_hyperparams {
      op: FC
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        variance_scaling_initializer {
```

```
            factor: 1.0
            uniform: true
            mode: FAN_AVG
          }
        }
      }
    }
    second_stage_post_processing {
      batch_non_max_suppression {
        score_threshold: 0.0
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 300
      }
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0
    second_stage_classification_loss_weight: 1.0
  }
}

train_config: {
  batch_size: 1
  optimizer {
    momentum_optimizer: {
      learning_rate: {
        manual_step_learning_rate {
          initial_learning_rate: 0.0002
```

```
      schedule {
        step: 2000
        learning_rate: .00002
      }
      schedule {
        step: 5000
        learning_rate: .000002
      }
      schedule {
        step: 7500
        learning_rate: .0000009
      }
    }
  }
  momentum_optimizer_value: 0.9
 }
 use_moving_average: false
}
gradient_clipping_by_norm: 10.0
fine_tune_checkpoint:
"C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\models\\frcnn_inception_coco\\model.ckpt"
from_detection_checkpoint: true
# Note: The below line limits the training process to 200K steps, which we
# empirically found to be sufficient enough to train the COCO dataset. This
# effectively bypasses the learning rate schedule (the learning rate will
# never decay). Remove the below line to train indefinitely.
num_steps: 200000
data_augmentation_options {
 random_horizontal_flip {
```

```
    }
  }
  data_augmentation_options {
    random_crop_image {
      min_object_covered: 0.0
      min_aspect_ratio: 0.75
      max_aspect_ratio: 3.0
      min_area: 0.75
      max_area: 1.0
      overlap_thresh: 0.0
    }
  }
}

train_input_reader {
  label_map_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\label_map.pbtxt"
  tf_record_input_reader {
    input_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\training.record"
  }
  shuffle: true
}
eval_config {
  num_examples: 532
  num_visualizations: 50
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\label_map.pbtxt"
```

```
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\eval.record"
  }
}
```

A2: SSD+FPN Mobilenet v1 config file

```
model {
  ssd {
    num_classes: 1
    image_resizer {
      fixed_shape_resizer {
        height: 640
        width: 640
      }
    }
    feature_extractor {
      type: "ssd_mobilenet_v1_fpn"
      depth_multiplier: 1.0
      min_depth: 16
      conv_hyperparams {
        regularizer {
          l2_regularizer {
            weight: 3.99999989895e-05
          }
        }
        initializer {
          random_normal_initializer {
```

```
      mean: 0.0
      stddev: 0.00999999977648
     }
    }
    activation: RELU_6
    batch_norm {
     decay: 0.996999979019
     scale: true
     epsilon: 0.0010000000475
    }
  }
  override_base_feature_extractor_hyperparams: true
}
box_coder {
  faster_rcnn_box_coder {
    y_scale: 10.0
    x_scale: 10.0
    height_scale: 5.0
    width_scale: 5.0
  }
}
matcher {
  argmax_matcher {
    matched_threshold: 0.5
    unmatched_threshold: 0.5
    ignore_thresholds: false
    negatives_lower_than_unmatched: true
    force_match_for_each_row: true
    use_matmul_gather: true
```

```
  }
}
similarity_calculator {
  iou_similarity {
  }
}
box_predictor {
  weight_shared_convolutional_box_predictor {
    conv_hyperparams {
      regularizer {
        l2_regularizer {
          weight: 3.99999989895e-05
        }
      }
      initializer {
        random_normal_initializer {
          mean: 0.0
          stddev: 0.00999999977648
        }
      }
      activation: RELU_6
      batch_norm {
        decay: 0.996999979019
        scale: true
        epsilon: 0.0010000000475
      }
    }
    depth: 256
    num_layers_before_predictor: 4
```

```
    use_dropout: true
    dropout_keep_probability: 0.500000011920929
    kernel_size: 3
    class_prediction_bias_init: -4.59999990463
  }
}
anchor_generator {
  multiscale_anchor_generator {
    min_level: 3
    max_level: 7
    anchor_scale: 4.0
    aspect_ratios: 1.0
    aspect_ratios: 2.0
    aspect_ratios: 0.5
    scales_per_octave: 2
  }
}
post_processing {
  batch_non_max_suppression {
    score_threshold: 0.300000011921
    iou_threshold: 0.600000023842
    max_detections_per_class: 100
    max_total_detections: 100
  }
  score_converter: SIGMOID
}
normalize_loss_by_num_matches: true
loss {
  localization_loss {
```

```
      weighted_smooth_l1 {
      }
     }
     classification_loss {
      weighted_sigmoid {
      }
     }
     classification_weight: 1.0
     localization_weight: 1.0
    }
    encode_background_as_zeros: true
    normalize_loc_loss_by_codesize: true
    inplace_batchnorm_update: true
    freeze_batchnorm: false
  }
}
train_config {
  batch_size: 2
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_crop_image {
      min_object_covered: 0.0
      min_aspect_ratio: 0.75
      max_aspect_ratio: 3.0
      min_area: 0.75
      max_area: 1.0
```

```
    overlap_thresh: 0.0
   }
  }
  sync_replicas: true
  optimizer {
   momentum_optimizer {
    learning_rate {
     exponential_decay_learning_rate {
      initial_learning_rate: 0.0004000000189989805
      decay_steps: 465
      decay_factor: 0.85000000000
      staircase: true
     }
    }
    momentum_optimizer_value: 0.899999976158
   }
   use_moving_average: false
  }
  fine_tune_checkpoint:
"C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\models\\frcnn_inception_coco\\model.ckpt"
  from_detection_checkpoint: true
  load_all_detection_checkpoint_vars: true
  num_steps: 12500
  startup_delay_steps: 0.0
  replicas_to_aggregate: 8
  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
}
train_input_reader {
```

```
  label_map_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\label_map.pbtxt"
  tf_record_input_reader {
    input_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\training.record"
  }
  shuffle: true
}
eval_config {
  num_examples: 532
  num_visualizations: 50
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\label_map.pbtxt"
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\eval.record"
  }
}
```

A3: SSD Mobilenet v2 config file

```
model {
  ssd {
    num_classes: 1
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
```

```
    height_scale: 5.0
    width_scale: 5.0
  }
}
matcher {
  argmax_matcher {
    matched_threshold: 0.5
    unmatched_threshold: 0.5
    ignore_thresholds: false
    negatives_lower_than_unmatched: true
    force_match_for_each_row: true
  }
}
similarity_calculator {
  iou_similarity {
  }
}
anchor_generator {
  ssd_anchor_generator {
    num_layers: 6
    min_scale: 0.2
    max_scale: 0.95
    aspect_ratios: 1.0
    aspect_ratios: 2.0
    aspect_ratios: 0.5
    aspect_ratios: 3.0
    aspect_ratios: 0.3333
  }
}
```

```
image_resizer {
  fixed_shape_resizer {
    height: 300
    width: 300
  }
}
box_predictor {
  convolutional_box_predictor {
    min_depth: 0
    max_depth: 0
    num_layers_before_predictor: 0
    use_dropout: true
    dropout_keep_probability: 0.5
    kernel_size: 1
    box_code_size: 4
    apply_sigmoid_to_scores: false
    conv_hyperparams {
      activation: RELU_6,
      regularizer {
        l2_regularizer {
          weight: 0.00004
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.03
          mean: 0.0
        }
      }
```

```
    batch_norm {
      train: true,
      scale: true,
      center: true,
      decay: 0.9997,
      epsilon: 0.001,
    }
  }
 }
}
feature_extractor {
 type: 'ssd_mobilenet_v2'
 min_depth: 16
 depth_multiplier: 1.0
 conv_hyperparams {
  activation: RELU_6,
  regularizer {
   l2_regularizer {
    weight: 0.00004
   }
  }
  initializer {
   truncated_normal_initializer {
    stddev: 0.03
    mean: 0.0
   }
  }
  batch_norm {
   train: true,
```

```
    scale: true,
    center: true,
    decay: 0.9997,
    epsilon: 0.001,
   }
  }
}
loss {
  classification_loss {
   weighted_sigmoid {
   }
  }
  localization_loss {
   weighted_smooth_l1 {
   }
  }
  hard_example_miner {
   num_hard_examples: 3000
   iou_threshold: 0.99
   loss_type: CLASSIFICATION
   max_negatives_per_positive: 3
   min_negatives_per_image: 3
  }
  classification_weight: 1.0
  localization_weight: 1.0
}
normalize_loss_by_num_matches: true
post_processing {
  batch_non_max_suppression {
```

```
      score_threshold: 1e-8
      iou_threshold: 0.6
      max_detections_per_class: 100
      max_total_detections: 100
    }
    score_converter: SIGMOID
  }
 }
}

train_config: {
 batch_size: 2
 optimizer {
  rms_prop_optimizer: {
   learning_rate: {
    exponential_decay_learning_rate {
     initial_learning_rate: 0.0004
     decay_steps: 500
     decay_factor: 0.85
    }
   }
   momentum_optimizer_value: 0.9
   decay: 0.9
   epsilon: 1.0
  }
 }
 fine_tune_checkpoint:
"C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\models\\ssd_mobilenet_v2_coco\\model.ckpt
"
```

```
  from_detection_checkpoint: true
  # Note: The below line limits the training process to 200K steps, which we
  # empirically found to be sufficient enough to train the pets dataset. This
  # effectively bypasses the learning rate schedule (the learning rate will
  # never decay). Remove the below line to train indefinitely.
  num_steps: 200000
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_crop_image {
      min_object_covered: 0.0
      min_aspect_ratio: 0.75
      max_aspect_ratio: 3.0
      min_area: 0.75
      max_area: 1.0
      overlap_thresh: 0.0
    }
  }
}

train_input_reader {
  label_map_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\label_map.pbtxt"
  tf_record_input_reader {
    input_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\training.record"
  }
  shuffle: true
}
```

```
eval_config {
  num_examples: 532
  num_visualizations: 50
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\label_map.pbtxt"
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "C:\\Users\\Storm\\Documents\\Optoscale\\FishRecog\\data\\eval.record"
  }
}

graph_rewriter {
  quantization {
    delay: 48000
    weight_bits: 8
    activation_bits: 8
  }
}
```

B: Python scripts

B1: Match images to binary images and rename binary images

```
from PIL import Image
import os
import glob
import numpy as np
```

```
import shutil


currentDir = os.path.dirname(os.path.realpath(__file__))
allBottomDirs = []
for dirpaths, dirnames, filenames in os.walk(currentDir):
	if not dirnames:
		allBottomDirs.append(dirpaths)


numDirs = len(allBottomDirs)
countDir = 1


countDir = 0
for dirPath in allBottomDirs:
	imageFiles = glob.glob(dirPath + '\\*.jpg')
	binaryFiles = glob.glob(dirPath + '\\*.png')
	countFiles = 0
	for imgPath in imageFiles:
		#1 finne dato og klokkeslett og isolere alle bilder tilhørende dette
		filename = os.path.basename(imgPath)
		if('_FullCam1' in filename):
			date = filename.replace('.jpg','').split('_FullCam1')[0]
			#2 finne dag
			day = date.split('_')[-1]
			date = date.replace('_' + day ,'')
			for binaryPath in binaryFiles:
				binaryName = os.path.basename(binaryPath)
				nameParts = binaryName.split('_')
				nameParts = nameParts[0:6]
```

```
                    for idx, part in enumerate(nameParts):
                        if((idx+1) != len(nameParts)):
                            nameParts[idx] += '_'
                        else:
                            nameParts[idx] = nameParts[idx].split('F')[0]

                    binaryDate = ''.join(nameParts)

                    if(date == binaryDate):
                        binaryDay = binaryName.replace(date,'').split('_')[1]
                        binaryLabel = binaryName.replace(date,'').split('_')[0]
                        if(day == binaryDay and not 'Boundary' in binaryName):

        shutil.copy(imgPath,'C:\\Users\\Storm\\Desktop\\Training_set\\ImagesRoi' + '\\' + filename)

        shutil.copy(binaryPath,'C:\\Users\\Storm\\Desktop\\Training_set\\ImagesRoi' + '\\' +
filename.replace('.jpg','_' + binaryLabel + '.png'))
            countFiles +=1
            print('{}/{}'.format(countDir,countFiles))

        countDir += 1

        input('Done')
```

B2: Removal of images that are too close in time when tracking fish

```
#Note: script must be in top dir of images directories


from PIL import Image
import os
import glob
import numpy as np
import shutil


class FishTypes:
	F1 = 0
	F2 = 1
	F3 = 2
	F4 = 3
	F5 = 4


class FishImage:
	def __init__(self, imagePath, binaryPath, day, block):
		self.imagePath = imagePath
		self.binaryPath = binaryPath
		self.day = int(day)
		self.block = int(block)-1


def keepImages(images):
	for fimg in images:
		imageName = os.path.basename(fimg.imagePath)
		binaryName = os.path.basename(fimg.binaryPath)
```

```
                shutil.copy(fimg.imagePath,
'C:\\Users\\Storm\\Desktop\\Training_set\\filteredImages\\' + imageName)
                shutil.copy(fimg.binaryPath,
'C:\\Users\\Storm\\Desktop\\Training_set\\filteredImages\\' + binaryName)



def findBlockImages(matchedPairs):
        blockImages = [[],[],[],[],[]] #F1-5

        for pair in matchedPairs:
                blockImages[pair.block].append(pair)

        return blockImages

def filterImages(blockImages):
        for block in blockImages:
                if(len(block) > 0):
                        matchedPairs = sorted(block, key=lambda x: x.day, reverse=False)
                        keepBin = []
                        #for entry in matchedPairs:
                        #       print(entry.binaryPath)

                        for idx, pair in enumerate(matchedPairs):
                                if((idx*5) >= len(matchedPairs)):
                                        pass
                                else:
                                        keepBin.append(matchedPairs[idx*5])
                        keepImages(keepBin)
```

```
currentDir = os.path.dirname(os.path.realpath(__file__))
allBottomDirs = []

allBottomDirs.append(currentDir + '\\ImagesRoi')

for dirPath in allBottomDirs:
        imageFiles = glob.glob(dirPath + '\\*.jpg')
        binaryFiles = glob.glob(dirPath + '\\*.png')



alreadyHandled = []
for imgPath in imageFiles:
        filename = os.path.basename(imgPath)
        date = filename.replace('.jpg','').split('_FullCam1')[0]
        #2 finne dag
        day = date.split('_')[-1]
        date = date.replace('_' + day ,'')
        if(date in alreadyHandled):
                continue
        else:
                alreadyHandled.append(date)
        matchedPairs = []
        for binaryPath in binaryFiles:
                binaryName = os.path.basename(binaryPath)
                binaryDate = binaryName.split('_FullCam1')[0]
                binaryDay = binaryDate.split('_')[-1]
                binaryDate = binaryDate.replace('_' + binaryDay ,'')

                if(binaryDate == date):
```

```
                imagePath = binaryPath.split('FullCam1_')[0] + 'FullCam1.jpg'
                block = binaryPath.split('FullCam1_F')[1].split('.')[0]
                matchedPairs.append(FishImage(imagePath,binaryPath,binaryDay, block))

    blockImages = findBlockImages(matchedPairs)

    filterImages(blockImages)
```

B3: Find the boundary box of each binary image and store it as a ".txt" file

#Note: script must be in top dir of images directories

```
from PIL import Image
import os
import glob
import numpy as np
import shutil

class FishImage:
    def __init__(self, imagePath):
        self.imagePath = imagePath
        self.binaryPaths = []

    def addBinaryPath(self,path):
        self.binaryPaths.append(path)

class BBox:
    def __init__(self, minx, maxx, miny, maxy):
        self.minx = int(minx)
        self.maxx = int(maxx)
```

```
            self.miny = int(miny)
            self.maxy = int(maxy)
            self.height = self.maxy-self.miny
            self.width = self.maxx-self.minx



def findBoxData(binaryPath):
    img = Image.open(binaryPath)
    binLab = np.asarray(img)

    bboxData = []
    try:
        for idx, x in enumerate(binLab.T):
            for idx2, y in enumerate(x):
                if(binLab[idx2][idx] >= 1):
                    bboxData.append(idx)
                    break
            if(len(bboxData) > 0):
                break

        #max x
        for idx, x in reversed(list(enumerate(binLab.T))):
            for idx2, y in enumerate(x):
                if(binLab[idx2][idx] >= 1):
                    bboxData.append(idx)
                    break
            if(len(bboxData) > 1):
                break
```

```
            #min y
            for idx, y in enumerate(binLab):
                for idx2, x in enumerate(y):
                    if(binLab[idx][idx2] >= 1):
                        bboxData.append(idx)
                        break
                if(len(bboxData) > 2):
                    break
            #max y
            for idx, y in reversed(list(enumerate(binLab))):
                for idx2, x in enumerate(y):
                    if(binLab[idx][idx2] >= 1):
                        bboxData.append(idx)
                        break
                if(len(bboxData) > 3):
                    break

        except:
            print('except')
            return []

        if(len(bboxData) < 4):
            print('len')
            return []
        else:
            return BBox(bboxData[0],bboxData[1],bboxData[2],bboxData[3])

def save(boxData,fimg, bugged=False):
```

```
        dirName = os.path.dirname(os.path.realpath(__file__)) +'\\boundaryData\\'
        fileName = os.path.basename(fimg.imagePath).replace('.jpg', '.txt')
        path = dirName + fileName

        with open(path,'a+') as f:
                if(bugged == False):
                        f.write(str(boxData.minx) + ' ' + str(boxData.maxx) + ' ' +str(boxData.miny) + ' '
+str(boxData.maxy) + ' ' +str(boxData.height) + ' ' +str(boxData.width) + '\n')
                else:
                        f.write('bugged')



currentDir = os.path.dirname(os.path.realpath(__file__))
allBottomDirs = []

allBottomDirs.append(currentDir + '\\filteredImages')

for dirPath in allBottomDirs:
        imageFiles = glob.glob(dirPath + '\\*.jpg')
        binaryFiles = glob.glob(dirPath + '\\*.png')



alreadyHandled = []
allImages = []
for imgPath in imageFiles:
        filename = os.path.basename(imgPath)
        date = filename.replace('.jpg','').split('_FullCam1')[0]
        #2 finne dag
        if(date in alreadyHandled):
```

```
                continue
        else:
                alreadyHandled.append(date)
        fimg = FishImage(imgPath)
        for binaryPath in binaryFiles:
                binaryName = os.path.basename(binaryPath)
                binaryDate = binaryName.split('_FullCam1')[0]

                if(binaryDate == date):
                        fimg.addBinaryPath(binaryPath)
        allImages.append(fimg)

countImgs = 1
totalImgs = len(allImages)
for fimg in allImages:
        for binaryPath in fimg.binaryPaths:
                boxData = findBoxData(binaryPath)
                if(boxData):
                        save(boxData, fimg, bugged=False)
                else:
                        save(boxData, fimg, bugged=True)
        print('{}/{}'.format(countImgs,totalImgs))
        countImgs += 1
```

B4: Remove structured light and put the grayscale image in each of the channels of an RGB image

```
from PIL import Image
import os
import glob
import numpy as np
import shutil

currentDir = os.path.dirname(os.path.realpath(__file__))
allBottomDirs = []

allBottomDirs.append(currentDir + '\\filteredImages')

for dirPath in allBottomDirs:
        imageFiles = glob.glob(dirPath + '\\*.jpg')

saveDir = currentDir + '\\grayImages\\'

countImgs = 1
totalImgs = len(imageFiles)
for imgPath in imageFiles:
        print(imgPath)
        img = Image.open(imgPath)
        r,g,b=img.split()

        red = np.asarray(r,dtype=np.uint8)
        green = np.asarray(g,dtype=np.uint8)
        blue = np.asarray(b,dtype=np.uint8)
```

```
        newImg = np.zeros_like(red,dtype=np.uint8)


        for idx, y in enumerate(red):
                for idx2, x in enumerate(y):
                        newPixelValue = np.uint8((int(red[idx][idx2]) + int(green[idx][idx2])+
int(blue[idx][idx2]))/3)
                        newImg[idx][idx2] = newPixelValue


        newPilImg = Image.fromarray(newImg, 'L')


        rgbimg = Image.new("RGB", r.size)
        rgbimg.paste(newPilImg)
        filename = os.path.basename(imgPath)
        rgbimg.save(saveDir + filename)
        print('{}/{}'.format(countImgs,totalImgs))
        countImgs += 1
```

B5: Create the TFRecord file with metadata on fish locations

```
import tensorflow as tf
from object_detection.utils import dataset_util
import os
import glob
import random
from PIL import Image
import shutil



class ImageData:
```

```
        def __init__(self, height, width, filename, source_id, encoded_img, xmin, xmax, ymin, ymax,
labels, texts):
                self.height = height
                self.width = width
                self.filename = filename
                self.source_id = source_id
                self.encoded_img = encoded_img
                self.xmin = xmin
                self.xmax = xmax
                self.ymin = ymin
                self.ymax = ymax
                self.texts = texts
                self.labels = labels


def create_tf_example(data):
        # TODO START: Populate the following variables from your example.
        height = data.height # Image height
        width = data.width # Image width
        filename = data.filename # Filename of the image. Empty if image is not from file
        encoded_image_data = data.encoded_img # Encoded image bytes
        image_format = b'jpeg'

        xmins = data.xmin # List of normalized left x coordinates in bounding box (1 per box)
        xmaxs = data.xmax # List of normalized right x coordinates in bounding box
                                                # (1 per box)
        ymins = data.ymin # List of normalized top y coordinates in bounding box (1 per box)
        ymaxs = data.ymax # List of normalized bottom y coordinates in bounding box
                                                # (1 per box)
```

```
        classes_text = data.texts # List of string class name of bounding box (1 per box)
        classes = data.labels # List of integer class id of bounding box (1 per box)
        # TODO END
        tf_label_and_data = tf.train.Example(features=tf.train.Features(feature={
                        'image/height': dataset_util.int64_feature(height),
                        'image/width': dataset_util.int64_feature(width),
                        'image/filename': dataset_util.bytes_feature(str.encode(filename)),
                        'image/source_id': dataset_util.bytes_feature(str.encode(filename)),
                        'image/encoded': dataset_util.bytes_feature(encoded_image_data),
                        'image/format': dataset_util.bytes_feature(image_format),
                        'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
                        'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
                        'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
                        'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
                        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
                        'image/object/class/label': dataset_util.int64_list_feature(classes),
        }))
        return tf_label_and_data

def main(_):
        writer = tf.python_io.TFRecordWriter('training.record')
        evalwriter = tf.python_io.TFRecordWriter('eval.record')
        testwriter = tf.python_io.TFRecordWriter('test.record')

        currentDir = os.path.dirname(os.path.realpath(__file__))
        allBottomDirs = []

        allBottomDirs.append(currentDir + '\\grayImages')
```

```
for dirPath in allBottomDirs:
        imageFiles = glob.glob(dirPath + '\\*.jpg')

random.shuffle(imageFiles)
valNumber = int((len(imageFiles)*0.7))
testNumber = int((len(imageFiles)*0.9))

txtDir = currentDir + '\\boundaryData\\'
countImgs = 0
totalImgs = len(imageFiles)
buggedImgs = 0
evalSamples = 0
testSamples = 0
for imgPath in imageFiles:
        filename = os.path.basename(imgPath)
        txtPath = txtDir + filename.replace('.jpg','.txt')
        file = open(txtPath, 'r')
        text = file.read()
        if(not text == 'bugged'):
                with Image.open(imgPath) as img:
                                                img_width, img_height = img.size
                allBoxes = text.split('\n')
                xmin = []
                xmax = []
                ymin = []
                ymax = []
                labels = []
                texts = []
                for box in allBoxes:
```

```
                    if(box != ''):
                        if(box != 'bugged'):
                            values = box.split(' ')
                            xmin.append(float(values[0])/float(img_width))
                            xmax.append(float(values[1])/float(img_width))
                            ymin.append(float(values[2])/float(img_height))
                            ymax.append(float(values[3])/float(img_height))
                            labels.append(0)
                            texts.append(str.encode('Fish'))
                        else:
                            print('not good')



                height = int(img_height)
                width = int(img_width)
                file.close()


                encoded_img = tf.gfile.FastGFile(imgPath,'rb').read()
                filename = os.path.basename(imgPath)
                #self, height, width, filename, source_id, encoded, xmin, xmax, ymin, ymax,
text, label
                data = ImageData(height, width,
filename,filename,encoded_img,xmin,xmax,ymin,ymax, labels, texts)
                tf_example = create_tf_example(data)
                if(countImgs < valNumber):
                    writer.write(tf_example.SerializeToString())
                    shutil.copy(imgPath, 'C:\\Users\\Storm\\Desktop\\Training_set\\train\\'
+ filename)
                elif(countImgs < testNumber):
                    evalwriter.write(tf_example.SerializeToString())
```

```
                                evalSamples += 1
                                shutil.copy(imgPath, 'C:\\Users\\Storm\\Desktop\\Training_set\\eval\\'
+ filename)
                        else:
                                testwriter.write(tf_example.SerializeToString())
                                testSamples += 1
                                shutil.copy(imgPath, 'C:\\Users\\Storm\\Desktop\\Training_set\\test\\'
+ filename)
                else:
                        buggedImgs += 1



                countImgs += 1
                print('{}/{}'.format(countImgs,totalImgs))

        writer.close()
        evalwriter.close()

        print('Num bugged imgs: {}'.format(buggedImgs))
        print('Eval samples: {}'.format(evalSamples))
        print('Test samples: {}'.format(testSamples))
        print('Total samples: {}'.format(totalImgs))

if __name__ == '__main__':
        tf.app.run()
```

B6: Load extracted graph and run inference to do measure fps

```
import tensorflow as tf

import numpy as np

import os

import glob

from utils import label_map_util

from utils import visualization_utils as vis_util

from PIL import Image

from matplotlib import pyplot as plt

import time

plt.switch_backend('TkAgg')


def load_image_into_numpy_array(image):

        (im_width, im_height) = image.size

        return np.array(image.getdata()).reshape((im_height, im_width, 3)).astype(np.uint8)


def run_inference_on_images(graph):


        with graph.as_default():


                with tf.Session() as sess:


                        ops = tf.get_default_graph().get_operations()

                        all_tensor_names = {output.name for op in ops for output in op.outputs}

                        tensor_dict = {}

                        for key in [

                                'num_detections', 'detection_boxes', 'detection_scores',

                                'detection_classes', 'detection_masks'

                        ]:
```

```
                tensor_name = key + ':0'
                if tensor_name in all_tensor_names:
                        tensor_dict[key] =
tf.get_default_graph().get_tensor_by_name(tensor_name)
        image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')


        count = 0
        start_time = time.time()
        currentDir = os.path.dirname(os.path.realpath(__file__))
        dirFiles = glob.glob(currentDir + '\\test\\*.jpg')
        for imgPath in dirFiles:
                if(count == 10):
                        start_time = time.time()
                img = Image.open(imgPath)
                image = load_image_into_numpy_array(img)
                # Get handles to input and output tensors


                # Run inference
                output_dict = sess.run(tensor_dict, feed_dict={image_tensor:
np.expand_dims(image, 0)})


                # all outputs are float32 numpy arrays, so convert types as appropriate
                output_dict['num_detections'] = int(output_dict['num_detections'][0])
                output_dict['detection_classes'] =
output_dict['detection_classes'][0].astype(np.uint8)
                output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
                output_dict['detection_scores'] = output_dict['detection_scores'][0]
                if 'detection_masks' in output_dict:
                        output_dict['detection_masks'] =
output_dict['detection_masks'][0]
```

```
                    count += 1



                end_time = time.time()
                fps = ((266-10)/(end_time-start_time))
                print('FPS: {}'.format(fps))


PATH_TO_LABELS = 'label_map.pbtxt'
PATH_TO_FROZEN_GRAPH = 'frozen_inference_graph.pb'


category_index = label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
use_display_name=True)
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')


run_inference_on_images(detection_graph)
```