NTNU
Norwegian University of
Science and Technology

Deep learning in Dynamic Imager

A convolutional neural network module

Johan Nikolai Slettevold

Master of Science in Informatics	
Submission date:	June 2018
Supervisor:	Frank Lindseth, IDI
Co-supervisor:	Ketil Bø, Trollhetta AS

Norwegian University of Science and Technology
Department of Computer Science

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Deep Learning in Dynamic Imager

A Convolutional Neural Network Module

by

Johan Nikolai Slettevold

Supervised by

Frank Lindseth

Ketil Bø

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

June 2018

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

This thesis investigates the extent of which deep learning methods can be used for automatic detection of salmon in images. It also investigates the extent of which a module with deep learning functionality can be integrated into an already existing program.

To solve these tasks, a comparison of state-of-the-art object detectors using convolutional neural networks is conducted. A dataset consisting of underwater images of salmon is annotated, then used to train a model based on the YOLOv2 (You Only Look Once version 2) object detection method. A module using the Darknet software framework is created for use with the image processing program Dynamic Imager. Finally, the trained models are evaluated based on their accuracy and speed.

The best performing model achieve an average precision score of 80.2%. Models tested using a graphical processing unit achieves a prediction time of 0.8 frames per second. Models tested in Dynamic Imager without access to a graphical processing unit achieve a prediction time of 0.1 frames per second.

Contents

List of Figures

List of Tables

YOLOv2	**Y**ou **O**nly **L**ook **O**nce **v**ersion **2**
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DL	**D**eep **L**earning
ML	**M**achine **L**earning
CNN	**C**onvolutional **N**eural **N**etwork
RBM	**R**estricted **B**oltzmann **M**achine
AI	**A**rtificial **I**ntelligence
RQ	**R**esearch **Q**uestion
SEKOM	**SE**nter for Faglig **KOM**munikasjon
ANN	**A**rticifial **N**eural **N**etwork
tanh	**h**ypoerbolic **tan**gent
ReLU	**Re**ctified **L**inear **U**nit
MSE	**M**ean **S**quared **E**rror
SGD	**S**tochastic **G**radient **D**escent
AP	**A**verage **P**recision
IoU	**I**ntersect **o**ver **U**nion
COCO	**C**ommon **O**bjects in **CO**ntext
PASCAL VOC	The **P**attern **A**nalysis **S**tatistical Modelling and **C**omputational **L**earning **V**isual **O**bject **C**lasses
MAP	**M**ean **A**verage **P**recision
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
GPU	**G**raphical **P**rocessing **U**nit
NMS	**N**on-**M**aximum **S**uppression
FPS	**F**rames **P**er **S**econd
SSD	**S**ingle **S**hot Multibox **D**etector
R-FCN	**R**egion-Based **F**ully **C**onnected **N**etwork
CPU	**C**entral **P**rocessing **U**nit
DL4J	**D**eep **L**earning for **J**ava
BSD	**B**erkley **S**oftware **D**istribution
MIT	**M**assachusetts **I**nstitute of **T**echnology
	Massachusetts Institute of Technology
NTNU	**N**orges **T**eknisk-**N**aturvitenskapelige **U**niversitet
DLL	**D**ynamic **L**ink **L**ibrary
JSON	**J**ava**S**cript **O**bject **N**otation

VM	**V**irtual **M**achine
gcloud	**g**oogle **cloud** Platform
OS	**O**perating **S**ystem
CLI	**C**ommand **L**ine **I**nterface
SDK	**S**oftware **D**evelopment **K**it
SSH	**S**ecure **SH**ell
cuDNN	**cu**da **D**eep **N**eural **N**etwork **L**ibrary
OpenCV	**Open** Source **C**omputer **V**ision Library
VS	**V**isual **S**tudio
SUS	**S**ystem **U**sability **S**cale
GUI	**G**raphical **U**ser **I**nterface
MSI	**M**icro-**S**tar **I**nternational
RAM	**R**andom **A**ccess **M**emory

Chapter 1

Introduction

This chapter will introduce the reader to the the main subject of the thesis and the reasons for why this work is important. It will also list the thesis' research questions, describe its contribution, and describe its structure.

1.1 Deep Learning

Deep learning (DL) is a field of research which has gained a lot of attention the last years. According to Web of Science, the number of published articles on the subject has grown rapidly from 3 publications in 2014, to 23, 78, and 71 publications in the following three years respectively [1]. DL is a subset of machine learning (ML) containing methods that are able to learn high level abstraction of data using computational models composed of multiple processing layers [2] [3]. The term deep or depth refers to the number of layers in a model, and a model is usually referred to as deep if it contains two or more layers [1]. DL methods can be used to process data such as images, video, audio, speech, and text, and have successfully been used for problems such as image classification, object detection, semantic segmentation, image retrieval, human pose estimation, video analysis, and speech recognition [3] [2] [4]. DL methods can be split into the four categories: convolutional neural networks (CNN), restricted boltzmann machines (RBM), autoencoders, and sparse coding [2]. However, CNN methods are the most utilized and most suitable for processing images [2].

1.2 Motivation

This thesis will explore automatic detection of salmon in images from fish farms, investigating the generality of object detection using CNNs. The motivation is to help Trollhetta AS to integrate CNNs to their solutions, and help the fish farm industry to estimate biomass in farms.

Because DL methods such as CNNs can be used to solve a large variety of problems, it is important for a company focusing on machine vision and artificial intelligence (AI) such as Trollhetta, to be able to readily utilize such methods in their work [5]. A CNN module implemented for one of their image processing programs, Dynamic Imager, will help them to do so.

Being able to identify and measure salmon in images could help the fish farming industry with estimations of biomass. Estimation of biomass in fish farms is important for several reasons and bad estimates has consequences throughout the production pipeline [6]. These consequences include misfeeding and mismedicating the fish, and not being able to tell whether the amount of fish has surpassed the legal maximum [6]. In a farm with a yearly production of 900 000 ton, a five percent overestimation of biomass can lead to a loss of 870 million Norwegian kroner (NOK) per year [6]. While a five percent underestimation of biomass can lead to a loss of 126 million NOK per year [6]. Mismedication of the fish could also result in the development of vaccine resistance in lice [6].

1.3 Research Questions

The goals of this thesis is to achieve automatic detection of salmon in images from fish farms using DL, and to implement a module capable of this task into the program Dynamic Imager. Sub-questions were created for each of the main research questions to be able to easier answer these. The research questions are the following:

RQ 1: To what extent can deep learning methods be used to detect salmon in images?

- **(a)** Which deep learning methods are suitable for detecting salmon in images?
- **(b)** How accurate are these deep learning methods?

(c) How fast are these deep learning methods?

RQ 2: To what extent can a deep learning module be integrated into an existing program?

(a) Which software frameworks and libraries are suitable for implementing such a module for DynamicImager?

(b) How accurate is such a module integrated into DynamicImager?

(c) How fast is such a module integrated into DynamicImager?

1.4 Contribution

This thesis provides the following contributions:

- Evaluations of state of the art CNN object detectors;
- Annotations of a dataset containing 632 images of salmon;
- Examples and results of high performing YOLOv2 models;
- Implementation of a DL module for Dynamic Imager, capable of testing images, training models, and validating models.

1.5 Thesis Structure

This thesis is structured into six parts as recommended by NTNU Centre for Academic and Professional Communication (SEKOM). The introduction chapter contains an introduction to the field of study, and the details surrounding the project and thesis. The background chapter contains descriptions to the various research, topics, and terms used in the thesis. The method chapter contains the details about the literature review conducted, and how the various project tasks were performed. The results chapter contains project results and their explanations. The discussion chapter contains my thoughts about the results and their ability to answer the research questions. And finally the conclusion chapter contains the answers to the research questions, thoughts about the importance of the work, and ideas for further work.

Chapter 2

Background

This chapter will introduce the reader to the research used to answer the research questions. It will also explain why these studies are relevant for this thesis. This chapter will aim to only discuss theory which is utilized in later chapters.

2.1 Artificial Neural Networks

A CNN is a type of a artificial neural network (ANN). ANNs are a DL method that is inspired by the biological neural networks in the brains of animals [7]. ANNs consists of a network of neurons connected together with adjustable weights [8]. Each neuron is also connected to a bias weight [8]. The neurons are divided into layers. The first layer of neurons is called the input layer, the last layer is called the output layer, and the layers in between are called hidden layers [9]. The neurons in these layers connected together in order from input layer to output layer.

2.1.1 Types of Learning

ML can be divided into supervised and unsupervised learning [9]. In supervised learning, a model is trained to find structure in data using additional information about the data [9]. This information is in the form of a training set with input/output pairs (labeled data), or in the form of a performance function in reinforcement learning [9]

[7]. In unsupervised learning, a model is trained to find structure in data without using additional information about the data [9].

There are two kinds of supervised learning problems using labeled data, classification and regression [8]. In classification problems, the goal is to classify the input as correct class out of a finite set of classes [8]. In regression problems, the goal is to approximate the correct continuous value based on the input [8]. To reach these goals, a model is trained for its task. Training a ANN model consists of giving the model an input from the training set, calculating the output, measuring the error of the output based on the correct output, and finally adjusting the weights [8].

2.1.2 Activation Functions

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

When a neuron receives a set of inputs, it can calculate its output based on an activation function [8]. In equation 2.1, y represents the output of a neuron, f represents the activation function, n represents the number of iterations, i represents the current iteration, w_i represents the current weight, x_i represents the current input signal, and b represents the bias [8]. This allows the ANN to calculate (propagate) the values from the neurons in the input layer, through the hidden layers, to the output layer. An activation function defines the output of a neuron given a set of inputs, and could be compared to a switch turning on or off [10]. Linear activation functions can be used, but non-linear activation functions are required to be able to learn non-convex structures [10]. Activation functions can also make sure that neuron output values are in specific ranges [9]. Examples of activation functions include the sigmoid function, the hyperbolic tangent (tanh) function, the rectified linear unit (ReLU), the leaky ReLU, and the softmax function [8] [10] [11]. Equation 2.2 shows the sigmoid function with range is $(0, 1)$ [12] [8]. Equation 2.3 shows the tanh function with range is $(-1, 1)$ [8]. Equation 2.4 shows the ReLU function with range is $[0, \inf)$ [8]. Equation 2.5 shows the leaky ReLU function with range $(-\inf, \inf)$, a is a positive number lower than one [10]. Using the softmax function on a layer returns the categorical probability distribution [11]. It has a range of $(0, 1)$ and the total sum of the neurons in the layer is set to one [11]. The ReLU function is the most used activation function in ANNs [7].

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

$$f(x) = \frac{2}{1+e^{2x}} - 1 \tag{2.3}$$

$$f(x) = \max(x, 0) \tag{2.4}$$

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

2.1.3 Loss Functions

A loss function is used to measure the error of an output (prediction) based on the correct label [9]. The goal of training is to minimize this error, and it is often normalized to a score between 0 and 1 [8] [10]. The most commonly used loss functions are the cross entropy function and mean squared error function (MSE) for classification and regression problems, respectively [10]. Equation 2.6 shows the cross entropy function. W represents a weight matrix, N represents the number of training examples, i represents the current training example, t_i represents the correct values for training example i, and y_i represents the output values for i [13] [10]. Equation 2.7 shows the MSE function [10]. Its notation is similar to equation 2.6. $||t_i - y_i||$ represents the magnitude of the matrix $t_i - y_i$ [10].

$$E(W) = -\frac{1}{N}\sum_{i=1}^{N} t_i \log y_i + (1-t_i)\log(1-y_i)) \tag{2.6}$$

$$E(W) = \frac{1}{N}\sum_{i=1}^{N} ||t_i - y_i||^2 \tag{2.7}$$

2.1.4 Optimization Algorithms

To adjust the weights, different optimization algorithms such as stochastic gradient descent (SGD), RMSProp, Adamax, or Adam are utilized [7] [14] [15]. If the network has more than two layers, a method called backpropagation is also used [10]. Backpropagation calculates partial derivatives (gradients) of the loss function with respect to the neurons in the network [8]. The values are calculated layerwise from the output layer backwards through the hidden layers. Optimization algorithms use these gradients to decide how much the neurons weights should change. The gradients can be multiplied with a learning rate to control the rate of change in the weights [10]. The weights in the earlier layers takes more time to train than the weights in the late layers because of the vanishing gradient problem [12]. The vanishing gradient problem describes the effect of the gradients decreasing in size as they are backpropagated [12]. Smaller gradients lead to smaller changes in the weights.

2.1.5 Training

During the training of a model, the same training data is often passed through the network several times [8]. For every time the model has trained on the entire training set, an epoch is completed [8]. These epochs are usually split into batches, and weights are adjusted after each batch [8]. However, the model can become overfit if it trains too much on the training data [8]. Overfitting is when the model learns something that was specific to the training data, but which is not a trend in other data [9]. When a model is overfit it will perform very well on the training data, but it is not able to generalize [9].

2.2 Convolutional Neural Networks

CNN assumes that its inputs are images [15]. Instead of processing the input pixel by pixel, it processes the image using two dimensional patches of the image [15]. Two problems CNNs can be created and trained to solve are image classification and object detection. In image classification, the goal is to identify one or more objects in an image [9]. In object detection, the goal is to identify and also locate the objects in an image [9]. Locations are usually represented using axis aligned bounding boxes, which are simply rectangles containing the object [9]. The labels in a object detection training set consists of bounding boxes called ground truth boxes [9]. The location of an object in an image is usually represented as four numbers, and can be posed as a regression problem [16].

There are three main kinds of layers in CNNs: convolutional layers, pooling layers, and fully connected layers [8].

2.2.1 Convolutional Layers

Convolutional layers are composed of multiple two dimensional arrays of neurons called feature maps, and are connected through multiple two dimensional patches of weights called filters or kernels [8] [7]. It is called convolutional because it uses a form of mathematical convolution instead of matrix multiplication to calculate its output [7]. In equation 2.8, s is the convoluted function of t, and x and w are functions [17]. The asterix between x and w does not represent multiplication nor dot product. Equation 2.9 is a discrete convolution with steps from negative infinity to positive infinity [17]. Equation 2.10 is a two-dimensional discrete convolution with steps from (0,0) to (m, n) [17]. Here, (i, j) represents coordinates in the output feature map S, $(i - m, j - n)$ represents the coordinates in input I, and (m, n) represents the coordinates in the kernel K [17]. However, many convolutional neural network libraries uses a function called cross-correlation rather than a convolution [17]. Equation 2.11 is a two-dimensional discrete cross-correlation with steps from (0,0) to (m, n) [17]. One of the reason convolutional layers perform well is because of sparse connectivity [7] [18]. Sparse connectivity means that not every neuron in two neighbouring layers are connected [7] [18]. This results in a lower amount of operations to compute [7] [18]. In a convolutional layer, the kernel is shared for every input image from the previous layers [7]. Because the kernels are smaller than the input images, this reduces the amount of weights in the network [7]. A network with fewer weights requires less memory to store, and is less likely to be overfitted [7] [10].

$$s(t) = (x * w)(t) \tag{2.8}$$

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.9}$$

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(i-m, j-n)K(m,n) \tag{2.10}$$

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.11}$$

2.2.2 Pooling and Fully Connected Layers

Pooling layers are used to reduce the size of the feature maps, which also reduces the number of parameters in a network [8]. The most common pooling method is maxpooling using a kernel with size 2x2 and a stride of two [8]. The stride refers to the distance between each utilized patch in the input image. Maxpooling outputs the maximum value in each patch [9]. In fully connected layers, every input neuron is connected to every output neuron [8]. A common way to organize the layers in a CNN is to use a couple of convoluational layers followed by a pooling layer, then repeat this pattern until the feature maps are quite small [10]. In the final layers it is common to use fully connected layers [10]. In the early layers of a CNN, features such as edges and corners are detected [7] [3]. Then, for each layer in the network the representations are more abstracted [7] [3]. In the later layers more specific patterns can be detected [7] [3].

2.2.3 Measuring Performance

$$Precision = \frac{True\ positives}{Number\ of\ ground\ truth\ boxes} \tag{2.12}$$

$$Recall = \frac{True\ positives}{True\ positives + False\ positives} \tag{2.13}$$

$$IoU = \frac{Area\ of\ intersect\ between\ prediction\ and\ label}{Area\ of\ union\ between\ prediction\ and\ label} \tag{2.14}$$

To measure the accuracy of a object detection system, it is common to use average precision (AP) [9]. AP is calculated using a precision score and a recall score [9]. Precision represents how correct the predictions are, and recall represents how many of the ground truth boxes were included in the output [9]. These two scores can be calculated using the number of true positives, false positives, and the number ground truth boxes from a prediction and label of an image [9]. Equation 2.12 shows how to calculate precision [9]. Equation 2.13 shows how to calculate recall [9]. True positives are often defined as a bounding box having an intersection over union (IoU) of 0.5 or larger with a ground truth box of the same class [9] [19]. Equation 2.14 shows how to calculate IoU [20]. Each prediction also has a confidence score [9]. By adjusting the threshold of the minimum amount of confidence a predicted box must have to be included in the output, it is possible to calibrate the balance between having a high precision and a high recall [9]. The precision-recall curve is the plot of average precision at every value of recall [21]. This plot can be approximated using the values of precision and recall at every confidence threshold between zero and one [21]. AP is defined as the area under the precision-recall curve [21]. Equation 2.15 is the definition of AP [21]. $p(r)$ represents precision p at a given recall value r [21]. Equation 2.16 shows the approximated AP [21]. k represents a set of confidence threshold values, $p(k)$ represents precision p at a given confidence value k, and $\Delta r(k)$ represents the change in recall r at a given confidence value k [21]. Some authors uses the interpolated average precision instead [21]. The interpolated AP doesn't use the precision at each confidence threshold, but instead uses the maximum precision found with the same recall for each confidence threshold [21].

$$AP = \int_0^1 p(r)dr \tag{2.15}$$

$$AP \approx \sum_k p(k)\Delta r(k) \tag{2.16}$$

2.3 ANN Design Patterns

These are some of the frequently discussed ANN design patterns from my literature review. They are relevant because of their use in CNN methods.

2.3.1 Momentum

Momentum helps the optimization algorithm to not get stuck in a sub optimal minima [10]. It is a value set between zero and one, and it controls how much the weights should consider their last adjustment when updating [10]. If the momentum is set to zero, the update in weights is only dependant on the gradient of the loss function [10]. If the momentum is set to one, the update in weights is only dependant on the last weight adjustment [10]. A commonly used momentum value is 0.9 [10] [22] [23] [24].

2.3.2 Learning Rate Annealing

Learning rate annealing helps the optimization algorithm to reach minimas [25]. If the learning rate is too large the optimization algorithm might fluctuate near a minima without being able to reach it [25]. Therefore, it is generally a good practice to reduce the learning rate over time [25]. This can be done by slowly lowering the learning rate after each weight update, which is called exponential annealing [25]. It can also be done by updating the learning rate by a large amount after a set amount of weight updates, this is called step learning annealing [25].

2.3.3 Weight Decay

Weight decay can help the model to generalize better and avoid overfitting [10]. Weight decay decreases the weights over time [10]. The decrease is proportional to the weights, therefore, large weights are decreased more than small weights [10]. This forces the network to find a solution using small weights [10]. L2 regularization is a weight decay method. It adds an additional term to the loss function [25]. The term is a set fraction of the sum of the weights squared [25]. This causes larger weights to have larger errors which leads to smaller weights [25]. Weight decay factors ranging from 0.0005 to 0.002 are the most common [10] [22] [23].

2.3.4 Dropout

Dropout can also help the model generalize better and avoid overfitting [9]. Dropout works by dropping random neurons and all their connections during a training step [9]. The weights in these connections are not updated during the training step [10]. Neurons that can be affected by dropout are usually placed in a type of layer called a dropout layer [25]. The neurons are dropped with a probability of $1 - p$, where p is a number between zero and one [10].

2.3.5 Batch Normalization

Batch normalization helps speed up the training process by normalizing the inputs to a network layer [10]. This solves the internal covariate shift problem [10]. The internal covariate shift problem describes an issue with value distribution in network layers [10]. When the weights in one layer changes, it also changes the distribution of values in the next layer [10]. This shifts all the values in the network, reducing the difference between the weights which slows down training [10].

2.3.6 Transfer Learning

Transfer learning is a commonly used design method to mitigate the effects of too few training images or the lack of processing power [15] [26] [27]. It works by reusing weight configuration from a pre-trained network in a new network [15] [26] [27]. Transfer learning works because the early layers of a CNN typically learns basic features such as edges and corners, which many objects have in common [15] [28]. There are two types of transfer learning [15] [29]. With frozen layers, the imported weights do not change during further training, but with fine tuning, the imported weights are further trained [15] [29].

2.3.7 Data Augmentation

Data augmentation is used to avoid overfitting by increasing the size of the training set [10]. It does this by adding modified versions of the training set images to the training set. Ways to modify an image include: scaling the image, shifting the image, stretching the image, flipping the image, cropping the image, changing brightness, changing color saturation, changing color hue, changing the colors, and adding noise [14] [9] [10].

2.3.8 Dataset Splitting

Dataset splitting is used to increase the testability of a model [15] [30]. It is essential to test the model on data it has not seen during training, therefore the dataset is often split [9]. It is common to split the data into either a training set and a test set, or a training set, a test set, and a validation set [15] [30] [8]. The training set is used for training, the validation set is used for model comparison, and the test set is used for evaluating the model [15] [30]. If a validation set is included, it is common to use a distribution of 60% data in the training set, 20% data in the validation set, and 20% data in the test set [15]. Otherwise, it is common to use distributions of 90%, 80%, or 70% data in the training set, and 10%, 20%, or 30% data in the test set, respectively [8]. By tracking the accuracy of these datasets during training, it is possible to determine when the model has reached peak performance [8]. If the training set error is decreasing over time, while the validation or test set error is not, the model is overfit to the training set [15] [30].

2.4 CNN Datasets

These are some of the frequently discussed datasets from my literature review. They are relevant because of their ability to benchmark solutions.

2.4.1 MNIST

MNIST is a dataset of handwritten digits with dimensions 28x28 pixels [31] [32]. It has 60 000 training examples and 10 000 testing examples [31] [32]. The dataset is labeled with the correct digits for each image, thus it is a classification problem. The MNIST website has a list of example solutions and their test error scores [32]. The example solution with the lowest error is a CNN with and error of 0.23% [32].

2.4.2 COCO

Common Objects in Context (COCO) is a dataset with images labeled for several computer vision problems [33]. The COCO object detection dataset has over 200 000 images with 80 different classes [34] [35]. They include 80 000 training images, 40 000 validation images, and 80 000 testing images [34] [35]. The COCO website hosts a leaderboard with the most successful solutions to their computer vision challenges [36]. The top rated object detection solution from the 2017 COCO challenge has an AP of 0.526 [36].

2.4.3 PASCAL VOC

The Pattern Analysis, Statistical Modelling and Computational Learning Visual Object Classes (PASCAL VOC) is a computer vision challenge that ran from 2005 to 2012 [37] [9] [19]. The datasets from the challenges in 2007, 2010, and 2012 are still being used to benchmark object detection solutions [9] [19]. The datasets from these challenges can be combined to form a dataset of 27 090 images with 20 different classes [9] [19]. The PASCAL VOC website hosts a leaderboard with the most successful solutions to their challenges [38]. This leaderboard is split into solutions that has only trained on the PASCAL VOC training data, and solutions that has pre-trained on additional data [38]. The top rated object detection solution from the 2012 PASCAL VOC challenge has 74.1% mean AP (MAP) and 88.8% MAP, for solutions only trained on PASCAL VOC data and solutions pre-trained on additional data, respectively [38]. The MAP represents the mean of the APs of every type of object (class).

2.4.4 ImageNet

ImageNet is a dataset that has more than 14 million images covering more than 20 000 classes [39]. There are over a million images with object detection labels [39]. ImageNet hosts a yearly computer vision challenge called the Large Scale Visual Recognition Challenge (ILSVRC) [40]. Their website hosts a leaderboard with the most successful solutions to their challenges [41]. This leaderboard is split into solutions that has only trained the ImageNet training data, and solutions that has pre-trained on additional data [41]. The top performing object detection solutions for ILSVRC 2017 have 0.732227 MAP and 0.731613 MAP, for solutions only trained on ImageNet training data and solutions pre-trained on additional data, respectively [41].

2.5 CNN Classification Methods

These are some of the frequently discussed CNN classification methods from my literature review. They are relevant because of their ability identify and locate objects in images. Although classification methods cannot be used for object detection, their network structures may be reused in object detection methods.

2.5.1 AlexNet

AlexNet is a CNN method that won the classification and the localization challenge in the ILSVRC 2012 [42]. The network has 60 million parameters and 650 000 neurons [22]. AlexNet uses convolutional, maxpooling, and fully connected layers [22]. It uses ReLU activations and the SGD optimization algorithm with a batch size of 128 [22]. AlexNet implements several design patterns, such as: momentum of 0.9, step learning annealing starting at 0.01 and reduced three times, weight decay of 0.0005, dropout layers with $p = 0.5$, and data augmentation including cropping, flipping, and changing color intensity [22]. The model was trained for five to six days on a Nvidia GTX 580 3GB graphical processing unit (GPU) [22]. Table 2.1 shows the AlexNet layer architecture. The number next to the network name is the depth of the network [22] [30] [43]. The number before the layer names is short for several layers of the same type in sequence. The number next to the layer names is the size of the kernels in that layer. The numbers following the layer names are the size and dimension of the output of that layer. Code is available at `https://code.google.com/archive/p/cuda-convnet/` [22].

AlexNet-8	
input	227x227x3
conv-11	55x55x96
maxpool-3	27x27x96
conv-5	27x27x256
maxpool-3	13x13x256
2 x conv-3	13x13x384
conv-3	13x13x256
maxpool-3	6x6x256
2 x fully connected	4096
fully connected	1000
softmax	

TABLE 2.1: AlexNet

2.5.2 VGGNet

VGGNet is a CNN method that won the localization challenge and came in second on the classification challenge for solutions only trained on provided training data in the ILSVRC 2014 [44]. The two most commonly used versions of VGGNet are the VGG-16 and VGG-19 networks [12]. The VGG-16 architecture has 138 million parameters and the VGG-19 architecture has 144 million parameters [45]. It uses convolutional, maxpooling, and fully connected layers, but it is also popularly used without its fully connected layers [45] [12]. Without its fully connected layers it only has about 15 million parameters [12]. VGGNet uses ReLU activations and the SGD optimization algorithm with a batch size of 256 [45]. It implements several design patterns, such as: momentum of 0.9, step learning annealing starting at 0.01 and reduced three times, weight decay of 0.0005, dropout layers with $p = 0.5$, and data augmentation including cropping, rescaling, flipping, and color shifting [45]. The model was trained for 370 000 iterations [45]. Table 2.2 shows the VGGNet layer architectures [45] [46]. The notation is similar to table 2.1. Weights and layer configurations are available at `http://www.robots.ox.ac.uk/~vgg/research/very_deep/` [45].

VGG-16		**VGG-19**	
input	224x224x3	input	224x224x3
2 x conv-3	224x224x64	2 x conv-3	224x224x64
maxpool-2	112x112x128	maxpool-2	112x112x128
2 x conv-3	112x112x128	2 x conv-3	112x112x128
maxpool-2	56x56x256	maxpool-2	56x56x256
3 x conv-3	56x56x256	4 x conv-3	56x56x256
maxpool-2	28x28x512	maxpool-2	28x28x512
3 x conv-3	28x28x512	4 x conv-3	28x28x512
maxpool-2	14x14x512	maxpool-2	14x14x512
3 x conv-3	14x14x512	4 x conv-3	14x14x512
maxpool-2	7x7x512	maxpool-2	7x7x512
2 x fully connected	4096	2 x fully connected	4096
fully connected	1000	fully connected	1000
softmax		softmax	

Table 2.2: VGGNet

2.5.3 GoogLeNet

GoogLeNet is a CNN method that won the object detection challenge for solutions pre-trained on additional data in the ILSVRC 2014 [44]. It also won the classification challenge and came in second on the localization challenge for solutions only trained on provided training data in the ILSVRC 2014 [44]. GoogLeNet has been improved since and the newest version is called Inception v4 [12]. GoogLenet has about 6.8 million parameters, and it uses convolutional, maxpooling, averagepooling, and fully connected layers [23]. In addition, GoogLeNet uses a layer module called an inception module [23]. The inception module runs the same input through four separate pipelines before joining their inputs [23]. Table 2.3 shows the inception module layer architecture

[23]. GoogLeNet uses ReLU activations and a SGD variant called asynchronous SGD [23]. It implements several design patterns such as: momentum of 0.9, step learning annealing decreasing with 4% every eight epoch, a dropout layer with $p = 0.7$, and data augmentation including cropping, flipping, and resizing [23]. Table 2.4 shows the GoogLeNet layer architecture [23]. The notation is similar to table 2.1.

Inception Module			
input			
↓	↓	↓	↓
conv-1	conv-1	conv-1	maxpool-3
↓	↓	↓	↓
↓	conv-3	conv-5	conv-1
↓	↓	↓	↓
filter concatenation			

Table 2.3: Inception Module

GoogLeNet-22	
input	224x224x3
conv-7	112x112x64
maxpool-3	56x56x64
conv-3	56x56x192
maxpool-3	28x28x192
inception	28x28x256
inception	28x28x480
maxpool-3	14x14x480
3 x inception	14x14x512
inception	14x14x528
inception	14x14x832
maxpool-3	7x7x832
inception	7x7x832
inception	7x7x1024
averagepool	1x1x1024
fully connected	1000
softmax	

Table 2.4: GoogLeNet

2.5.4 ResNet

ResNet is a CNN method that won the classification challenge using only provided training data in the ILSVRC 2015 [47]. Together with a method called R-CNN it also won the object detection and localization challenge using only provided training data in the ILSVRC 2015, and the object detection challenge in COCO 2015 [47] [36].

ResNet uses a shortcut connection between layers which enables backpropagation to be calculated more efficiently [14]. This allows the ResNet to be very deep and still very fast [14]. ResNet created five networks using depth of 18, 50, 101, and 152 layers [48]. Very roughly, these networks have 35, 35, 35, 65, and 65 million parameters respectively [49]. ResNet uses convolutional, maxpooling, averagepooling, and fully connected layers [48]. ResNet uses ReLU activations and the SGD optimization algorithm with a batch size of 256 [48]. It implements several design patters such as: momentum of 0.9, step learning annealing starting at 0.1 and reduced by a factor of ten every time the rate of change in error stagnates, weight decay of 0.0001, batch normalization, and data augmentation including cropping, rescaling, and color changes [48]. Their model was trained for 600 000 iterations [48]. Table 2.6 shows ResNet 18, 34, and 50, while table 2.7 shows ResNet 101 and 152 [48] [50]. The notation is similar to table 2.1.

ResNet-18		**ResNet-34**		**ResNet-50**	
input	224x224x3	input	224x224x3	input	224x224x3
conv-7	112x112x64	conv-7	112x112x64	conv-7	112x112x64
maxpool-3	56x56x64	maxpool-3	56x56x64	maxpool-3	56x56x64
4 x conv-3	56x56x64	6 x conv-3	56x56x64	3 x	$\begin{bmatrix} \text{conv-1} & \text{56x56x64} \\ \text{conv-3} & \text{56x56x64} \\ \text{conv-1} & \text{56x56x256} \end{bmatrix}$
4 x conv-3	28x28x384	8 x conv-3	28x28x128	4 x	$\begin{bmatrix} \text{conv-1} & \text{28x28x128} \\ \text{conv-3} & \text{28x28x128} \\ \text{conv-1} & \text{28x28x512} \end{bmatrix}$
4 x conv-3	14x14x256	12 x conv-3	14x14x256	6 x	$\begin{bmatrix} \text{conv-1} & \text{14x14x256} \\ \text{conv-3} & \text{14x14x256} \\ \text{conv-1} & \text{14x14x1024} \end{bmatrix}$
4 x conv-3	7x7x512	6 x conv-3	7x7x512	3 x	$\begin{bmatrix} \text{conv-1} & \text{7x7x512} \\ \text{conv-3} & \text{7x7x512} \\ \text{conv-1} & \text{7x7x2048} \end{bmatrix}$
averagepool	1x1x512	averagepool	1x1x512	averagepool	1x1x2048
fully connected	1000	fully connected	1000	fully connected	1000
softmax		softmax		softmax	

TABLE 2.5: ResNet 18, 34, and 50

2.6 CNN Detection Methods

These are some of the frequently discussed CNN detection methods from my literature review. They are relevant because of their ability identify and locate objects in images.

ResNet-101		**ResNet-152**	
input	224x224x3	input	224x224x3
conv-7	112x112x64	conv-7	112x112x64
maxpool-3	56x56x64	maxpool-3	56x56x64
3 x	$\begin{bmatrix}\text{conv1} & \text{56x56x64}\\ \text{conv3} & \text{56x56x64}\\ \text{conv1} & \text{56x56x256}\end{bmatrix}$	3 x	$\begin{bmatrix}\text{conv1} & \text{56x56x64}\\ \text{conv3} & \text{56x56x64}\\ \text{conv1} & \text{56x56x256}\end{bmatrix}$
4 x	$\begin{bmatrix}\text{conv1} & \text{28x28x128}\\ \text{conv3} & \text{28x28x128}\\ \text{conv1} & \text{28x28x512}\end{bmatrix}$	8 x	$\begin{bmatrix}\text{conv1} & \text{28x28x128}\\ \text{conv3} & \text{28x28x128}\\ \text{conv1} & \text{28x28x512}\end{bmatrix}$
23 x	$\begin{bmatrix}\text{conv1} & \text{14x14x256}\\ \text{conv3} & \text{14x14x256}\\ \text{conv1} & \text{14x14x1024}\end{bmatrix}$	36 x	$\begin{bmatrix}\text{conv1} & \text{14x14x256}\\ \text{conv3} & \text{14x14x256}\\ \text{conv1} & \text{14x14x1024}\end{bmatrix}$
3 x	$\begin{bmatrix}\text{conv1} & \text{7x7x512}\\ \text{conv3} & \text{7x7x512}\\ \text{conv1} & \text{7x7x2048}\end{bmatrix}$	3 x	$\begin{bmatrix}\text{conv1} & \text{7x7x512}\\ \text{conv3} & \text{7x7x512}\\ \text{conv1} & \text{7x7x2048}\end{bmatrix}$
averagepool	1x1x2048	averagepool	1x1x2048
fully connected	1000	fully connected	1000
softmax		softmax	

Table 2.6: ResNet 101 and 152

2.6.1 Faster R-CNN

Faster R-CNN is a CNN method that has been a part of solutions winning many computer vision challenges between 2015 and 2017. Such methods won the object detection with and without additional training data in the ILSVRC 2015 and ILSVRC 2016 [47] [51]. They also won the object detection challenge in COCO 2015 and COCO 2016 [36]. In addition, the third highest ranking solution in the PASCAL VOC 2012 object detection challenge also uses a Faster R-CNN variant [38]. Faster R-CNN uses a modified version of the VGG-16 network [52]. It uses the network to find about 2000 regional proposals which are run through separate sub networks trained for classification and bounding box regression [9] [52] [53]. The region proposals are found using kernels of different sizes (anchors) often shaped as rectangles [52]. Faster R-CNN uses SGD with a batch size of 256, and implements several design patterns such as: momentum of 0.9, step learning annealing starting at 0.001 and dropping to 0.0001 after 60 000 batches, weight decay of 0.0005, transfer learning with a model pre-trained on imagenet, and data augmentation including rescaling [52]. Faster R-CNN also uses non-maximum suppression (NMS) to choose only one of several bounding boxes when there are bounding boxes with a large overlap from the same class [12] [52]. The method has a test run time of 0.2 seconds per image, or 5 frames per second (FPS) [52]. Code is available for Python and Matlab at https://github.com/rbgirshick/py-faster-rcnn, and https://github.com/shaoqingren/faster_rcnn respectively [52].

2.6.2 SSD

The single shot multibox detector (SSD) is a CNN method that uses a modified version of the VGG-16 network [54]. The network does not include fully connected layers, and does instead have a pipeline of convolutional layers each outputting prediction scores at the end of the VGGNet [12] [54]. It uses this network to find a fixed size collection of predictions per image which it runs regression on [54]. The prediction boxes are found using kernels of different sizes (default boxes) often shaped as rectangles [54]. SSD uses SGD with a batch size of 32, and implements several design patterns such as: momentum of 0.9, step learning annealing, weight decay of 0.0005, transfer learning with a model pre-trained on imagenet, NMS, and data augmentation including cropping, rescaling, flipping, and distortions [54]. The method has different step learning annealing parameters and training time depending on the task, and uses batch normalization in some cases [54]. SSD also uses hard negative mining, which removes some of the negative training examples / prediction boxes during training [54]. This is done to adjust the ratio between positive and negative training examples [54]. SSD has two versions, SSD300 uses input images with dimensions 300x300 pixels, and SSD512 uses input images with dimensions 512x512 pixels [54]. SSD300 reaches 59 FPS while SSD512 reaches 22 FPS [54]. Code is available at https://github.com/weiliu89/caffe/tree/ssd [54].

2.6.3 R-FCN

Region-based fully connected network (R-FCN) is a CNN method used in the solutions that has first and third place in the object detection challenge using additional training data in PASCAL VOC 2012 [55]. It uses a modified version of ResNet-101, without the last averagepool and fully connected layer [55]. It uses the network to find regional proposals which it runs classification and bounding box regression on [55] [53]. R-FCN is fully convolutional and shares all its layers with the entire image instead of using sub networks [55]. The method uses a batch size of 8, and implements several design patterns such as: momentum of 0.9, step learning annealing starting at 0.001 and dropping to 0.0001 after 20 000 batches, transfer learning with a model pre-trained on imagenet, and

NMS with a threshold of 0.3 IoU [55]. R-FCN has a test run time of 170 milliseconds per image, or 5.9 FPS [55]. Code is available at `https://github.com/daijifeng001/r-fcn` [55].

2.6.4 YOLOv2

You only look once version two (YOLOv2) is a CNN method that uses its own network architecture called Darknet-19 [56]. Darknet has about 21 million parameters and uses convolutional and maxpool layers [56]. It uses the network to find a fixes size collection of predictions per image which it runs regression on [56]. The prediction boxes are found using kernels of different sizes (anchors) [56]. In YOLOv2, the anchor shapes are based on a k-means clustering algorithm run on the ground truth boxes from the training set [56]. The method implements several design patterns such as: momentum of 0.9, step learning annealing starting at 0.001 and reduced by a factor of 10 after 60 and 90 epochs, weight decay of 0.0005, batch normalization, and data augmentation including cropping and color shifting [56]. YOLOv2 also uses multi-scale training in which it rescales images during training to make the model more robust to different resolutions [56]. The authors train the model for 160 epochs [56]. YOLOv2 has several versions using different input image dimensions, which include 288x288, 352x352, 416x416, 480x480, and 544x544 pixels [56]. These models reaches 91, 81, 67, 59, and 40 FPS, respectively [56]. Table 2.7 shows the Darknet-19 224 layer architecture [56]. When this model is used for classification or regression, a couple of task specific layers are appended to the end of the architecture [56]. The notation is similar to table 2.1. Code, models, and weights are available at `http://pjreddie.com/yolo9000/` [56].

2.7 Software Frameworks and Libraries

These are some of the frequently discussed software frameworks and libraries from my literature review. They are relevant because of their ability to implement CNN methods.

Darknet-19 224	
input	224x224x3
conv-3	224x224x32
maxpool-2	112x112x32
conv-3	112x112x64
maxpool-2	56x56x64
conv-3	56x56x128
conv-1	56x56x64
conv-3	56x56x128
maxpool-2	28x28x128
conv-3	28x28x256
conv-1	28x28x128
conv-3	28x28x256
maxpool-2	14x14x256
conv-3	14x14x512
conv-1	14x14x256
conv-3	14x14x512
conv-1	14x14x256
conv-3	14x14x512
maxpool-2	7x7x512
conv-3	7x7x1024
conv-1	7x7x512
conv-3	7x7x1024
conv-1	7x7x512
conv-3	7x7x1024

TABLE 2.7: Darknet

2.7.1 Theano

Theano is a open-source library written in Python and C, created for use with Python [57] [58]. The library is used for efficient computing with multi-dimensional arrays [58]. Theano can run on the central processing unit (CPU) but can also produce instructions for the GPU using Nvidia CUDA [58]. Theano is released under the Berkeley software distribution (BSD) licence, and can be distributed freely as long as the licence, copyright notices, and disclaimer are provided together with the source code and binaries [57] [58].

2.7.2 DL4J

Deep dearning for Java (DL4J) is a open-source library written in Java, JavaScript, and Scala, created for use with Python, Java, and Scala [59]. The library is used for creating DL programs and running them on distributed systems [59]. It can also import neural network models from many large frameworks via Keras [59]. DL4J can run on the

CPU but can also produce instructions for the GPU using CUDA [59]. DL4J is released under the Apache 2.0 licence, and can be distributed freely as long any modifications are marked, and any derivated work contains the proper copyrights, patents, trademarks, and attributions [59] [60].

2.7.3 Caffe2

Caffe2 is a open-source library written in C++, Python, and C, created for use with C++, Python, and Matlab [61]. The library is used for creating DL programs and offers pre-trained models through Caffe Model Zoo [61]. Caffe2 can run on the CPU but can also produce instructions for the GPU using CUDA [61] [62]. Caffe2 is released under the Apache 2.0 licence, and can be distributed freely as long any modifications are marked, and any derivated work contains the proper copyrights, patents, trademarks, and attributions [61] [60].

2.7.4 TensorFlow

TensorFlow is a open-source framework written in C++, Python, Go, and Java, created for use with C++, Python, Go, and Java [63]. The framework is used for creating ML programs and offers pre-trained models through the TensorFlow Model Zoo [63]. TensorFlow can run on the CPU but can also produce instructions for the GPU using CUDA [63]. TensorFlow is released under the Apache 2.0 licence, and can be distributed freely as long any modifications are marked, and any derivated work contains the proper copyrights, patents, trademarks, and attributions [63] [60].

2.7.5 Darknet

Darknet is a open-source framework written in C, created for use with C [64]. The framework is used for creating and running neural networks, especially using the YOLOv2 method, and offers pre-trained models [64]. Darknet can run on the CPU but can also produce instructions for the GPU using CUDA [64]. Darknet is released under many licences, but it seems that they all state that the software is completely free for any use [64].

2.7.6 Torch

Torch is a open-source framework written in C, Lua, and C++, created for use with LuaJIT [65]. The framework is used for creating programs that uses ML algorithms [65]. Torch can run on the CPU but can also produce instructions for the GPU using CUDA [65]. Theano is released under the BSD licence, and can be distributed freely as long as the licence, copyright notices, and disclaimer are provided together with the source code and binaries [65].

2.7.7 MXNet

Apache MXNet is a open-source library written in Python, C++, Perl, and Scala, created for use with Python, Scala, R, Julia, C++, Perl, Go, and JavaScript [66]. The library is used for creating DL programs and offers pre-trained models through Gluon Model Zoo [66]. MXNet can run on the CPU but can also produce instructions for the GPU using CUDA [66]. MXNet is released under the Apache 2.0 licence, and can be distributed freely as long any modifications are marked, and any derivated work contains the proper copyrights, patents, trademarks, and attributions [66] [60].

2.7.8 Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit is a open-source toolkit written in C++, Python, and C#, created for use with C++, Python, and C# [67]. The toolkit is used for creating programs that can train DL algorithms and it offers pre-trained models [67]. Microsoft Cognitive Toolkit can run on the CPU but can also produce instructions for the GPU using CUDA [67]. Microsoft Cognitive Toolkit is released under the Massachusetts Institute of Technology (MIT) licence, and can be distributed freely as long as any derivated work contains the proper copyright notice, permission notice, and disclaimer [67].

2.7.9 Keras

Keras is a high-level open-source library written in Python, created for use with Python [68]. The library can run on top of TensorFlow, Microsoft Cognitive Toolkit, or Theano, and is used for creating DL programs [68]. Keras can run on the CPU but can also produce instructions for the GPU using CUDA [68]. Keras is released under the MIT licence, and can be distributed freely as long as any derivated work contains the proper copyright notice, permission notice, and disclaimer [68].

Chapter 3

Method

This chapter will introduce the reader to the methods used to to answer the research questions. It will describe the process used to find relevant literature and the process of producing measurable results.

3.1 Literature Review

To be thorough and non biased towards the current state of the art research on DL, I decided to design and perform a structured literature review. I started from NTNU's homepage, and through SEKOM, I found NTNU Viko. NTNU Viko provides information about how and where to search for source material. The strength of structured literature reviews is that it provides access to a diverse collection of research, however, it is extremely time consuming.

3.1.1 Keywords

I chose to only use a few central fields of research as keywords in my literature review. I did this because the amount of research on DL is very large and fairly complex. The keywords I used were:

1. Deep Learning;
2. Convolutional Neural Networks;

3. Object Detection.

However, these keywords by themselves would not necessarily provide any useful material. CNN's are used for other tasks than object detection, and object detection is not always performed using DL methods. Therefore I chose to use the following combination of keywords:

1. "Deep Learning" AND "Object Detection";
2. "Convolutional Neural Networks";
3. "Convolutional Neural Networks" AND "Object Detection".

3.1.2 Libraries

Using NTNU Viko, I found five online research collections. These web pages typically provided sorting search results on date, usage, and relevance. I found that sorting on usage usually provided the most interesting results, however, I chose to use results from all three sorts in my literature review so I would not miss very new or very relevant research. Using this method I would end up with about five collections times three search sorts times three search keywords, which is 45 lists of search results. This lead me to choose the fairly low number of ten results per search to investigate. I used the following research collections:

1. NTNU Open;
2. Web of Science;
3. Scopus;
4. NTNU Oria *University Library*;
5. NTNU Oria *Norwegian Academic Library*;
6. Google Scholar.

3.1.3 Filtering Process

Using this method would leave me with about 450 sources of material to investigate, which is a lot to read. Therefore I chose to use a filtering process to divide useful from useless research. I would begin by finding interesting titles and reading their abstracts. In the abstracts I would look for interesting topics, such as precise definitions, information about the keywords, or implementations and discussion of results. If an interesting abstract was found, I would download the material, read it, and make a note of why the material was of interest. This way I would keep the content locally, and know for what purpose I had stored it.

3.2 DL Method Comparison

Reviewing the literature on DL, there seemed to be a consensus on the fact that CNN methods are the most suitable DL method for working with images. The fact that the highest performing computer vision methods in several large competitions are all CNN methods, supports this idea. These reasons lead me to choose to work with CNN methods.

The problem of finding and identifying salmon in images, should not be posed as a classification problem because of the necessity to describe the location of the objects. Neither should the it be posed as a localization problem, as localization methods only predicts the location of a single object. The problem could be posed as a semantic segmentation problem, but this would most certainly increase the complexity of any solution. The problem should instead be posed as a object detection problem, since methods solving these problems can predict both multiple classes and locations. The bounding boxes produced by a object detection system could also provide information such as the length and height of the salmons.

Because the salmon detection problem should not be posed as a classification, localization, or semantic segmentation problem, I will regard the CNN methods focusing on these types of problems as unsuitable. These CNN methods include: AlexNet, VGGNet, GoogleNet, and ResNet. The other CNN methods from Chapter 2 should all be suitable for the task of detecting salmon in images as they have previously been used to solve object detection problems quite successfully. These methods include: Faster R-CNN, SSD, R-FCN, and YOLOv2.

3.2.1 CNN Method Accuracies

The papers describing Faster R-CNN, SSD, R-FCN, and YOLOv2 do all include performance measures using their respective method. However, they are not all tested with the same sort of configuration.

The papers describing Faster R-CNN, SSD, R-FCN, and YOLOv2 all present results on the PASCAL VOC 2007, PASCAL VOC 2012, and COCO 2015 challenges. They do however experiment with which training data they use for each task. While R-CNN, SSD, and R-FCN all show results for the PASCAL VOC challenges using training and validation (trainval) data for all three challenges, YOLOv2 only shows results on the PASCAL VOC challenges using PASCAL VOC trainval data. This lead me to compare the four methods using results from PASCAL VOC 2007 and 2012 using only PASCAL VOC trainval data, and from COCO 2015 using only COCO trainval data. Table 4.1 shows the accuracy in MAP for the compared methods in these challenges using the same training data. According to this comparison, R-FCN is the most accurate method.

	PASCAL VOC 2007	PASCAL VOC 2012	COCO 2015	Mean
Faster R-CNN	73.2%	70.4%	42.7%	62.1%
SSD512	79.8%	**78.5%**	48.5%	68.9%
R-FCN	**80.5%**	77.6%	**53.2%**	**70.4%**
YOLOv2 544	78.6%	73.4%	44.0%	65.3%

Table 3.1: Method Accuracy Comparison

3.2.2 CNN Method Speeds

Many of the papers describing the suitable methods contain information about how fast the network runs at test time, and some of them contain information about how long their model needs to be trained. However, this information is not given using the same metrics or harware through the source material.

The methods are trained and tested using different hardware. Faster R-CNN and R-FCN uses Nvidia K40 GPUs, while SSD and YOLOv2 uses Nvidia Titan X GPUs. These GPUs are not equally fast, but comparing the speeds of these methods with respect to the speed of the hardware in use is beyond the scope of this thesis. However, because of the parallel computing capabilities of these processors, they will greatly benefit work such as training and testing CNNs. Any CPU implementation will lack this parallel computing benefit and will thus run much slower.

The papers describing the suitable methods do provide information about their speed at test time, but only SSD and YOLOv2 specifies the batch size used during testing. I will compare the methods using a batch size of one at test time, and and presume that this is the test time batch size used by Faster R-CNN and R-FCN as well. The test time speeds of these methods are the following:

- Faster R-CNN: 5 FPS
- SSD512: 19 FPS
- R-FCN: 5.9 FPS
- YOLOv2 544: **40 FPS**

Comparing the time it takes to train a model is a tough task. This is mostly due to the amount of factors that affect this value. These factors include: amount of adjustable model parameters, performance of pre-trained model or luck in initialization of weights, amount of training data, the presence of shortcut connection such as in ResNet, and the efficiency of the hardware. Some of the papers include interesting information about the training speed. The R-FCN paper writes that the method uses 0.45 secnods to process an image during training, and the YOLOv2 paper writes that their model is usually trained for 160 epochs. The other papers do however not provide similar information. The suitable methods all use fairly well known network architectures, which leads me to believe that pre-trained models exists for all of them. Faster R-CNN and SSD uses variants of the VGGNet-16 network, R-FCN uses a variant of the ResNet-101 network, and YOLOv2 uses the Darknet-19 network. These architectures contain roughly the following amounts of parameters:

- Faster R-CNN: **15 million**
- SSD512: **15 million**

- R-FCN: 63 million
- YOLOv2 544: 21 million

3.2.3 CNN Method Choice

Ketil Bø informed me that the Dynamic Imager module was expected to run as a 32 bit dynamic link library (DLL) on a Microsoft Windows computer, not necessarily having a GPU compatible for CUDA. Either training or testing a CNN on a CPU is much slower than on a GPU. This lead me to prioritize fast methods. Because the training time of the methods were hard to estimate, I decided to use the test time comparison instead. In this comparison the YOLOv2 544 method is clearly faster than the others, therefore I chose to use YOLOv2 in my Dynamic Imager module.

3.3 Software Framework and Library Comparison

The software frameworks and libraries reviewed in Chapter 2 all give the impression that they contain functionality for implementataion of DL methods such as CNN. However, not all them are equally suitable to use for implementing a Dynamic Imager module. Dynamic Imager is written in C++, which makes frameworks and libraries written in or for C++ or C more suitable for the module implementation. These frameworks and libraries include: Caffe2, TensorFlow, Darknet, MXNet, and Microsoft Cognitive Toolkit. The Darknet framework does however stand out as it is created by the authors of YOLOv2, for training and testing purposes on the YOLOv2 architecture, containing easily attainable network configurations and pre-trained weights. This lead the me to choose the Darknet framework for implementation of the Dynamic Imager module.

3.4 Salmon Dataset

Trollhetta provided me with a image dataset of salmons from fish farms. The dataset consisted of 3253 images in three different formats of size and image quality. These images were of the sizes 801x532, 1626x1236, and 704x576 pixels. After inspecting the images, I found that the image sets were taken from video streams with different time intervals in between images. The image sets also included several duplicates. Because I knew I had to annotate the dataset, I decided to reduce the amount of images by removing images visibly in short intervals from each other and duplicate images. From the image sets with a low time interval in between images, I decided to use one in every 20 images. After reducing the number of images, I was left with a dataset with 632 images of a higher visual variety than the original dataset.

3.4.1 Dataset Annotation

My supervisor from Trollhetta, Ketil Bø, informed me that the detections from a system such as the one I was making, could be used to estimate biomass in fish farms by using the length of the fish. Therefore I decided to only annotate fish where the length was visually obvious, usually by being able to identify both the head and tail fin of the fish. Fish far from the camera, swimming in a vertical direction, or towards or against the camera were not annotated because it would not be possible to determine their lengths. By only annotating clearly identifiable fish, I was also more confident that a DL system would be able to identify them as well.

I wanted to annotate the the dataset on my Apple Mac computer, and found the Mac program RectLabel which allowed me to draw bounding boxes around fish and save labels and coordinates to JavaScript object notation (JSON) files. The program stored the box coordinates as x position, y position, width, and height.

3.5 Virtual Machine Setup

I decided to find an efficient operating environment in order to run some tests and produce some preliminary results using the Darknet framework with the salmon dataset. As I did not already have access to any such operating environments for prolonged periods of time, I decided to create a virtual machine (VM) in the cloud, with a decent amount of hardware associated with it. There are several services providing cloud VMs, but I did not think it would matter much which provider I choosed. They do however cost money to rent, so I based my choice on the Google Cloud Platform (glcoud) on the fact that they gave me 2322 NOK in promotion credit. I created a gcloud VM instance in their gcloud europe-west1-b region, because it is close to Norway, and because it provides access to GPUs. I chose Ubuntu 16.04 as the operating system (OS) because Linux computers provide good terminal functionality, and Ubuntu 16.04 is a stable Linux version. Then I applied to Google for the use of a Nvidia Telsa K80 GPU for the use of CNN training and testing, which was approved. I wanted to use the VM from my Mac latop, and found that gcloud provides a command line interface (CLI) usable from Mac terminals. I downloaded the gcloud software development kit (SDK) and configured the VM for secure shell (SSH) access. The SSH access to the server allowed to me to use a terminal on the VM from my laptop.

Using Git, I cloned the official Darknet Github project from `https://github.com/pjreddie/darknet` (pjreddie/darknet) to my VM. The project is dependant on having CUDA installed and greatly benefits from having access to a GPU. Optionally, you can compile it with the Nvidia CUDA deep neural network library (cuDNN) to accelerate training, or the open source computer vision library (OpenCV) for increasing the number of image formats supported. I installed drivers for the GPU, then installed CUDA and cuDNN, before I compiled the Darknet project on the VM.

3.5.1 YOLOv2 Configuration

To be able to able to train a model in Darknet with your own training data, you have to provide the framework with some configuration files. This includes a file describing the location of training and test data, a file describing the structure of the network, and a file containing model weights. In addition, the location of training and test data must reference to a single file containing a list of images, and these images must be placed in directories with an annotation file for each image. Appendix A and Appendix B contains examples for such configuration files. Appendix A contains examples of various files including a data configuration file. Appendix B contains a network configuration example, namely yolo-voc.2.0.cfg from pjreddie/darknet. This network configuration can not be found in the current pjreddie/darknet Github version, you may find it in git commit 1e729804f61c8627eb257fba8b83f74e04945db7.

I wrote a Python script that took the annotations from RectLabel, rearranged them to fit the Darknet annotation style, and saved them in the same directory as the images. Then I had to split the dataset into subsets for training, validation, and testing. I chose to use a 80% / 10% / 10% dataset split because I did not have a very large dataset, and wanted to make sure I had enough data for training. I wrote another Python script saving the paths for every four out of five images to a text file for training, every tenth file starting with the fifth image to a file for validation, and every tenth file starting with the tenth image to a file for testing. The pjreddie/darknet repository contains many network configuration examples, and I chose to start testing using a network configuration created for use the the PASCAL VOC dataset called yolo-voc.2.0.cfg. This configuration contained the Darknet19 architecture with output layers for detection, it used input images of 416x416 pixels, but was configured for 20 classes instead of 1. By changing the filter of the final layer, and the *classes* value in the configuration, I was able to configure the network for a single class. Finally, I downloaded weights pre-trained on ImageNet from the YOLO website at `https://pjreddie.com/darknet/yolo/`, namely darknet19_448.conv.23. This allowed me to train and test a model with the Darknet19 architecture for detection of salmon on the VM.

3.6 Dynamic Imager Module Implementation

From Trollhetta, I was given a copy of 32 bit Dynamic Imager version 3.0.3.4 for Windows, as well as a development tutorial, documentation, and programming SDK for use during the integration of the DL module. Dynamic Imager is well suited for module integration, and contains functionality to import and use DLLs. These DLLs must contain the Dynamic Imager SDK, and contain two DLL entry point functions for use when importing and running modules. These modules can be written in either C++ or C.

3.6.1 Development Environment

Since Dynamic Imager is a Windows program, I chose to work on a Windows computer during development. The pjreddie/darknet project is however only tested on Linux and Mac computers, so to ease the development of the DL module, I decided to search for forks of Darknet more suited for use on Windows. Through this search I found the very well documented Github project AlexeyAB/darknet from `https://github.com/AlexeyAB/darknet`. It is a Github fork of pjreddie/darknet repository, but it is created for use with Visual Studio (VS) on Windows. Additionally, it contains several VS project configurations, including configurations for use with CPU only, and for compiling to DLL.

I decided to use AlexeyAB/darknet in my DL module, and cloned the repository to my computer. I decided not to keep the repository's commit history, so I deleted the original .git file and re initiated git. Then I created a private repository on Github, and set this repository as the local repository's remote. This would allow me to push my changes to the AlexeyAB/darknet project to my own private repository, making it simpler to share the project with Trollhetta. Appendix C contains my Github repository's readme. I decided to use AlexeyAB/darknet's VS project configuration called yolo_cpp_dll_no_gpu.sln as basis for the DL module. This VS project configuration was created for use with CPU only, and for compiling to DLL. It was originally created for compiling 64 bit DLLs, but by compiling it without OpenCV and using an older version of CUDA, I was able to compile it as a 32 bit DLL. CUDA seems to have stopped development on 32 bit libraries in version 7.0, so I had to use version 6.5. The project contained a set of DLL entry points providing access to the projects functionality, such as training or testing models. However, Dynamic Imager only uses two DLL entry points, one for importing and one for running. Therefore I removed the original entry points entirely and created a file just called main.cpp that would contain the new entry points. Then I imported the Dynamic Imager SDK into the project. Using the header files and libraries contained in this SDK, I was able to use the Dynamic Imager functionality in my main file, allowing me to create modules usable by the program.

Studying the source code in the AlexeyAB/darknet repository, I was able to identify a couple of functions that I would like to use in the module. I knew that functionality for training and testing a module was necessary for the module to be useful, but I also wanted the module to somehow be able to indicate when a training process was complete. This can be done using the error of the loss function for the training and validation set. However, Darknet does not calculate the loss of the validation set during training, and I was not able to efficiently append it to the functionality. Fortunately, developers of AlexeyAB/darknet had added the function validate_detector_map in the detector.c file. This function calculates the MAP for a model on a dataset. Validate_detector_map could have been used to check the MAPs for the training and test dataset thoughout the training process, providing an alternative evaluation metric to the loss function error. Calculating the MAP for a model does however take a fair amount of time because it works by testing every image in the dataset at several levels of confidence. Because the module was anticipated to run slowly as a result of it running on a CPU, I did not see it fit to use MAP checks for the task of indicating when the training process was complete. I decided to use the functions train_detector, test_detector, and validate_detector_map all found in detector.c, for use during training, testing, and validation of a model, respectively. Declarations of these C functions where added to the main file, which allowed me to use them through the Dynamic Imager entry points.

3.6.2 User Story

The main use story for the module would be to first train a model for the amount of time you had available, having the module save intermediate weights at a chosen interval during training. Then validating a number of weights from the end of the training on both the training and validation set to determine which of them had the highest performance. Before finally using the highest performing model to test images, which would produce both a visual and textual representation of the results. The intermediately saved weights would be usable for further training of the model.

3.6.3 Dynamic Imager

Dynamic Imager is a image processing tool for stepwise modification of images using a network layout. This generally means that every Dynamic Imager module has input and output ports for connecting them to other modules. As an example one could import an image using a *Import Image* module, send it on to some sort of image modification module, before sending it on to a *Save Image* module saving the image in a chosen format. These three connected modules would then represent a network in the program. The original idea for the DL module was to use a network layout for both training and testing. During training one could use a *Import Images* module to send several training images on to the training module, which would use these images for training and save output weights to file. During testing, one could use a similar *Import Images* module to send images through whatever modification modules before entering the testing module. This module could then either present the images containing predictions, pass them on to other modules, or save them to file. I did however have some issues with image formats, and the way they were stored as data in Dynamic Imager SDK and Darknet.

I was given access to a *Import Image* module, and investigated how it represented an image, then compared this to the way Darknet would represent the same image, and they were not the same. This issue was not resolved, and I decided to make the module work without any connections. This would make the module less user friendly, but the functionality of modifying an image before use in the DL module was still possible. By running a image modification network ending with saving the image to file would mean that you could later use this modified image just as any other with the DL module.

3.6.4 Darknet Modification

By including the Darknet source files in the DL module project, and then placing declarations for Darknet functions in the main file, I was able to call the functions train_detector, test_detector, and validate_detector_map from main. They were however not functioning exactly the way I needed them to, so I modified them. When modifying the Darknet code base I changed some larger functions that I knew only I would be calling from main, but I did also need to modify some smaller more frequently used functions. Instead of changing these functions, I decided to create and use slight copies of them. This way I would not affect other code using the original functions. All modified and added functions were commented as such in the source code. The modifications provided the following features:

- Being able to set the save location for intermediate weights during training;
- Being able to decide the frequency of weight saving during training;
- Automatically setting the filter size of the final output layer to a valid value based on the *classes* parameter from the data configuration.
- Automatically setting the *classes* parameter in the network to the same value as the *classes* parameter in the data configuration;
- Being able to test several images as a batch job;
- Being able to choose whether to validate a model using the training or validation dataset.

3.6.5 User Input

I did not want to hardcode any of the Darknet functionality into the module, and decided to use the same amount of user input in the DL module as in the original Darknet terminal program described in detector.c. This choice was based on the fact that I believed I could make even a highly configurable module user friendly. The user input would include data configuration, network configuration, and weight file. By using the Dynamic Imager SDK I was able to create input fields on the modules for training, testing, and validation of models. To train a model using the DL module you would need to input the following:

- A data configuration file containing: the number of classes, the path to the training data, and the path to a names file containing the names of each detectable class;
- A network configuration file containing the structure of the network and hyperparameters;
- A weights file containing trained or untrained weights for the model;
- A chosen frequency for saving intermediate weights;
- A directory to save the output files.

This would produce trained weight files with batch interval equal to the chosen frequency, to the chosen output directory. To test a model using the DL module you would need to input the following:

- An images file containing paths to the images to be tested;
- A data configuration file containing: the number of classes, the path to a names file containing the names of each detectable class, and the path to a labels folder containing a font;
- A network configuration file containing the structure of the network and hyperparameters;
- A weights file containing trained weights for the model;
- A chosen confidence level;

- A directory to save the output files.

This would produce both visually labeled images and annotation files in YOLOv2 format to the chosen output directory. To validate a model using the DL module you would need to input the following:

- A data configuration file containing: the number of classes, the path to the training data, the path to the validation data, and the path to a names file containing the names of each detectable class;
- A network configuration file containing the structure of the network and hyperparameters;
- A weights file containing trained weights for the model;
- A chosen confidence level;

This would produce a Dynamic Imager message describing the MAP of the model, as well as the precision, recall, true positives, false positives, false negatives, and average IoU of the chosen confidence level.

3.7 Solutions Testing

Integration testing of the earlier versions of the module revealed some issues with the implementation. This testing was performed running Dynamic Imager as the VS debugging command, and using VS interfaces such as callstack, breakpoints, and immediate window. Darknet is written in C where there is no automatic garbage collection, this can cause memory leaks. The Darknet terminal program is also created in a way such that it terminates after each action, such as testing an image or validating a model. This makes this program largely unaffected by possible memory leaks during execution. In contrast, the DL module would need to be usable several times before termination of Dynamic Imager. By using the performance profiler in VS, I was able to detect several memory leaks in the Darknet framework. To fix this issue I modified some Darknet functions to release their allocated memory upon completion. During testing of resource use, I discovered that the CPU and memory use of the DL module was much lower than the Darknet terminal program. This would result in the DL module needing more time to complete tasks. I attempted to provide the module with more resources, but this issue was not resolved. Because of the limitation in memory the module was able to use, I had to reduce the training batch size substantially. By reducing the batch value in the network configuration from 64 to 10, the module was able to run without issues. During testing of the module's usability, I discovered the need for user feedback, as well as error handling and error messaging. Attempting to provide a solid error handling using try catch functionality was not successful. C does not provide try catch functionality, and I was not able to wrap the C functionality in C++ try catch statements. The best solution available was then to preemptively check if the input parameters existed, and that the program had access to them. This solution provides the user with specific information about errors in the input, such as that the path to training data is unavailable. This solution would however fail if the file exists but has the wrong format. The user feedback provides the user with information about errors in the input, feedback if the input check is successful, and feedback when a job is complete.

3.7.1 Accuracy Testing

To evaluate the detection accuracy of YOLOv2 on the salmon dataset, I trained a model using my modified yolo-voc.2.0.cfg network configuration both on the VM and using the DL module. As the programs did not provide the loss function error of the validation set during training, I decided to use MAP score to identify the highest performing model. The intermediate weights produced during training would be used after training to graph the development of performance on both the training and validation set. Then I would pick the best weights and calculate their MAP on the test set. The DL module was however very slow, so I decided to investigated how similar it operated in comparison with the AlexeyAB/darknet terminal program. I discovered that the train_detector functionality contained some randomization, thus it was not possible to determine if the two programs executed this job identical. The test_detector and validate_detector_map functionality did however not contain any randomization, and compiling the AlexeyAB/-darknet terminal project without OpenCV proved to provide the exact same results as the DL module. This lead me to believe that the test_detector functionality was also similar in the two programs, and I decided to do the last half of the DL module training in the terminal program using GPU instead of in Dynamic Imager.

3.7.2 Configuration Testing

After producing valid models using the rather standard yolo-voc.2.0.cfg network configuration, I decided to search for a more optimal configuration for the salmon dataset in order better evaluate the accuracy of YOLOv2 on the task. Possible ways to reconfigure the network and hyperparameters includes: changing the output layers of the network, changing the learning rate and learning rate annealing, changing the bounding box anchors, and toggling multi-scale training. The preddie/darknet and AlexeyAB/-darknet repositories contains more recent network configurations than yolo-voc.2.0.cfg, and many of these configurations describing the Darknet19 architectures uses a different structure of output layers than yolo-voc.2.0.cfg. These network configurations include yolov2-voc.cfg from preddie/darknet, which uses one more convolutional layer in the output layers than yolo-voc.2.0.cfg. By closer investigation of the YOLOv2 research paper, it seems that the authors are actually describing the network structure found in yolov2-voc.cfg rather than the one found in yolo-voc.2.0.cfg. The YOLOv2 paper states that a learning rate of 0,001 and a step learning rate annealing with a factor of ten after 60 and 90 epochs was used. The yolo-voc.2.0.cfg configuration does instead use a leaning rate of 0,0001, which it increases to 0,001 after 100 iterations. After this point it seems that it reduced the learning rate by a factor of ten for the equivalent of 60 and 90 epochs on the PASCAL VOC training set. Because the salmon dataset contains 632 images, 60 and 90 epochs of the salmon dataset would amount to processing 37920 and 56880 images, respectively. This would be achieved after 593 and 889, or 3792 and 5688 iterations, depending on whether a batch size of 64 of 10 is used The anchors in the mentioned configurations are undoubtedly created for the PASCAL VOC training set. According to the YOLOv2 research paper, the authors created these anchors using a five cluster k-means with IoU as the distance metric. The AlexeyAB/darknet repository contains a Python script for generating new anchors. In the search for a good configuration, I decided to test several combination of hyperparameters. I prioritized a configuration as similar as possible to what is described in the YOLOv2 research paper. This meant starting with a modified version of yolov2-voc.cfg, containing modified learning rate, learning rate annealing, anchors generated from the salmon dataset, and using multi-scale training.

3.7.3 Speed Testing

To evaluate the speed of Darknet YOLOv2 on the salmon dataset, I decided to investigate how much time the different Darknet solutions used for a set of tasks. I also wanted to test how much speed was gained by using better or more suitable hardware. The computers I had available for testing was the Linux VM, a Windows desktop computer, and a Mac laptop. On the VM and the Mac, I was able to compile and run the pjreddie/darknet terminal program, while on the Windows computer, I was able to compile and run the AlexeyAB/darknet terminal program for CPU and GPU, as well as the DL module in Dynamic Imager. As I had been most concerned with the ability to train, test, and validate models during the development of the DL module, I decided to test the speeds of these functions. I split the test cases into: loading network and weights, testing model on one image, validate model on four images, and training for one iteration with a batch size of ten. All tests would use the same configuration, and the same images from the image set with dimensions 1626x1236 pixels. The tests on the pjreddie/darknet and AlexeyAB/darknet terminal programs were done by modifying the source code to print the time difference between certain stages of execution, while the tests on the DL module was done by using breakpoints in VS.

3.7.4 Usability Testing

To evaluate the usability of the DL module in Dynamic Imager, I decided to plan a user test. The DL module contains functionality for training, testing, and validating models, therefore I created a user test simply containing these three tasks. The user would be given access to configuration files and the Dynamic Imager program during the test. Before the test I would do an introduction of the module, configuration files, the tasks. After this the user would perform the tasks with as little communication with me as possible. At the end of the last task, I would ask the participant to fill in a system usability scale (SUS) form. The SUS form is a standardized usability test, and produces a usability score between 0 and 100. Typically, a score below 51 is bad, a score in between 51 and 80 is OK, while a score of 80 or above is great. As the people at Trollhetta were the only people I knew with access to Dynamic Imager, I did the user testing with them. Unfortunately, only one person was able to test the module. The test was scheduled to last for about one hour.

Chapter 4

Results

This chapter will introduce the reader to the results produced to be able to answer the research questions. It will also aim to analyze these results.

4.1 Salmon Dataset

Figure 4.1 shows an example of the short interval between images before reducing the dataset. Figure 4.2 shows how this interval has changed after reducing the dataset. These are images from image set 3, with dimensions 704x576 pixels. The job of annotating the 632 image set took approximately 2 minutes per image, which is just over 21 hours. There are 5666 annotations, which means there are about 9 annotated salmon per image. Figure 4.3 shows examples of annotated images. These are the same images as in Figure 4.2.

4.2 Virtual Machine Training Results

Training a model using a modified version yolo-voc.2.0.cfg network configuration and the salmon dataset for 1600 iterations with a batch size of 64 took approximately 5.3 hours on the VM. Figure 4.4 shows how the MAP developed through training. The weights saved at iteration 900 were the highest performing, and the following iterations show signs of overfitting. This model scored a MAP of 82.4% on the training set, 80.9% on the validation set, and **79.5%** on the test set. These are fairly good results. Figure

4.5, 4.6, 4.7, 4.8, 4.9, and 4.10 contains examples of annotations compared to good and bad results on the three different image sets predicted with this model. All of these images are from the image test set. In many cases, the erroneous predictions made by the model does contain salmons, but salmons that have not been annotated. This is a sign of generalization by the model, which is good. It is however hard to determine the actual length of some of these predicted fish, due to the fish being partly out of frame or behind other fish. The images presented as good predictions are very similar to the annotations, and additional detections in these images, which are not contained in the annotation, does typically contain either a whole salmon or large portions of one. The images presented as bad predictions are not very similar to the annotations, and several of these contain detections which only contains small portions of one or more salmon.

4.3 Dynamic Imager Module Results

The DL module was completed with functionality for training, testing, and validation of models. It did however not appear to have a high usability as the user have to plug in a high amount of input. The return for the poor usability is a high configurability as the user can pretty much use any network described using the Darknet configuration notation. Figure 4.11 shows the Dynamic Imager graphical user interface (GUI). In this image, the module *Test Model* is currently open and the user input fields are visible in the bottom left corner. Training a model using a modified version of the yolo-voc.2.0.cfg network configuration and the salmon dataset for 700 iterations with a batch size of 10 took approximately 58.3 hours in the DL module. Figure 4.12 shows how the MAP developed through training with the DL module. This training was continued using the AlexeyAB/darknet terminal program on Windows using GPU. Training the exact same model from iteration 700 to 1600 took approximately 18 minutes using this program. This is a huge difference! Figure 4.13 shows how the MAP developed through training with the AlexeyAB/darknet terminal program. I could unfortunately not determine why the MAP values crashed during the last two hundred iterations. The weights saved at iteration 1300 were the highest performing, scoring a MAP of 83.4% on the training set, 80% on the validation set, and **80.2%** on the test set. This is better than the results from the VM, but only by 0.7%. It is interesting that the DL module needed to process a lot fewer images than the VM during training to reach good MAP values.

It is likely that this is partly due to the fact that the learning rate is very low in yolo-voc.2.0.cfg until iteration 100. While the DL module reaches iteration 100 after processing 1000 images, the VM reaches iteration 100 after 6400 images. I am however not certain that this is the only reason for the difference. Figure 4.14, 4.15, 4.16, 4.17, 4.18, and 4.19 contains examples of annotations compared to good and bad results on the three different image sets predicted with this model. All of these images are from the image test set. The images presented as good predictions are very similar to the annotations, and additional detections in these images, which are not contained in the annotation, does typically contain either a whole salmon or large portions of one. The images presented as bad predictions are not very similar to the annotations, and several of these contain detections which only contains small portions of one or more salmon.

4.4 Configuration Testing Results

To find a well suited network configuration for the salmon dataset, I did a fairly extensive search of the hyperparameter space, with over a dozen models trained using the VM. Models trained with the yolov2-voc.cfg did very poorly. Some of the models suddenly crashed such as in Figure 4.13, some never passed 1% MAP, while others produced mediocre results such as MAP values below 70%. Using a low learning rate during the first iterations such as in yolo-voc.2.0.cfg helped stabilize the models, but did not provide any good results. Neither did turning off the multi-scale training or adding anchors generated for salmon dataset. Figure 4.20 shows YOLOv2 output grids with the anchors for yolo-voc.2.0.cfg and the anchors generated for the salmon dataset. By comparing the anchors of yolo-voc.2.0.cfg and the generated anchors with the shapes of the bounding boxes from the annotated images, it is clear that the generated anchors are more similar in shape to the salmon. However, the bounding boxes from yolo-voc.2.0.cfg are clearly more suited for general purposes, as it contains a higher variety of shapes. Models trained with yolo-voc.2.0.cfg did better than those trained with yolov2-voc.cfg. I identified that this was largely due to its anchors working well. Reducing the learning rate after 60 and 90 epochs had virtually no effect on the results, and it seemed that just using the standard learning rate of 0,001 worked best. In all cases, using the multi-scale training resulted in unstable training. While increasing the input dimensions would most likely increase the accuracy, it would also definitely reduce efficiency. Therefore, I did not include this parameter in the optimization process. In the end, I was not able to improve on the yolo-voc.2.0.cfg configuration.

4.5 Speed Testing Results

Table 4.1 shows the results of the speed testing. The three computers used for the testing had the following specifications: Linux VM running on 2 virtual CPUs and a Nvidia Tesla K80 GPU; Windows desktop computer running on a Intel Core i5 3570K CPU and a Micro-Star International (MSI) Geforce GTX 1050 Ti GPU; Mac laptop running on a Intel Core i7 4650U CPU. The labels DL Module, Win CPU, Win GPU, VM GPU, and Mac CPU refers to Windows running the DL module, Windows running AlexeyAB/darknet with CPU, Windows running AlexeyAB/darknet with GPU, Linux running pjreddie/darknet using GPU, and Mac running pjreddie/darknet on CPU, respectively. The VM GPU and Mac CPU does not have results for validation, this is because pjreddie/darknet does not provide this functionality. The results presented in Table 4.1 describes the speed of: loading the network and weights, testing prediction on one image, validating the MAP of a validation set of four images, and training one model one iteration with a batch size of ten images. The *validate 1* and *train 1* fields in the table simply describe the times for *validate 4* and *train 10* divided by four and ten, respectively. As one might expect, the GPU solutions runs much faster than the CPU versions. The VM did however use strangely long to load the network and weights. This might be caused by limitations of its disk, random access memory (RAM), or CPU. It is also strange that the GPU solutions performs much worse than 40 FPS, which is what the authors of the method achieved. Why the time to test an image in these solutions is slower than training one is also a good question. By further investigation of the code, it seems that actions such as printing to terminal and saving results to file is to blame for these large values, and the isolated detection time of an image on the VM GPU is closer to 0.05 seconds, which is 20 FPS. The Win CPU and Mac CPU test times are fairly similar, so are the Win GPU and VM GPU test times. This implies that the two Darknet projects are not very different. I also suggests that the Darknet framework doesn't require a high-end GPU to perform well. The DL module is the slowest of the solutions. It trains and validates models more than twice as slow as Win CPU, and uses 30% more time to test an image than Win CPU and Mac CPU. The speed test results for the DL module might be slightly off because of overhead caused by VS, but not by a lot. A retest of load and training with the DL module outside VS resulted in a time 311 seconds.

4.6 Usability Testing Results

One usability test of just over an hour was conducted. The test candidate was familiar with Dynamic Imager, but did struggle a bit with the DL module. The amount of time the computer required to run the tasks did not ease the test process. At one point I had to intervene with the test to help the candidate with Windows file paths. However, all the three tasks were completed successfully. After the test, the candidate filled a SUS form. The score result of the SUS form was **60**. This is a low score, which means that the module probably has a lot of room for improvement. The result of this test is however not a very good indicator of usability as the test was only conducted on a single user.

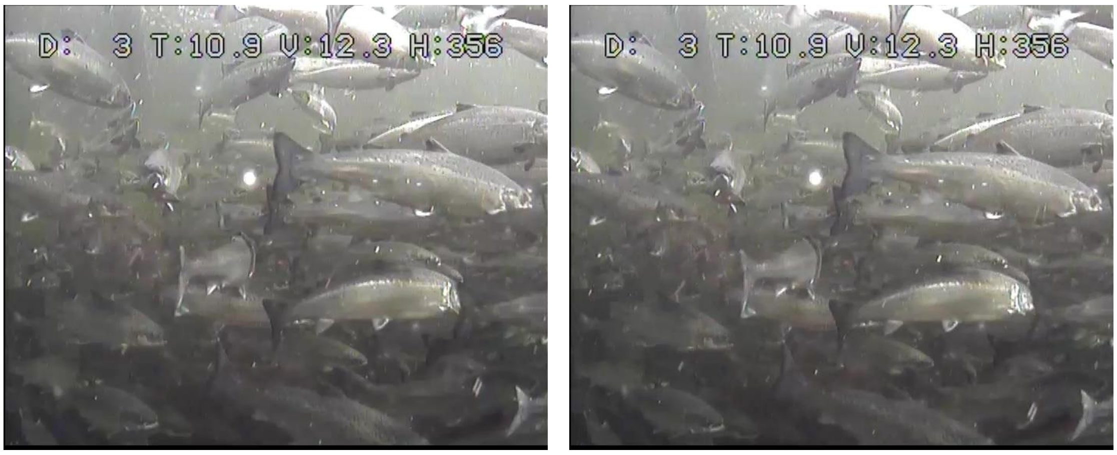


Figure 4.1: Original set 3, image 1 (left) Original set 3, image 2 (right)

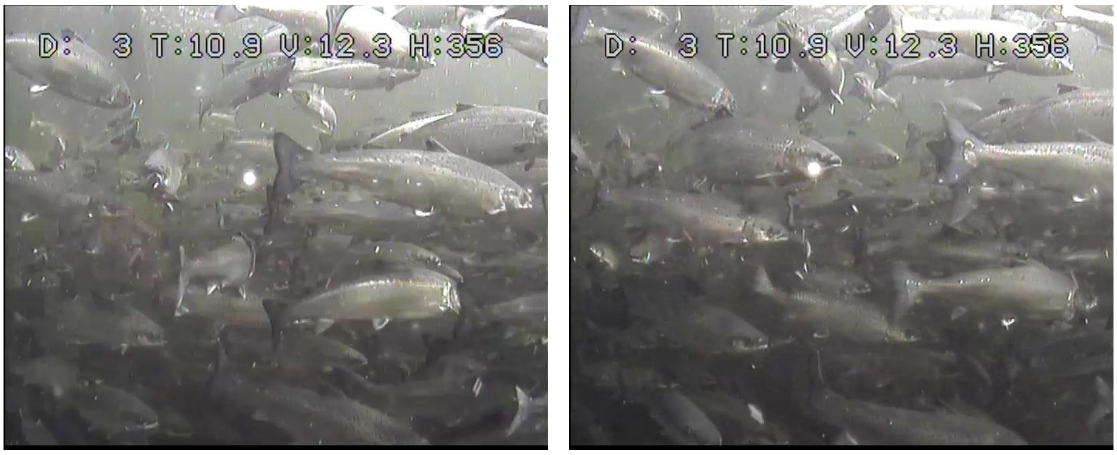


Figure 4.2: New set 3, image 1 (left) New set 3, image 2 (right)

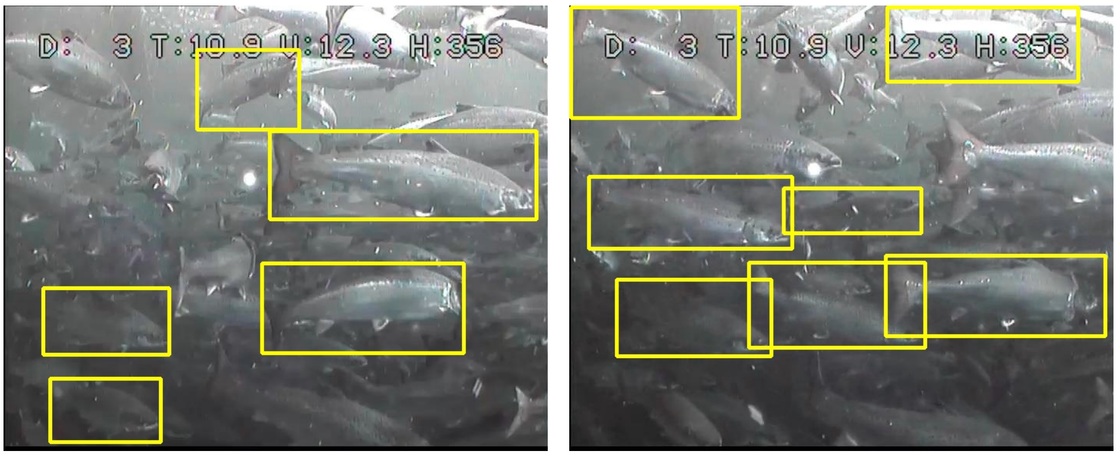


Figure 4.3: Annotation of set 3, image 1 (left) Annotation of set 3, image 2 (right)

	DL Module	Win CPU	Win GPU	VM GPU	Mac CPU
Load	1.4 s	1.1 s	1.9 s	13.7 s	**1 s**
Test Image	9 s	6.2 s	1.5 s	**1.18 s**	6.1 s
Validate 4	32.8 s	12.5 s	**1.1 s**	-	-
∼Validate 1	8.2 s	3.1 s	**0.3 s**	-	-
Train 10	316.8 s	131.1 s	1.9 s	**1.4 s**	180 s
∼Train 1	32.7 s	13.1 s	0.2 s	**0.1 s**	18 s

TABLE 4.1: Speed Testing Results

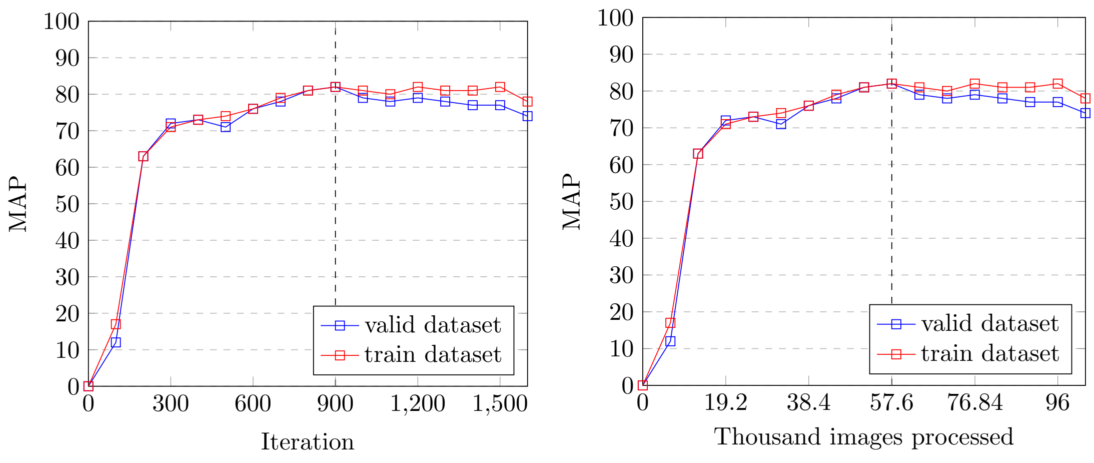


FIGURE 4.4: Training results of yolo-voc.2.0.cfg using pjreddie/darknet

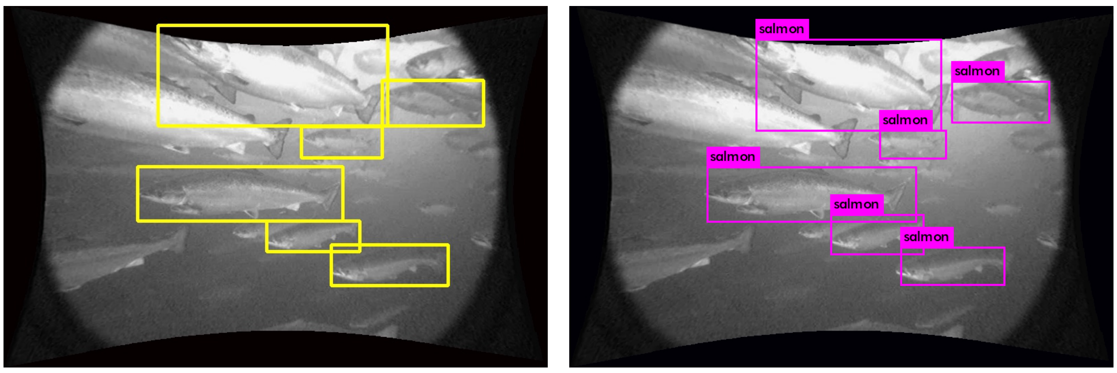


FIGURE 4.5: Annotation from set 1 (left) Good prediction from set 1 (right)

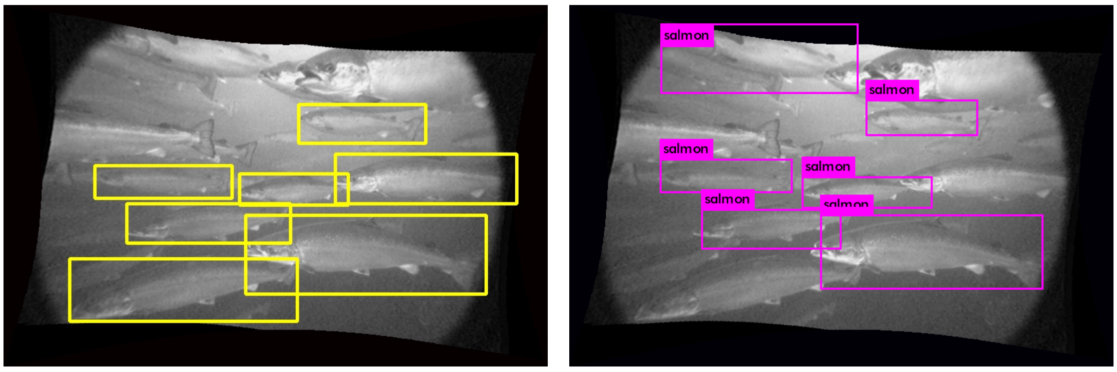


FIGURE 4.6: Annotation from set 1 (left) Bad prediction from set 1 (right)

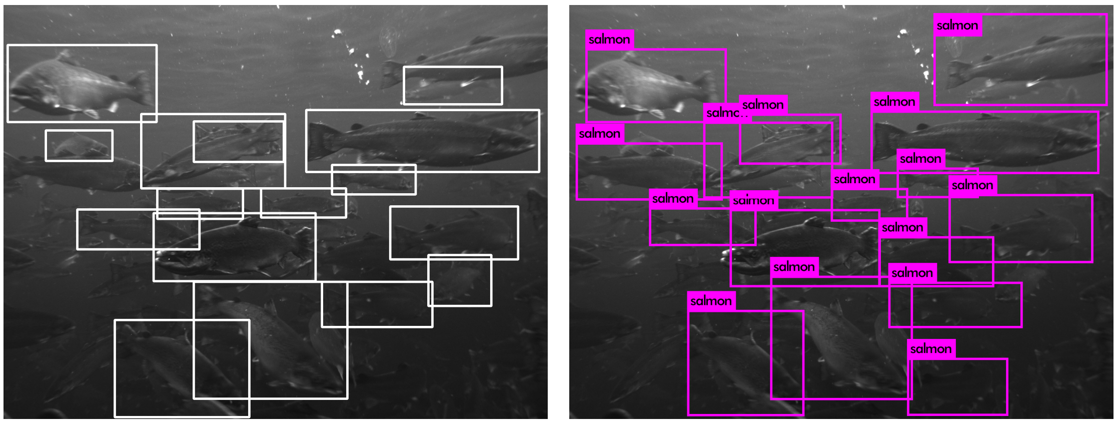


FIGURE 4.7: Annotation from set 2 (left) Good prediction from set 2 (right)

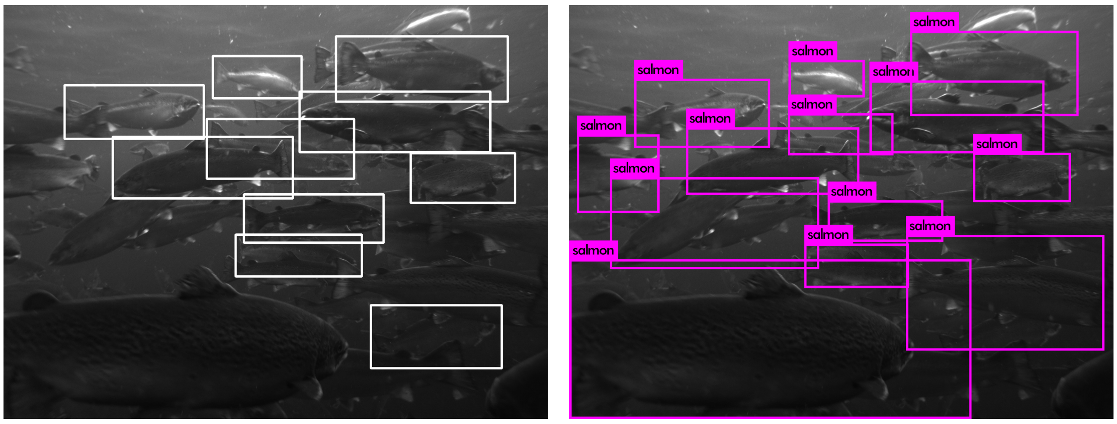


FIGURE 4.8: Annotation from set 2 (left) Bad prediction from set 2 (right)

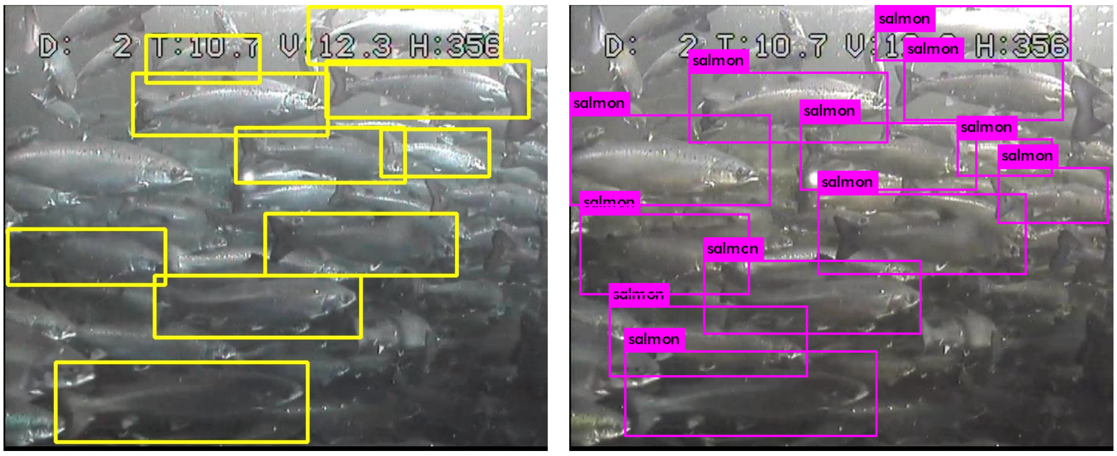


FIGURE 4.9: Annotation from set 3 (left) Good prediction from set 3 (right)

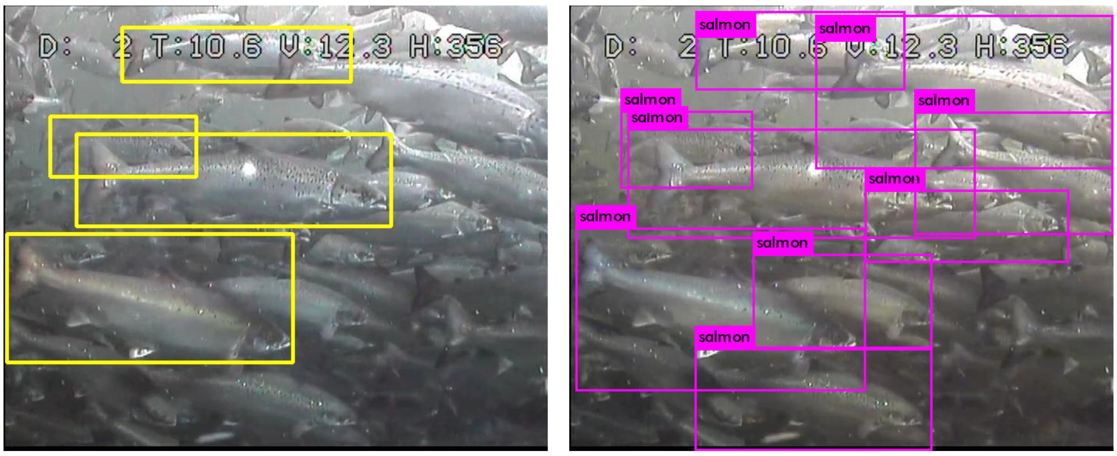
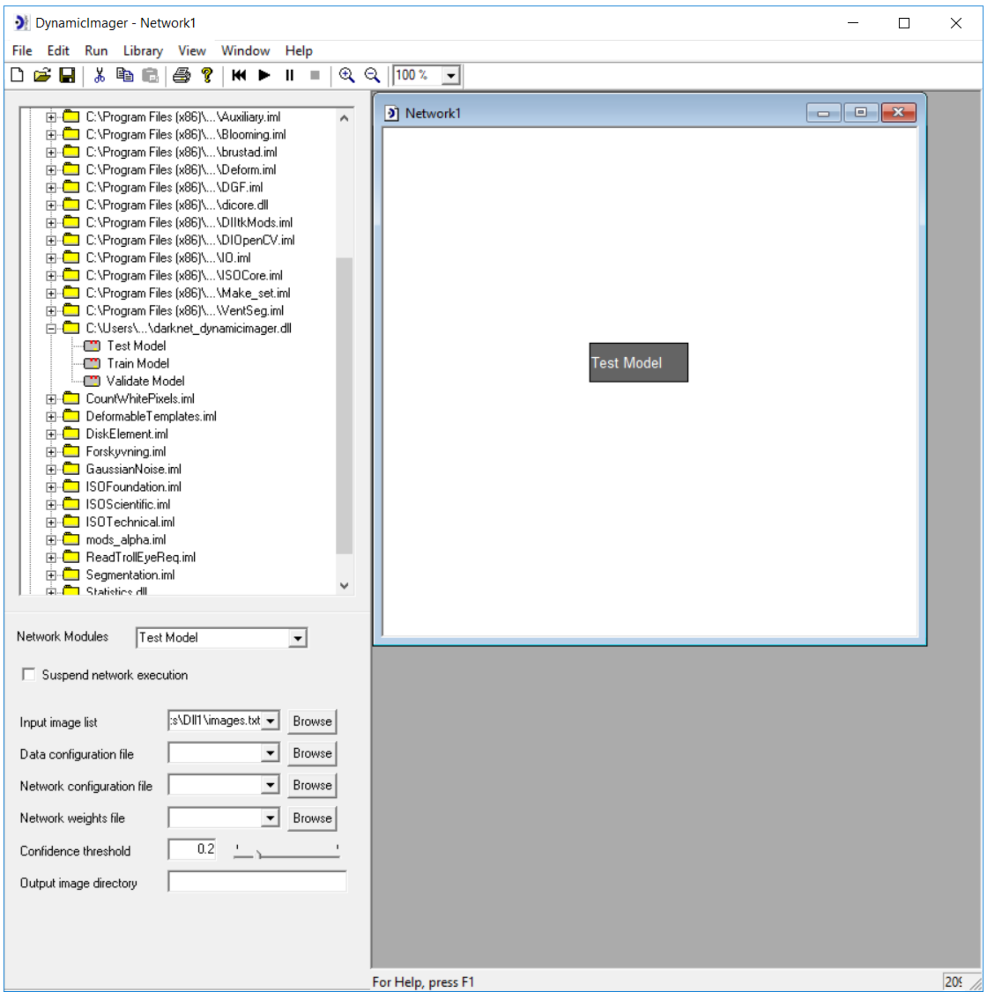


Figure 4.10: Annotation from set 3 (left) Bad prediction from set 3 (right)



```
C:\Program Files (x86)\...\Auxiliary.iml
C:\Program Files (x86)\...\Blooming.iml
C:\Program Files (x86)\...\brustad.iml
C:\Program Files (x86)\...\Deform.iml
C:\Program Files (x86)\...\DGF.iml
C:\Program Files (x86)\...\dicore.dll
C:\Program Files (x86)\...\DIItkMods.iml
C:\Program Files (x86)\...\DIOpenCV.iml
C:\Program Files (x86)\...\IO.iml
C:\Program Files (x86)\...\ISOCore.iml
C:\Program Files (x86)\...\Make_set.iml
C:\Program Files (x86)\...\VentSeg.iml
C:\Users\...\darknet_dynamicimager.dll
    Test Model
    Train Model
    Validate Model
CountWhitePixels.iml
DeformableTemplates.iml
DiskElement.iml
Forskyvning.iml
GaussianNoise.iml
ISOFoundation.iml
ISOScientific.iml
ISOTechnical.iml
mods_alpha.iml
ReadTrollEyeReq.iml
Segmentation.iml
Statistics.dll
```

Network Modules: Test Model

☐ Suspend network execution

Input image list :s\Dll1\images.txt Browse
Data configuration file Browse
Network configuration file Browse
Network weights file Browse
Confidence threshold: 0.2
Output image directory:

Figure 4.11: Dynamic Imager GUI

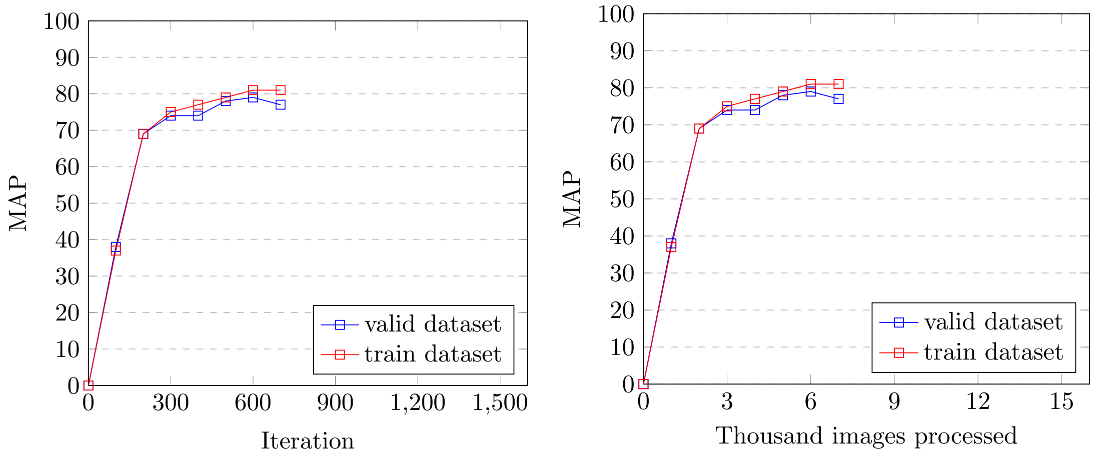


FIGURE 4.12: Training results of yolo-voc.2.0.cfg using DL module in Dynamic Imager

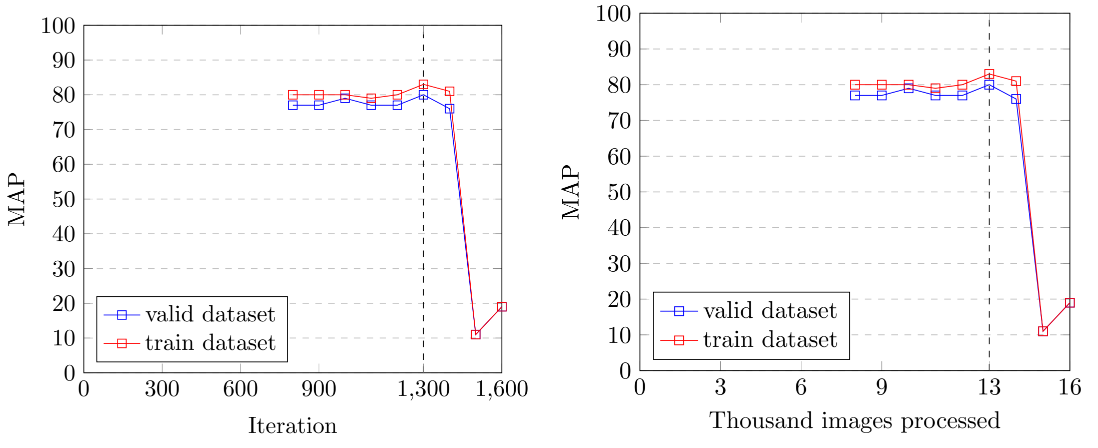


FIGURE 4.13: Training results of yolo-voc.2.0.cfg using AlexeyAB/darknet

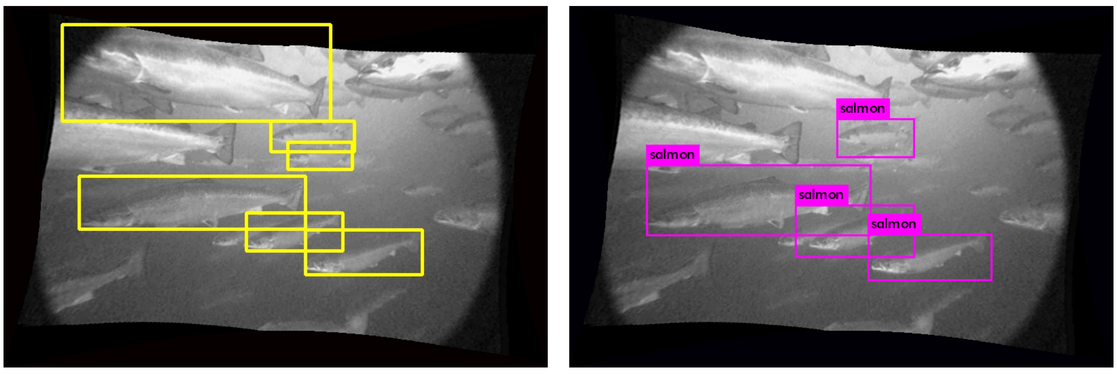


FIGURE 4.14: Annotation from set 1 (left) Good prediction from set 1 (right)

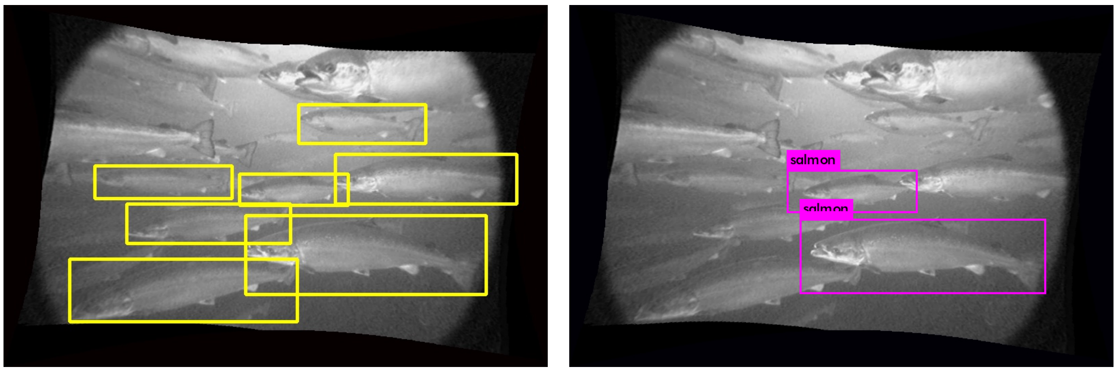


FIGURE 4.15: Annotation from set 1 (left) Bad prediction from set 1 (right)

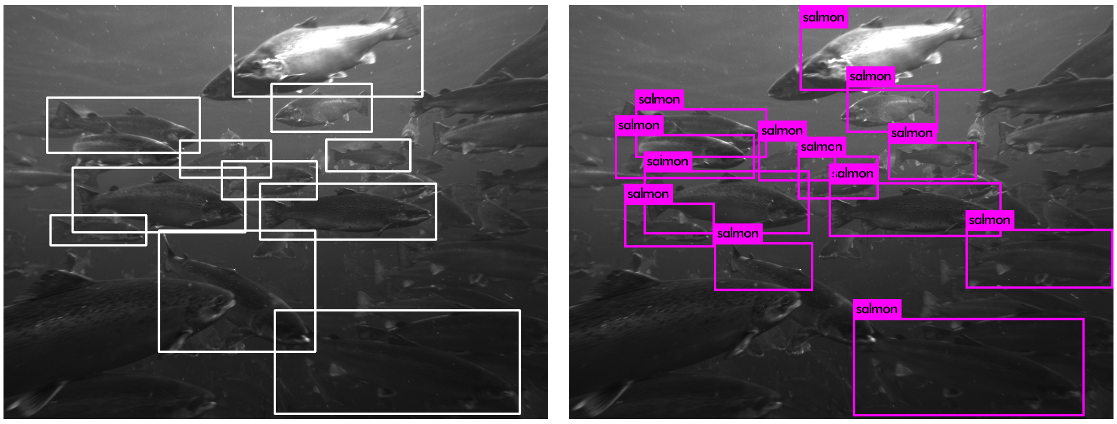


FIGURE 4.16: Annotation from set 2 (left) Good prediction from set 2 (right)

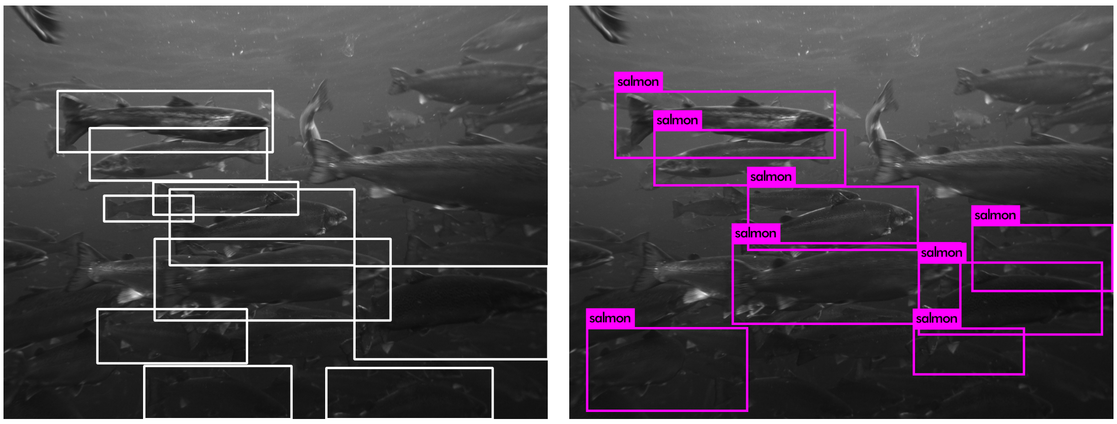


FIGURE 4.17: Annotation from set 2 (left) Bad prediction from set 2 (right)

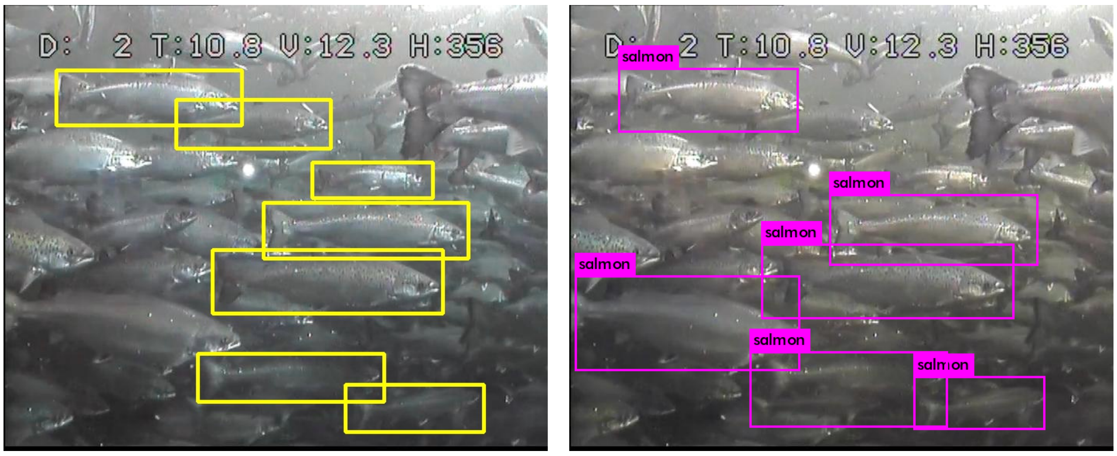


FIGURE 4.18: Annotation from set 3 (left) Good prediction from set 3 (right)

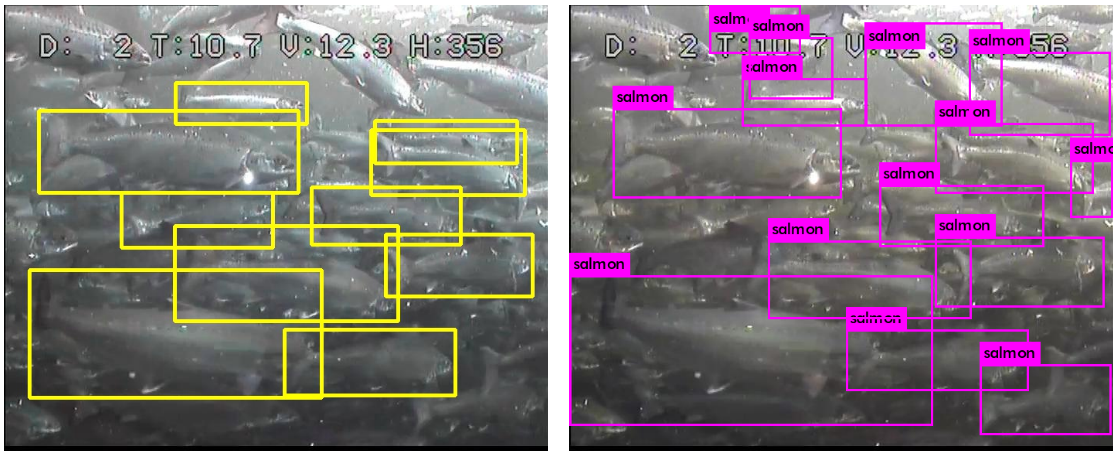


FIGURE 4.19: Annotation from set 3 (left) Bad prediction from set 3 (right)

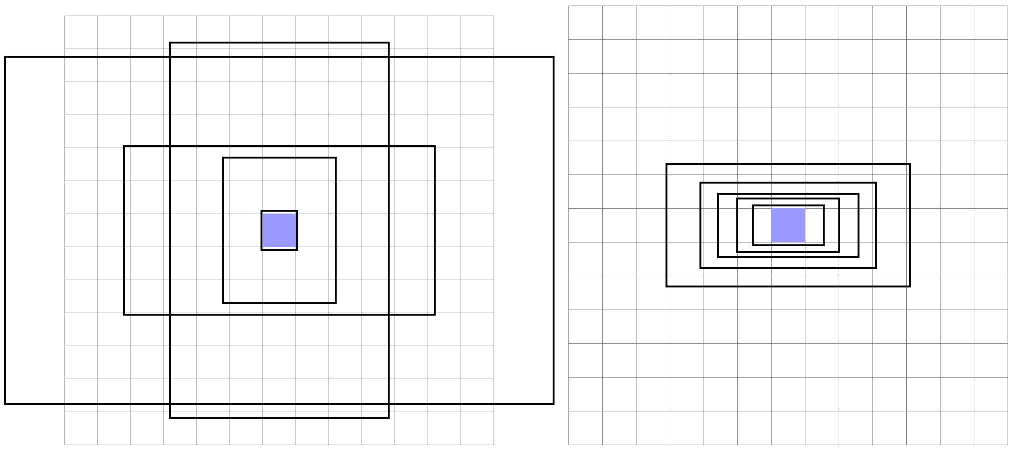


FIGURE 4.20: Yolo-voc.2.0.cfg anchors (left) Custom salmon anchors (right)

Chapter 5

Discussion

This chapter will discuss the strengths and weaknesses of the methods used, and the results they have produced. It will also discuss how the results can be used to answer the research questions.

5.1 Literature Review

The literature review took a lot of time, and I believe it could have been more efficient with less structure. As an example I could probably have saved time by not going through obviously uninteresting search results, and instead prioritizing to read the sources used by interesting papers and theses. A structured literature review does however seem more reliable, as it guarantees that a certain number of papers from a certain number of research collections are reviewed.

5.2 DL Method Comparison

The DL methods presented in this thesis includes state-of-the-art object detectors. I did make the choice to only focus on CNN object detection methods. Although the reasoning for this choice is rather solid, as there seems to be evidence that CNN are the most suitable DL method for processing images, and object detection CNNs are the most suitable CNN methods for the task of predicting bounding boxes for objects, the DL method comparison would have been more comprehensive if other type of DL methods had also been evaluated. The comparison could for instance have included DL methods using semantic segmentation or unsupervised learning. The method used to compare DL methods compares them based on accuracy on popular datasets, and test time prediction speeds. Although these are sensible metrics for evaluation, the comparison conducted is only based on what their authors have stated in the respective research papers. As the results provided in these papers are produced on different hardware and with different configurations, the comparison is less reliable. It would have been more interesting to test the suitable methods using the provided salmon dataset in a single computer environment. Even though the DL method comparison could have been more comprehensive and reliable, I believe this thesis has provided a good insight into suitable methods, and a reasonable comparison of these methods. The suitable methods identified can be used in the answer of RQ 1a. The accuracies and speeds of the suitable methods can be used in the answer of RQ 1b and 1c, respectively.

5.3 Software Framework and Library Comparison

The DL software frameworks and libraries presented in this thesis includes the state of the art in DL software. This software was found through mentions of their use in papers found during the literature review. The choice of using the Darknet framework was largely due to the fact that I chose to use the YOLOv2 method. These are highly compatible. The choice of software could have made based on a more comprehensive evaluation. The suitable software frameworks and libraries can be used in the answer of RQ 2a.

5.4 Dataset Annotation

The annotation task was done with the focus on being able to identify the length of the each fish. This was done because I was informed that the length of the fish would be valuable for the task of calculating biomass in fish farms. By only annotating some of the fish, leaving ones that had visually unidentifiable horizontal lengths out, the problem of only seeking to identifying a portion of the presented objects in images was introduced. Neither the task of annotating fish or only identifying a portion of presented objects in an image were thoroughly research passed the point of a couple of unstructured web searches. Doing a review of the research on these areas could have been valuable for the thesis. Although the annotation process was done in a rather ad-hoc fashion, I believe that the choice of only annotating fish with visually identifiable horizontal lengths was advantageous. This is because the CNN would then be trained on identification of the fish head and tail fin for every fish example.

5.5 Detection of Salmon

Research question 1 asks "To what extent can deep learning methods be used to detect salmon in images". The sub-questions ask which methods could be used for the task, and how accurate and fast these methods are.

To answer this research question, work on reviewing DL methods and evaluating their performance was done. This thesis presents the performance results of two YOLOv2 models on the salmon dataset. These results include achieving an MAP score of as high as 80.2%, and test times, including overhead, of as low as 1.5 seconds per image on a desktop computer with an off-the-shelf GPU. By detecting salmon in images not used for training and salmon not contained in the annotations, the models show their ability to generalize. I regard these results as very good. There is however a good chance that even better MAP scores are achievable. Looking at the prediction results in Figure 4.18 and 4.19, there seems to be several non-annotated salmon detected that should arguably have been annotated. Such examples include the large fish in the bottom left corner in Figure 4.18, and the small fish in the bottom right corner in Figure 4.19. By being more consistent during annotation I could have improved the trained models. The configuration of these models also seem to be improvable. I find it very strange that even after over a dozen models trained with a seemingly more optimized configuration, the initial configuration yolo-voc.2.0.cfg still produced the highest performing models. Based on the fact that this configuration was optimized for PASCAL VOC, I deem it likely that a configuration could be optimized for the salmon dataset as well. Additionally, I could have conducted experiments with various confidence thresholds at prediction time. Such experiments would not have affected the MAP score, but would have affected the IoU scores, which could have produced better predictions. The verification of the fact that YOLOv2 can be used for detecting salmon in images can be used in the answer of

RQ 1a. The results of testing accuracy and speed with YOLOv2 on the salmon dataset can be used in the answer of RQ 1b and 1c, respectively.

5.6 Module Implementation

Research question 2 asks "To what extent can a deep learning module be integrated into an existing program". The sub-questions ask which software frameworks and libraries are suitable for a Dynamic Imager module, and how accurate and fast such a module is.

To answer this research question, work on reviewing software, implementing a DL module for use with 32 bit Dynamic Imager, and evaluating its performance was done. This thesis presents the performance on results on a YOLOv2 model partially trained with the DL module. This model achieved a MAP score of 80.2%, and a test time, including overhead, of 9 seconds per image on a desktop computer with an off-the-shelf CPU. I regard these results are fairly good. The training time per image did however require a lot of time, using 32.7 seconds per image. This is almost twice the amount of time used to train an image using the same framework with CPU, on the same computer, not run through Dynamic Imager. 32.7 seconds per image is also almost 164 times slower than the same framework with GPU, on the same computer, not run through Dynamic Imager. I regard this result as poor. During development of the DL module, I discovered an important distinction between ways of integrating DL into an already existing program. This distinction is between modules providing functionality for either: just prediction; prediction and training; and prediction, training, and network configuration. A module just doing prediction would only need access to test images, one network configuration, and one set of trained weights. Because such a module would only require a single network configuration, and set of weights, these parameters could be hidden from the user, thus leaving the input images as the only user input. A module doing prediction and training, could consequently hide the network configuration from the user. However, a module doing prediction, training, and network configuration would not be able to hide any of the mentioned input from the user, and would require an interface for the task of modifying the configuration. This seems to demonstrate a compromise between configurability and usability. The DL module implemented attempted to be suited for prediction, training, and network configuration. Lacking a proper interface for the large amount of input required from the user resulted in a solution where some of the configuration has to be done via a text editor. This is definetly sub-optimal. I believe a better solution to the user interface would have been to hide some of the inputs, for instance hardcoding initial weights, standard network, and output directory into the module. The result of the usability testing was a SUS score of 60. I regard this as a fairly poor result. The results of testing accuracy and speed with the DL module on the salmon dataset can be used in the answer of RQ 2b and 2c, respectively.

Chapter 6

Conclusion

This chapter will provide answers to the research questions. It will also briefly discuss the importance of this work, and suggest further work.

6.1 Research Questions

Research question 1 asks ”To what extent can deep learning methods be used to detect salmon in images”. To answer this question I will first answer its sub-questions. The DL methods suitable for detecting salmon in images include: Faster R-CNN, SDD, R-FCN, and YOLOv2. The accuracies of these methods, caluclated from a mean MAP score from PASCAL VOC 2007, PASCAL VOC 2012, and COCO 2015, are 64.1%, 68.9%, 70.4%, and 65.3% for Faster R-CNN, SSD512, R-FCN, and YOLOv2 544, respectively. The YOLOv2 method has achieved a MAP of 80.2% using the salmon dataset. The prediction speeds of the suitable methods are 5 FPS, 19 FPS, 5.9 FPS, and 40 FPS for Faster R-CNN, SSD512, R-FCN, and YOLOv2 544, respectively. The YOLOv2 method achieves a speed of 0.8 FPS including overhead, or 20 FPS excluding overhead, on prediction of large images from the salmon dataset. Because there are several accurate and fast object detection methods, one of which has been demonstrated to produce good reults on the salmon dataset, I conclude that DL methods can be used to detect salmon in images to a large extent.

Research question 2 asks ”To what extent can a deep learning module be integrated into an existing program”. To answer this question I will first answer its sub-questions. The software frameworks and libraries suitable for implementing a DL module for Dynamic Imager include: Caffe2, TensorFlow, Darknet, MXNet, and Microsoft Cognitive Toolkit. The DL module integrated into Dynamic Imager achieved a MAP of 80.2% on the salmon dataset. The DL module achieved a prediction speed of 0.1 FPS on large images from the salmon dataset. In addition to the answers to the sub-questions, it is important to address that the DL module trains models very slowly, and has poor usability. On one hand there are several software frameworks and libraries capable of implement a DL module and the implemented DL module achieved a high accuracy. On the other hand the DL module achieved a somewhat slow prediction time, a poor training time, and has poor usability. To be able to create a fast and user friendly DL module, suitable hardware and an extensive user interface is required. Based on these factors I conclude that DL methods can be integrated into an existing program to a moderate extent.

6.2 Importance of Work

The importance of the work done in this thesis is largely rooted in the project motivations described in Chapter 1. By developing a DL module capable of prediction, training, and network configuring for use with Dynamic Imager, I have helped Trollhetta AS in development of CNN solutions. The work on the DL module has also revealed interesting obstacles one could face during the implementation of CNN functionality. The discussion of these obstacles could aid others in similar projects. By annotating the salmon dataset and using it to train high performing CNN models, I have demonstrated that automatic detection of salmon in images is possible. This conclusion can aid the fish farm industry in salmon biomass estimation, possibly even using the same training set, network configuration, and trained weights created for this project. Improving the salmon biomass estimation in fish farms could lead to less misfeeding and mismedication of the fish, which could also halt the development of vaccine resistance in lice.

6.3 Further Work

Beneficial further work on this project includes:

- Assessing the quality of the annotations, possibly adding or removing some annotated fish;
- Testing the prediction quality of either the model I have trained, or a similar model, in a real world scenario;
- Investigating the performance of other CNN methods using the salmon dataset;
- Further investigating the possibilities of hyperparameter optimization on the network configuration used in this work.
- Creating a GPU version of the Dynamic Imager module;
- Adding network connections for the modules *Test Model*, *Train Model*, and *Validate Model* in Dynamic Imager;
- Increasing the DL module usability by reducing user input and adding more user feedback;
- Implementing functionality to automatically determine when training is complete in the DL module;
- Testing the network configuration with other datasets, investigating how generally applicable it is.

Appendix A

Various File Examples

Data Configuration File (.data)

```
classes = 1
train = /path/to/training_set.txt
valid = /path/to/validation_set.txt
names = /path/to/name_file.txt
backup = /path/to/output_directory
labels = /path/to/font_directory # Line added for DL module
```

Training Dataset File (.txt)

```
/path/to/image1.jpg
/path/to/image2.jpg
/path/to/image3.jpg
/path/to/image4.jpg
/path/to/image6.jpg
/path/to/image7.jpg
/path/to/image8.jpg
/path/to/image9.jpg
/path/to/image11.jpg
/path/to/image12.jpg
```

Validation Dataset File (.txt)

```
/path/to/image5.jpg
/path/to/image15.jpg
/path/to/image25.jpg
/path/to/image35.jpg
/path/to/image45.jpg
/path/to/image55.jpg
/path/to/image65.jpg
/path/to/image75.jpg
/path/to/image85.jpg
/path/to/image95.jpg
```

Test Dataset File (.txt)

```
/path/to/image10.jpg
/path/to/image20.jpg
/path/to/image30.jpg
/path/to/image40.jpg
/path/to/image50.jpg
/path/to/image60.jpg
/path/to/image70.jpg
/path/to/image80.jpg
/path/to/image90.jpg
/path/to/image100.jpg
```

Names File (.names)

```
salmon
person
bicycle
car
motorbike
aeroplane
bus
train
truck
boat
traffic light
```

Appendix B

Network Configuration

Yolo-voc.2.0.cfg

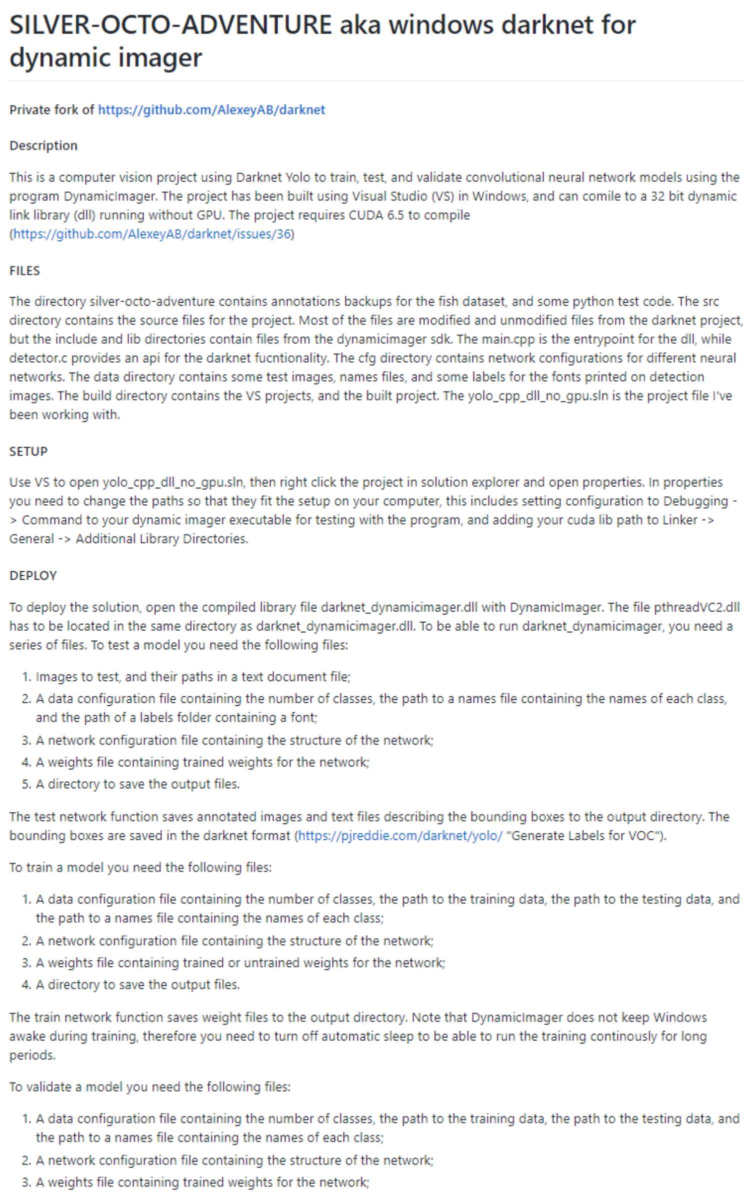
```
[net]
batch=64 #value of 10 used in DL module
subdivisions=8 #value of 10 used in DL module
height=416
width=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.0001
max_batches = 45000
policy=steps
steps=100,25000,35000
scales=10,.1,.1

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=64
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
```

```
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

#######

[convolutional]
batch_normalize=1
size=3
stride=1
pad=1
filters=1024
activation=leaky

[convolutional]
batch_normalize=1
size=3
stride=1
pad=1
filters=1024
activation=leaky

[route]
layers=-9

[reorg]
stride=2

[route]
layers=-1,-3

[convolutional]
batch_normalize=1
size=3
stride=1
```

```
pad=1
filters=1024
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=125 #Commented out in DL module
activation=linear

[region]
anchors = 1.08,1.19, 3.42,4.41, 6.63,11.38, 9.42,5.11, 16.62,10.52
bias_match=1
classes=20 #Commented out in DL module
coords=4
num=5
softmax=1
jitter=.2
rescore=1

object_scale=5
noobject_scale=1
class_scale=1
coord_scale=1

absolute=1
thresh = .6
random=0
```

Appendix C

Github Readme



SILVER-OCTO-ADVENTURE aka windows darknet for dynamic imager

Private fork of https://github.com/AlexeyAB/darknet

Description

This is a computer vision project using Darknet Yolo to train, test, and validate convolutional neural network models using the program DynamicImager. The project has been built using Visual Studio (VS) in Windows, and can comile to a 32 bit dynamic link library (dll) running without GPU. The project requires CUDA 6.5 to compile (https://github.com/AlexeyAB/darknet/issues/36)

FILES

The directory silver-octo-adventure contains annotations backups for the fish dataset, and some python test code. The src directory contains the source files for the project. Most of the files are modified and unmodified files from the darknet project, but the include and lib directories contain files from the dynamicimager sdk. The main.cpp is the entrypoint for the dll, while detector.c provides an api for the darknet fucntionality. The cfg directory contains network configurations for different neural networks. The data directory contains some test images, names files, and some labels for the fonts printed on detection images. The build directory contains the VS projects, and the built project. The yolo_cpp_dll_no_gpu.sln is the project file I've been working with.

SETUP

Use VS to open yolo_cpp_dll_no_gpu.sln, then right click the project in solution explorer and open properties. In properties you need to change the paths so that they fit the setup on your computer, this includes setting configuration to Debugging -> Command to your dynamic imager executable for testing with the program, and adding your cuda lib path to Linker -> General -> Additional Library Directories.

DEPLOY

To deploy the solution, open the compiled library file darknet_dynamicimager.dll with DynamicImager. The file pthreadVC2.dll has to be located in the same directory as darknet_dynamicimager.dll. To be able to run darknet_dynamicimager, you need a series of files. To test a model you need the following files:

1. Images to test, and their paths in a text document file;
2. A data configuration file containing the number of classes, the path to a names file containing the names of each class, and the path of a labels folder containing a font;
3. A network configuration file containing the structure of the network;
4. A weights file containing trained weights for the network;
5. A directory to save the output files.

The test network function saves annotated images and text files describing the bounding boxes to the output directory. The bounding boxes are saved in the darknet format (https://pjreddie.com/darknet/yolo/ "Generate Labels for VOC").

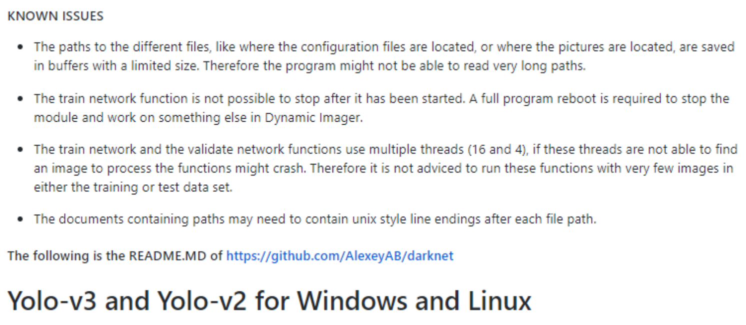
To train a model you need the following files:

1. A data configuration file containing the number of classes, the path to the training data, the path to the testing data, and the path to a names file containing the names of each class;
2. A network configuration file containing the structure of the network;
3. A weights file containing trained or untrained weights for the network;
4. A directory to save the output files.

The train network function saves weight files to the output directory. Note that DynamicImager does not keep Windows awake during training, therefore you need to turn off automatic sleep to be able to run the training continously for long periods.

To validate a model you need the following files:

1. A data configuration file containing the number of classes, the path to the training data, the path to the testing data, and the path to a names file containing the names of each class;
2. A network configuration file containing the structure of the network;
3. A weights file containing trained weights for the network;



KNOWN ISSUES

- The paths to the different files, like where the configuration files are located, or where the pictures are located, are saved in buffers with a limited size. Therefore the program might not be able to read very long paths.
- The train network function is not possible to stop after it has been started. A full program reboot is required to stop the module and work on something else in Dynamic Imager.
- The train network and the validate network functions use multiple threads (16 and 4), if these threads are not able to find an image to process the functions might crash. Therefore it is not adviced to run these functions with very few images in either the training or test data set.
- The documents containing paths may need to contain unix style line endings after each file path.

The following is the README.MD of https://github.com/AlexeyAB/darknet

Yolo-v3 and Yolo-v2 for Windows and Linux

...

Bibliography

[1] Xiao Xiang Zhu et al. Deep learning in remote sensing: A review. October 2017.

[2] Yanming Guo et al. Deep learning for visual understanding: A review. November 2015.

[3] Yann LeCun et al. Deep learning. May 2015.

[4] Hao Zhai. Research on image recognition based deep learning technology. 2016.

[5] Trollhetta as. Visited February 2018. URL http://www.trollhetta.com.

[6] Ingvild Johanne Aarhus et al. Konsekvensanalyse - utredning rundt flgene av feilestimering av biomasse i sjbasert oppdrett. September 2009.

[7] Torgeir Haaland. Investigating the generality of deep learning. June 2016.

[8] Alexander Martin Olofsson. Tactile-sensitive robotic grasping of food compliant objects with deep learning as a learning policy. July 2017.

[9] Stian Jensen and Andreas Løve Selvik. Using 3d graphics to train object detection systems. July 2016.

[10] Hogne Jørgensen. Automatic license plate recognition using deep learning techniques. July 2017.

[11] Ji Yang. Relu and softmax activation functions. February 2017. URL https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions.

[12] Markus Teigen Pike. Computer vision and deep learning in autonomous drones. June 2017.

[13] Michael Nielsen. Neural networks and deep learning; improving the way neural networks learn. December 2017. URL http://neuralnetworksanddeeplearning.com/chap3.html.

[14] Øyvind Kjeldstad Grimnes. End-to-end steering angle prediction and object detection using convolutional neural networks. June 2017.

[15] Nikita Uvarov. Multi-label classification of a real-world image dataset. June 2017.

[16] Fei-Fei Li et al. Cs231n lecture 8 - localization and detection. June 2016. URL https://www.youtube.com/watch?v=_GfPYLNQank.

[17] Convolutional networks. Visited February 2018. URL http://egrcc.github.io/docs/dl/deeplearningbook-convnets.pdf.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[19] Mark Everingham et al. The pascal visual object classes challenge: A retrospective. June 2014.

[20] Adrian Rosebrock. Intersection over union (iou) for object detection. November 2016. URL https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.

[21] Sancho McCann. Average precision. September 2011. URL https://sanchom.wordpress.com/tag/average-precision/.

[22] Alex Krizhevsky et al. Imagenet classifcation with deep convolutional neural networks. June 2017.

[23] Christian Szegedy et al. Going deeper with convolutions. September 2014.

[24] Jianmin Chen et al. Revisiting distributed synchronous sgd. Mars 2017.

[25] Hamed Habibi Agham and Elnaz Jahani Heravi. *Guide to convolutional neural networks a practical application to traffic-sign detection and classification*. Springer, 2017.

[26] Maxime Oquab. Learning and transferring mid-level image representations using convolutional neural networks. 2014.

[27] Sinno Jialin Pan. A survey on transfer learning. 2009.

[28] Matthew D. Zeiler. Visualizing and understanding convolutional networks. 2014.

[29] Jason Yosinski. How transferable are features in deep neural networks? November 2014.

[30] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. Visited February 2018. URL http://cs231n.github.io/convolutional-networks/.

[31] Patrice Y. Simard et al. Best practices for convolutional neural networks applied to visual document analysis. August 2003.

[32] Yann LeCun et al. The mnist database of handwritten digits. Visited February 2018. URL http://yann.lecun.com/exdb/mnist/.

[33] Tsung-Yi Lin et al. Coco common objects in context. Visited February 2018. URL http://cocodataset.org/#home.

[34] Junwei Han et al. Advanced deep-learning techniques for salient and category-specific object detection a survey. January 2018.

[35] Tsung-Yi Lin et al. Microsoft coco: Common objects in context. February 2015.

[36] Coco detection leaderboard. Visited February 2018. URL http://cocodataset.org/#detections-leaderboard.

[37] Mark Everingham et al. The pascal visual object classes homepage. Visited February 2018. URL http://host.robots.ox.ac.uk/pascal/VOC/.

[38] Leaderboards for the evaluations on pascal voc data. Visited February 2018. URL http://host.robots.ox.ac.uk:8080/leaderboard/main_bootstrap.php.

[39] Jia Deng et al. Imagenet: A large-scale hierarchical image database. August 2009.

[40] Imagenet. 2016. URL http://www.image-net.org.

[41] Imagenet large scale visual recognition challenge 2017 (ilsvrc2017). 2017. URL http://image-net.org/challenges/LSVRC/2017/results.

[42] Imagenet large scale visual recognition challenge 2012 (ilsvrc2012). 2012. URL http://image-net.org/challenges/LSVRC/2012/results.html.

[43] Bacon. How does krizhevsky's '12 cnn get 253,440 neurons in the first layer? October 2016. URL https://stats.stackexchange.com/questions/132897/how-does-krizhevskys-12-cnn-get-253-440-neurons-in-the-first-layer.

[44] Imagenet large scale visual recognition challenge 2014 (ilsvrc2014). 2014. URL http://image-net.org/challenges/LSVRC/2014/results.

[45] Karen Simonyan et al. Very deep convolutional networks for large-scale image recognition. April 2015.

[46] Adrian Rosebrock. Imagenet: Vggnet, resnet, inception, and xception with keras. Mars 2017. URL https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/.

[47] Imagenet large scale visual recognition challenge 2015 (ilsvrc2015). 2015. URL http://image-net.org/challenges/LSVRC/2015/results.

[48] Kaiming He et al. Deep residual learning for image recognition. December 2015.

[49] Alfredo Canziani et al. An analysis of deep neural network models for practical applications. April 2017.

[50] Alexis Cook. Global average pooling layers for object localization. April 2017. URL https://alexisbcook.github.io/2017/global-average-pooling-layers-for-object-localization/.

[51] Imagenet large scale visual recognition challenge 2016 (ilsvrc2016). 2016. URL http://image-net.org/challenges/LSVRC/2016/results.

[52] Shaoqing Ren et al. Faster r-cnn: Towards real-time object detection with region proposal networks. January 2016.

[53] Felix Lau. Speed/accuracy trade-offs for modern convolutional object detectors. Mars 2017. URL https://medium.com/@phelixlau/speed-accuracy-trade-offs-for-modern-convolutional-object-detectors-bbad4e4e0718.

[54] Wei Liu et al. Ssd: Single shot multibox detector. December 2016.

[55] Jifeng Dai et al. R-fcn: Object detection via region-based fully convolutional networks. June 2016.

[56] Joseph Redmon et al. Yolo9000: Better, faster, stronger. December 2016.

[57] Jason Brownlee. Introduction to the python deep learning library theano. May 2016. URL https://machinelearningmastery.com/introduction-python-deep-learning-library-theano/.

[58] LISA lab. Theano documentation. November 2017. URL http://deeplearning.net/software/theano/index.html.

[59] Skymind. Deep learning for java. 2017. URL https://deeplearning4j.org.

[60] The Apache Software Foundation. Apache lincence, version 2. 2017. URL https://www.apache.org/licenses/LICENSE-2.0.

[61] Facebook Open Source. Caffe2 a new lightweight, modular, and scalable deep learning framework. Visited February 2018. URL https://caffe2.ai.

[62] Leonardo Araujo dos Santos. Deep learning libraries. Visited February 2018. URL https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/deep_learning_libraries.html.

[63] Tensorflow. Visisted February 2018. URL https://www.tensorflow.org.

[64] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[65] Ronan et al. Torch, a scientific computing framework for luajit. Visited February 2018. URL http://torch.ch.

[66] The Apache Software Foundation. Apache mxnet a flexible and efficient library for deep learning. Visited February 2018. URL https://mxnet.apache.org.

[67] Microsoft. The microsoft cognitive toolkit. 2018. URL https://www.microsoft.com/en-us/cognitive-toolkit/.

[68] Keras: The python deep learning library. Visited February 2018. URL https://keras.io.