MAMN-MAB

Zero-shot classification of salmon lice images by siamese neural networks

Master's thesis in Applied and Computational Mathematics

Author: Kristian Mølbach Lian

Supervisors: Kristian Gundersen & Erik Andreas Hanson

Written: H2022-V2023

> *Artificial intelligence is the quest for machines to acquire knowledge, reason, and exhibit intelligent behavior, bridging the gap between human intellect and computational power.*
>
> *Machine learning is the transformative force that empowers machines to learn patterns, adapt, and improve their performance through experience, echoing the essence of human learning.*
>
> *Deep learning unravels the hidden intricacies of data, unveiling complex representations and insights, while mirroring the intricate layers of human cognition.*
>
> ChatGPT, *2023*

Contents

I would like to express my deepest gratitudes for the people that have assisted in breathing this thesis to life. Specifically, to my supervisors Dr. Kristian Gundersen and Dr. Erik Andreas Hanson for being eagerly invested in my endeavours, for allocating spare time for counseling, for suggesting interesting reads and literatures, and for providing essential and constructive feedback. Also, to my friends and family for their investment and motivational support, your impact has been priceless and vital for the finished product. Lastly, to my better half Kaja for challenging my tendency of procrastination, for supporting me during hard times, and for being patient with me, the time as a master's student would not be the same without you.

1

Abstract

Deep learning models, such as neural networks and its variations, have proven exceptionally useful in the current state of society. However, facilitating competitive performances requires large amounts of data for the models to train on, which is especially true in the problem of classification. Addressing this issue for the scarce image datasets containing salmon lice images used in this thesis, can be done by recasting the problem of *"which class does this image belong to?"*, to rather be a question of image similarity, i.e. *"is image X_1 similar to X_2?"*.

In regards to this thesis, siamese neural networks are employed to distinguish images, rather than to explicitly classify them, which has the effect of producing more data points for training. Exactly how many data points for training is readily developed in this thesis (specifically *triplet cardinality*).
Furthermore, the thesis extensively compares the performance measures of F1-score and TAR@FAR(p) in regards to siamese neural networks, and finds that they differ in terms of prediction strictness and what elements of the confusion matrix they focus on. Specifically, TAR@FAR is designed to be more strict because a bound can be set on the allowance of percentage p of false accepts, whereas F1-score also considers false rejects.
Moving on, the thesis is the first work to cover the procedure of cylindrical convolution [21] in siamese neural networks, and shows that they in fact contribute in addressing the problem of rotated images. Additionally, cylindrical convolution seemingly solves the problem of inconsistent distribution of data.

Conclusively, the best model at predicting image similarity on the synthetic dataset was `Siamese_LeNet5_var` with cylindrical convolutions. On this dataset augmented 100 times, it performed a testing F1-score of $72.5 \pm 2.6\%$ and a testing TAR of $72.8 \pm 3.0\%$ ($mean \pm std$). In terms of the real dataset, testing performances could not be calculated due to dataset scarcity. Regardless, the model that performed the best on the validation dataset was also `Siamese_LeNet5_var` with cylindrical convolutions. On this dataset augmented 100 times, it performed a median validation F1-score of 60.9% and a median TAR@FAR(0.01) of 46.7%.

2

Introduction

The seafood industry

Norwegian export of seafood has grown to become an integral part of its overall export industry, amounting to about 5,8% of its total value [9]. Collectively, the Norwegian Seafood Council (*Norges sjømatråd*)[2] reports Norwegian exported seafood goods in a quantity of 2,9 million tons for a total of 151,4 billion NOK in 2022, which is an increase of approximately 50 billion NOK from 2019, and 100 billion NOK from 2012. Although uncertain economic times have had some effect on the Norwegian currency, the global demand for Norwegian seafood is not to be underestimated. Put into perspective for the average consumer, the quantity of exported seafood amounts to about **40 million meals** worldwide, *every day*.

The exportation of salmon accounted for nearly 70% of the reported Norwegian seafood exportation value as of 2022, and is followingly a vital part of the seafood industry. As with most profitable sources of revenue comes inevitable and difficult problems to solve. The infestation of salmon lice (*Lepeophtheirus salmonis*) is an increasing problem in the salmon farming industry [46], and affects factors such as law-regulated animal welfare (see paragraph §24.b in the Animal Welfare Act [32]), economic favorability, and the ecosystem existing outside the fish farms where salmons are kept [44].

Ecology and labelling of salmon lice

In order to aid with the research on salmon lice and their ecology, there is a need for end-to-end tools in order to monitor and regulate salmon louse ecosystems. A study conducted by Mennerat et al. [29] utilizes microchips glued to salmon lice in order to classify and keep track of louse individuals. This labeling procedure has the advantage of being somewhat practical to use once the microchips have been properly attached. Identifying the louse is done by a simple scanning procedure, and the louse does not need to be removed from its habitat during scanning. Disadvantageous to this approach, is that the chips occasionally fall off the lice, and the initial labelling procedure is cumbersome. The chips also have a tendency to stop working, and are compatible with female lice only due to their body size.

Conducting ecology research on salmon lice by **imaging** could provide several advantages as opposed to the procedure of manually tagging them. By imagery, the animal is subject to less mental and physical stress, and is more cost-efficient than using microchips. Image data also provides unique data with a label, as opposed to a microchip that only give the label without any unique data. The caveat to using image data, which also serves as motivation for this thesis, is that it has to be processed by appropriate image analysis schemes.

Artificial intelligence to classify individuals of salmon lice

Recent years have seen a drastic increase of interest in the field of artificial intelligence (often abbreviated as AI, speaks to algorithms and procedures mimicking human intelligence and cognition), and can be attributed to things such as the availability of evergrowing datasets, education tools and advances in computer hardware and software. Novel advancements in natural language processing and generative models have introduced the general population to accessible tools such as ChatGPT and DALL-E [37],[38] that can produce human-like text and images, respectively. The impact of AI on society is consequently difficult to overstate, and continues as of writing this thesis to impact society in revolutionary ways.

The field of AI delivers a wide range of algorithms and techniques suitable for a variety of different types of data and tasks. Problems thought to require advanced and custom algorithms, now commonly run through a process of "*can we use artifical intelligence to solve this*". What's more, the field sees frequent suggestions and refinements through continuous research and development. Resultingly, this thesis aims to suggest and explore the use AI-tools in the subfield of *deep learning* to present end-to-end means of classifying individuals of salmon lice, in order to aid with salmon ecology research.

As is typical with most problem-solvers, a natural question is the one of pros and cons to such an apporach. Advantageous to the use of AI, is high flexibility and adaptability to different types of data, as well as the ability to process large volumes of information. These attributes are likely to contribute to significant time and cost savings, as well as improving efficiency and accuracy in analyzing data. Additionally, AI algorithms are often capable of identifying patterns and relationships in data that may not be immediately apparent to analysis performed by humans. Once trained by proper means, they offer potentially powerful tools for decision-making and problem-solving.

On the other hand, the approach of AI requires in many cases a sizeable dataset which may be expensive to acquire. Moreover, a large dataset might introduce other problems such as biased and prejudiced predictions through uneven and unrepresentative samples [10]. In the field of computer vision, correctly handling the dataset is also of importance for a variety of reasons such as achieving invariance properties and managing the storage requirement. Another important factor to consider is that very deep models are generally seen as black boxes, a term often used in the domain of AI to describe functions that lack interpretability as they, more often than not, are simply too complex.

Regardless, this thesis deploys a handful of existing deep learning models tweaked to approach the problem of classifying individuals of salmon lice. They are of differing complexities in order to quantify computational needs, and the models are of the class "convolutional neural networks" (CNNs) [26] which have shown great promise in a wide variety of computer vision problems. They are set up and trained as "siamese neural networks" (SNNs) [7],[4], which is motivated by the nature of the datasets this thesis works with, and that the goal of these networks is to facilitate a way of measuring the similarity between pairs of salmon louse images. Training such a network setup is performed by utilizing the Triplet Loss function [40], which has proven useful to the domain of face recognition. Furthermore, the images in the datasets appear rotated, which poses a big hiccup for baseline CNNs generalizability. To combat this, this thesis performs polar transformation of images, and extends its reach to the method of "cylindrical convolution" [21], in which the layers of the CNNs performing convolution are replaced with the cylindrical convolution operation to address rotated images. This in turn recasts rotation into translation. Furthermore, the thesis will investigate attributes of salmon lice images that impact the performance of *zero-shot classification*. In due course, all of these terms are elaborated to paint a clearer picture.

Classification by similarity

A noteworthy comment to the thesis' approach to investigate and classify images of salmon lice,

is that it largely deviates from the classical approach to classifying images. A standard approach to solve the problem of classifying images, is to have the images be sophistically encoded by a CNN to extract a vector of meaningful features from the input image. This vector would then be fed into a classification layer to predict which class the input image belongs to. This thesis however, looks at the problem of using similarity to predict whether two images are of the same class. CNNs are still employed to encode feature vectors, but the main difference is that the CNN is trained by the means of predicting similarity, rather than which class the input belongs to. These approaches of predicting classes and similarities are elaborated in Chapter 4 along with their general pros and cons, but is appropriately introduced here to make the reader aware of how deep learning models are utilized in this thesis.

Relevant works

By classical image processing means, Merz et al. [30] shows that the melanophore patterns found on salmon heads can be used for long term individual recognition.

In the domain of salmon lice, Pettersen et al. [35] developed a regression model based on hyperspectral imaging to count lice and classify their maturity on dynamic underwater images.

A master's thesis from UiB [23] explores image registration methods for classifying salmon lice individuals, whereas some of its preprocessing work in cropping and translating salmon louse images is used in this thesis.

Training neural networks to relate/distinguish two inputs from each other was first developed by Baldi and Chauvin [4] to verify images of signatures, and later named by Bromley et al. [7] where the problem was to match fingerprints. These works are the origins of siamese neural networks, which is reiteratively how this thesis will approach the problem of relating/distinguishing two images of salmon lice. In this thesis, it is also referred to as a *siamese framework*.

Another master's thesis, extended to be a research article by the name of *FishNet* [28] deploys several convolutional neural networks (CNNs) in a siamese framework by the use of the Triplet Loss function [40]. Aimed towards image data of salmon faces in farming cages, they achieve a true positive rate of 96% (evaluated at a false accept rate of 0.1%).

Software important to the thesis

The code used for this thesis can be found in the GitHub repository: `https://github.com/kisen123/mastersthesis`. It will, however, require refurbishing and proper commenting for future use.

The code to perform cylindrical convolution can be found in the GitHub repository: `https://github.com/mcrl/CyCNN`. Note that the code for `CyConv` layers have been slightly modified to accompany the hyperparameter of *groups*, which is relevant for MobileNetV3 model used in this thesis. The modified script is located in `cyconvlayer.py` inside the `models` folder of the `/mastersthesis` repository.

2.1 Problem formulation

In order to gain a better understanding of the ecosystem of salmon and lice (*Lepeophtheirus salmonis*), Mennerat et al. [29] have conducted extensive labeling work through population monitoring by manually gluing microchips to female lice that infest salmons in fish tanks, whereas male lice are unlabeled due to their smaller size. The lice are all imaged, and the images are labeled according to the available microchips.

In more detail, the sampling procedure of the lice in the fish lab is briefly summarized as follows: A number of fish are held in a set of fish tanks. As time passes, salmon lice can tend to switch hosts, and it is of interest to quantify this constantly changing ecosystem. The lice that occupy a labeled fish are imaged, and re-attached to the fish. This sampling procedure is then re-done after some amount of time, and Figure 1 attempts to illustrate this lab pipeline:

(a) Image taken 19$^{\text{th}}$ of February, 2021

(b) Image taken 26$^{\text{th}}$ of March, 2021

Figure 1: The lice from one fish are extracted, and placed on a piece of paper in a petri dish. Male lice appear smaller than female lice, in which female lice are easily distinguishable by their lower body consisting of anatomical parts for reproduction.

Figure 1 merits an elaboration, as these images are the basis of the entire thesis. Specifically, Figure 1 shows two images of all the salmon lice picked from fish number 7, which resides in fish tank number 5 (K5F7). The captions of subfigures 1a and 1b indicate their sampling date (also written in the top-right part of the images), and some very interesting questions arise from these images: *"What happened to the lice in-between sampling dates?", "What made fish number 7 suddenly uninteresting for the lice at a later date?", ...*

In accelerating the acquisition of results for the thesis, the salmon louse images come preprocessed in terms of circularly cropping the images[1], with the louse's mouth in the center of the image as Figure 2 illustrates:

[1]Cropped images are results of "in-house preprocessing", courtesy of co-supervisor Dr. Erik Hanson

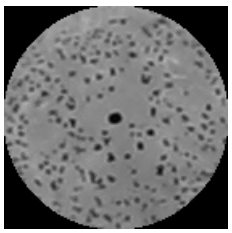


Figure 2: The louse's mouth (the big black dot in the image center) is the circular crop center. The images are circularly cropped to avoid any padding artefacts due to rotational augmentation performed on the images, something that will be re-visited later in this thesis (ref. section 4.3.7).

In aiding the salmon louse ecosystem study, this thesis attempts to present means of classifying images of salmon lice. Specifically, the task is to perform **binary classification** in order to deduce a pair of salmon lice images as a *genuine pair* (same identity), or an *impostor pair* (different identity).

Furthermore, the thesis explores what attributes of salmon louse images are impactful to the performance of classifying image pairs as genuine and impostor. To do this, a program has been developed to synthetically produce salmon louse images[2]. The variables that produce the images can, to the user's own liking, be set to mimic real images of salmon lice. Due to this, the thesis has the ability to pose hypotheses on *both* the dataset, and which model to employ. For illustrative purposes, a synthetically made salmon louse image is shown in Figure 3:

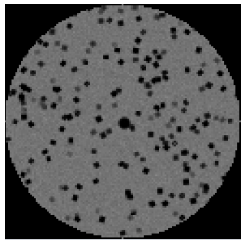


Figure 3: A synthetically produced salmon louse image. They also follow the tradition of being circularly cropped for convenience purposes.

Comprehensively, the thesis aims to answer the following *Research Questions*:

RQ1. Will zero-shot classification by siamese neural networks be able to correctly relate salmon louse images with similar IDs to a satisfactory degree?

RQ2. What attributes of salmon lice images have a significant effect on the performance of zero-shot classification.

2.2 Contributions

This thesis presents detailed theory on the size of a triplet dataset, and means to still include unlabeled images as part of the triplet dataset. To the best of our knowledge, this theory has

[2]An important distinction to make, is that the melanophore patterns on the salmon lice are assumed relatively fixed over time such that they are able to be visually related.

not been presented as detailed as this thesis does it (see section 4.3.6).

Performing cylindrical convolution has not been implemented in siamese neural networks, to the best of the author's knowledge. Achieving rotation invariance in a zero-shot problem formulation is important to consider, as rotated images could violate the perfomance of networks that do not consider rotation.

The master's thesis by Kvernevik [23] performs studies on the same image database, but by classical image processing means. Although this does fall under the definition of artifical intelligence, one might hope that salmon louse identification by deep learning methods is an approach that achieves imporved performance and efficiency.

The synthetic dataset script, along with the pipeline developed for this project is readily available at GitHub: `https://github.com/kisen123/mastersthesis`. It serves to verify the results achieved in this thesis, and acts as a stepping stone for future development and research. More than that, it poses as a playground to experiment with parameters not covered in this thesis, and is applicable to other data domains regarding circular images with dots (e.g. images of bacteria colonies and celestial bodies). In building more sophisticated software to further research the area, this thesis represents a *proof-of-concept* of the presented methodologies.

2.3 Thesis outline

The thesis is outlined as follows:

Chapter 2 introduced the research field in which the thesis is based on. It presented the *research questions* the thesis attempts to solve, along with images of the datasets and how they are sampled. The thesis' contributions towards further work are also appropriately covered here.

Chapter 3 covers general machine learning/deep learning theory required to understand and build on the theory more specific to the thesis. Specifically, the chapter covers details on neural networks, how they are trained, and means to prevent neural networks from overfitting.

Chapter 4 elaborates and motivates more specific theory in order to answer *RQ1* and *RQ2*. It presents the datasets in a more detailed manner, covering their cardinalities, which loss function to use for training, and appropriate performance measures.

Chapter 5 presents the pipeline of training and evaluating models, and displays the results and discussions of *RQ1* and *RQ2*.

Chapter 6 condenses the results and discussions regarding *RQ1* and *RQ2*. It also reiterates the findings and contributions this thesis has made to the research field, whilst alluding to future work that requires attention.

3

Background

This chapter covers basic concepts and methodologies from the literatures of machine learning and deep learning. It is primarily reliant on theory as it is presented in the following resources:

- *Deep Learning* (accessible by `https://www.deeplearningbook.org/`) [12]
- *Dive into Deep Learning* (accessible by `https://d2l.ai`) [47]
- *Digital Image Processing* [11]
- *Pattern Recognition and Machine Learning* [5]

Furthermore, the reader is advised to confer with the Machine Learning Glossary [1] for any terms not specifically elaborated in this thesis, and can be accessed by the following URL: `https://developers.google.com/machine-learning/glossary/`. Note that this glossary mostly supports the TensorFlow machine learning library, and that this thesis is based on PyTorch.

3.1 Neural networks

Neural networks are a rich and powerful family of models, with applications to a large variety of fields. In a world where computational hardware is evolving and education is more available, intricacies of these models are still very much an area of research today. This section starts by showing and elaborating on a simple neural network known as a multi-layer perceptron (MLP).

Artificial perceptrons

To begin with, a linear transformation on a D–dimensional input vector $\boldsymbol{x} = (x_1, ..., x_D)^T$ with a D–dimensional set of parameters/weights $\boldsymbol{w} = (w_1, ..., w_D)^T$ is performed. The layer in the neural network is denoted by square brackets $^{[]}$ in the superscript. Further, a *bias*-term b is added, and this summation ultimately produces pre-activations z as such:

$$z^{[1]} = {\boldsymbol{w}^{[1]}}^T \boldsymbol{x} + b^{[1]} = w_1 x_1 + w_2 x_2 + ... + w_D x_D + b \tag{1}$$

Note 3.1. *In some literatures, the bias term b is instanced as the product $w_0 x_0$, where x_0 is conveniently defined as* 1*. This achieves a more compact notation, but adding the bias term as Equation (1) does will be how this thesis denotes the bias term.*

The observant reader will recognize Equation (1) as nothing more than the equation for linear regression. Following this linear transformation, perhaps the most important operation for any functioning neural network is performed; the linear transformation is wrapped in a **non-linear** activation function (typically this output is referred to as a):

$$a^{[1]} = g^{[1]}(z) = g^{[1]}\left({\boldsymbol{w}^{[1]}}^T \boldsymbol{x} + b^{[1]}\right) \tag{2}$$

The importance of $g()$ being a non-linear function is elaborated in a separate section (section 3.2). A graphical representation of an artificial perceptron is depicted in Figure 4, where the blue circle corresponds to the network's only artificial perceptron:

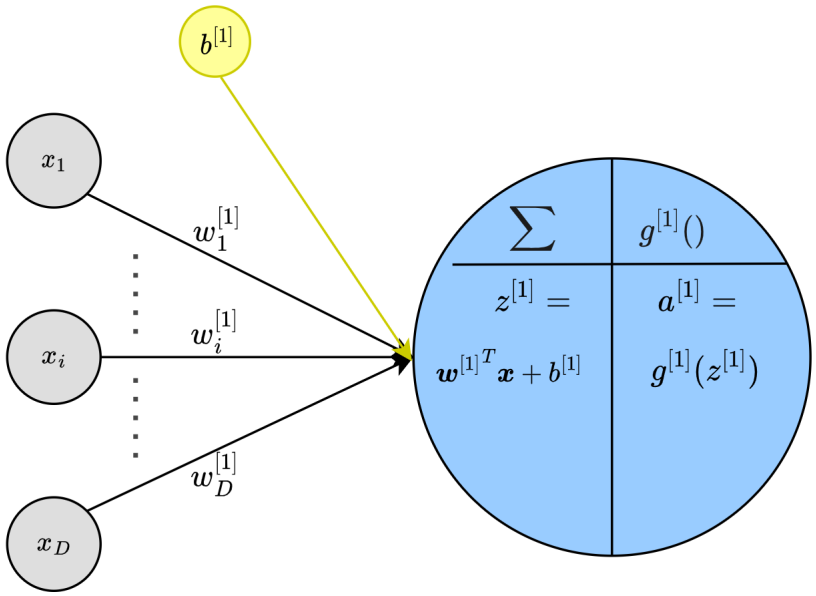


Figure 4: The artificial perceptron performs a weighted sum of the weights $\boldsymbol{w}$ and the input vector $\boldsymbol{x}$, plus a bias term b. This sum is then fed through an activation function $g^{[1]}()$ to produce an output $a^{[1]}$.

Multi-layer perceptron (MLP)

Instead of restricting the network to have just one perceptron and one layer, it can be extended by adding L layers, and $m^{[l]}$ neurons to the $l-$th layer. As mathematical tradition would have it, adding neurons and layers introduces the need for matrix notation. The following equation is Equation (2) expanded to work with a multi-layer perceptron:

$$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}, \qquad \boldsymbol{a}^{[0]} = \boldsymbol{x} \tag{3}$$

Importantly, the neurons must be passed through an activation function:

$$\boldsymbol{a}^{[l]} = g^{[l]}\left(\boldsymbol{z}^{[l-1]}\right) \tag{4}$$

Machine learning terminology has this way of using matrix notation named *vectorizing*, which serves to make the notation for neural networks as compact as it can be.

For simplicity, this example restricts the number of layers to just one hidden layer. In writing out $\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$, the *vectorizing* term hopefully becomes apparent:

$$\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]} = \begin{bmatrix} — & \boldsymbol{w}_1^{[1]T} & — \\ — & \boldsymbol{w}_2^{[1]T} & — \\ & \vdots & \\ — & \boldsymbol{w}_i^{[1]T} & — \\ & \vdots & \\ — & \boldsymbol{w}_m^{[1]T} & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} + \begin{bmatrix} b_1{}^{[1]} \\ b_2{}^{[1]} \\ \vdots \\ b_i{}^{[1]} \\ \vdots \\ b_m{}^{[1]} \end{bmatrix} = \begin{bmatrix} z_1{}^{[1]} \\ z_2{}^{[1]} \\ \vdots \\ z_i{}^{[1]} \\ \vdots \\ z_m{}^{[1]} \end{bmatrix} = \boldsymbol{z}^{[1]} \tag{5}$$

Now, each layer of neurons can be represented by vectors as demonstrated by Equation (5). Note that each layer contains a number of $m^{[l]}$ neurons, and that the weights vectors $\boldsymbol{w_i}$ are

$D-$dimensional. This notation is dropped inside vectors and matrices purely for aesthetic purposes.

A graphical representation of such a classifier with one hidden layer is depicted in Figure 5:

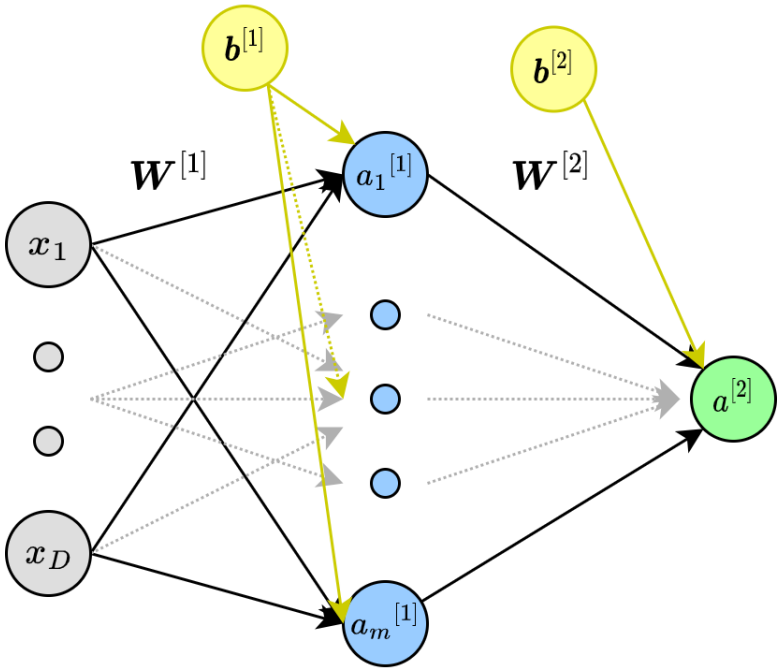


Figure 5: The computation graph of an MLP with one hidden layer illustrated. The gray nodes denote the input features $\boldsymbol{x}$. The blue nodes denote the hidden features, in which there is just a single layer denoted by the superscript $^{[1]}$. The green node denotes the output of the network, which in this case is just a single number $a^{[2]}$. The dotted lines are present to illustrate the connections not explicitely visualized by the figure, as illustrations of the computational graphs of MLPs tend to get quite messy.

Following this illustration of an MLP, Figure 4 can be updated to visualize a general artifical neuron in an MLP:

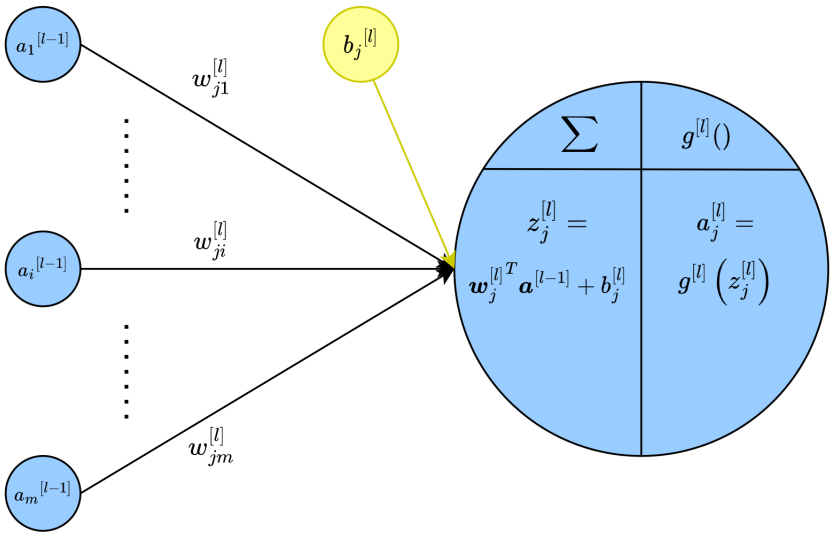


Figure 6: Visual representation of how the j-th hidden neuron in layer l is calculated by a weighted sum (weights by $\boldsymbol{w}_{ji}$) of the previous layer's activations $\boldsymbol{a}^{[l-1]}$, along with adding the bias term depicted in yellow. Note that this is a general representation of a perceptron inside an MLP.

With the general description of using multiple artificial perceptrons covered, one can build an MLP with an arbitrary number of layers and perceptrons. Before moving on to describing the theory of *activation functions*, the formulas for *forward-propagation* and the training data notation must be covered.

Forward-propagation, and adding training data to the MLP notation

Crucial to the performance of neural networks, is the need for substantial amounts of training data. Followingly, a need for matrix notation is important when considering multiple input vectors of training data.

Suppose an MLP is fed a number of N vectors, where N is the number of training data vectors. The input matrix $\boldsymbol{X}$ is then:

$$\boldsymbol{X} = \begin{bmatrix} | & | & & | & & | \\ \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \dots & \boldsymbol{x}^{(n)} & \dots & \boldsymbol{x}^{(N)} \\ | & | & & | & & | \end{bmatrix} \tag{6}$$

With N training data vectors, there is resultingly a number of N pre-activations $\boldsymbol{Z}^{[l]}$ and activations $\boldsymbol{A}^{[l]}$:

$$\boldsymbol{Z}^{[l]} = \begin{bmatrix} | & | & & | & & | \\ \boldsymbol{z}^{[l](1)} & \boldsymbol{z}^{[l](2)} & \dots & \boldsymbol{z}^{[l](n)} & \dots & \boldsymbol{z}^{[l](N)} \\ | & | & & | & & | \end{bmatrix} \tag{7}$$

$$\boldsymbol{A}^{[l]} = \begin{bmatrix} | & | & & | & & | \\ \boldsymbol{a}^{[l](1)} & \boldsymbol{a}^{[l](2)} & \dots & \boldsymbol{a}^{[l](n)} & \dots & \boldsymbol{a}^{[l](N)} \\ | & | & & | & & | \end{bmatrix} \tag{8}$$

With the previously covered notation conventions, the formulas of *forward-propagation* is as follows (recall that $\boldsymbol{A}^{[0]} = \boldsymbol{X}$):

$$\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{b}^{[1]} \tag{9}$$

$$\boldsymbol{A}^{[1]} = g^{[1]}\left(\boldsymbol{Z}^{[1]}\right) \tag{10}$$

$$\boldsymbol{Z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]} + \boldsymbol{b}^{[2]} \tag{11}$$

$$\boldsymbol{A}^{[2]} = g^{[2]}\left(\boldsymbol{Z}^{[2]}\right) \tag{12}$$

$$\vdots$$

$$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]} \tag{13}$$

$$\boldsymbol{A}^{[l]} = g^{[l]}\left(\boldsymbol{Z}^{[l]}\right) \tag{14}$$

3.2 Activation functions

What makes neural networks so flexible can be attributed to many things such as their variability in layers and the number of neurons, but the necessity of activation functions cannot be understated when building a neural network. The theory in this section is additionally based on the works of Lederer [27].

Assume a neural network containing no activation functions, just neurons containing linear regressions. Forward-passing the input $\boldsymbol{X}$ yields a large, yet simple set of linear transformations:

$$\boldsymbol{A}^{[l]} = \boldsymbol{W}^{[l]}\left[\boldsymbol{W}^{[l-1]}\left[\boldsymbol{W}^{[l-2]}\left[\ldots\boldsymbol{X}\right] + \boldsymbol{b}^{[l-2]}\right] + \boldsymbol{b}^{[l-1]}\right] + \boldsymbol{b}^{[l]} \tag{15}$$

The point being made by Equation (15), is that the model will become just one gigantic linear transformation. By adding non-linear activation functions in between each linear transformation, the weight matrices $\boldsymbol{W}$ of the individual layers are allowed to vary independently, enabling the neural network to learn complex and non-linear relationships between the inputs and outputs. Adding activation functions to Equation (15), Equations (9)-(14) can be compactly written as follows:

$$\boldsymbol{A}^{[l]} = g^{[l]}\left(\boldsymbol{W}^{[l]}\left[g^{[l-1]}\left(\boldsymbol{W}^{[l-1]}\left[g^{[l-2]}\left(\boldsymbol{W}^{[l-2]}\left[\ldots\boldsymbol{X}\right] + \boldsymbol{b}^{[l-2]}\right)\right] + \boldsymbol{b}^{[l-1]}\right)\right] + \boldsymbol{b}^{[l]}\right) \tag{16}$$

Interestingly, the only requirement for an activation function, is that it must be non-linear. Discontinuities are allowed, and non-differentiable functions are allowed, as long as a derivative is defined for the entire range of inputs. For completeness, a non-linear function **fails to satisfy** any of the linear function conditions presented by Definition 3.1:

Definition 3.1 (*Linearity*)**.** Suppose α is a scalar, and x, y are vectors in some $d-$ dimensional vector space. A function $g()$ is linear if it satisfied additivity, and homogeneity:

$$\text{linearity of } g() : \begin{cases} g(x+y) = g(x) + g(y), & \text{additivity} \\ g(\alpha x) = \alpha g(x), & \text{homogeneity} \end{cases} \tag{17}$$

Listed below are some common activation functions, whereas some of them are used in the models of this thesis. Figure 7 displays the activation functions covered in the following paragraphs:

Sigmoid: Not used in this thesis, but is an activation function commonly used to model logistic regression. The σ symbol is common to denote this function, and is defined mathematically as:

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{18}$$

It effectively "squashes" all values to the range between 0 and 1, and can be deployed to model probability values and logistic regression because of this. They are, however, rarely ever used in hidden layers, as other activation functions have empirically shown more promise. One of the reasons becomes apparent when studying their effects on input values that have a high absolute value. When performing backpropagation, high-valued inputs to a sigmoid function yields a significantly small gradient, causing the parameters to train very slowly. This phenomenon is commonly referred to as "saturation".

Tanh: Achieves a similar effect as the sigmoid activation function, squashing inputs into a range between -1 and 1. Mathematically, it is defined as such:

$$g(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{19}$$

Noteworthy, is that it also suffers from gradients being saturated when inputs have a high absolute value inputs. On the other hand, the expressivity of the tanh function is doubled in that the values are now mapped to a range between -1 and 1, and it is centered around 0. In regards to this thesis, the model named LeNet5-var uses tanh() as activation functions.[3]

ReLU (Rectified Linear Unit): One of the more commonly activation functions used for deep learning today. It can be argued to follow the principle of Occam's razor, namely *simpler is often better*, and is mathematically defined as follows:

$$g(x) = \text{relu}(x) = \max(0, x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases} \tag{20}$$

For completeness, its derivative at $g'(x = 0)$ is defined to be 0, as it mathematically is non-differentiable at $x = 0$. Although simple to define and compute, it is widely used in many deep learning architectures that achieve state-of-the-art performances. In regards to this thesis, the models named MobileNetV3-var and ResNet18-var use relu() as activation functions.[4]

A noteable caveat to this function, is that it is subject to the "dying-ReLU"-phenomenon. The phenomenon is encountered when many of the network nodes are negative prior to activation by

[3]LeNet5 originates from [25], and LeNet5-var is the thesis' own version of this network. Section 4.4.1 covers the architecture of this network.

[4]MobileNetV3 and ResNet18 originate from the works of Howard et al. [17], and the works of He et al. [14]. MobileNetV3-var and ResNet18-var is the thesis' own versions of these networks, and subsection 4.4.1 covers the architectures of these networks.

the ReLU activation function. Too many neurons being effectively zero during training might prevent the model from learning complex relationships. To remedy this, there are alterations of the ReLU function (see [27]), which is out of the scope of this thesis.

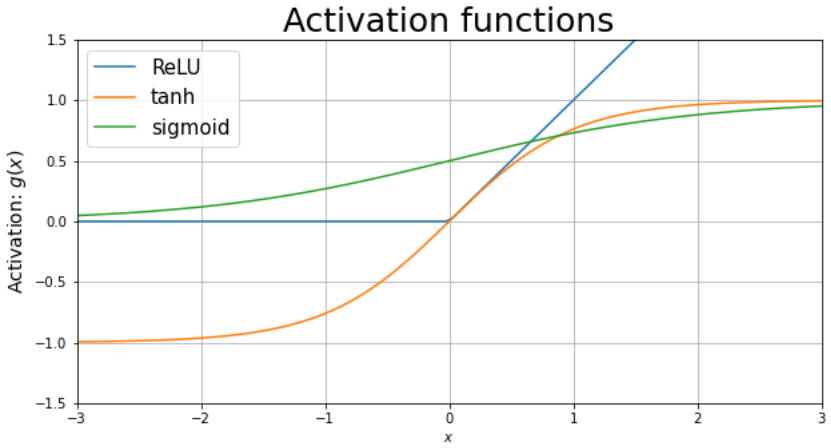


Figure 7: The activation functions as elaborated in this section. The sigmoid function is displayed in green, tanh is in orange, and ReLU is in blue.

3.3 Learning from data

The thesis includes an extensive summary of linear regression in this section as it is a well-known concept, and describes the idea of loss functions in a short and intuitive sense.

Accompanying most optimization problems is a sensible way of measuring how well the model is performing. Classical examples of optimization can be found in the problem of **curve-fitting**, where the goal is to find a suitable function that mimics the trend of a set of data points. Seasoned mathematicians will also recall from their early careers the problem of *least-squares fitting*, where one aims to minimize the sum of squared residuals. This will be used as a running example to motivate the idea of inherently learning model parameters from data the model is presented with:

Least-squares fitting

To state in mathematical terms, least-squares fitting attempts to optimize a linear combination of input variables such that their sum of squared residuals is as low as possible in a given function basis ϕ. A simple basis is the linear basis, where each dimension x_i is orthogonal to every other dimension x_j. In this running example of least-squares fitting, i denotes the $i-$th dimension in which the input point $\boldsymbol{x}$ lives, and M denotes the number of dimensions considered in making a regression model:

$$\hat{y}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}\boldsymbol{x}^T + b = w_1 x_1 + \ldots + w_M x_M + b = \sum_{i=1}^{M} w_i x_i + b \tag{21}$$

Linear basis functions are limited in terms of their complexity, deeming them unfit for many applications. On the other hand, by extending the idea of representing an output with a linear

combination of $\boldsymbol{x}$ in other non-linear bases, a much more general family of *regression models* spawns:

$$\hat{y}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}\boldsymbol{\phi}^T(\boldsymbol{x}) + b = w_1\phi_1(\boldsymbol{x}) + ... + w_M\phi_M(\boldsymbol{x}) + b = \sum_{i=1}^{M} w_i\phi_i(\boldsymbol{x}) + b \tag{22}$$

Note 3.2. *The term b is often defined by extending the input $\boldsymbol{x}^T = [1, x_1, x_2, ..., x_D]$ and the weights $\boldsymbol{w}^T = [w_0, w_1, w_2, ..., w_M]$ by a dummy input. In the case of basis functions, a dummy function $\phi_0(\boldsymbol{x}) = 1$ is added.*

Determining the appropriate basis functions can sometimes be trivial, but other times painstakingly difficult. For explanatory purposes, such nuances will be left out in favor of examples that are simple to understand. Illustrating the need for such basis functions, linear (23), and monomial (24) basis functions are illustrated:

$$\{\phi_i(\boldsymbol{x})\}_{i=1}^{M} = \{\boldsymbol{x}, \boldsymbol{x}, \boldsymbol{x}, ..., \boldsymbol{x}\} \tag{23}$$

$$\{\phi_i(\boldsymbol{x})\}_{i=1}^{M} = \{\boldsymbol{x}, \boldsymbol{x}^2, \boldsymbol{x}^3, ..., \boldsymbol{x}^M\} \tag{24}$$

The choice of which basis function to use, and how many basis functions M to use is referred to as **hyperparameters**, and is generally required by the user to set as optimally as possible[5]. Figure 8 illustrates these basis functions in action, attempting to fit a regression model from a set of data points:

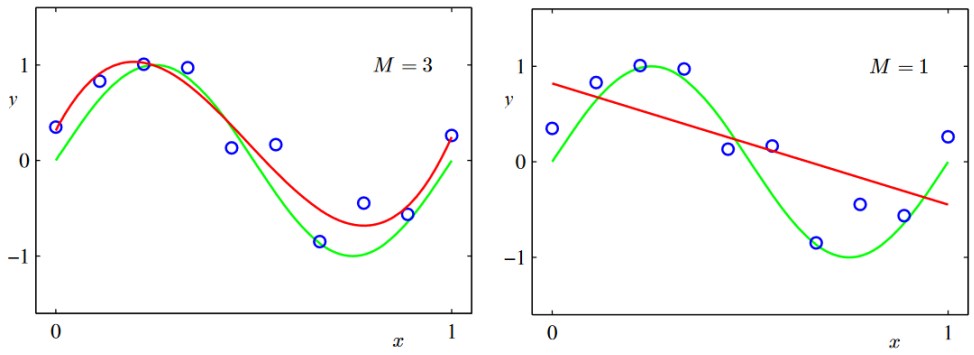


Figure 8: The green curve is the true curve. The blue circular points t_n are drawn from the green curve, but random Gaussian noise has been added to distance them a bit from the green curve. The red curve shows example regression models, where M corresponds to the maximum number of powers of the exponential basis functions. The left figure illustrates a regression model utilizing monomial basis functions, and the right figure illustrates a regression model utilizing the linear basis functions. Figure and plots are adapted from [5].

Arriving at these regression models, the models have been through a "training step". Specifically, the weights $\boldsymbol{w}$ from Equation (22) have been determined by some methodology. This methodology of acquiring the appropriate weights is the backbone to *any* model optimization problem. In determining a model's weights, a robust *metric* must first be defined to evaluate a model's goodness-of-fit. A classic metric in regards to curve-fitting is a curve's *mean-squared-error*:

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}\left[\hat{y}(\boldsymbol{x}_n, \boldsymbol{w}) - y_n\right]^2 \tag{25}$$

[5]There are ways to automate the search for optimal hyperparameters, which is out of the scope of this thesis.

The term inside the summation term, namely $[\hat{y}(\boldsymbol{x}_n, \boldsymbol{w}) - y]^2$, is denoted by the function $\mathcal{L}_n(\boldsymbol{w})$, and is defined to be a **loss function** for the training point denoted as n. Furthermore, the average sum of all the losses is one of many ways to evaluate the total loss of a model, leaving the average sum to be a function itself. This thesis refers to this as the **cost function**. In model training, the loss function $\mathcal{L}$ is important, because it defines what objective the model should aim to train for. The cost function $\mathcal{C}$ is more responsible for appropriately scaling the loss, where the cost function of averaging the loss is interpretable for many different problems. Equation (25) can simplified followingly:

$$\text{MSE} = \mathcal{C}_{\text{avg}}(\mathcal{L}_n(\boldsymbol{w})) \tag{26}$$

Note 3.3. *The literature tends to disagree on how to properly define a cost function. Some literatures define the cost function as the metric itself (such as Equation (25)), whilst others refer to the cost function as a measure of a model's loss on a group of data points. This thesis operates with the latter, in that a metric is defined by the cost of losses. This has the benefit of separating terms that are easy to interchange.*

Note 3.4. *In regards to this thesis, appropriate loss functions in the context of siamese neural networks are elaborated in Chapter 4.*

Motivated by measuring how far off true points y_n are from the model curve $\hat{y} = \hat{y}(\boldsymbol{x}, \boldsymbol{w})$, the MSE value is high for points far away from the model curve, and low for points near the model curve. Generally, a lower MSE tends to describe a better model, and will be the optimizing goal of this example.[6].

For the purpose of the example, assume the data to be linearly distributed. Modelling this linear data is then performed by employing the linear basis functions, and finding the lowest MSE-value. For ease of notation and computations, the problem is recast to finding the lowest *sum-of-squared-errors* (SSE), i.e. the cost function $\mathcal{C}$ is set to be $\mathcal{C}_{\text{sum}}$:

$$\text{SSE} = \mathcal{C}_{\text{sum}}(\mathcal{L}_n(\boldsymbol{w})) = \sum_{n=1}^{N} [\hat{y}(\boldsymbol{x}_n, \boldsymbol{w}) - y_n]^2 \tag{27}$$

Followingly, such a linear model can be written in vector notation:

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} \tag{28}$$

Note 3.5. *This system of equations is generally underdetermined, meaning one cannot simply multiply by X^{-1} on both sides to yield a closed form solution to $\boldsymbol{w}$.*

Determining the optimal weights $\hat{\boldsymbol{w}}$ is in the case of linear regression done by examining the vectorized SSE metric on the dataset:

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) \tag{29}$$

Differentiating the vectorized SSE with respect to $\boldsymbol{w}$ and setting it to be equal to 0, yields the lowest SSE and hence gives a closed form solution $\hat{\boldsymbol{w}}$:

$$\partial_w \left[(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) \right] = 0 \tag{30}$$

[6]Lower is **not** always better as the concept of overfitting describes, which is covered in section 3.7

$$\vdots$$

$$-2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = 0 \tag{31}$$

Rewriting this in terms of $\boldsymbol{w}$ followingly yields $\hat{\boldsymbol{w}}$:

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{32}$$

Note 3.6. *The matrix multiplication* $\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T$ *is also known as the pseudoinverse of* $\boldsymbol{X}$*.*

For most machine learning problems, such closed form solutions do not exist. The remedy to such problems is often found in iterative methods, whereas optimization by gradient descent is widely used to compute parameters. The following sections elaborate on this procedure of determining the model's parameters given a training dataset.

Model selection pipeline

To inherently *learn from data* and decide on what the best hyperparameters are, the dataset responsible for determining the parameters $\hat{\boldsymbol{w}}$ must be split up in an appropriate manner.

First, a fraction of the dataset must be allocated for the purpose of "training" the model (i.e. determining some intermediate parameters $\hat{\boldsymbol{w}}$). This partition of the dataset is referred to as the *training dataset*, and its only purpose is to train the model.

Onwards, another part of the dataset is required to make decisions about the model's hyperparameters. This part of the dataset is known as the *validation dataset*, and it serves as a means to fine-tune the model's performance and select the optimal hyperparameters in order to maximize its generalizability.

Finally, once the model has been trained and validated, it needs to be evaluated on an independent dataset to assess its true performance and generalization ability. This independent set is called the *testing dataset*. The testing dataset provides an unbiased assessment of the model's effectiveness, as its data has not been used for any decision making of which model to employ.[7]

An illustration of the dataset partitioning pipeline is presented in Figure 9:

[7]It is very important to stress that the testing dataset is not to be used for other purposes than testing the performance of the model that performs the best on the validation dataset.

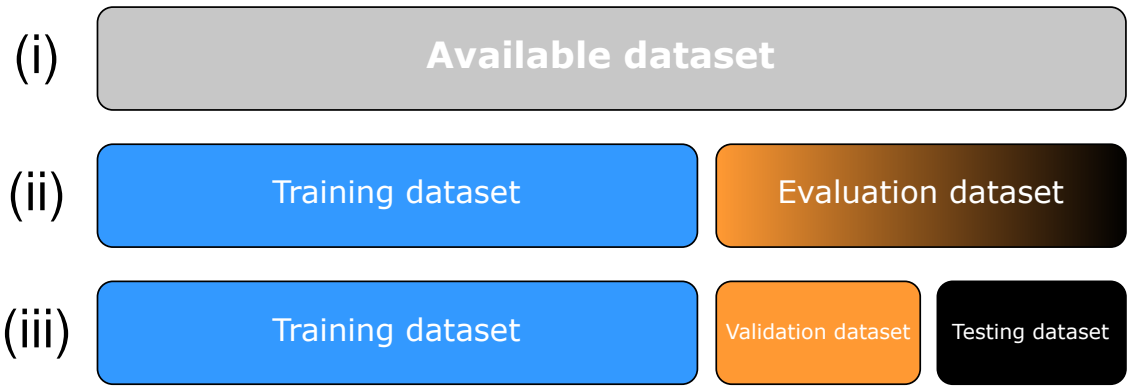


Figure 9: The gray box denoted as step **(i)** represents the entire dataset, without any form of partitioning. The boxes colored blue and a mixture of orange and black denoted as step **(ii)** splits the dataset into a part for training, and a part for evaluation. The resulting boxes from step **(iii)** further partitions the evaluation set into a validation and testing dataset.

The colorization convention of the dataset partitions in Figure 9 is carried on throughout the thesis, and operates as a means to associate a color to what role a data point plays in a machine learning pipeline.

3.4 Gradient descent

*As discussed in the previous section (**Learning from data** - section 3.3), measuring a model with some appropriate metric presents a way of evaluating the model's goodness-of-fit, as a function of its parameters* $\boldsymbol{w}$*. As most machine learning problems do not present closed form solutions, one must look to iterative methods to calculate* $\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{C}\left(\mathcal{L}_n(\boldsymbol{w})\right)$.

This section discusses the algorithm of gradient descent, along with some optimizers that are widely used in deep learning applications in order to search for $\hat{\boldsymbol{w}}$*. For extensive details regarding optimizers, the reader is referred to [47] (Chapter 12) and [12] (Chapter 8).*

The linear regression model from section 3.3 is kept as a running example throughout this section for readability purposes. Denoting the linear regression model to be a function $\mathcal{L}$ of its parameters $\boldsymbol{w}$, its gradient is as follows:

$$\nabla_{\boldsymbol{w}} \mathcal{C}(\mathcal{L}_n(\boldsymbol{w})) = \left[\frac{\partial \mathcal{C}}{\partial w_0}, \frac{\partial \mathcal{C}}{\partial w_1}, \ldots, \frac{\partial \mathcal{C}}{\partial w_M}\right]^T = \left[\frac{\partial \mathcal{C}}{\partial b}, \frac{\partial \mathcal{C}}{\partial w_1}, \ldots, \frac{\partial \mathcal{C}}{\partial w_M}\right]^T \tag{33}$$

The gradient has the property of pointing in the function's steepest direction, such that stepping in the direction of $-\nabla_{\boldsymbol{w}} \mathcal{C}(\mathcal{L}_n(\boldsymbol{w}))$ will result in traversing in the direction where the function $\mathcal{C}(\mathcal{L}_n(\boldsymbol{w}))$ decreases most rapidly. To illustrate this, linear regression by least squares fitting demonstrates how one can "descend" in the cost space by utilizing the gradient of the cost:

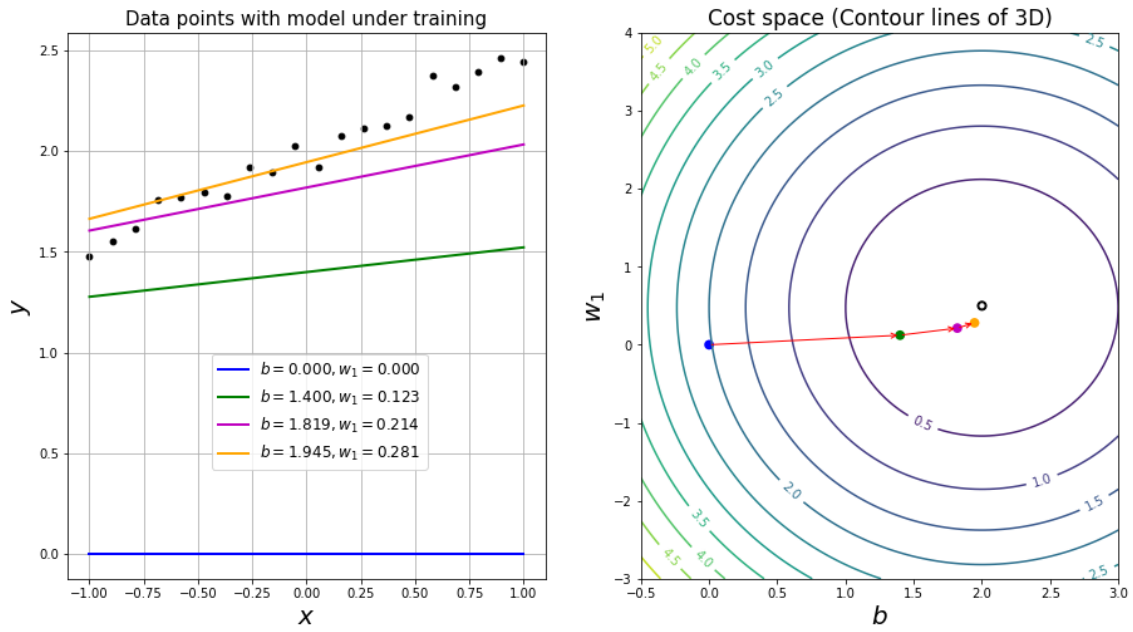


Figure 10: The plot to the left illustrates several color-annotated linear regression models with different options for the parameters b and w_1. They attempt to describe the distribution of black training points, and the colored lines depict different regression models with parameters b and w_1 being trained through stepping in the cost space according to the method of gradient descent. The plot to the right illustrates the same linear regression models, but as colored points in cost space. The two visible axes are in terms of the model parameters b and w_1, whilst the last axis (drawn as contour lines) is the cost of the model, namely $\mathcal{C}(\mathcal{L}_n(\boldsymbol{w})) = \mathcal{C}(\mathcal{L}_n(b, w_1))$. The red arrows attempt to show how the model updates according to $\nabla_{\boldsymbol{w}}\mathcal{C}(\mathcal{L}_n(\boldsymbol{w}))$, whilst the small black circle illustrates the lowest training cost.
Plots are courtesy of `https://scipython.com/blog/visualizing-the-gradient-descent-method/`.

Although the models in Figure 10 are simple, they illustrate the core idea of iteratively stepping in the cost space by moving in the direction of $-\nabla_{\boldsymbol{w}}\mathcal{C}(\mathcal{L}_n(\boldsymbol{w}))$. Mathematically, it can be described as such, where t denotes the $t-$th optimization step:

$$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} + \eta \cdot (-\nabla_{\boldsymbol{w}}\mathcal{C}(\mathcal{L}_n(\boldsymbol{w}_{t-1})) \tag{34}$$

This equation introduces the learning rate hyperparameter η, which effectively decides the size of the gradient steps. In regards to Figure 10, the red arrows on the plot to the right show the impact of a set value for η. Choosing inappropriate values for η can lead to overshooting (η is too large), which describes situations where the parameters fail to converge. There is also the risk of inefficient training, where the model trains too slow (η is too small).

To use the idea of gradient descent in real-life applications, utilizing the entire dataset for one gradient step is often inefficient, and typically consumes more memory than the hardware system can handle. To overcome this issue, one can instead *estimate* the cost space by feeding the cost function **mini-batches** (indexed by m_b), which are stochastically subsampled from the training dataset:

$$\mathcal{C}(\mathcal{L}_n(\boldsymbol{w})) \approx \frac{1}{M_b}\sum_{m_b=1}^{M_b}\mathcal{C}_{m_b}(\mathcal{L}(\boldsymbol{w})) \tag{35}$$

In words, the cost $\mathcal{C}$ of the entire training dataset is *approximated* by averaging a number of M_b mini-batch costs $\mathcal{C}_{m_b}$. The mini-batch costs $\mathcal{C}_{m_b}$ can be any cost function, but the average mini-batch cost is usually the default mini-batch cost function.

$$\mathcal{C}_{m_b-avg}(\mathcal{L}(\boldsymbol{w})) = \frac{1}{N_{m_b}} \sum_{n=1}^{N_{m_b}} \mathcal{L}_n(\boldsymbol{w}) \tag{36}$$

Note 3.7. *The number of training samples in a mini-batch is denoted by N_{m_b}, where the subscript m_b denotes the m_b−th mini-batch. The last mini-batch typically has less training samples than the other mini-batches, which occurs if the number of training data points in a mini-batch is not divisible by the training data point cardinality.*

Performing gradient descent on the cost $\mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}))$ derived from mini-batches instead serves to increase computational efficiency, whilst retaining an approximate distribution of the training dataset. This method is referred to as mini-batch stochastic gradient descent (SGD), and can be stated algorithmically as follows:

Algorithm 1 Mini-batch stochastic gradient descent

Require: Learning rate η, initialized values $\boldsymbol{w}_0$
Require: Training dataset $\boldsymbol{X}, \boldsymbol{Y}$
  while *stopping criterion not met* **do**
    Sample a mini-batch m_b from $\boldsymbol{X}, \boldsymbol{Y}$
    Compute the estimated gradient $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}))$
    Update the parameters $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \boldsymbol{g}$
  end while

It is important to emphasize that such an optimizing routine does not guarantee the localization of the global minimum in the cost space, nor a local minimum if η is not properly set. To add insult to injury, a promising local minimum in the cost space might model the training data very well, but perform poorly on unseen data[8].

Moreover, the mini-batch SGD algorithm as it appears in Algorithm 1 tends to be slow to learn due to factors such as variance in the stochastic gradient, and an ill-conditioned Hessian matrix of the cost space[9]. In remedying this, [36] suggests accumulating past gradients by an exponentially decaying moving average (EDMA), and performing gradient descent by taking this EDMA into account. The parameter update from Algorithm 1 is altered as follows:

$$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \boldsymbol{v}_t \tag{37}$$

where $\boldsymbol{v}_t$ is iteratively computed ($v_0 \equiv 0$) by EDMA:

$$\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + (1 - \beta_1) \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}_{t-1})) \tag{38}$$

[8]This is generally referred to as overfitting, and is discussed in section 3.7.

[9]The Hessian matrix refers to the matrix that describes a function's double derivatives. An ill-conditioned Hessian will reveal the curvature of the cost space to be highly sensitive to changes in its variables.

Remark 3.1. *Equation (38) is usually abbreviated by getting rid of the* $(1-\beta_1)$ *term:*

$$\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}_{t-1})) \tag{39}$$

Note 3.8. $\beta_1 \in (0,1)$ *is a momentum hyperparameter that controls how much preceding gradients should be weighted. As a rule of thumb,* $\beta_1 = 0.9$ *approximately corresponds to the last* 10 *gradients being of highest relevance. Onwards,* $\beta_1 = 0.99$ *approximately corresponds to the last* 100 *gradients being of highest relevance, and is motivated by the limit of the infinite sum of the momentum hyperparameters* $\sum_{t=1}^{\infty} \beta_1^t = \frac{1}{1-\beta_1}$.

Such an adjustment to the gradient descent helps to speed up training, because gradients that are consistently aligned through multiple iterations get accelerated. This introduces the need to define the momentum SGD algorithm:

Algorithm 2 Mini-batch stochastic gradient descent with momentum

Require: Learning rate η, momentum hyperparameter β_1
Require: Initialized values $\boldsymbol{w}_0$
Require: Training dataset $\boldsymbol{X}, \boldsymbol{Y}$
  while *stopping criterion not met* **do**
    Sample a mini-batch m_b from $\boldsymbol{X}, \boldsymbol{Y}$
    Compute the estimated gradient $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}))$
    Update the momentum term $\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + \boldsymbol{g}$
    Update the parameters $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \boldsymbol{v}_t$
  end while

Remark 3.2. *Some frameworks operate by placing the learning rate* η *in front of the estimated gradient* $\nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}))$ *in Equation (39), instead of having it in front of* $\boldsymbol{v}_t$ *in Equation (37). PyTorch is the framework used in this thesis, which operates with the latter momentum definition*[10].

Another way of letting past gradients influence the next gradient step, is by RMSProp [15]. Unlike Algorithm 2, the estimated gradients are now squared, and the learning rate is adaptively changed:

$$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \frac{\eta}{\sqrt{\boldsymbol{s}_t + \varepsilon}} \odot \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}_{t-1})) \tag{40}$$

where $\boldsymbol{s}_t$ is iteratively computed $(s_0 \equiv 0)$ by EDMA:

$$\boldsymbol{s}_t \leftarrow \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2) \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}_{t-1}))^2 \tag{41}$$

Note 3.9. *The symbol* $\odot$ *is used to denote elementwise multiplication of matrices. Furthermore, the hyperparameter* β_2 *is different from* β_1*, although it also contributes in assessing the past gradients by EDMA.*

[10] More details can be found in the PyTorch documentation at `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html`

By Equations (40) and (41), the RMSProp optimizing algorithm can be defined as:

Algorithm 3 Mini-batch stochastic gradient descent by RMSProp

Require: Learning rate η, smooting hyperparameter β_2, small constant $\varepsilon = 10^{-8}$
Require: Initialized values $\boldsymbol{w}_0$
Require: Training dataset $\boldsymbol{X}, \boldsymbol{Y}$
  while *stopping criterion not met* **do**
    Sample a mini-batch m_b from $\boldsymbol{X}, \boldsymbol{Y}$
    Compute the estimated gradient $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}))$
    Update the smoothing term $\boldsymbol{s}_t \leftarrow \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Update the parameters $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \frac{\eta}{\sqrt{\boldsymbol{s}_t + \varepsilon}}\boldsymbol{g}$
  end while

Note 3.10. *The term ε is introduced in order to control numerical instability through division by significantly small numbers.*

The terms $\boldsymbol{v}_t$ and $\boldsymbol{s}_t$ have a tendency to be "slow to start", and is attributed to their initialization. To remedy this, they are "bias-corrected":

$$\tilde{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-{\beta_1}^t} \tag{42}$$

$$\tilde{\boldsymbol{s}}_t = \frac{\boldsymbol{s}_t}{1-{\beta_2}^t} \tag{43}$$

By combining the ideas behind the gradient descent alterations presented in Algorithm 2 (momentum), Algorithm 3 (RMSProp), and Equations (42) and (43) for bias correction, comes the ADAM (ADAptive Moment) algorithm [22]:

Algorithm 4 Mini-batch stochastic gradient descent by ADAM

Require: Learning rate η, small constant $\varepsilon = 10^{-8}$
Require: Momentum hyperparameter β_1, smoothing hyperparameter β_2
Require: Initialized values $\boldsymbol{w}_0$
Require: Training dataset $\boldsymbol{X}, \boldsymbol{Y}$
  while *stopping criterion not met* **do**
    Sample a mini-batch m_b from $\boldsymbol{X}, \boldsymbol{Y}$
    Compute the estimated gradient $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{w}} \mathcal{C}_{m_b}(\mathcal{L}_n(\boldsymbol{w}))$
    Update the momentum term $\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + (1-\beta_1)\boldsymbol{g}$
    Bias-correct the momentum term: $\tilde{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-{\beta_1}^t}$
    Update the smoothing term $\boldsymbol{s}_t \leftarrow \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Bias-correct the smoothing term: $\tilde{\boldsymbol{s}}_t \leftarrow \frac{\boldsymbol{s}_t}{1-{\beta_2}^t}$
    Update the parameters $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \frac{\eta}{\sqrt{\tilde{\boldsymbol{s}}_t + \varepsilon}}\tilde{\boldsymbol{v}}_t$
  end while

Note 3.11. *The momentum term in ADAM keeps the $(1-\beta_1)$ term, as opposed to Algorithm 2. Furthermore, the parameter update is now performed with $\tilde{\boldsymbol{v}}_t$ instead of the estimated gradient $\boldsymbol{g}$ as Algorithm 3 does it.*

3.5 Backpropagation

This section requires familiarity with all of the preceding sections. Furthermore, the notation provided assumes the network to be an MLP, as described in the previous sections.

Generally, backpropagation refers to the algorithm of determining the gradient $\nabla_{\boldsymbol{w}}\mathcal{C}(\mathcal{L}_n(\boldsymbol{w}))$ that is to be used for gradient descent (see Equation (33) for its definition). Calculating this gradient is essentially performed by the chain rule of differentiation[11], and is done for every possible parameter $\boldsymbol{w}$. For simplicity of notation, a *local gradient* $\delta_i^{[l]}$ is defined, which corresponds to a node's current chain-rule "progress". Conceptually, the local gradient can be understood as "every step of differentiation required to reach the final differentiation of a specific parameter". Multiplying this local gradient by the last derivative leading to the weight differential $\partial w_{i,j}^{[l]}$ and bias differential $\partial b_i^{[l]}$ makes the local gradient a powerful notation tool.

To motivate this, the forward propagation described in Equations (3) and (4) are extended by index notation. Consider the $i-$th node in layer l being a weighted sum of the previous activations $\boldsymbol{a}^{[l-1]}$ indexed by j:

$$z_i^{[l]} = b_i^{[l]} + \sum_j w_{i,j}^{[l]} a_j^{[l-1]} \tag{44}$$

Differentiating $z_i^{[l]}$ by its parameters would yield the final differentiations to calculate the gradient:

$$\frac{\partial z_i^{[l]}}{\partial w_{i,j}^{[l]}} = a_j^{[l-1]} \tag{45}$$

$$\frac{\partial z_i^{[l]}}{\partial b_i^{[l]}} = \mathbf{1} \tag{46}$$

Reaching this final differentation from $\nabla_{\boldsymbol{w}}\mathcal{C}$ is denoted by the local gradient $\delta_i^{[l]}$:

$$\frac{\partial \mathcal{C}}{\partial w_{i,j}^{[l]}} = \delta_i^{[l]} \times \frac{\partial z_i^{[l]}}{\partial w_{i,j}^{[l]}} = \delta_i^{[l]} \times a_j^{[l-1]} \tag{47}$$

For the biases, it is simply:

$$\frac{\partial \mathcal{C}}{\partial b_i^{[l]}} = \delta_i^{[l]} \times \frac{\partial z_i^{[l]}}{\partial b_i^{[l]}} = \delta_i^{[l]} \tag{48}$$

For the general notation of this local gradient in an MLP, there are *two* cases to consider: The **output layer**, and the **hidden layers**.

Hidden layers: The *local gradients* for the hidden layers are:

$$\delta_i^{[l]} = \frac{\partial \mathcal{C}}{\partial a_i^{[l]}} \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} = \left(\sum_{k=1}^{m^{[l+1]}} \frac{\partial \mathcal{C}}{\partial a_k^{[l+1]}} \frac{\partial a_k^{[l+1]}}{\partial a_i^{[l]}} \right) \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}}, \qquad l \in [1, \dots, L-1] \tag{49}$$

[11]The chain rule states: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$

Including the derivative of the activation function $g^{[l]}()$ in the $l-$th layer, the local gradient can be simplified as:

$$\delta_i^{[l]} = \sum_{k=1}^{m^{[l+1]}} \delta_k^{[l+1]} w_{k,i}^{[l+1]} \times g_i'^{[l]}(z_i^{[l]}), \qquad l \in [1, \ldots, L-1] \tag{50}$$

Output layer: This very special case is dependent on what is chosen to be the loss function $\mathcal{L}$ for the parameters. The local gradient for the output layer L is as follows:

$$\delta_i^{[L]} = \frac{\partial \mathcal{C}}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial z_i^{[L]}} \tag{51}$$

For simplicity, say just a single training example is considered, leaving $\mathcal{C}(\mathcal{L}_n(\boldsymbol{w}))$ to be just $\mathcal{L}(\boldsymbol{w})$. The term $\frac{\partial \mathcal{C}}{\partial \hat{y}_i}$ would then be nothing more than the derivative of the loss function, with respect to the predicted point $\hat{y}_i$:

$$\delta_i^{[L]} = \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial z_i^{[L]}} \tag{52}$$

If the loss function is for example the squared distance loss:

$$\mathcal{L}(\boldsymbol{w}) = \left(\hat{y}(\boldsymbol{x}, \boldsymbol{w}) - y\right)^2 \tag{53}$$

Then its derivative is simply:

$$\frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \hat{y}_i} = 2 \cdot \left(\hat{y}(\boldsymbol{x}, \boldsymbol{w}) - y\right) \tag{54}$$

Note that backpropagation as presented here, is for the example of an MLP architecture. For more details regarding backpropagation through convolutional neural networks (covered in section 3.6 in this thesis), the reader is referred to the *Digital Image Processing* book [11].

3.6 Convolutional neural networks

When considering using machine learning for any particular task, one is inclined to consider what sort of data is to be worked on. In the field of computer vision, ***convolutional neural networks*** *(CNNs) have been proven hugely successful in solving problems such as object localization, detection, recognition and segmentation and others. They make use of the fact that a neighborhood of pixel values contains information that make up specific objects and patterns.*

A prime example of this would be a human's face, in which the pixel values and their spatial distribution work together to define what person is in the image, and if there is a person at all in an image. One could hope that the performances of CNNs extend fairly seamlessly in the problem of salmon lice images, seeing as the problem of recognizing salmon lice is generally a tough task for even humans.

This section introduces the mathematical operation of convolution, and explores how this operation is an integral part of CNNs. Furthermore, the concepts of pooling and stride are covered, as well as cylindrical convolution.

3.6.1 Convolution/cross-correlation

Since the thesis deals with grayscale images in the form of pixel arrays, the concept of convolution is introduced with a 2D example.

2D convolution kernel

The convolution of a 2D input $a_{i,j}$ around the point (i, j) is given by:

$$z_{i,j} = \sum_m \sum_n w_{m,n} a_{i-m,j-n} \tag{55}$$

The subscripts m, n denote the indices of the weight matrix, and the kernels w are typically set to be odd-sized in order to have a well-defined center of the kernel. $*$ is used to denote the convolution operation as such:

$$z_{i,j} = w * a_{i,j} = \sum_m \sum_n w_{m,n} a_{i-m,j-n} \tag{56}$$

A noteworthy deviation from the literature that deserves more than a footnote, is that most machine learning libraries in fact implement the cross-correlation operation, denoted by $\star$:

$$z_{i,j} = w \star a_{i,j} = \sum_m \sum_n w_{m,n} a_{i+m,j+n} \tag{57}$$

Cross-correlation and convolution both perform weighted summation of a pixel neighborhood, but the difference is that convolution effectively *rotates* the kernel prior to the summation. The take-away from mentioning this deviation, is that although the operations have small differences, the machine learning literature has adopted the term *convolution* to refer to the weighted summation of an input tensor. As such, this thesis operates with the same convention as the literature, meaning cross-correlation is renamed as convolution.

In aiding the mathematical description of convolution, a comprehensive and descriptive illustration of how convolution is shown in Figures 11 and 12:

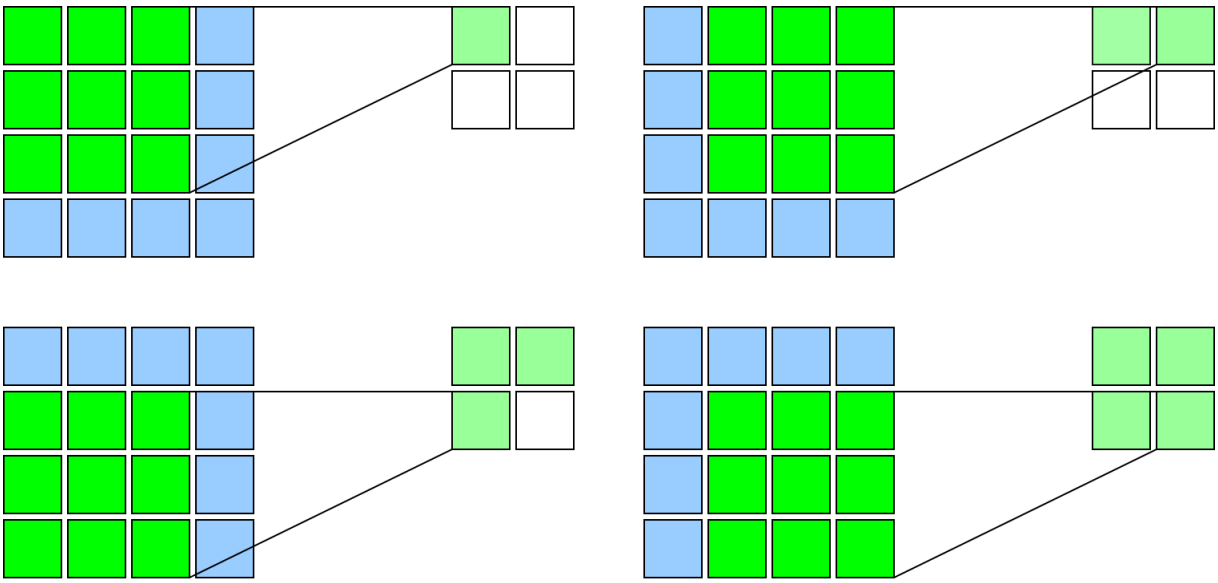


Figure 11: An illustration of the operation of convolution. The kernel in sharp green shifts over the blue input features, where the output features of the convolution operations are depicted in soft green. Note that these output features are called the receptive field of the blue input features, as information about the blue spatial neighborhood is downscaled into the soft green spatial neighborhood.

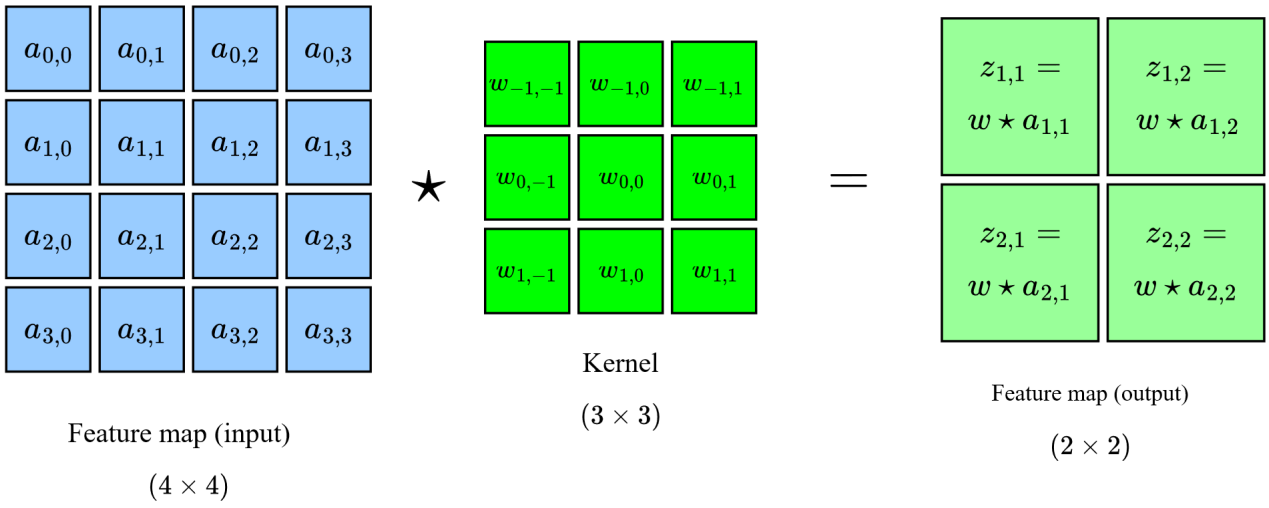


Figure 12: This figure illustrates the output of convolution (described by Equation 57). Notice that the indices of the output values $z_{i,j}$ do not coincide with the indices of the input values $a_{i,j}$.

With the convolution operation covered, one can build a neural network that utilizes kernel matrices w as parameters of the network. It is important to note that there must added a bias-term b for every kernel, and that this is left out only for simplicity of explanation.

Analogous with the MLP from section 3.1, there are layers, weights and biases to consider. Consider an input image x of dimensions: $n_W^{[l]} \times n_H^{[l]} \times n_C^{[l]}$, and that this image is convolved with a number of $n_C^{[l+1]}$ distinct kernels. The first hidden layer of the network can be described as such:

$$z_{c,i,j}^{[1]} = w^{[1]} * a_{i,j}^{[0]} + b^{[1]} \tag{58}$$

Following the same conventions as in section 3.1, the 0−th activation $a^{[0]}$ is defined as being the input x. Before moving on too far ahead, there is a need to elaborate 3D convolutions, and assign the proper notation.

3D convolution kernel

Stated before Equation 58, a number of $n_C^{[l+1]}$ convolutions is performed, and the result is a number of $n_C^{[l+1]}$ *feature maps* with dimensions $n_W^{[l+1]} \times n_H^{[l+1]}$. Here is the important part:

When building such a network even deeper, full connectivity can be enforced by using 3D-kernels that are $n_C^{[l+1]}$ deep. And when traversing to the next layer, a number of $n_C^{[l+2]}$ kernels that have a depth of $n_C^{[l+1]}$ is needed. The notation for 3D kernels is a straightforward extension of 2D kernels:

$$z_{c,i,j}^{[l]} = \sum_d \sum_m \sum_n w_{c,d,m,n}^{[l]} a_{d,i+m,j+n}^{[l-1]} + b_c^{[l]} \tag{59}$$

The subscript c denotes the c^{th} kernel $(1, \ldots, c, \ldots n_C^{[l]})$ and the subscript d denotes the input channel $(1, \ldots, d, \ldots, D^{[l]})$. As before, the reader is reminded that the superscript notation of the subscript indices to allocate the layer in Equation (59) is left out of the equation purely for aesthetic purposes.

An arbitrary convolution layer of a CNN

To nicely wrap up these concepts, and their contributions in CNNs, a simple demonstration of a forward propagation is performed in a representative convolution layer.

Suppose we are in the middle of a CNN, and the feature volume has dimensions $(n_W^{[l]}, n_H^{[l]}, n_C^{[l]}) = (32 \times 32 \times 64)$. Further, suppose the convolution layer has hyperparameters $\{f^{[l]}, n_C^{[l+1]}, s^{[l]}, p^{[l]}\} = \{5, 128, 1, 1\}$. Applying the formulas for 2D convolution output:

$$n_W^{[l+1]} = \left\lfloor \frac{n_W^{[l]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \tag{60}$$

$$n_H^{[l+1]} = \left\lfloor \frac{n_H^{[l]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \tag{61}$$

yields an output feature volume of dimensions $(30 \times 30 \times 128)$. Conceptually, this new feature volume contains a number of 128 2-D feature maps, attempting to describe something meaningful about the previous feature volume.[12] Figure 13 depicts a representative illustration of the operations of a convolutional layer:

[12]What the feature maps represent is more often than not non-trivial, but there are some areas of deep learning research that attempt to dive into their meaning, namely explainable AI.

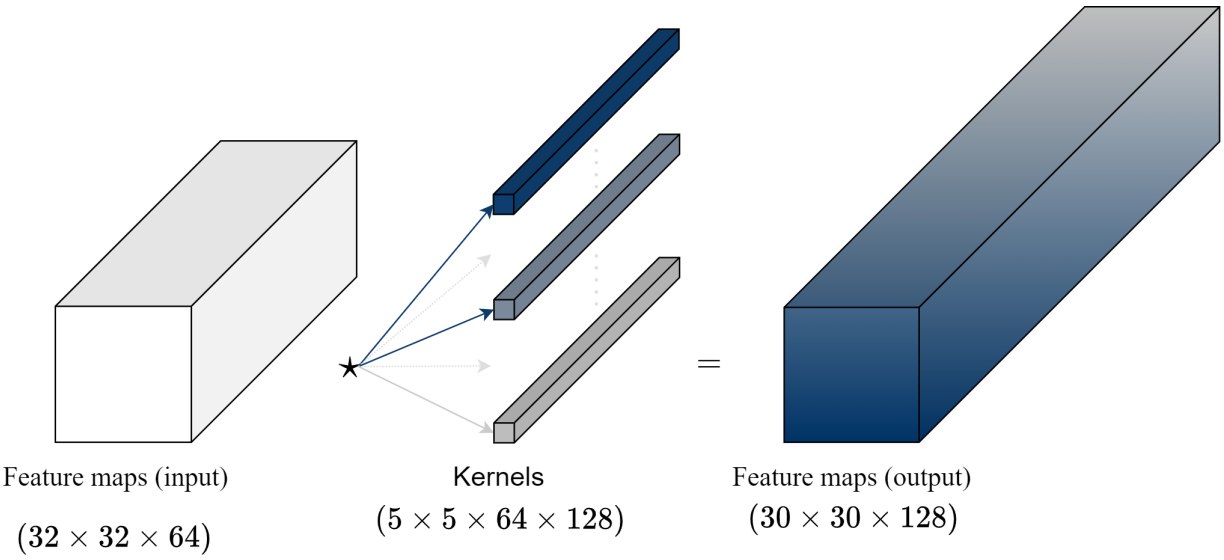


Figure 13: This figure illustrates a volume of feature map inputs with dimensions $(32 \times 32 \times 64)$ convolved with a set of 128 3D-kernels. Notice that the 3D-kernels along with the output feature volume are gradient-colored. This is an attempt to visualize that convolving the input feature volume with for example the first 3D-kernel (royal blue kernel), results in a 2D output (royal blue feature map in the output). The output convolutions are then stacked, and this results in a volume of feature map outputs. Note that the bias-vector is omitted here in favor of simplicity, as well as wrapping the output feature volume inside an activation function.

Following this new convolution layer, is non-linearization by activation functions as described in section 3.2. Depending on the architecture being used, further downsampling by pooling is a common approach to handle input data of high dimensions. This will be elaborated in the following subsection.

3.6.2 Convolution hyperparameters

Upping the sophistication of neural networks tends to introduce a larger variety of hyperparameters, and convolutional neural networks is no stranger to this. This subsection will cover the kernel size $f^{[l]}$, number of kernels $n_C^{[l]}$, stride $s^{[l]}$, padding $p^{[l]}$ and groups $g^{[l]}$, with the hyperparameter of number of layers L being trivial. Left out of the subsection and the thesis is the hyperparameter of dilation, as it is not a common hyperparameter to tune.

Kernel size $f^{[l]}$: This hyperparameter refers to the dimensions of the convolution kernels. A common design choice is to let kernels be quadratic in size, and this thesis is no exception to this. Common sizes of kernels range from $(3, 3)$ up to $(5, 5)$, but larger kernels tend to be used in earlier layers for the purposes of the convolutional neural network covering large receptive fields. Another common feature among kernel sizes, is that they are odd in dimensions, in order to have a well-defined center.

Number of kernels $n_C^{[l]}$: A CNN's depth is generally attributed to its amount of layers L, and how many kernels one chooses to convolve a layer's feature volume with. As a rule of thumb in building a CNN, $n_C^{[l]}$ tends to increase as the dimensions $n_W^{[l]}$ and $n_H^{[l]}$ gets lower. Worth noting is a computational trick involving convolving feature volumes with a kernel size of $(1, 1)$ in an

attempt to bottleneck the number of channels $n_C^{[l+1]}$ in the next layer. This keeps the dimensions $(n_W^{[l+1]}, n_H^{[l+1]})$ of the features, whilst reducing the depth of the feature volume $(n_C^{[l+1]} < n_C^{[l]})$.

Stride $s^{[l]}$: A way of reducing the height and width $(n_W^{[l]}, n_H^{[l]})$ of feature volumes is achieved by introducing convolution striding. When a convolution kernel convolves over an input image, it does so by calculating the weighted sum of the appropriate area around a point (i, j) as Equation (55) describes nicely. What it does not describe, is where the next point of convolution should be, which is what the stride $s^{[l]}$ aims to describe. This hyperparameter is usually set to be different from 1 in earlier layers as a way of downsizing the height and width $(n_W^{[l]}, n_H^{[l]})$ of feature volumes, and set to be 1 later in the architecture.

Padding $p^{[l]}$: When convolving an input image, it becomes clear that pixels not found on the boundaries are more involved than pixels on the boundary of the image. Attempting to resolve this, one can introduce padding to the image, as a way of making boundary pixels more represented in the output. Essentially pixels are added to the boundaries, making the input height and width $(n_H^{[l]} + 2p^{[l]}, n_W^{[l]} + 2p^{[l]})$ bigger by $2p^{[l]}$ pixels. Conventionally, pixels added to boundaries in this way are set to be 0.

Groups $g^{[l]}$: The hyperparameter of groups refers to dividing the input and output feature maps into separate subgroups. Instead of having every single feature map from the input be connected to every single feature map in the output as Figure 13 illustrates, one can let a subset of feature maps in the input be connected to a subset of feature maps in the output. For example, having $g^{[l]}$ be for example the same as $n_C^{[l]}$, means that each group consists of one input feature map connected to one output feature map. Conversely, setting $g^{[l]}$ to 1 implies standard 3D convolution with full connectivity, where all input feature maps are connected to all output feature maps. Regardless, $n_C^{[l-1]}$ and $n_C^{[l]}$ needs to be divisible by $g^{[l]}$ in order to obtain a valid number of groups. For the purposes of this thesis, this hyperparameter is covered as MobileNetV3 utilizes group convolutions. Its model architecture will be covered in section 4.4.

Note 3.12. *Conventionally, kernel size $f^{[l]}$, stride $s^{[l]}$ and padding $p^{[l]}$ are set to be quadratic. This means that the thesis makes no notational distinction between the width and height of these hyperparameters, and that e.g. $f^{[l]}$ refers to kernels of dimensions $(f^{[l]}, f^{[l]})$.*

3.6.3 Pooling

As images can grow quite large in terms of what potential information they might hold, a need for downsampling procedures is inevitable. An efficient and traditionally a parameter-less option is by forcibly downsampling the image by pooling. This has desirable properties such as controlling a model's ability to overfit, and to some extent, provide robustness towards some warping of the input. For the purposes of this thesis, max pooling and average pooling are elaborated in this subsection.

Max pooling

This sort of pooling can be explained as the principle of "winner-takes-it-all". Similar to convolutional layers, a filter window of size f is shifted over the input image with a stride s. The filter window resultingly selects the largest value, before striding on to the next filter region. Important to note is also that filter windows can coincide with each other, which happens when $s < f$. A simple illustration is depicted in Figure 14:

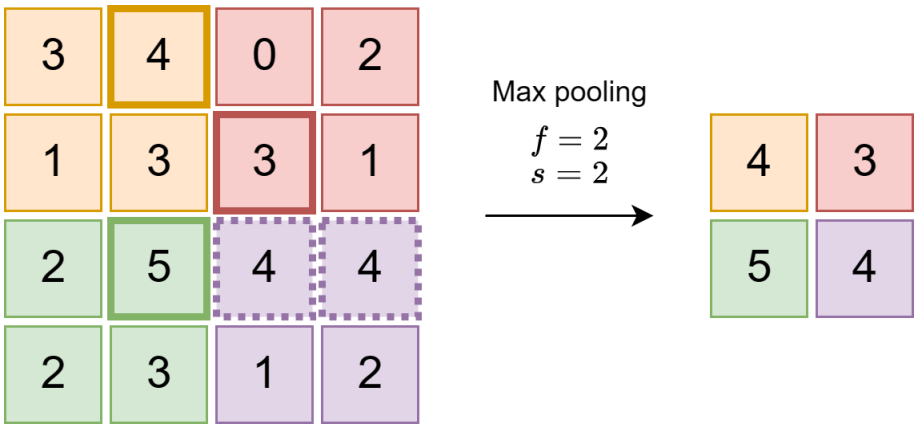


Figure 14: Max pooling downsamples the input by choosing the maximal value inside a filter window of size $f = 2$, and the stride $s = 2$ determines the next filter window. Pay attention to the ambiguous values in the lower right corner, denoted by dotted peripheral lines. Although their spatial locations differ, the max-pooling procedure makes no distinctions as to which pixel got selected - the value is all that matters.

As made apparent by Figure 14, the spatial orientation of pixels is not necessarily kept and effectivly warps the input. Additionally, possibly important pixels are cut to give way for the brightest pixel inside a filter window.

Average pooling

In an attempt to preserve information when performing pooling, an option is to average all the values inside the filter windows. Instead of a "winner-takes-it-all" strategy, each value inside a filter window is now weighted according to the pixel neighborhood size.

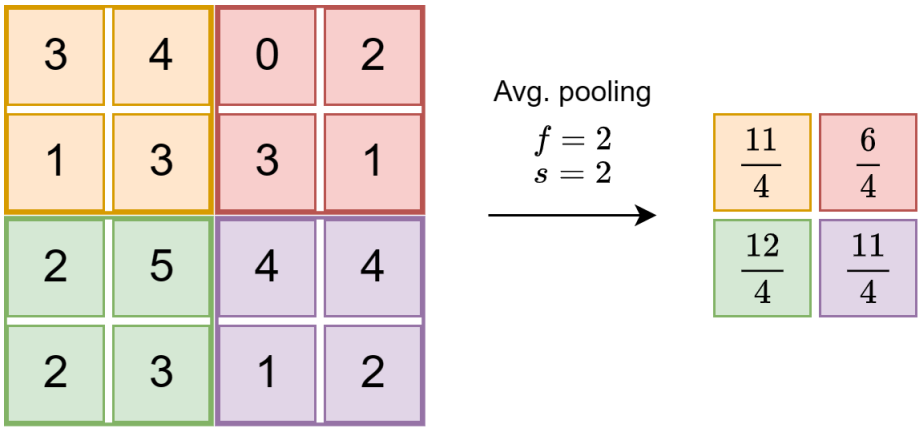


Figure 15: Average pooling downsamples the input by averaging the values inside a filter window of size $f = 2$, and the stride $s = 2$ determines the next filter window.

However, a key property of averaging operators is that they effectively filter out high frequency

information. In domains where high frequency information matters, average pooling introduces the risk of smoothing out possibly important information.

Common to pooling operations in general, is that they potentially throw out important information. To remedy this, there exists research to instead downsample by larger strides in convolutional layers, suggested by the works of Springenberg et al. [41]. Furthermore, there are intricacies regarding the output size, in that values for f and s do not coincide with the input if the entire image's pixels are to be taken into consideration. Not using "even"-sized pooling options effectively "skips" boundary pixels instead of padding them.

3.6.4 Polar and cylindrical convolution

In dealing with the problem of rotated images, this thesis extends its reach to the method of "cylindrical convolution" of polar transformed images[13]*. This subsection provides a short, but comprehensive overview the method, and its application in processing rotated images.*

Historically, there have been many attempts to address the issue of rotation in works on convolutional neural networks. Some examples are the works of Jaderberg et al. [19] performing learnable affine transformation parameters on the feature maps inside a convolutional network, and the works of Jiang and Mei [20] look to polar transform the input image to achieve rotation-invariant feature learning. The procedure of polar transforming the image, and performing regular convolutions on this image, is referred to as **polar convolution** in this thesis.

Another method proposed and developed by Kim et al. [21] named **cylindrical convolution**, attempts to solve the rotation problem of convolutional neural networks by also recasting the images into a polar coordinate space. The additional trick of cylindrical convolution is to pad the top and bottom parts of the image with pixels from the bottom and top, respectively, and have this be a part of backpropagation. This has the effect of "completing" the polar transformed image, as Figure 16 demonstrates.

[13] Polar transformation of images is covered in subsection 4.3.8.

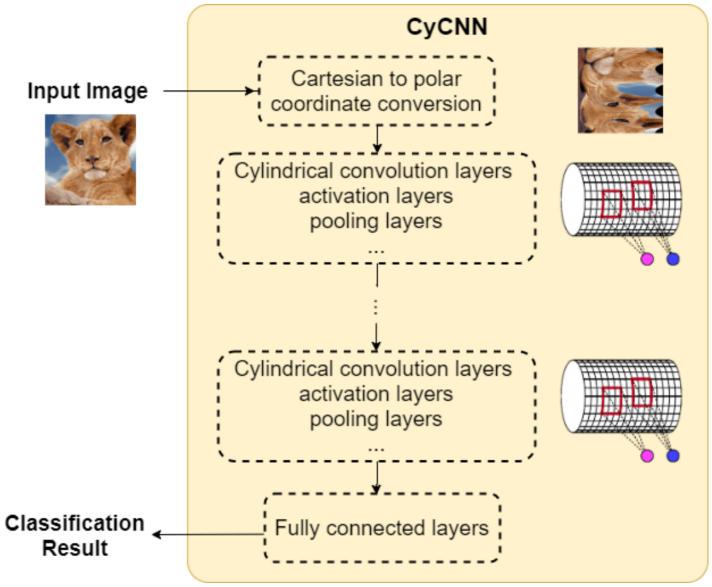


Figure 16: The cylindrical convolution pipeline as described and illustrated in the original paper of cylindrical convolutions by Kim et al. [21]. Notice the authors' attempt to conceptually visualize how the method of cylindrical convolution can be thought of wrapping the image around itself. The red outlines show the kernel convolving the pixels on the "cylinder".

3.7 Overfitting and regularization

When training complex models, there is a risk of the model "overfitting" to its training data. Because the goal of any predictive model is to generalize well to unseen data, tackling the issue of "overfitting" is critical to ensure the effectiveness of said model. This section presents regularization methods deployed to tackle overfitting in this thesis, whilst alluding to other techniques commonly used.

Before delving into the regularization techniques to prevent overfitting, they are properly defined by Definition 3.2 and 3.3 as they are described in Chapter 5 in the *Dive into Deep Learning* book [47]:

Definition 3.2 (*Overfitting*)**.** Overfitting occurs when the gap between the performance on the training data and validation data is too large.

Definition 3.3 (*Regularization*)**.** Regularization is any modification we make to a learning algorithm that is intended to increase a model's performance on validation data, but not necessarily increase the model's performance on training data.

These definitions speak to the fact that training a model is not a trivial task, and measures need to be taken to ensure that a model in fact is able to generalize well. To condense these two terms, regularization speaks to techniques to tackle the problem of overfitting. Below are some common approaches to perform regularization in deep learning problems.

Dropout: Neurons in a network have a tendency to co-adapt, meaning that neurons can become too dependent on each other. Co-adaptation can lead to the model overfitting on

the training data, and resultingly yields a model unfit to perform predictions on unseen data. Preventing co-adaptation is commonly done by the process of stochastically "dropping out" neurons (i.e. setting neurons' activation values to 0) during training, which effectively generates a plethora of smaller models learning the parameters, as opposed to one big model. Dropping out neurons followingly discourages co-adaptivity, as neurons that might co-adapt will not necessarily have the chance to do so [16], [42]. Figure 17 illustrates a possible path of a dropout network:

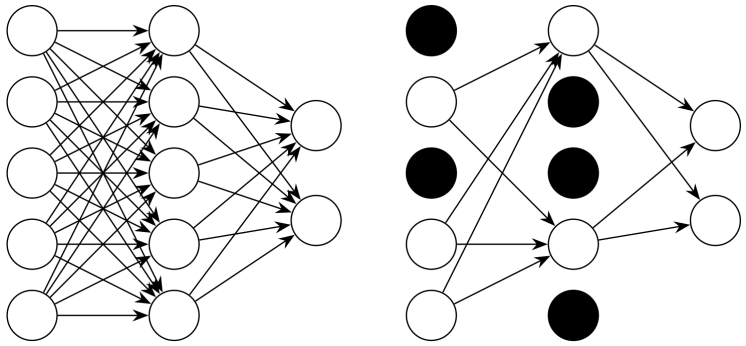


Figure 17: An illustration of a neural network having some of its neurons "dropped out", visualized as black neurons. The corresponding arrows show which weights contribute to the forward pass, and has the effect of producing ensembles of different forward-passes. Figure is reprinted from the works of Labach et al. [24].

Performing dropout in CNNs can be done in several ways, as one can argue what constitutes "features" in a CNN. One approach is to randomly drop out activations in the feature volumes themselves, contributing to a variability in the network's activations. This effectively warps the activations in the following feature volumes, which acts as a regularization measure because it adds noise to the hidden units (performed by `nn.Dropout()` in PyTorch). [33]

A caveat of this sort of dropout, is that images tend to have neighboring pixels be spatially correlated. Excluding potentially vital spatial information might hurt the model's ability to train, as pointed out in the works of Tompson et al. [45].

Another way of implementing dropout in CNNs, is by dropping out entire feature channels instead of individual activations. This has the effect of making the CNN more independent on its feature channels. (performed by `nn.Dropout2d()` in PyTorch)

Data augmentation: Increasing the size of the training dataset is usually preferable in deep learning tasks, as their performance are dependent on the training dataset to pose a representative distribution of the domain of data that is to be learnt. Adding to the training set in one way or another is therefore preferable, and can be done in numerous ways. In regards to images, one can for example vary the image's rotation and bluriness, which are the augmentation techniques applied in this thesis. Other augmentation tricks involve for example translating the image, adding different types of noise, and channel shifting to simulate different brightness conditions.

Batch normalization: The distribution of each layer's inputs are subject to change through training, which effectively slows down training (described as *internal covariate shift*) [18]. Addressing this problem can be done by normalizing layer inputs for each mini-batch during training, and acts as a regularizer. Furthermore, batch normalization allows for being less careful about parameter initialization, higher learning rates, and in some cases eliminates the need for dropout regularization [18]. For details regarding the algorithm to perform this, the reader is referred to Algorithm 1 and Algorithm 2 in the works of Ioffe and Szegedy [18].

Early stopping: In training fairly complex models, a common observation is a decreasing performance on the unseen data, whilst the performance on the training data is improving. This observation signals that the model is overfitting, which one can attempt to prevent by prematurely stopping model training. Protocols for stopping model training is described as early stopping, and can be implemented by saving the model parameters after a certain multiple of epochs/gradient steps. Another way of early stopping is to save the parameters only if the performance on the unseen data improves until a stopping criteria is met, but evaluating the model performance after each gradient step might be expensive to perform.

Other methods: A popular approach to regularization, is by penalizing a model's representational capacity. Common ways to do this, are penalizing a high number of basis functions a regression model uses (see section 3.3), or by penalizing large model weights by adding distance functions such as L1-norm $||\boldsymbol{w}||_1$ or the L2-norm $||\boldsymbol{w}||_2$ to the cost function $\mathcal{C}$. Generally, the notation for this is as follows:

$$\tilde{\mathcal{C}}(\boldsymbol{w}) = \mathcal{C}(\boldsymbol{w}) + \lambda R(\boldsymbol{w}) \tag{62}$$

where λ is a hyperparameter to determine the strength of the penalization term.[14]

[14]Biases in neural networks are seldom regularized as it could lead to for example underfitting (see page 226 in the *Dive into Deep Learning* book [47] for details).

4

The Image Similarity Approach

This chapter presents motivation as to why siamese neural networks are employed, and how the models are trained. Practically, it serves as a theory section more specific to the thesis' approach to RQ1 and RQ2.

4.1 Motivation

Reiterating the core problem of this thesis, is that it attempts to relate an image of a salmon louse to an assigned label, namely classification. Achieveing such predictions followingly requires an image dataset to be labeled, but more importantly demands a way of measuring if an image corresponds to a certain label. Instead of asking the question *"which class (ID) does image X belong to?"*, the problem can be posed as a problem of measuring image similarity, i.e. repurposing the question as *"does image X_1 belong to the same class (ID) as image X_2"*.

In the domain of deep learning, deploying *Siamese Neural Networks* to solve the problem of measuring the similarity between inputs is a common approach. Motivating the approach of SNNs can be done by first looking at some of the shortcomings from how this general problem of multi-class classification is commonly approached.

Multi-class classification by category prediction: Traditionally, neural networks attempting to solve multi-class classification tasks are equipped with an output layer (neurons denoted as $\{a_i{}^{[L]}\}_{i=1}^{m}$) activated by the *softmax* function:

$$\text{softmax}(a_i) = \frac{e^{a_i}}{\sum_{i=1}^{m} e^{a_j}} \tag{63}$$

The intuition behind this activation function, is that the output layer's values resemble class probabilities, as they fulfill the requirement of $\sum_j a_j = 1$ (i.e. the sum of all probabilities are 1). Training such a network is commonly done by letting the loss function be the negative log-likelihood function[15], and the label be one-hot encoded[16].

A classic example to showcase of the advantages of category prediction can be found in the classification of the MNIST dataset of handwritten digits [25]. Approaching this problem with the use of fairly regular CNN architectures, and an output layer activated by the log softmax function[17], test accuracies has been reported to be as high as 99.91% [3] as of writing this thesis. A simple illustration of such a multi-class classification apporach, is shown by Figure 18:

[15]This loss function is motivated by ideas from entropy and information theory, details can be found in the *Pattern Recognition and Machine Learning* book Chapter 1 [5].). Furthermore, one could train the classification layer directly by the cross-entropy loss function, without needing to activate the output layer by softmax. For inference however, one would still need to activate the output by softmax to get meaningful predictions.

[16]One-hot encoding frames the label as a perfect probability distribution, where the label is assigned a probability of 1 and all other labels are assigned a probability of 0.

[17]Taking the log of Equation (63) is generally preferred to combat numerical instability.

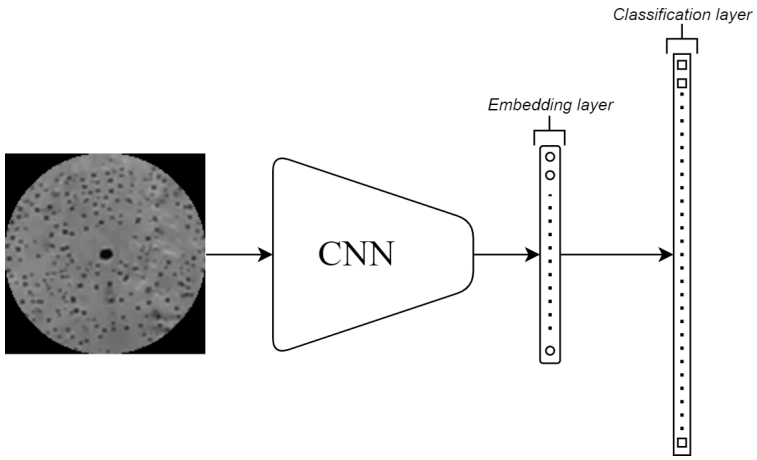


Figure 18: The input image is first fed through a CNN architecture. This serves to encode the image into its most important components, represented by the embedding layer, which is a $1 \times D$ dimensional vector (illustrated as a rounded rectangle with circles to illustrate the neurons in the embedding layer). The embeddings are then connected to a classification layer that attempts to one-hot encode the input image. Its length is the number of classes to predict.

To this approach exists significant drawbacks. Important to any deep learning approach, is a sizeable dataset in order to teach a class' general distribution to a model. In regards real world applications, it might be both expensive and time-consuming to collect large amounts of data. Although one can look to *data augmentation* (see sections 3.7 and 4.3.7) to address this, acquiring original data is key for most deep learning problems.

Another limitation of this multi-class classification approach, is that it is not designed to easily accommodate new classes. Should a new class be important to include, the entire network will have to be retrained and a new neuron will have to be added to the classification layer.

Multi-class classification by similarity prediction: The presented weaknesses in category prediction can be overcomed by instead having the model learn to detect similarity or dissimilarity between two images. As sections 4.3.5 and 4.3.6 will readily demonstrate, there are many more data points to a dataset if one instead frames the problem as measuring the similarity between inputs.

As previously mentioned, training a neural network to measure the similarity between inputs is commonly done by Siamese Neural Networks (abbreviated as SNNs). The idea behind SNNs is to set up multiple instances of the same network that share both the network architecture and parameters. The job of the network architecture is then to sophistically decode/down-dimensionalize the inputs into an *embedding space*, such that inputs that belong to the same class are mapped close together in some sense, whereas inputs belonging to different classes are mapped far away from each other. The following figures aims to comprehensively capture the core idea of such architecture/parameter-sharing, and how a trained decoding network maps similar inputs close together:

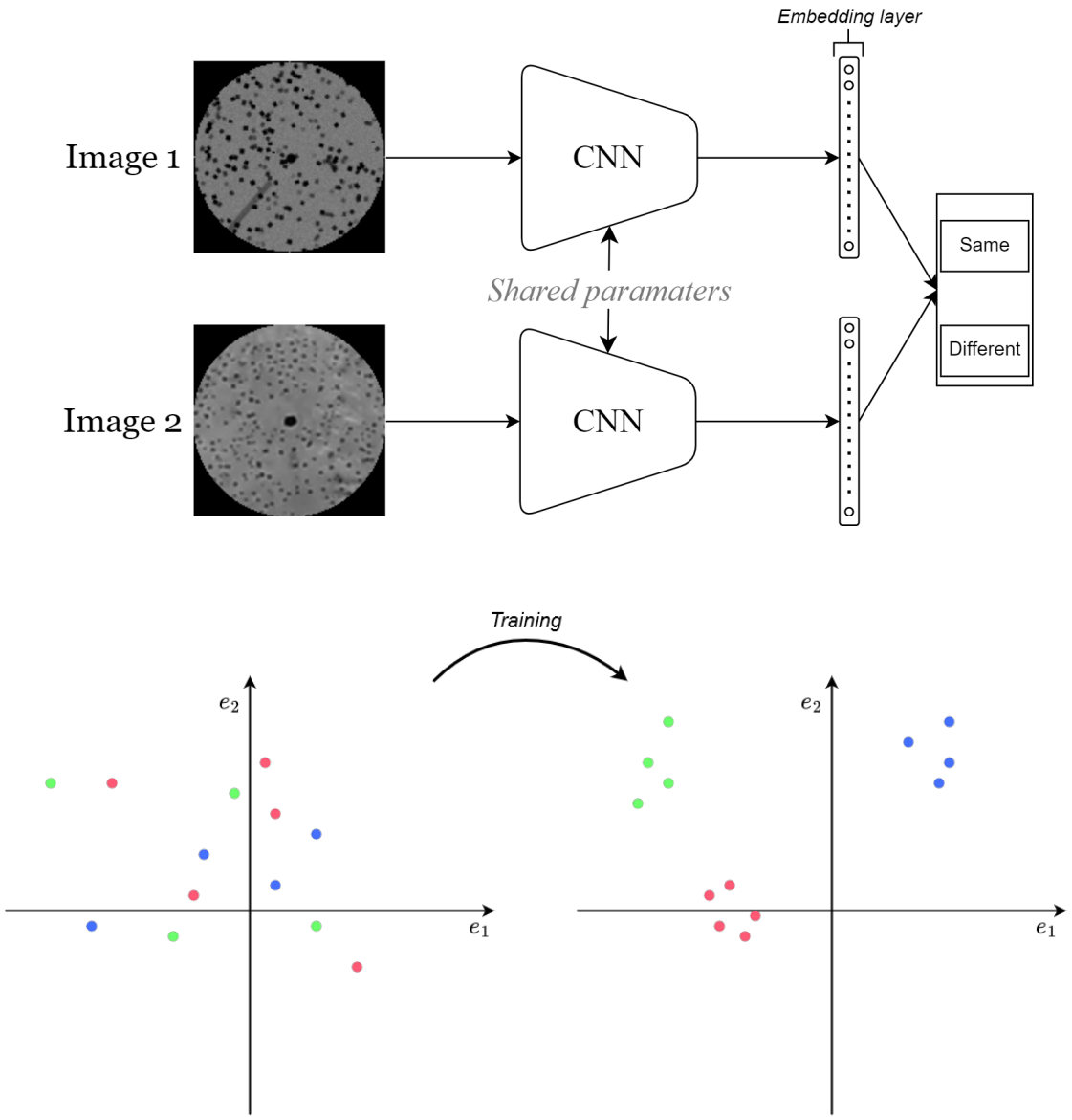


Figure 19: The left figure shows color-labeled images in the embedding space of an untrained model. The right figure shows the same color-labeled datapoints, but in the embedding space of a trained model. The axes e_1 and e_2 span a 2D embedding space for visual purposes, but can be extended to any number of dimensions.

Training an SNN as Figure **??** depicts is commonly referred to as the *face verification problem*[18], and comes from the fact that this pipeline can be used to determine if two images of faces belong to the same person or not. Appropriately, pairs of images are also referred to as a "genuine" pair, or an "impostor" pair. These terms (same/genuine and different/impostor) are used interchangeably throughout the thesis.

The idea of architecture/parameter-sharing as depicted in Figure **??** was first seen in research in the early 1990s. They were first named by Bromley et al. [7] in order to verify signatures, and

[18]There are some details left out of the face verification problem, such as how one would connect the embedding layer to the output layer, which loss function to use etc. More details on this can be found in the DeepFace paper [43].

independently developed by Baldi and Chauvin [4] to match pairs of fingerprints[19]. Following their inception, they have proven more effective in recent years due to hardware improvements, as they generally take a long time to train.

In regards to this thesis: Common for category and similarity prediction as presented, is that they learn to produce an embedding vector *indirectly* (i.e. the loss function is not designed to work directly with the embeddings). For this thesis, training is performed by considering the embedding vector rather than the outputs of a classification layer that outputs "same" or "different". This is motivated from the results of training the embeddings directly as presented in the FaceNet-paper [40] outperforming the results of the DeepFace-paper [43], which uses a classification layer to train the embeddings.

4.2 Loss functions for siamese neural networks

4.2.1 Contrastive loss

*Although this thesis does **not** use Contrastive Loss in training and evaluating results, it is included as a subsection to show how siamese neural networks were traditionally trained prior to the invention of Triplet Loss.*

The objective of the Contrastive Loss function, originally proposed by the works of Hadsell et al. [13], is to cluster together *similiar* images (indicated by $I = 0$), and distance *dissimilar* images (indicated by $I = 1$). Defining a mathematical formula for Contrastive Loss (denoted by $\mathcal{L}_C$) is done by a superposition of both *similarity loss* (denoted as $\mathcal{L}_S$) and *dissimilarity loss* (denoted as $\mathcal{L}_D$):

$$\mathcal{L}_C = \mathcal{L}_S + \mathcal{L}_D \tag{64}$$

The indication variable I is a part of each conditional loss, making the superposition formulation possible. The similarity loss and dissimilarity loss is elaborated further, and assume further that X_1 and X_2 are input images that are either similar or dissimilar. Lastly, let m be a margin hyperparameter which decides how big the distance between dissimilar image embeddings must be before the model actively trains on dissimilar image pairs:

$$\mathcal{L}_S = (1 - I) \cdot \frac{1}{2} \cdot (\mathcal{D}(X_1, X_2))^2 \tag{65}$$

$$\mathcal{L}_D = I \cdot \frac{1}{2} \cdot (\max\{0, [m - \mathcal{D}(X_1, X_2)]\})^2 \tag{66}$$

Inside the formulations for similarity and dissimilarity loss for the Contrastive Loss function is a distance measure $\mathcal{D}$, which aims to quantify the distance between the embeddings $f()$ of the images X_1 and X_2:

[19] A more comprehensive overview of the history of siamese neural networks can be found in the book named *Artifical Neural Networks: An Overview* [8].

$$\mathcal{D}(X_1, X_2) = \|f(X_1) - f(X_2)\| \tag{67}$$

4.2.2 Triplet loss

First described in the FaceNet-paper [40], the Triplet Loss function achieves training of a siamese neural network by letting image embeddings be *triplets*. These triplets contain an anchor point (A), positive point (P) and a negative point (N). The points A and P will share the same label, and N has a label different from A and P.

The objective of the Triplet Loss function will be to pull together similar image embeddings (A and P), whilst also push away dissimilar image embeddings (A and N) as illustrated in Figure 20

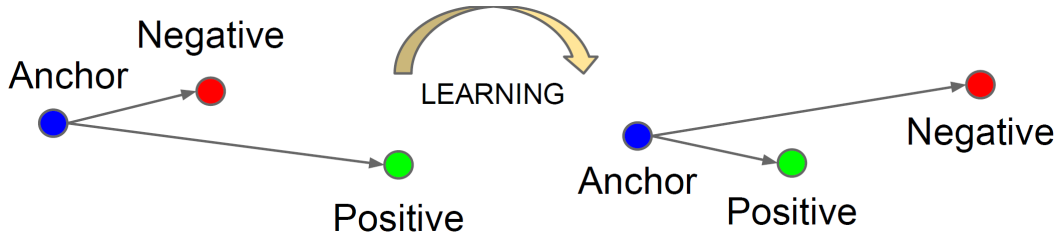


Figure 20: Illustrative example of what the Triplet Loss attempts to achieve in order to directly learn image embeddings. Notice that the distance between A and P is not directly paid attention to. Figure is reprinted from the FaceNet-paper [40].

In motivating a loss function with triplets of embedded images, the authors of the FaceNet-paper proposes initially to constrain the distances to the points in a triplet as such:

$$\|f(A) - f(P)\| + m < \|f(A) - f(N)\| \tag{68}$$

Following the notation from Section 4.2.1, from Equation 67, this constraint can be written in a more compact form:

$$\mathcal{D}(A, P) + m < \mathcal{D}(A, N) \tag{69}$$

The margin hyperparameter m is set to encourage P to be at most a distance m from A. Formalizing this inequality, the Triplet Loss function is achieved:

$$\begin{aligned} \mathcal{L}_T &= \max\{\mathcal{D}(A, P) - \mathcal{D}(A, N) + m, 0\} \\ &= \text{ReLU}\left(\mathcal{D}(A, P) - \mathcal{D}(A, N) + m\right) \end{aligned} \tag{70}$$

Remark 4.1. *A remark is made in regards to the Triplet Loss in that the image embeddings vector $f(X)$ is set to be normalized in the FaceNet-paper [40]. This thesis follows this custom.*

4.2.3 Triplet selection (Data mining)

As stated by the FaceNet-paper [40], selecting A, P, N at random will likely yield negative samples that easily satisfy the constraint (69), resulting in $\mathcal{L}_T = 0$. Henceforth, such triplets will not contribute to improving the model parameters, and will result in significantly slow convergence. In mitigating this issue, one solution is online data mining, which refers the process of selecting triplets that will contribute to model training as it trains.

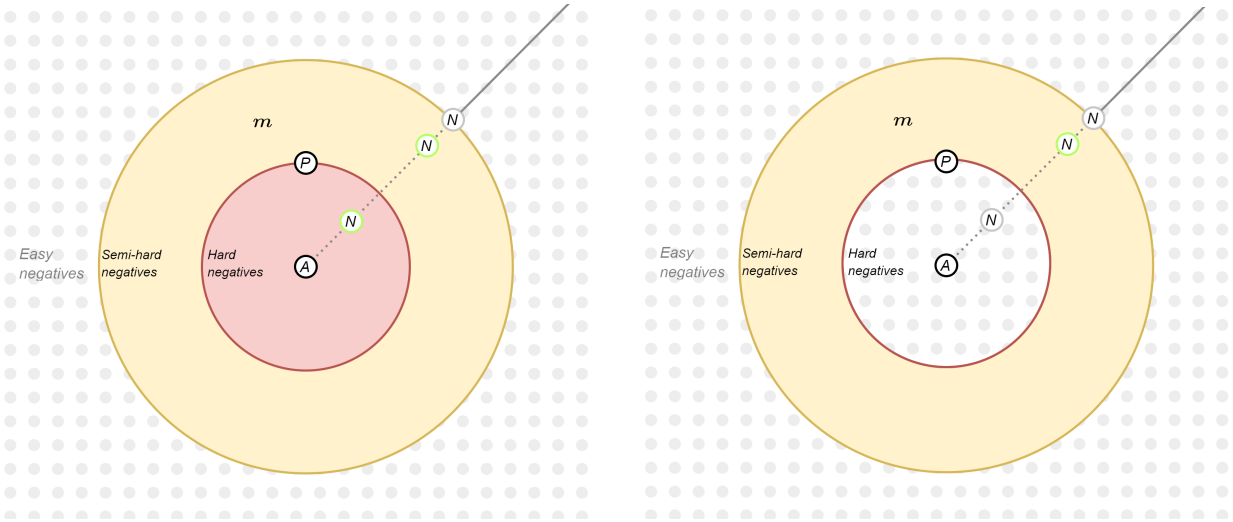
This thesis utilizes semi-hard data mining. This is achieved by choosing embeddings that **violate** (69):

$$\mathcal{D}(A, P) + m \geq \mathcal{D}(A, N) \tag{71}$$

but **satisfy** this semi-hard constraint:

$$\mathcal{D}(A, P) < \mathcal{D}(A, N) \tag{72}$$

Masking the triplets by these two constraints is illustrated as follows:



(a) Constraining triplet selection by (71). (b) Constraining triplet selection by (71) and (72).

Figure 21: Illustration of how one triplet would be selected for a euclidean distance function. The points A, P, N are placed for illustrative purposes, but it is important to stress that the anchor and positive points define the red circle, where hard negatives (achieved by removing m from 71, or by violating 72) can lie. Notice that the green circles around the negative point indicate that the point will be selected according to the constraints set for triplet selection. Gray circles indicate that the triplet will not be selected.

As is apparent from Figure 21, by setting the constraints (71) and (72), triplets that exhibit negative points inside the margin m will be selected for backpropagation, and resultingly the model trains on these triplets whilst neglecting the rest.

Emphasized by the FaceNet-paper [40], is that the hardest triplets often in practice leads to the model collapsing (i.e. $f()$ maps all points to 0), such that the model does not learn anything. This is easily spotted when the average training loss is exactly the same as m, where selecting the semi-hard negatives is a way of mitigating this. Altering the triplet selection constraints mid-training was experimented with, but the results were inconclusive and is out of the scope of this thesis.

Remark 4.2. *Choosing to not square the distance function $\mathcal{D}()$ yielded a more stable convergence in the experiments of this thesis, and will therefore deviate from the semi-hard triplet selection from the FaceNet-paper [40].*

4.3 Dataset details

Necessary to the neural network approach, is large amounts of data in order to properly train such models. As with many real applications, the dataset provided for this thesis is no stranger to the lack of data, and will be an important problem to address. Furthermore, because the thesis deals with Siamese Neural Networks, there is a need to cover some specific definitions in regards to the dataset, namely **individual**s (**ID**s), **samples** and **data points**.

IDs: A louse individual (ID) corresponds to one *individual*. IDs can freely jump from fish to fish, but an ID is assumed to possess a similar melanin pattern as time passes. An ID can be viewed as the label of the salmon louse

Samples: Images of IDs are taken after a set amount of time. As one ID is photographed over different time instances, an ID accumulates reference images. For simplicity, such reference images of the same ID is defined as *samples*.

Data points: Like most machine learning pipelines, we define inputs to the Siamese Neural Networks as data points. Unlike most machine learning pipelines, a data point, in the sense of this thesis, is not a singular image, but rather a *collection of images*. As described in sections 4.2.1 and 4.2.2, a collection of images will depend on which loss function is being used to train the network. For the purposes of this thesis, data points are either defined as **triplets**[20] of images, or **pairs**[21] of images.

Below is a representative figure of how a small dataset would look like:

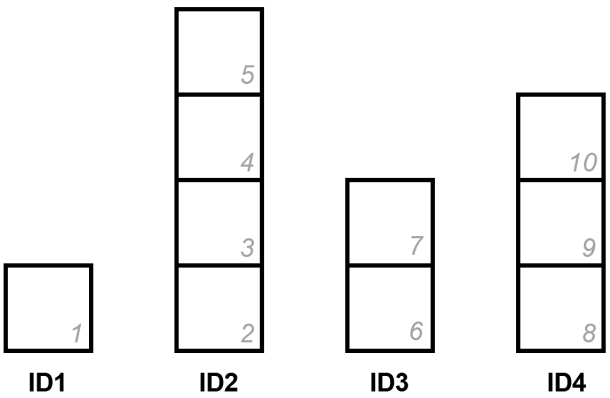


Figure 22: A box represents an image, where the numbers in the right hand corner of the boxes denotes the image number in the entire dataset. The stacks of boxes illustrate the set of unique images (samples) there are of each ID.

[20] As section 4.3.6 elaborates, a triplet is non-trivial and must satisfy certain constraints.

[21] Figure 22 is re-used in Figure 35 to visualize pairs of images.

4.3.1 Division of training and evaluation datasets

Training and evaluating a model to correctly identify the relationship between two images, can be split into two categories.

- **The few-shot classification problem:** Train a model on all existing IDs, holding out a percentage of samples, and use this model to attempt to recognize the held out samples.
- **The zero-shot classification problem:** Train a model on a percentage of IDs, letting all samples of these IDs be a part of the training dataset, and use this model to attempt to relate completely unseen IDs.

The few-shot classification problem

Training any instance of siamese neural networks requires data points that contain images from the same ID, and from different IDs. Sections 4.2.1 and 4.2.2 will be more comprehensive in explaining the notion of *pairs* and *triplets*, and how they can be set up for training the model. This is mentioned here to emphasize that any ID should contain **at least two** samples to facilitate training.

Below is a representative figure that divides the dataset according to the samples of IDs:

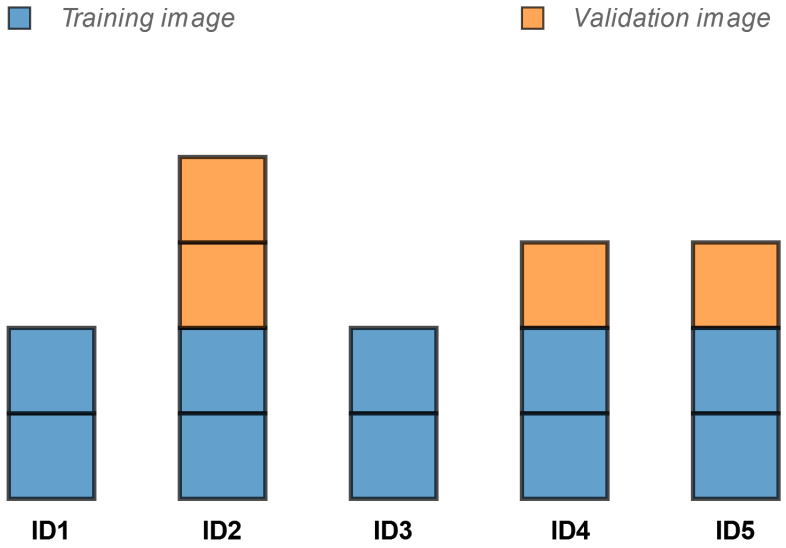


Figure 23: The blue color represents images belonging to the training dataset, whereas the orange color represents images belonging to the validation dataset. The validation dataset now contains unseen samples from IDs that are seen during training.

Solving the problem will yield a model trained to *recognize* already seen IDs, which can be ideal for making a dynamic lookup-table. One would simply feed a query image to the network, and the output would be some similarity score between the query image and IDs seen during training. By thresholding the similarity scores learned through model evaluation, the pipeline can return the most similar image, and this image's ID would be the predicted ID.

However, such a problem will require multiple samples of each ID, in which collecting more samples could be an expensive procedure. Furthermore, all the IDs have been seen during training, such that evaluating its ability to generalize to unseen IDs might be challenging.

The zero-shot classification problem

As is apparent from the *few-shot classification problem* there is a need to be able to generalize to unseen IDs, which is more or less why siamese neural networks were invented in the first place. Looking at the provided dataset (see section 4.3.2), the number of IDs greatly exceeds the number of samples.

Below is a representative figure that divides the dataset according to the IDs:

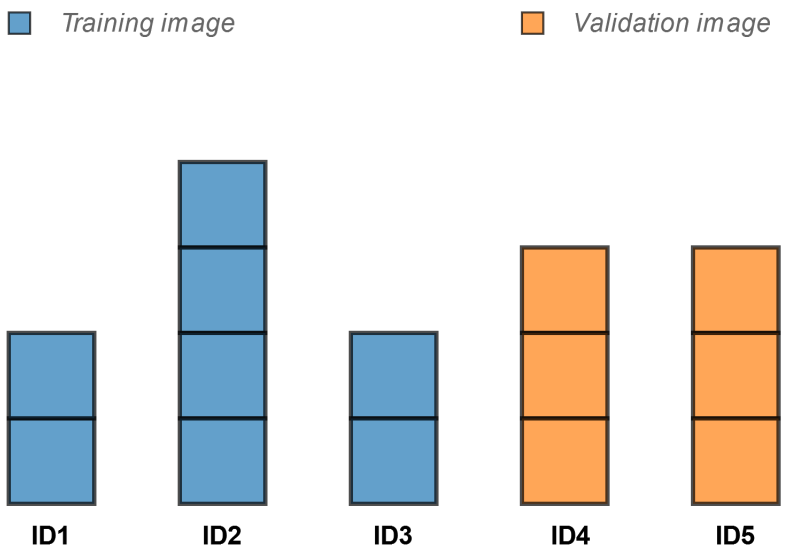


Figure 24: The blue color represents images belonging to the training dataset, whereas the orange color represents images belonging to the validation dataset. The validation dataset now contains IDs not seen during training.

Solving this problem will present an important tool for monitoring and understanding the ecology of salmon lice individuals that are not seen during training. Models able to generalize such a task will further prove that the siamese framework approach is able to give insight to how one can monitor arbitrary salmon louse ecology systems.

A caveat of this problem is that it is generally a much more difficult task to learn than few-shot classification. In the world of the network, it has only ever seen images of salmon lice, and will consequently require many different IDs and samples to be taught the similarities and dissimilarities between such images. Furthermore, the images presented in this thesis are still difficult to obtain, as each louse would have to be manually removed and imaged in a proper way. In future work, there is still a need to perform labeling and recognition on-the-fly, which is out of the scope of this thesis.

Moving on to the datasets themselves, the following subsections describe the real dataset

provided by Mennerat et al. [29], and a synthetic dataset produced to mimic the real dataset.

4.3.2 The real dataset

Specific to the problem of this thesis, is a dataset containing images of salmon lice. They are photographed in groups, and have been processed to segment individual lice in an image masked by a circle. A series of images from the real dataset are shown in Figure 25:

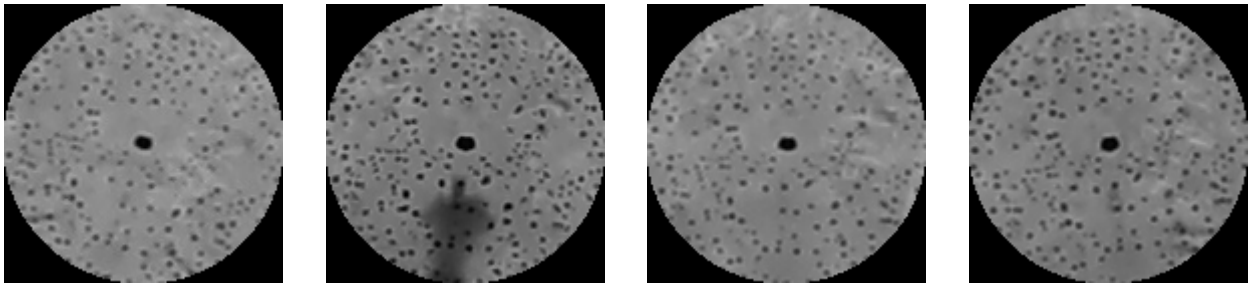


Figure 25: A number of four images of the same louse, taken at different times. In other words, there are four **samples** to this **ID**.

Onwards, the **dataset distribution** of the real dataset is elaborated on. The real dataset contains a number of 40 collections of imaged lice, ranging from 2-4 unique images. A number of 35 images are labeled, but lonely (the ID has 1 sample). Furthermore, 67 images are unlabeled. Below is a representative bar plot in order to illustrate the dataset, and is closely related to Figure 24:

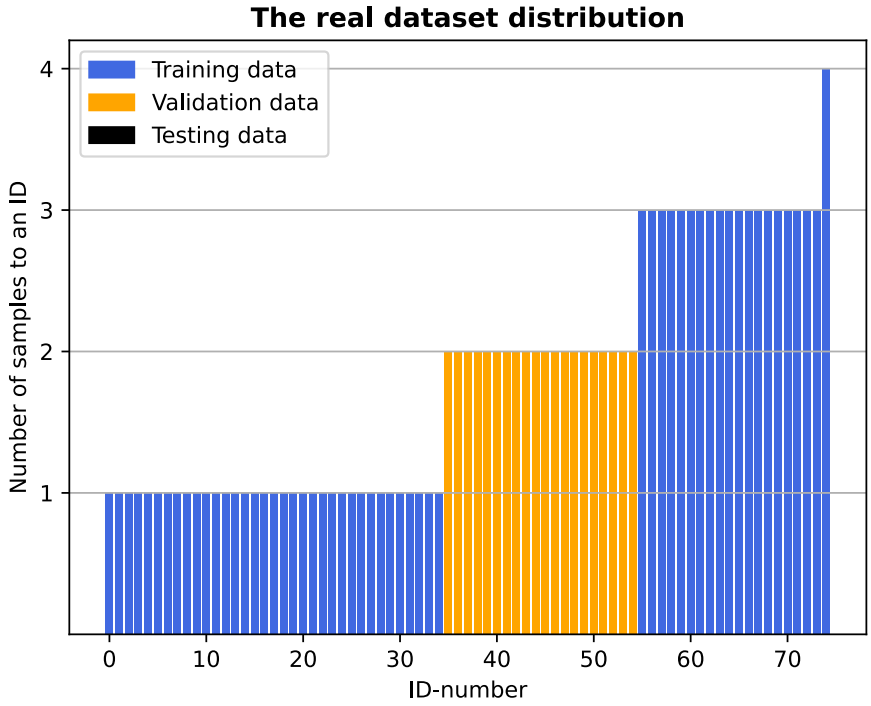


Figure 26: The ID/sample distribution of the real dataset. Note that the 67 unlabeled images are not present in the figure for the purposes of interpretability, and that dataset is too scarce to allocate images to a testing dataset.

Conclusively, there are instances of both labeled (present in Figure 44) and unlabeled lice, as well as labeled lice with one sample. Because data scarcity is a prominent issue in this thesis, data has to be used efficiently, and will hence demand detailed definitions as described by its *triplet cardinality* in section 4.3.6.

4.3.3 The synthetic dataset

The synthetic dataset is made to mimic the real dataset, with an ideal distribution of samples per image. Because the maximum number of samples in a real ID is 4, the entirity of the synthetic dataset has 4 samples to each ID. Furthermore, a number of 75 IDs was chosen because the real dataset has a number of 75 IDs labeled. Synthetically produced images also has the benefit of letting the user be able to artificially control how the synthetic salmon lice images look like. Some of the design choises in making this dataset is followingly elaborated:

Image background: The image background is synthesized by letting each background pixel intensity be collected from a Gaussian noise distribution with relatively low variance. The choice of a Gaussian noise distribution might not be the most optimal choice, but is simple to produce and develop countermeasures for, by for example slightly blurring the image.

Black dots: To mimic the melanin pattern of a salmon louse, a number of black rectangles are randomly distributed across the image. The rectangles' widths and heights are randomized by a uniform distribution. Furthermore, the positions of the individual black dots are slightly warped by a Gaussian distribution. The black center dot mimics a louse's mouth, and is kept circular and even-sized.

Rotation of images: Generally, the images cannot be guaranteed to be appear non-rotated. To account for this, the images are randomly rotated in the training pipeline as part of data augmentation.

Circle crop: In terms of cylindrical convolutions (see section 3.6.4), there is no need for the pixels that exist outside of the image radius r from the center. Furthermore, rotating the images would introduce unwanted rotation artefacts because of padding issues.

Image dimensions: To achieve symmetrical circular crops, the images were made to be odd in dimensions in order to have a properly defined center pixel. As the real dataset consists of images of dimensions 139×139, a choice was made to downsize the synthetic images to 127×127 as it is close to an exponent of 2 for computational purposes. On the notion of performance, the number of pixels available does indeed affect the performance. The effects on performance, however, are assumed to be relatively minimal as Table 4 in the FaceNet-paper suggests [40].

A more detailed documentation can be found in the `DATASET_HPs.py` file in the `hyperparameters` folder in the GitHub repository. Some examples of synthetically produced images are shown in Figure 27:

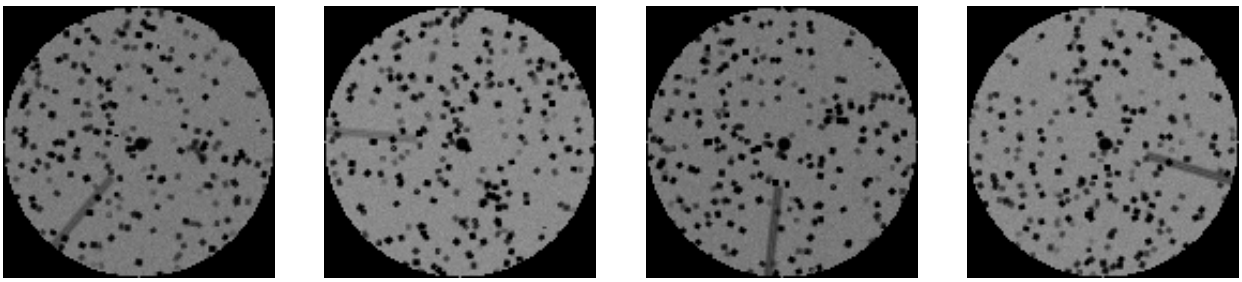


Figure 27: Four synthetic lice, with different generation seeds.

4.3.4 Standardization of data

In the works of Ioffe and Szegedy [18] and its sources therein, a point is made in that "whitening" the input (i.e. linearly transforming the input dataset such that its mean is set to be 0, and its variance to be set to 1) generally makes training go faster, which is a point that in fact motivates the procedure of batch normalization as section 3.7 describes. For clarity, the whitening term is translated as *standardization* in this thesis[22].

More specifically, the mean and variance responsible for standardizing datasets, are calculated from the training dataset. Suppose that N_p is the number of pixels in the training dataset, and p_i is the pixel value of the i–th pixel in the training dataset. The mean μ_{train} of the training dataset is defined as follows:

$$\mu_{\text{train}} = \frac{1}{N_p} \sum_i p_i \tag{73}$$

The variance σ^2_{train} depends on the mean. Its definition is as follows:

$$\sigma^2_{\text{train}} = \frac{\sum_i \left(p_i - \mu_{\text{train}}\right)^2}{N_p} \tag{74}$$

Employing these measures to standardize the data is done as follows:

$$\hat{\boldsymbol{X}} = \frac{\boldsymbol{X} - \mu_{\text{train}}}{\sqrt{\sigma^2_{\text{train}}}} \tag{75}$$

4.3.5 Pair cardinality $|\mathcal{P}|$

In order to define a proper sense of what an epoch is in regards to siamese neural networks, there is a need to describe the number of training possibilities. This subsection and the next defines the proper notion of an epoch in regards to which loss function is being used to train the model, and how many instances there are to evaluate in producing performance results.

Relevant to the **Contrastive Loss** function (was covered in subsection 4.2.1), and evaluating the performance of a siamese neural network, there is a need to work out the number of unique

[22]Standardization speaks to the dataset being scaled such that it has a mean of 0 and variance equal to 1. Normalization, on the other hand, is the procedure of skewing the entire dataset into the region of $[0, 1]$. The two terms are often mistaken for each other, and is worth noting as a footnote.

ways to pair two images, i.e. the dataset's pair cardinality. For starters, it is apparent that X_1 and X_2 can be interchanged without affecting the loss, meaning that the number the unique image pairs is essentially halved. Essentially, pair cardinality can be deducted from counting the number of elements in an upper triangual square matrix. Suppose there is a number of N images in a dataset. Then its unique pair cardinality $|\mathcal{P}|$ is[23]:

$$|\mathcal{P}| = |\{(1,2),(1,3)\dots(1,N),(2,1),(2,3),(2,4),\dots,(2,N)\dots,(N,N-2),(N,N-1)\}| \quad (76)$$

Stated with a more compact notation:

$$|\mathcal{P}| = \sum_{n=1}^{N}\sum_{m=n+1}^{N} 1 = \frac{N(N-1)}{2} \quad (77)$$

Note 4.1. *Self-similar image pairs, i.e. $\{(n,n)\}$, are left out in model evaluation as they are always a trivial solution to the problem of matching identities. This, however, can be partially solved by augmenting the images in this self-similar image pair, and is a way to enhance a dataset's evaluation cardinality. Augmentation of self-similar image pairs is not done in this thesis, but augmentation of image pairs that are not self-similar is done.*

4.3.6 Triplet cardinality $|\mathcal{T}|$

For the **Triplet Loss**, there is a need to describe the number of possible triplets, and is a significantly more difficult task than pair cardinality. First off, $((A,P),N) \neq ((P,A),N)$, meaning that order matters. Secondly, the triplet cardinality is highly dependent of how many samples there are to an ID, as opposed to the pair cardinality.

Triplet cardinality example

Starting off with an example, imagine the dataset $\mathcal{D}$ is rectangular (each ID has an equal amount of samples). Let s_i be the $s-$th sample for the $i-$th ID. Let also S_i be the number of samples for ID i, I be the number of IDs, and S be the total number of images in the entire dataset:

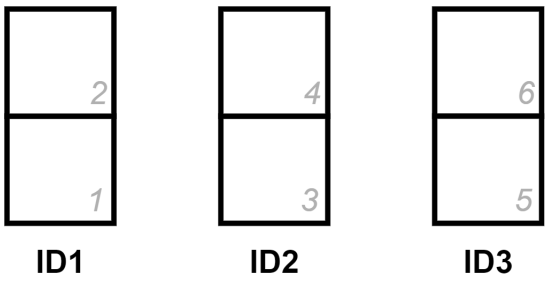


Figure 28: A rectangular dataset, meaning each ID has the same number of samples. For this example dataset, $S_i = 2 \;\; \forall \;\; i = \{1,\dots,I\}$, $I = 3$ and $S = \sum_i S_i = 6$.

Pairing up all the (A,P) possibilities from the first ID $i = 1$ (recall the definition of s_i) yields:

$$((1_1,2_1),3_2), \;\; ((1_1,2_1),4_2), \;\; ((1_1,2_1),5_3), \;\; ((1_1,2_1),6_3) \quad (78)$$

[23]A visual representation of image pairing can be found in Section 4.5

$$((2_1, 1_1), 3_2), \quad ((2_1, 1_1), 4_2), \quad ((2_1, 1_1), 5_3), \quad ((2_1, 1_1), 6_3) \tag{79}$$

Out of ID1 alone, there are 8 unique triplets. In order to generalize this, one needs to sample negatives from the remaining $(I-1)$ IDs, and each ID has $S_i(S_i-1)$ different permutations for anchor-positive pairing. Lastly, a number of S_i negatives from each ID are sampled. Stated in a more mathematical sense, this is formulated as:

$$(I-1) \cdot {S_i}^2 \cdot (S_i - 1), \qquad i = 1 \tag{80}$$

Completing the number of triplets for the dataset in Figure 28, Equation (80) is multiplied with its number of IDs. Seeing as each ID has an equal amount of samples, the dataset's triplet cardinality $|\mathcal{T}|$ can be calculated as such:

$$|\mathcal{T}| = I(I-1) \cdot {S_i}^2 \cdot (S_i - 1), \qquad i = 1 \wedge 2 \wedge 3 \tag{81}$$

$$|\mathcal{T}| = 3 \cdot (3-1) \cdot 2^2 \cdot 1 = 24 \tag{82}$$

In a more complete sense, with uneven amounts of samples in each ID, there is a need to sum over IDs and samples. In building a general formula for the number of triplets, the reader is reminded of S denoting the number of images in the entire dataset. By subtracting the entire dataset by the number of samples in the i–th ID to count the number of negatives that an anchor-positive pair can have, gives the expression:

$$|\mathcal{T}_{\text{unique}}| = \sum_{i=1}^{I} f(S_i) f\,(S_i - 1)\,(S - S_i) \tag{83}$$

Remark 4.3. *Truly unique anchor-positive pairs requires $f(S_i)f\,(S_i-1)$, where $f()$ is:*

$$f(x) = \begin{cases} 0, & x = 1 \\ x, & x \in \mathbb{N} \setminus \{1\} \end{cases} \tag{84}$$

*$f(x)$ is set to be 0 when the anchor image is paired with itself to create a positive image, as this is not a unique anchor-positive pair, and will lead to a trivial triplet data point. However, by augmenting the images in the dataset, one can theoretically produce infinitely many anchor-positive pairs. For definition purposes, an ID with one image is defined to have **one** anchor-positive pair. That is, the positive image is an augmented version of the anchor. This anchor-positive pair now only has a chance to be a trivial triplet data point, because of image augmentation.*

As described by Remark 4.3, one can pair an anchor with an augmented version of itself although it might lead to a trivial triplet data point. A triplet cardinality expression, where anchor-positive pairing with the same base image is allowed, is as follows:

$$|\mathcal{T}_{\text{full}}| = \sum_{i=1}^{I} \left(f(S_i) f\,(S_i - 1) + S_i\right)(S - S_i) \tag{85}$$

This updated expression now has added S_i inside the summation term in order to add all self-similar anchor-positive image pairs.

By combining the ideas behind these expressions for triplet cardinality, the cardinality used in practice is neither of these. Whenever an ID has only one sample (lonely ID), that sample is allowed to exist in an augmented self-similar anchor-positive pair. If, however, an ID has more than one sample, that ID is not allowed to have a self-similar anchor-positive pair. This is mainly done to include lonely IDs in as triplets in training, and to adjust the number training data points, as SNNs already take a long time to train.

This *design choice* is reflected in the following triplet cardinality expression:

$$|\mathcal{T}_{\text{mixed}}| = \sum_{i=1}^{I} S_i \left(S_i - 1\right) \left(S - S_i\right) \tag{86}$$

It is important to emphasize that this definition of triplet cardinality is specifically designed for inadequate datasets.

Regarding the real dataset

As mentioned in section 4.3.2, the scarcity of the real dataset requires cardinality nuances to be fleshed out.

Because there is a good chance an unlabeled louse is sampled multiple times, unlabeled images of lice can only be used as negative samples in conjunction with anchor-positive images that are labeled. Adding unlabeled lice as negatives will introduce new triplets for the model to train on, and will aid in increasing an already scarce dataset.

Letting unlabeled lice be negatives warrants an update to the expression for $|\mathcal{T}_{mixed}|$, and introduces the need denote the number of unlabeled lice as S_U:

$$|\mathcal{T}_{\text{mixed-U}}| = \sum_{i=1}^{I} S_i \left(S_i - 1\right) \left(S - S_i + S_U\right) \tag{87}$$

The expression $|\mathcal{T}_{\text{mixed-U}}|$ is read as follows: The cardinality of the triplet dataset, that is mixed with the triplet cardinality expressions (83) and (85), where unlabeled lice images denoted by U are included as negative images.

To summarize, $|\mathcal{T}_{unique}|$ is the triplet cardinality expression that yields the number of unique, non-trivial triplets in a dataset, and would be the go-to expression for larger datasets.

$|\mathcal{T}_{full}|$ still yields unique triplets, but some of them are trivial (i.e. the anchor and the positive image are the same) and need to be augmented to facilitate training for the trivial triplets.

$|\mathcal{T}_{mixed}|$ is a down-sized version of $|\mathcal{T}_{full}|$, only counting trivial triplets whenever an ID only has one sample to it.

$|\mathcal{T}_{mixed-U}|$ is the same as $|\mathcal{T}_{mixed}|$, but unlabeled images are now allowed to be set as negative images in acquiring triplets.

Note 4.2. *In reality, using unlabeled lice introduces problems regarding potential mislabeling. For completeness, this section works with the assumption that unlabeled louse is not a part of the labeled lice that has lost its labeling chip, but has accounted for the fact that unlabeled lice potentially do not exist as unique entities (i.e. the ID of one unlabeled image is different from the ID of another unlabeled image).*

4.3.7 Data augmentations

This subsection covers the augmentation techniques used for the datasets in this thesis.

A common practice in the field of machine learning, especially when data is scarce, is to augment the training dataset. This includes perturbing the training images in some way in order to artificially produce *new* images for the training dataset. This accomplishes variation to the dataset domain, which might be necessary in order to facilitate training and a model's ability to generalize.

Image rotation: The real dataset contains images of lice lying in arbitrary rotated positions. This dataset attribute poses the biggest hurdle to the performance of the network, and is a well known problem in the field of regular CNNs as subsection 3.6.4 mentions.

In order to cover the entire rotation range $\mathcal{R} = [-180°, 180°]$, the training images are randomly rotated in this range when sampled in the training loop. As might be apparent to the reader, this augmentation alone will provide in practice infinitely many augmentations as the range $\mathcal{R}$ contains real numbers.

Image blurring: To account for images taken with suboptimal focus, the training dataset is augmented by applying Gaussian blur with odd-sized blurring kernels of varying size k_{blur}. The kernels' sigma values are precomputed by OpenCV [6] as:

$$\sigma = 0.3 \cdot \left(\frac{k_{\text{blur}} - 1}{2} - 1 \right) + 0.8 \tag{88}$$

For this thesis, the range of kernel sizes was set to be odd numbers in the range $k_{\text{blur}} \in [1, 9]$, as any larger mask would potentially blur out the smallest speckles.

Evaluation augmentation

The datasets in this thesis are subject to data augmentation in evaluation as well as in training, and serves to account for the evaluation sets being lackluster. Performing augmentation of evaluation sets also allows for producing distributions of evaluation performances.

4.3.8 Polar transformation of images

Important to this thesis, is the procedure of transforming images into a polar coordinate space. This subsection covers the basics of polar coordinate transformation in the domain of images, and the methods used to perform this. The theory and figures are largely presented as the origin

paper of cylindrical convolution [21] does it, and is a continuation of polar convolution and cylindrical convolution first introduced in subsection 3.6.4.

In order to deal with rotated images, they are recast from the Cartesian coordinate system to the polar coordinate system. By doing this, the operation of rotation in Cartesian coordinates becomes translation in polar coordinates as Figure 29 illustrates:

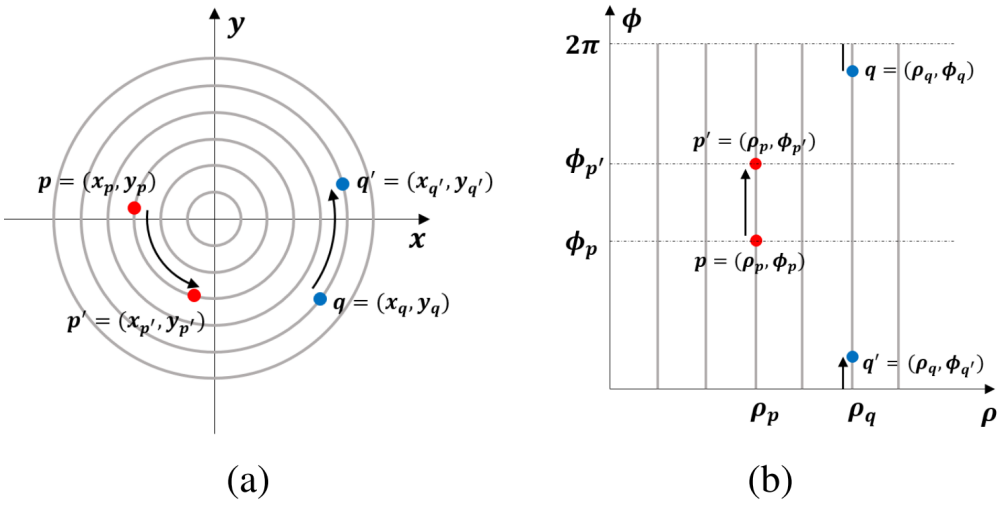


Figure 29: Illustration of how points from the Cartesian coordinate system are mapped to the polar coordinate system. In these figures, the points p and q undergo rotation in the Cartesian coordinate system, leaving them as points p' and q' in the polar coordinate system. Note also how $q \to q'$ passes the 2π boundary, leaving the two points spatially far from each other in polar coordinate space. For conversion reference, the gray circles in subfigure (a) correspond to the vertical lines in subfigure (b). The figure is reprinted from the origin paper of cylindrical convolutions [21].

To put it more formally, a point (x, y) in the Cartesian coordinate system is transformed to a point in the polar coordinate system (ρ, ϕ) by (89) and (90):

$$\rho = \sqrt{x^2 + y^2} \tag{89}$$

$$\phi = \begin{cases} \tan^{-1}\left(\frac{y}{x}\right) & \text{if } x > 0 \text{ and } y \geq 0, \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0, \\ \pi + \tan^{-1}\left(\frac{y}{x}\right) & \text{if } x < 0 \text{ and } y \geq 0, \\ \pi + \tan^{-1}\left(\frac{y}{x}\right) & \text{if } x < 0 \text{ and } y < 0, \\ \frac{3\pi}{2} & \text{if } x = 0 \text{ and } y < 0, \\ 2\pi + \tan^{-1}\left(\frac{y}{x}\right) & \text{if } x > 0 \text{ and } y < 0, \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases} \tag{90}$$

With any transformation of discrete data, comes the inevitable question of interpolation. As the paper on cylindrical convolution [21] notes, the chosen interpolation technique is bilinear interpolation. It is used in an attempt to mitigate aliasing artefacts introduced by the sampling procedure illustrated in Figure 29. For completeness, bilinear interpolation is briefly elaborated:

Bilinear interpolation: As previously stated, there are intricacies regarding transforming any discrete signal in that pixel information is only present at specific spatial locations. When transforming any image, there is a need to acquire pixel information in locations not covered by the original image's grid-structure. This process is known as interpolation, where the goal is to "fill in the blanks" and obtain pixel information from intermediate locations in the original image. Conceptually, bilinear interpolation performs a weighted sum between the four nearest neighbors, in which the weights are determined by where the desired interpolation point is. Figure 30 illustrates this concept:

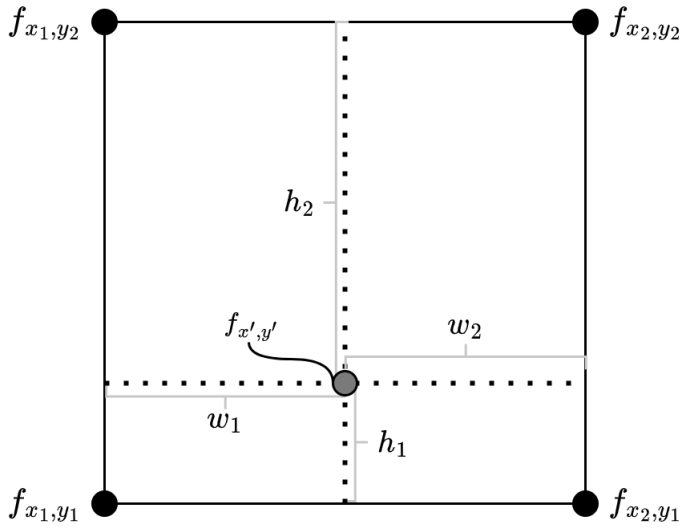


Figure 30: This figure shows a point $f_{x',y'}$ to be computed by bilinear interpolation. Solving it requires the neighboring pixel values denoted by f_{x_i,y_j}, and the horizontal and vertical distances w_i and h_j.

For starters, linear interpolation is first done in the x-direction (i.e. the two horizontal lines in Figure 30):

$$f_{x',y_1} = \frac{x_2 - x'}{x_2 - x_1} f_{x_1,y_1} + \frac{x - x_1}{x_2 - x_1} f_{x_2,y_1} \tag{91}$$

$$f_{x',y_2} = \frac{x_2 - x'}{x_2 - x_1} f_{x_1,y_2} + \frac{x - x_1}{x_2 - x_1} f_{x_2,y_2} \tag{92}$$

The same is done in the y-direction (i.e. the two vertical lines in Figure 30). Note that since $f(x', y_1)$ and $f(x', y_2)$ are now known, they are directly inserted into the linear interpolation equations for the y-direction:

$$f_{x',y'} = \frac{y_2 - y'}{y_2 - y_1} f_{x',y_1} + \frac{y' - y_1}{y_2 - y_1} f_{x',y_2} \tag{93}$$

This equation can be neatly represented in matrix form:

$$f_{x',y'} = \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x' & x' - x_1 \end{bmatrix} \begin{bmatrix} f_{x_1,y_1} & f_{x_1,y_2} \\ f_{x_2,y_1} & f_{x_2,y_2} \end{bmatrix} \begin{bmatrix} y_2 - y' \\ y' - y_1 \end{bmatrix} \tag{94}$$

In wrapping up this subsection, an example of a polar transformed salmon louse image is shown in Figure 31, where the bilinear interpolation technique resamples the image from intermediate pixel locations:

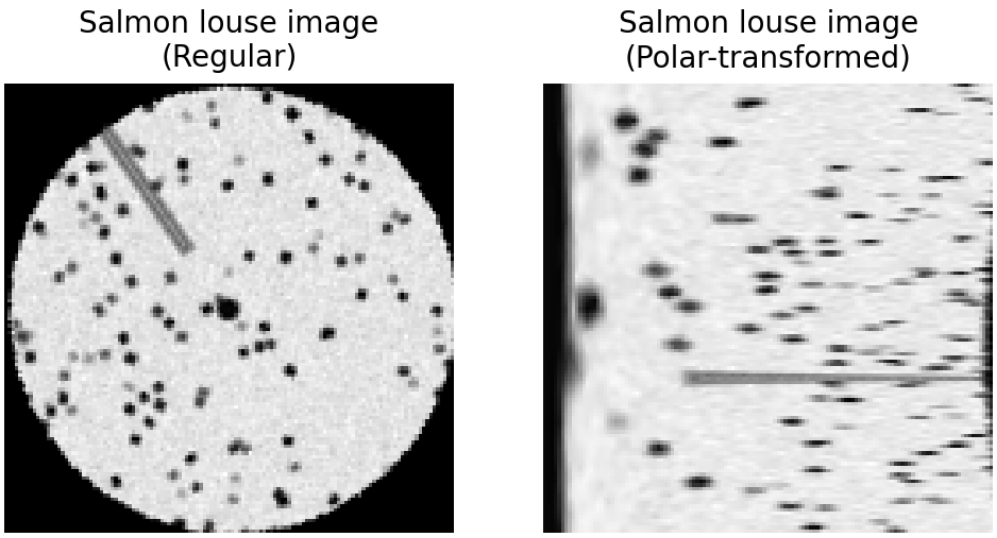


Figure 31: A salmon louse image polar transformed by bilinear interpolation. Note that because the radius is of the length of one half the width/height of the original image, the polar transformed image will be "stretched out" in the radius dimension to accommodate for this. For interpretability, the reader is advised to compare this figure to Figure 29.

Note 4.3. *Bilinear interpolation is the interpolation technique used when rotating the images by data augmentation.*

4.4 Convolutional Network Architectures

4.4.1 Model architectures

The thesis explores variants of three popular off-the-shelf model architectures: ***LeNet5*** *[25],* ***MobileNetV3*** *[17], and* ***ResNet18*** *[14]. These models were selected to cover a range of complexities and to demonstrate the effectiveness of the image similarity approach across varying architecture complexities.*

LeNet5-var The first model used is a variant of LeNet5, a CNN developed in the 1990s [25] for handwritten digit recognition. In an attempt to demonstrate simplicity versus complexity, the original LeNet5 architecture was subject to custom experimentation and tweaking. As such, its customized and tweaked architecture is elaborated in more detail than the two other models used in this thesis. The architecture is illustrated by Figure 32:

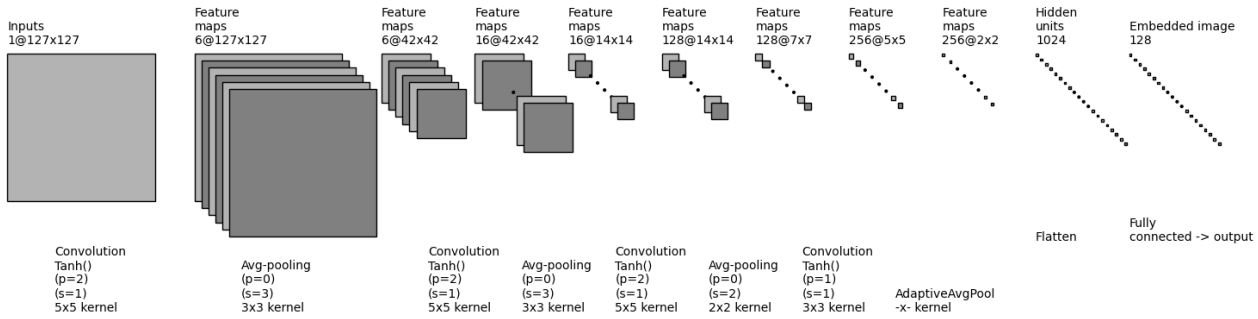


Figure 32: A flowchart illustrating the LeNet5-var architecture used in the thesis. Note that after every pooling layer is a dropout layer `nn.Dropout(p=0.2)`, and that the kernel size of the `AdaptiveAvgPool` is absent as it depends on the input and output.

The activation and pooling functions are generally kept. The number of feature channels have been increased, and padding for most convolutional layers has been included. In addition, dropout of individual feature map pixels by `nn.Dropout(p=0.2)` is done after every pooling layer.

Note 4.4. *The network is purposefully made non-optimal to illustrate if relatively arbitrary alterations to a simple network is sufficient to achieve adequate performance.*

MobileNetV3-var The second model used is a variant of the *small*[24] version of MobileNetV3 [17], a lightweight CNN designed for running on mobile devices with relatively limited memory specifications. Its architecture consists of a set of 11 InvertedResidual layers, wrapped with a convolutional layer in the beginning and the end of this set of layers. The deep convolutional part is then followed up by an adaptive average pooling layer and one hidden layer before the embedding output. Below is a figure of an InvertedResidual-block, reprinted from the paper where MobileNetV3 is introduced [17]:

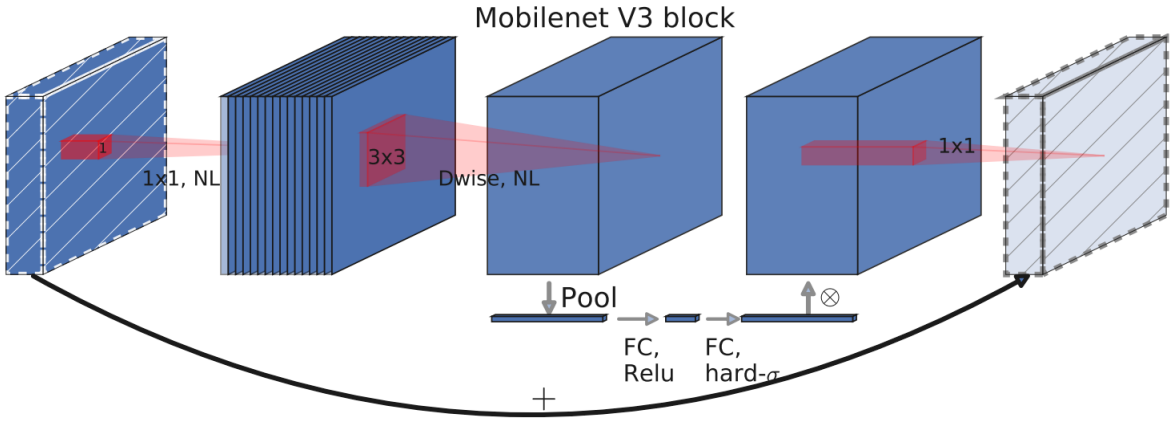


Figure 33: The InvertedResidual-block used in the MobileNetV3 architecture. "NL" is a general non-linearity (differs in some blocks), and "Dwise" corresponds to setting the `groups` variable in the `nn.Conv2d` to be the same as the number of output channels (each input channel is convolved by individual 2D kernels, yielding the same number of output channels). Below the depthwise layer is a so-called "Squeeze-and-excite" residual layer. More details are found in its origin paper [17].

[24]See Table 1 and Table 2 in the origin paper on MobileNetV3 [17] for their general differences.

Being consistent with the embedding output from LeNet5-var, the output layer of MobileNetV3-small is set to be 128-dimensional. Furthermore, the input size is adjusted to one color channel, as opposed to the original architecture needing three color channels in the input.

ResNet18-var The third model used is a variant of ResNet18 [14]. Having success with skip-connections[25] to form very deep networks, these "ResNets" were top performers in the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) 2015. Figure 34 shows a table from the original paper, describing some of the "ResNets" for different depths:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 34: Table 1 from the ResNet18-paper [14] showcasing some architecture variants used on the ImageNet dataset, where ResNet18 is highlighted by black lines. "Residual blocks" are shown in brackets with the number of stacked blocks. Each skip-connection is done after each block, meaning in ResNet18 there would be 8 skip-connections. Note that the output-layer is a 1000-dimensional softmax output, which is the number of classes to predict on the ImageNet1K [39] dataset.

The final output layer is adjusted to be 128-dimensional, and the input size is adjusted to one color channel.

Convolution versions

The model architectures just discussed are employed to generate image embeddings, and have been selected to cover models ranging from simple to complex. Furthermore, this thesis explores three different ways of convolving the input images:

- **Baseline convolution:** The input is convolved with standard `Conv` layers. This was covered in subsection 3.6.1.
- **Polar convolution:** The input image is polar transformed, and this image is convolved with regular `Conv` layers. This was covered in subsections 3.6.4 and 4.3.8.
- **Cylindrical convolution:** The input image is polar transformed, and regular `Conv` layers are swapped to `CyConv` layers. This was covered in subsections 3.6.4 and 4.3.8.

[25] Skip-connection generally refers to adding the input after some mapping function $\mathcal{F}$ has been performed on an input $\boldsymbol{x}$ as such: $\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$

In order to distinguish specific convolutions, the prefix "`Cy-`" is introduced to denote if `Conv` layers are replaced by `CyConv`, and the suffix "*-p*" to denote if the images have been polar transformed.

Common model attributes

Lastly, some hyperparameters are shared between the model variants, in order to make results fairly comparable with each other.

- ***Adaptive average pooling*** (`AdaptiveAvgPool2d`): Although most CNN architectures are built according to a specific image size, they are usually fitted with an adaptive pooling layer at their end. This makes the networks compatible with most input sizes, and the networks in this thesis apply this to the last convolutional layer.

- ***Padding in convolutional layers:*** Building on the compatibility feature of adaptive pooling, most convolutional layers in modern architectures are designed such that the spatial dimensions of the input and output are the same $\left(n_W^{[l]} = n_W^{[l+1]} \text{and } n_H^{[l]} = n_H^{[l+1]}\right)$. Furthermore, `CyConv`-layers effectively depend on padding, as it pads the top and bottom boundaries with pixels from the bottom and top, respectively (see subsection 3.6.4 for details).

- ***Embedding dimensionality:*** A common embedding dimensionality applied by several papers discussing Triplet Loss has been 128 [28], [40]. To make the results of the thesis fairly comparable, this is the chosen embedding dimensionality for the models.

- ***Dropout:*** In preventing overfitting to some degree, the layer after adaptive average pooling is equipped with a dropout layer `nn.Dropout(p=0.2)`. Performing `nn.Dropout2d(p=0.2)` after convolutional layers was experimented with, but resulted in significant training and performance drops.

4.5 Model Evaluation

Following the FaceNet-paper [40], the process of evaluating the network performance is done by evaluating the space of **all** image pairs. For clarity purposes, image pair sets are illustrated with an example:

Illustration example

Let $\mathcal{P}_{\text{same}}$ and $\mathcal{P}_{\text{diff}}$ be the set of all image pairs with equal and different identity, respectively. Furthermore, assume a dataset $\mathcal{D}$ consisting of **four** unique IDs, where each ID has an arbitrary amount of unique images. The dataset can be visualized as Figure 35 illustrates:

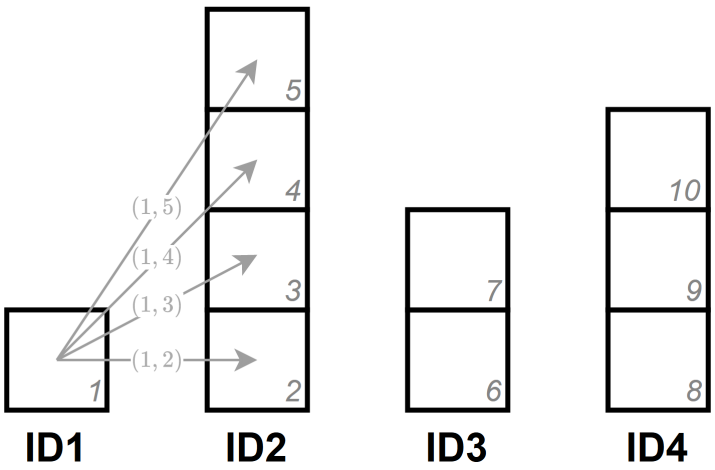


Figure 35: Figure 22 re-purposed to illustrate image pairs. The arrows show image pairs that are compared against each other.

The labels connected to the arrows can be interpreted as matrix entries with positions (i, j) of a big similarity matrix. The dataset in Figure 35 would have a similarity matrix as Figure 36 depicts, denoting 0 as a genuine pair and 1 as an impostor pair:

$$Y = \begin{pmatrix} \bar{0} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \bar{0} & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & \bar{0} & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & \bar{0} & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & \bar{0} & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & \bar{0} & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & \bar{0} & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & \bar{0} & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & \bar{0} & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \bar{0} \end{pmatrix}$$

Figure 36: The similarity matrix Y for the dataset illustrated in 35. Note that $\bar{0}$ denotes an image-image comparison where both images are identical, more specifically a self-similar image pair.

Note 4.5. ***This similarity matrix is ultimately what the models try to predict.***

The observant reader will have noticed that Y is symmetrical, and consequently only the upper triangular part of Y is required to acquire $\mathcal{P}_{\text{same}}$ and $\mathcal{P}_{\text{diff}}$, as comparing pair (i, j) gives the same outcome as comparing pair (j, i). For completeness, let the whole matrix be part of the upcoming definitions.

Definition 4.1. The set of all genuine image pairs $\mathcal{P}_{\text{same}}$, is defined as:

$$\mathcal{P}_{\text{same}} = \{(i, j) \mid y_{i,j} = 0,\ i \neq j\} \tag{95}$$

Definition 4.2. The set of all impostor image pairs $\mathcal{P}_{\text{diff}}$, is defined as:

$$\mathcal{P}_{\text{diff}} = \{(i, j) \mid y_{i,j} = 1,\ i \neq j\} \tag{96}$$

A useful number to figure out would be the set sizes of $\mathcal{P}_{\text{same}}$ and $\mathcal{P}_{\text{diff}}$, and is as follows:

Evaluation cardinality

Recall that pairing a sample from an ID with another sample from that same ID, is defined as a genuine pair. By examining the similarity matrix from Figure 36, it is apparent that the number of genuine pairs to an ID follows the same upper triangular matrix principle that defined the pair cardinality $|\mathcal{P}|$ from Equation (77). Because IDs can have varying samples, the definition needs to be defined as a sum:

$$|\mathcal{P}_{\text{same}}| = \sum_{i=1}^{I} \frac{S_i\,(S_i - 1)}{2} \tag{97}$$

For completeness, the cardinality of different pairs is defined:

$$|\mathcal{P}_{\text{diff}}| = |\mathcal{P}| - |\mathcal{P}_{\text{same}}| \tag{98}$$

It is noteworthy that zero/few-shot classification problems usually have a dataset distribution where $|\mathcal{P}_{\text{same}}| \ll |\mathcal{P}_{\text{diff}}|$. With this in mind, it is natural to discuss the case of defining a *similarity measure* to produce the binary labels $\hat{y} = \{0, 1\}$. Furthermore the *confusion matrix* is reviewed in order to define proper *performance measures* to evaluate the predicted similarity matrix $\hat{Y}$.

4.5.1 Euclidean similarity measure ($\mathcal{S}_E$)

In reality there are many different distance/similarity metrics, and one is not guaranteed to work better than the other for every problem. This thesis applies euclidean distance as its distance function $\mathcal{D}_E$ and similarity measure $\mathcal{S}_E$, mostly because the origin paper of Triplet Loss [40] employs euclidean distance.

Becuase the image embeddings are trained through some form of distance function/measure $\mathcal{D}$ (see section 4.2.1, 4.2.2), a natural way of evaluating the embedded image pairs will be to use that same distance function/measure that was used during training. As such, this thesis uses the Euclidean distance function $\mathcal{D}_E$ for training, and will use the this same metric for evaluating image pairs. The Euclidean similarity metric $\mathcal{S}_E$ is described as follows:

$$\mathcal{S}_E\,(f(X_1), f(X_2)) = \|f(X_1) - f(X_2)\|_2 \tag{99}$$

$$= \sqrt{[f_1(X_1) - f_1(X_2)]^2 + [f_2(X_1) - f_2(X_2)]^2 \ + \cdots + [f_D(X_1) - f_D(X_2)]^2} \tag{100}$$

$$= \sqrt{\sum_{d=1}^{D} [f_d(X_1) - f_d(X_2)]^2} \tag{101}$$

Note 4.6. *As the model is trained to map an image to an embedding space, relating new unlabeled points in the embedding space becomes a clustering problem. In approaching a real-life application, there will be a need to continuously perform and maintain clustering schemes, which is out of the scope of this thesis.*

Thresholding $\mathcal{S}_E$ by a value τ is the key to evaluating the models, and is by definition a similarity measure scheme $\mathcal{S}$, which determines if an image pair (X_1, X_2) are to be predicted as an *impostor pair*, or a *genuine pair*:

$$\mathcal{S}(X_1, X_2) = \begin{cases} 0, & \mathcal{S}_E(f(X_1), f(X_2)) \leq \tau \\ 1, & \mathcal{S}_E(f(X_1), f(X_2)) > \tau \end{cases} \tag{102}$$

This thesis selects a number of 500 thresholds evenly spaced in the range of the lowest possible threshold $\min_{i,j} \mathcal{S}_E(X_i, X_j)$, and the highest possible threshold $\max_{i,j} \mathcal{S}_E(X_i, X_j)$ in the evaluation dataset. This is chosen to search through the thresholds in a comprehensive manner.

4.5.2 Confusion matrix

As previously stated, evaluating the model on image pairs is the key to measure its performance. This section covers the performance measures applied in the thesis, and aims to cover their strengths and weaknesses.

Recall that $\mathcal{S}$ is a scheme to determine if two embeddings are equal or different, where $\hat{y}_{i,j} = \mathcal{S}(f_i, f_j) = 0$ denotes *genuine pair*, and $\hat{y}_{i,j} = \mathcal{S}(f_i, f_j) = 1$ denotes *impostor pair*. The concept of a *confusion matrix* is first introduced, and some important performance measures are followingly derived from it:

As the model aims to produce binary results in the form of $\hat{y} = \{0, 1\}$, a binary confusion matrix is helpful as Figure 37 illustrates:

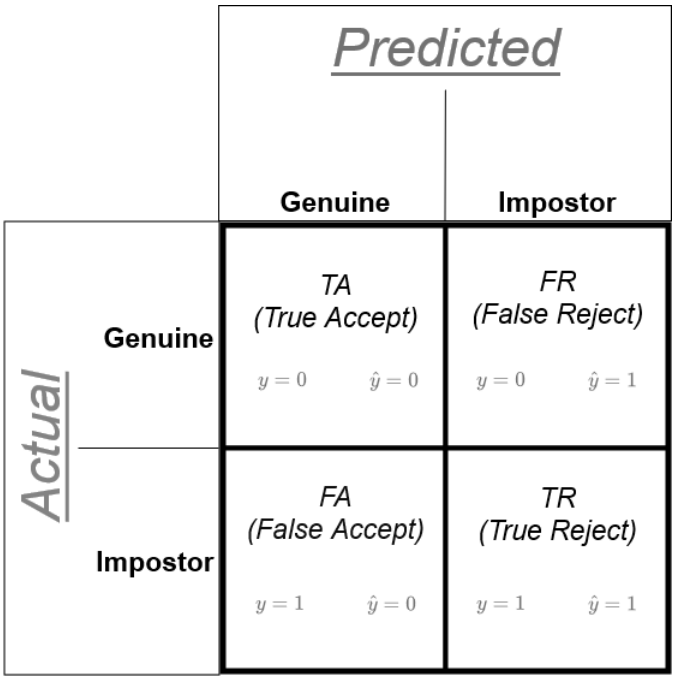


Figure 37: Confusion matrix for the problem of deciding if image pairs belong to the same ID.

The components of the confusion matrix are outlined as follows:

Definition 4.3. The set of all *true accepts*, given a similarity measure scheme between two images $\mathcal{S}(i,j)$ is defined as:

$$TA(\mathcal{S}) = \{(i,j) \in \mathcal{P}_{\text{same}} \mid \hat{y} = \mathcal{S}(i,j) = 0\} \tag{103}$$

Conversely, if two images are different and the similarity scheme $\mathcal{S}$ misclassifies two images (i,j) to be *genuine*, it belongs to the set of *false accepts*:

Definition 4.4. The set of all *false accepts*, given a similarity measure scheme between two images $\mathcal{S}(i,j)$ is defined as:

$$FA(\mathcal{S}) = \{(i,j) \in \mathcal{P}_{\text{diff}} \mid \hat{y} = \mathcal{S}(i,j) = 0\} \tag{104}$$

The other elements of the confusion matrix from Figure 37 are defined as follows:

Definition 4.5. The set of all *false rejects*, given a similarity measure scheme between two images $\mathcal{S}(i,j)$ is defined as:

$$FR(\mathcal{S}) = \{(i,j) \in \mathcal{P}_{\text{same}} \mid \hat{y} = \mathcal{S}(i,j) = 1\} \tag{105}$$

Definition 4.6. The set of all *true rejects*, given a similarity measure scheme between two images $\mathcal{S}(i,j)$ is defined as:

$$TR(\mathcal{S}) = \{(i,j) \in \mathcal{P}_{\text{diff}} \mid \hat{y} = \mathcal{S}(i,j) = 1\} \tag{106}$$

4.5.3 Performance measures

This subsection covers and justifies some standard performance measures originating from the confusion matrix, along with the performance measures used in this thesis.

Accuracy/Precision/Recall/F1-score

From the confusion matrix' components, comes a number of various performance measures. A commonly used performance measure from the confusion matrix is *accuracy*:

Definition 4.7. The number of all correct predictions, divided by the number of possible predictions is defined as *accuracy* (see Figure 36), and can be derived from the confusion matrix:

$$accuracy = \frac{TA + TR}{TA + FA + TR + FR} \tag{107}$$

Note 4.7. *Accuracy tends to work well when the data is evenly distributed, i.e. there are about as many genuine pairs as the are impostor pairs. However, if there were 1 genuine pair for every 99 impostor pairs, a majority label classifier would get 99% accuracy. Accuracy as a performance measure for the tasks in this thesis will therefore be highly misleading, as the real goal of a good classifier is to be able to correctly predict genuine pairs as well.*

As previously discussed, there is a good chance any randomly chosen image pair is labeled as an impostor pair (see section 4.5). Due to this, there is a significant need to monitor the number of *TAs*, *FRs* and *FAs* in the evaluated confusion matrix. To do this, the *precision* and *recall* performance measures are introduced.

Definition 4.8. The performance measure *precision* requires the number of *TAs* and *FAs*, and is defined as:

$$precision = \frac{TA}{TA + FA} \tag{108}$$

Definition 4.9. The performance measure *recall* requires the number of *TAs* and *FRs*, and is defined as:

$$recall = \frac{TA}{TA + FR} \tag{109}$$

Note 4.8. *Precision and recall as performance measures have the strengths in that each measures a hope one might have for the predictions. If the quality of the model's ability to predict genuine pairs is critical, the precision measure is a natural choice. If, however, the quality of the model's ability to predict impostor pairs is wanted, the recall measure is more a more appropriate choice.*

A mixture of *precision* and *recall* yields a more comprehensive picture of a model's ability to predict/mistake image pairs as genuine. The F1-score arrives from these expressions, and is defined as follows:

Definition 4.10. The performance measure F1-score requires the number of *TAs*, *FAs* and *FRs*, and is defined as:

$$\text{F1-score} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{110}$$

Remark 4.4. *The thesis utilizes F1-score as one of its performance measures as it balances false positives/negatives into its value, whilst not being affected by large amounts of impostor pairs. The downside to the F1-score is that it weights the recall and precision equally, which is unwanted in some prediction tasks (e.g. predicting a person to have cancer that needs expensive treatment).*

True Accept Rate/False Accept Rate

When dealing with a dataset that have so few genuine pairs compared to impostor pairs, it is important to monitor the models' genuine pair predictions, i.e. their true/false accepts. The following definitions cover the concepts of True Accept Rate (TAR) and False Accept Rate (FAR), and result from definitions derived from the confusion matrix (see section 4.5.2):

Definition 4.11. The *true accept rate* builds on Definition 4.3 by averaging Equation (103) by the number of genuine pairs $\mathcal{P}_{\text{same}}$:

$$TAR(\mathcal{S}) = \frac{|TA(\mathcal{S})|}{|\mathcal{P}_{\text{same}}|} \tag{111}$$

Definition 4.12. The *false accept rate* builds on Definition 4.4 by averaging Equation (104) by the number of impostor pairs $\mathcal{P}_{\text{diff}}$:

$$FAR(\mathcal{S}) = \frac{|FA(\mathcal{S})|}{|\mathcal{P}_{\text{diff}}|} \tag{112}$$

As Figure 36 suggests, there are many more *impostor pairs* than *genuine pairs*, meaning it would be much more impressive if the model correctly predicts the genuine pairs (true accepts), whilst keeping the number of impostor pairs that have been misclassified as genuine (false accepts), low. A natural way of co-measuring the TAR and FAR, is by examining their Receiver Operating Characteristics (ROC) curve.

Note 4.9. *The True Accept Rate is the same as recall (see Equation (109)), and is also referred to as "sensitivity" in some literatures. The False Accept Rate can be viewed as* $1 - specificity$, *with* $specificity = \frac{TR}{TR+FA}$.

4.5.4 Receiver Operating Characteristics curves (ROC)

When attempting to find an optimal threshold for some similarity measure $\mathcal{S}$, *there is always the question of "which threshold is the best". Building on definitions 4.11 and 4.12 from subsection 4.5.3, a potential performance measure to quantify this will be the area under curve (AUC) of the receiver operating characteristics (ROC) curve. This subsection explains briefly the meaning of these*

As there is not always a clear answer as to what the optimal threshold for similarity should be, one can instead attempt to gradually increase the threshold, and monitor the TAR and FAR of each threshold. By choosing thresholds ranging from the lowest distance to the largest distance between two points, there will be a range of TAR and FAR values. Plotting these against each other results in an ROC curve as Figure 38 illustrates:

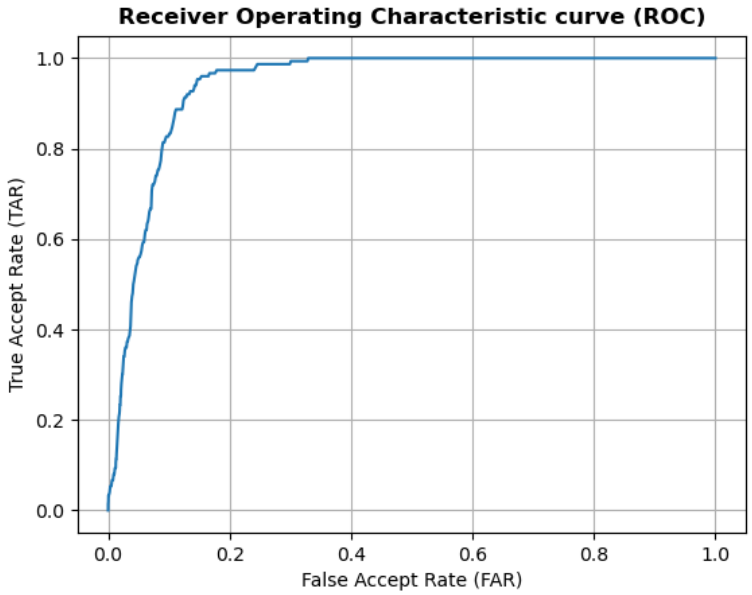


Figure 38: An example ROC curve attempting to showcase how one can monitor the trade-off between TAR and FAR, whilst also keeping an eye on the similarity thresholds. In this specific figure, interpreting the figure reveals that all the genuine pairs will be correctly classified, if the user accepts a 30% chance of falsely accepting an impostor pair.

In inspecting the ROC curve, there is now a clear picture as to the percentage of false accepts one is willing to sacrifice for a certain percentage of true accepts. With this curve, one sets an a priori value for FAR, and the corresponding TAR value is read off the ROC curve.

A downside to this approach is that the model's inability to classify an image pair as genuine is not represented (i.e. false rejects).

Note 4.10. *The FAR does not pose a continuous nature, so the FAR closest to 0.01 for the thresholds τ is picked to be the index for choosing the corresponding TAR value, hence the formulation TAR@FAR(0.01).*

5

Experiments and results

5.1 Experiment setup

In order to measure similarity between images, the models are set up as siamese neural networks, as section 4.1 motivates and justifies. Regarding the models, they are variations of three different off-the-shelf model architectures, namely LeNet5-var, MobileNetV3-var and ResNet18-var (see subsection 4.4.1 for a comprehensive overview). These models are then trained and stored, as the following paragraph elaborates. Further, the trained models are then employed in an attempt to answer *RQ1* and *RQ2*, in which sections 5.2-5.5 demonstrates and discusses the results.

The training pipeline

This paragraph will attempt to summarize the pipeline of training the models and evaluating them, in which the theory required to specifically build this pipeline is presented in Chapter 4. The pipeline components are neatly unpacked in a mostly chronological fashion, where the relevant sections are referred to:

1 Datasets are instantiated

a) Split the datasets into training, validation and testing datasets for the *zero-shot learning problem* (see subsection 4.3.1)

b) Standardize the datasets with the mean and standard deviation calculated from the training dataset (see subsection 4.3.4)

2 The training loop calls for minibatches of triplets

a) Select valid triplets to build a minibatch (see subsection 4.3.6)

b) Augment the images in the triplets (see subsection 4.3.7)

c) (Polar-transform the image if the model is supposed to train on such images, see subsection 4.3.8)

d) Run each image into the same model to produce image embeddings (see subsection 4.4.1)

e) Normalize the embeddings, perform triplet selection on the embeddings of the minibatch, and evaluate these embeddings by Triplet Loss (see subsections 4.2.2 and 4.2.3, and remark 4.1)

f) Adjust the parameters through backpropagation and the ADAM optimizing scheme (see sections 3.4 and 3.5)

An illustration of the training part of the pipeline is demonstrated by Figure 39:

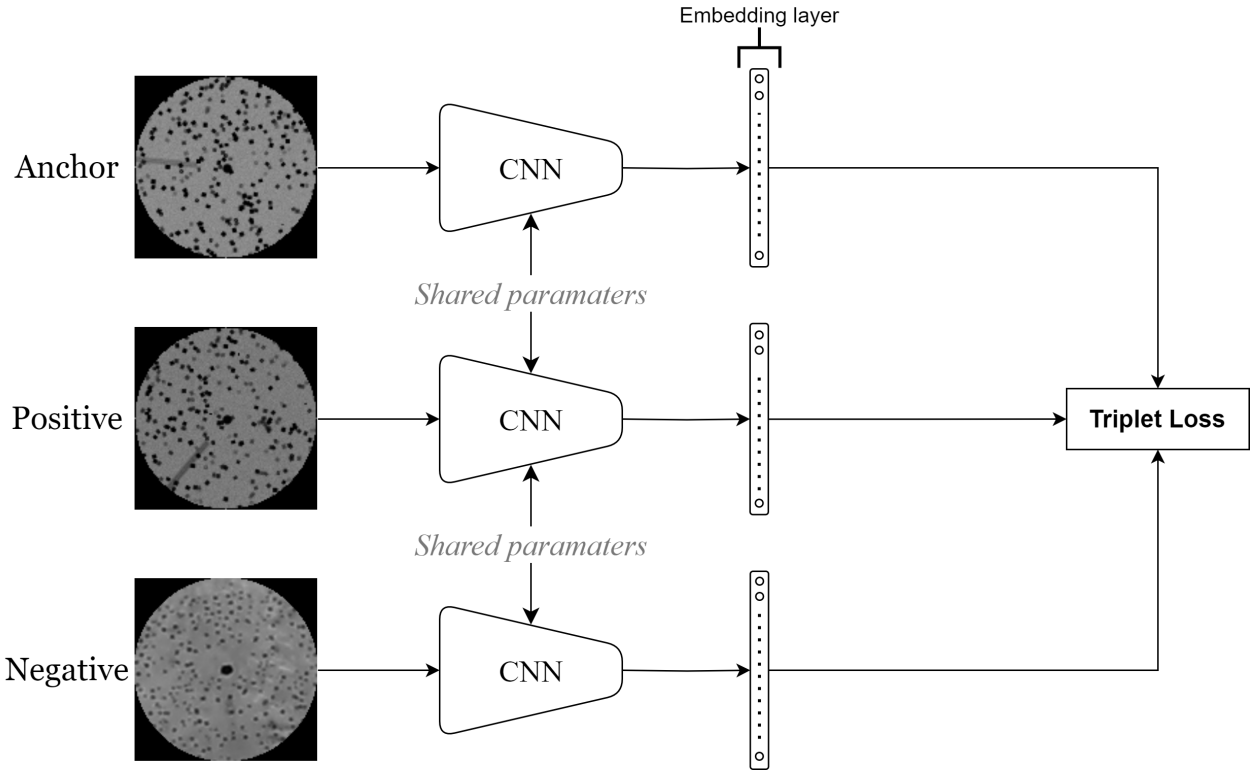


Figure 39: A flowchart illustration of a siamese neural network employing the Triplet Loss function to train the encoding network to produce meaningful embedding vectors. Note that steps such as normalization and triplet selection are missing in the figure.

3 Evaluation of the model performance in each epoch

a Produce normalized embeddings of all the images in the evaluation set (see subsection 4.4.1)

b Threshold each pair of embeddings by the Euclidean similarity measure $\mathcal{S}_E$ to obtain a predicted similarity matrix (see subsection 4.5.1)

c Evaluate the predicted similarity matrix and the true similarity matrix by appropriate performance measures, which in the case of this thesis are F1-score and TAR@FAR(0.01) (see subsections 4.5.2, 4.5.3, 4.5.4, and note 4.10).

d For each epoch, store the model parameters and evaluate the model on the validation dataset. The model parameters that yield the best validation performance are saved as the model's best parameters.

This pipeline elaboration briefly summarizes how one can acquire meaningful performances of a given set of hyperparameters, which will subsequently be the next paragraph to cover.

On the notion of hyperparameters

In the field of deep learning, the number of hyperparameters tends to grow proportionally with the complexities of the models and datasets, and the number of unique research questions. To keep the experiment results fairly compatible with each other, a choice has been made to

explore the hyperparameter of **which network** is being used to produce image embeddings, and **which convolution** is being used in the `Conv`-layers. As such, this paragraph is intended to act as a lookup table for tuneable hyperparameters that are *set* in the works of this thesis. The listed citations act as references for choosing these values in particular, whereas citation-less hyperparameters are justified accordingly (hyperparameters marked with * are additionally justified):

Fixed hyperparameters		
Hyperparameter name	Value	Citation(s)
Triplet margin (α)	0.2	[40]
Embedding dimensionality	128	[40], [28]
Optimizer	ADAM(`lr=0.0005`)	-
Number of epochs	50	-
Evaluation protocol	Hold-out validation	-
Train/hold-out ratio (synthetic)	$\frac{40}{75}/\frac{15}{75}/\frac{20}{75}$	-
Train/hold-out ratio (real)	$\frac{55}{75}/\frac{20}{75}/\frac{-}{75}$	-
Batch size	128	-
FAR-threshold	10^{-2}	[40], [28]*
Evaluation augmentations	100	-
Synthetic samples per ID	4/1	-

Table 1: Hyperparameter lookup table for the experiments in the thesis. Their values are set due to experimentation and/or other works having success with them.

Optimizer: The ADAM optimizer with the corresponding learning rate has worked well when experimenting and building the training pipeline. This optimizer is by no means guaranteed to be the best for each model, and serves to be an example of optimizer options that facilitates training and model performance to some extent.[26]

Number of epochs: As alluded to in the sections regarding pair cardinality and triplet cardinality (see subsections 4.3.5 and 4.3.6), the number of possible image combinations can become quite large. A number of 50 triplet epochs was set to be managable by the hardware setup that was available.

Evaluation protocol: Neural network pipelines are known to be computationally heavy, meaning cross-validation would be infeasible to consider for evaluation. An downside of applying hold-out validation worth mentioning, is that the real dataset is scarce. This means that the

[26]In regards to the ADAM optimizer, the PyTorch default values are set for the regularization hyperparameters. More details can be found in the PyTorch documentation at `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`

validation performance will be less representative of the model's true performance. If time consumption was not of significance, cross-validation could have been a more optimal choice for the given datasets.

Train/validation/test ratio (synthetic): The training fraction is set to acquire 40 IDs with multiple samples, as the real dataset also has 40 IDs of multiple samples. The validation and test fractions are split up from the number of IDs from the real dataset that has only 1 sample. By examining the corresponding validation and test datasets, they each contain cardinalities $|\mathcal{P}_{\text{same}}|$ as follows, by Equation (97):

$$|\mathcal{P}_{\text{same-val}}| = 15 \cdot \frac{4 \cdot (4-1)}{2} = 90 \tag{113}$$

$$|\mathcal{P}_{\text{same-test}}| = 20 \cdot \frac{4 \cdot (4-1)}{2} = 120 \tag{114}$$

For evaluation purposes, a number of approximately 100 genuine pairs for each evaluation dataset was deemed to be appropriate. The slight increase of test data points is set such that the selected best model has a performance that is less affected by performance deviations.

Train/validation/test ratio (real): Because the unlabeled images may contain several images of the same ID, they can only be used as negative images for pairing labeled anchor-positives. Further, a choice by design has been to have IDs that have 1, 3 and 4 samples be part of the training dataset. Because image augmentation of evaluation data effectively make infinitely many unique images to an ID, the validation dataset consists exclusively of a number of 2 truly unique images of an ID.

FAR-threshold: The datasets suffer from scarcity, which makes any TAR evaluation at low FAR values prone to vary. A value of 10^{-2} has been set in this thesis, as a genuine pair "trustworthiness" of 99 % is adequate for the purposes of this thesis due to the evaluation dataset cardinality pointed out by Equation (113) and (114)

Note 5.1. *Although IDs with 3-4 unique images are valuable to both the training dataset, and the validation dataset, they were allocated to the training dataset to increase the number of truly unique anchor-positive pairs. By re-visiting the section of triplet cardinality (subsection 4.3.6), it is apparent from the $S_i\,(S_i - 1)$ term (i.e. the number of samples in ID i) that adding samples to an ID significantly increases the triplet dataset.*

Batch size: As a rule of thumb, the batch size is set to be an integer power of 2. Setting it to 128 was what the hardware setup and programming pipeline at disposal was capable to stably work with.

Evaluation augmentations: A number of 100 evaluation augmentations where deemed appropriate to acquire a sufficient distribution of performances. Although more augmentations is not harmful, a choice was made as any more that 100 evaluation augmentations made some of the results take a long time to produce.

Synthetic samples per ID: Because adding samples to an ID significantly increases the size of the dataset (see section 4.3.6 for details about $|\mathcal{T}|$), a choice was made to have each synthetic ID only have 4 samples. This is also based on the real dataset distribution, where one ID in fact has 4 samples.

NOTE: The dataset hyperparameters responsible for producing the synthetic dataset were set by empiric observation, and they are found in the folder `hyperparameters`*, in the file* `DATASET_HPs.py` *in the GitHub repository.*

Software and hardware

Software: This thesis uses the PyTorch library [34] for setting up and executing model experimentation. `CyConv`-layers [21] in CUDA [31] are implemented through `https://github.com/mcrl/CyCNN`, with the addition of groups being developed by the author of this thesis. For image rotation and polar image conversion, the OpenCV [6] library is used.

Hardware: The experiments of this thesis were performed on an *MSI GP66 Leopard* laptop, with the following specifications:

- CPU: Intel Core i7-10870H
- GPU: NVIDIA GeForce RTX 3080 Laptop (VRAM: 8 GB)

Transfer learning

Before moving on the results, a final subsection is introduced regarding the notion of transfer learning, as the models in this thesis are pretrained on the synthetic dataset prior to being trained on the real dataset.

Transfer learning [12] is a powerful technique in the field of machine learning that allows models to gain knowledge from one dataset domain, and apply it to another dataset domain, often resembling the domain the model was initially trained on. It offers a way to approach the challenge of scarce datasets, while also potentially accelerating the training process.

More specifically, transfer learning is performed by "pretraining" a model's parameters on a source dataset, typically consisting of a relatively large amount of labeled data. Once the model has been trained on a sufficient source dataset, these pretrained parameters can be used as initial parameters in learning to target dataset. In the field of computer vision, transfer learning is especially useful in learning low level features such as shapes and edges.

In regards to this thesis, the synthetic dataset can be made arbitrarily rich with data. Pretraining the models on the synthetic dataset could then prove useful in learning the real dataset parameters.

5.2 Model results (*RQ1*)

Referring back to the problem formulation (see section 2.1), there are two main *Research Questions* to answer. This section and the next focuses on *RQ1* and will, with knowledge gained from this question, attempt to answer *RQ2*, found in sections 5.4 and 5.5. For the sake of readability, *RQ1* is restated here:

Will zero-shot classification by siamese neural networks be able to correctly relate salmon louse images with similar IDs to a satisfactory degree?

To answer this question, the model architectures presented in section 4.4.1 are employed and trained on the **synthetic dataset** and the **real dataset**[27] to produce a number of 100 validation performances[28]. The validation performances are illustrated in representative box-plots as a way of demonstrating the variation in model performance. Accompanying the box-plots of the validation performance results are the corresponding thresholds τ responsible for the performances on the synthetic dataset (Tables 2 and 3) and the real dataset (Tables 4 and 5).

In regards to the dataset hyperparameters, the images fed in are randomly rotated in the range of $\mathcal{R} \in [-180°, 180°]$ unless explicitly stated otherwise. The hyperparameters to reproduce the results of the pipeline are found in the folder named `hyperparameters` in the GitHub repository.

Note 5.2. *A bug was discovered late in the works of the thesis in that the thresholds τ were output incorrectly. To fix this, all the models would have to be retrained to acquire the performance results and corrected threshold values. As a remedy, the performance results and threshold values were recalculated for the models that performed the best on the validation dataset during the training phase (recall that all the models are performance evaluated at each completed epoch). The script that employs these best models is named* `results_script.py` *in the GitHub repository, and is responsible for plotting the results of RQ1 and RQ2.*

[27] Tying in the notion of transfer learning, the model parameters that yield the best *average* performance on the synthetic dataset, are set as initial parameters for training the models on the real dataset.

[28] All validation datasets are augmented as described in section 4.3.7.

Synthetic dataset results

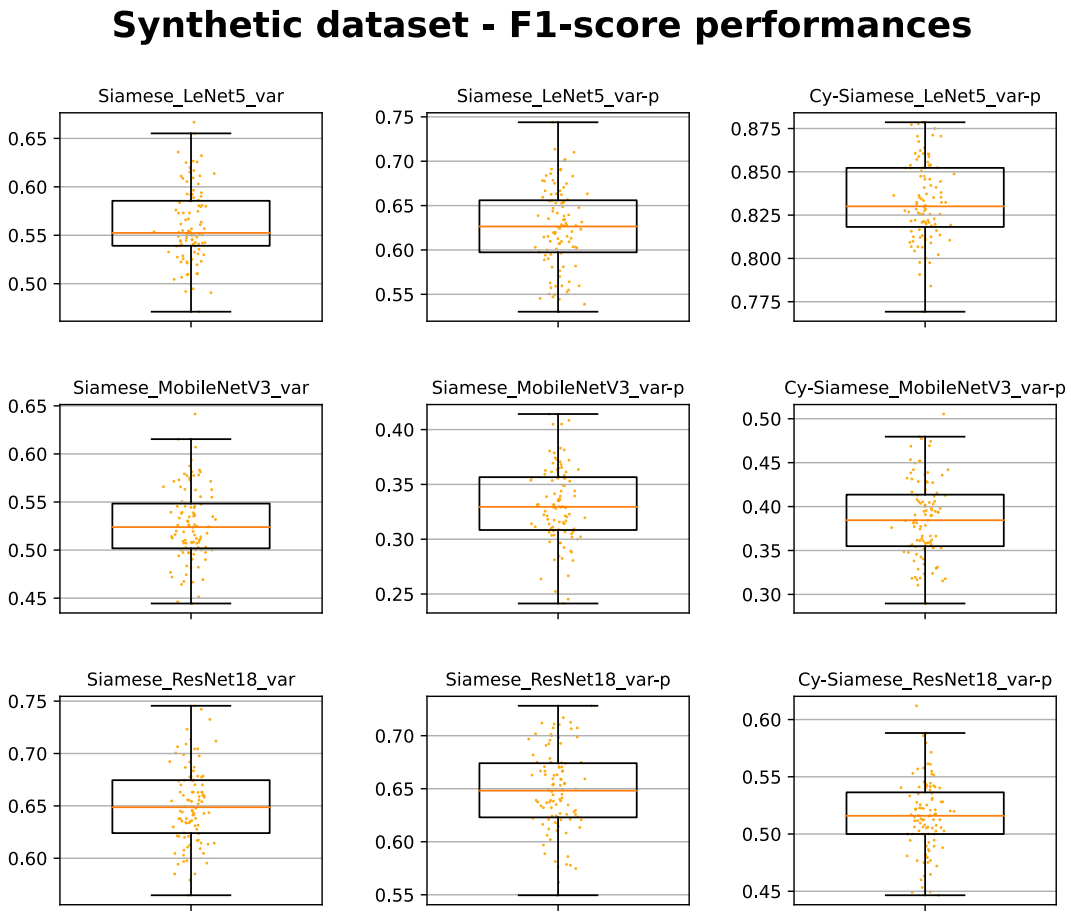


Figure 40: Box-plots of 100 validation F1-score performances on the synthetic dataset, where the title of each box-plot indicates the responsible model. One orange dot corresponds to one validation performance acquired by randomly augmenting the validation data, and the orange line corresponds to the median of these validation performances. Note that the orange dots are horizontally displaced only for readability purposes. The reader is also reminded to pay attention to the y-axis scale in these plots.

Synthetic dataset - F1-score thresholds (τ)			
Model names	-	*-p*	`Cy-`, *-p*
`Siamese_LeNet5_var`	0.5448 ± 0.0405	0.7432 ± 0.0343	0.6747 ± 0.0276
`Siamese_MobileNetV3_var`	0.3944 ± 0.0576	0.4836 ± 0.0920	0.5624 ± 0.0761
`Siamese_ResNet18_var`	0.4393 ± 0.0367	0.5626 ± 0.0441	0.4573 ± 0.0433

Table 2: Statistics of the thresholds τ responsible for the validation F1-score performances represented as orange dots in Figure 40 are presented in this table in terms of the mean and standard deviation of the thresholds.

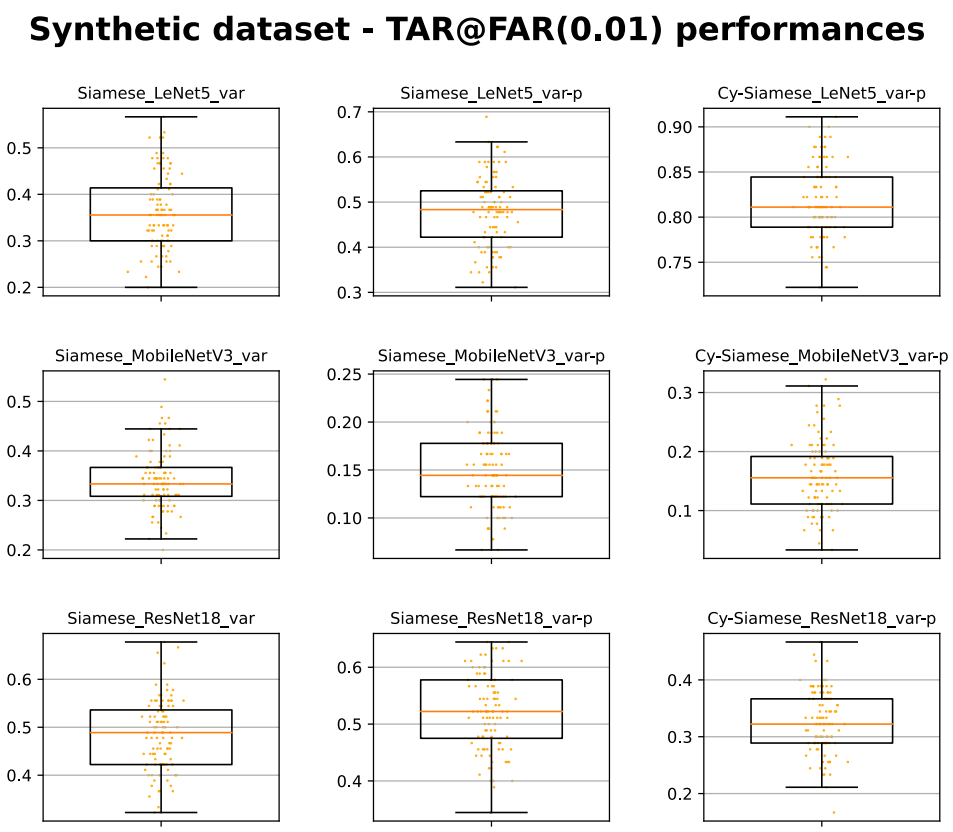


Figure 41: Box-plots of 100 validation TAR@FAR(0.01) performances on the synthetic dataset, where the title of each box-plot indicates the responsible model. One orange dot corresponds to one validation performance acquired by randomly augmenting the validation data, and the orange line corresponds to the median of these validation performances. Note that the orange dots are horizontally displaced only for readability purposes. The reader is also reminded to pay attention to the y-axis scale in these plots.

Synthetic dataset - TAR@FAR(0.01) thresholds (τ)			
Model names	-	*-p*	`Cy-`, *-p*
`Siamese_LeNet5_var`	0.4310 ± 0.0270	0.6801 ± 0.0295	0.6938 ± 0.0190
`Siamese_MobileNetV3_var`	0.2755 ± 0.0246	0.2740 ± 0.0303	0.3140 ± 0.0326
`Siamese_ResNet18_var`	0.3536 ± 0.0282	0.4970 ± 0.0278	0.3388 ± 0.0243

Table 3: Statistics of the thresholds τ responsible for the validation TAR@FAR(0.01) performances represented as orange dots in Figure 41 are presented in this table in terms of the mean and standard deviation of the thresholds.

In reviewing the box-plots illustrated in Figures 40 and 41, and their corresponding threshold statistics represented in Tables 2 and 3, the best performing model on the synthetic validation dataset was `Cy-Siamese_LeNet5_var-p`. In regards to F1-score, the median validation performance was 83.0%. In regards to TAR@FAR(0.01), the median validation performance was 81.1%.

Real dataset results

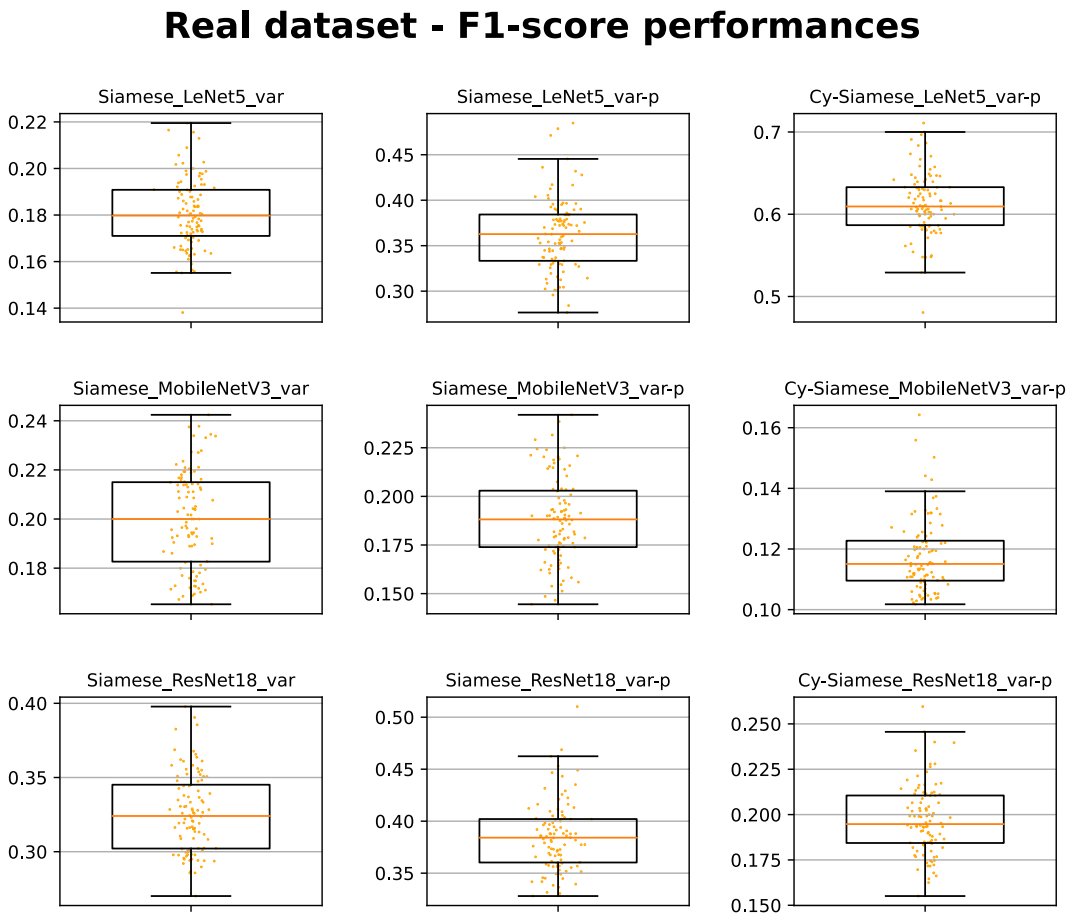


Figure 42: Box-plots of 100 validation F1-score performances on the real dataset, where the title of each box-plot indicates the responsible model. One orange dot corresponds to one validation performance acquired by randomly augmenting the validation data, and the orange line corresponds to the median of these validation performances. Note that the orange dots are horizontally displaced only for readability purposes. The reader is also reminded to pay attention to the y-axis scale in these plots.

Real dataset - F1-score thresholds (τ)			
Model names	-	*-p*	`Cy-`, *-p*
`Siamese_LeNet5_var`	0.4398 ± 0.0697	0.6680 ± 0.0505	0.5543 ± 0.0275
`Siamese_MobileNetV3_var`	0.4000 ± 0.0952	0.1109 ± 0.0271	0.4349 ± 0.2272
`Siamese_ResNet18_var`	0.2154 ± 0.0291	0.4582 ± 0.0493	0.5054 ± 0.0881

Table 4: Statistics of the thresholds τ responsible for the validation F1-score performances represented as orange dots in Figure 42 are presented in this table in terms of the mean and standard deviation of the thresholds.

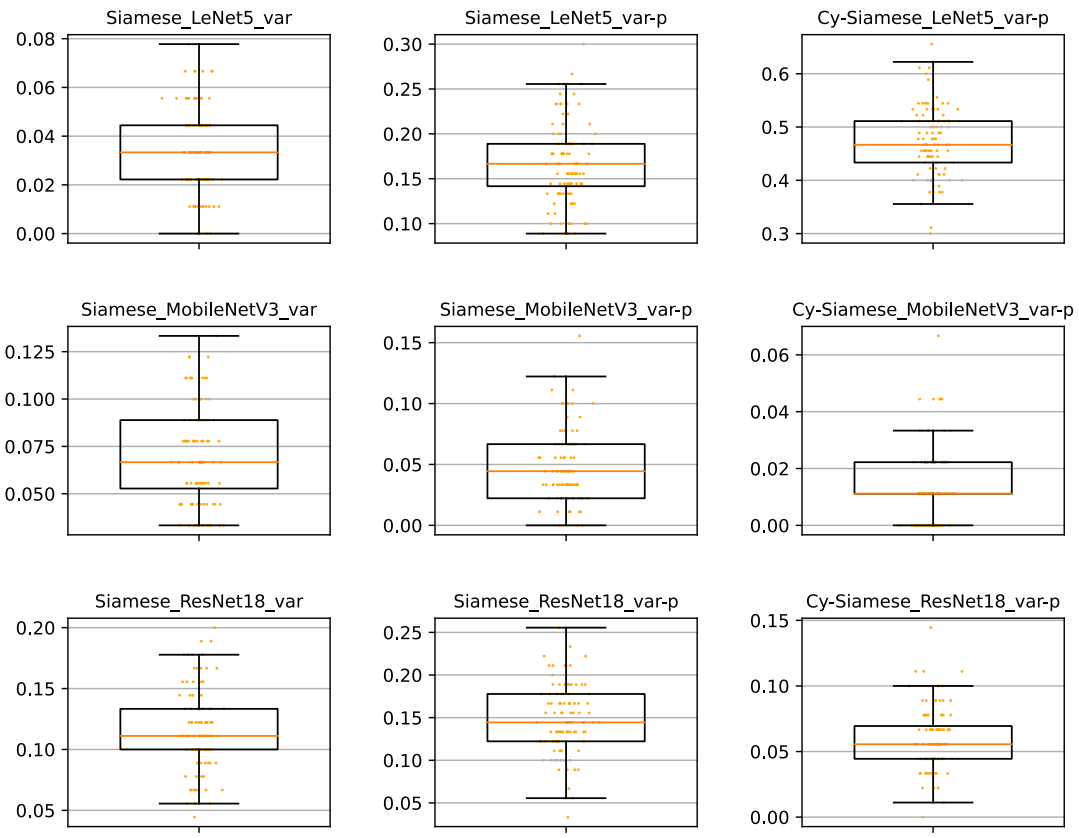


Figure 43: Box-plots of 100 validation TAR@FAR(0.01) performances on the real dataset, where the title of each box-plot indicates the responsible model. One orange dot corresponds to one validation performance acquired by randomly augmenting the validation data, and the orange line corresponds to the median of these validation performances. Note that the orange dots are horizontally displaced only for readability purposes. The reader is also reminded to pay attention to the y-axis scale in these plots.

Real dataset - TAR@FAR(0.01) thresholds (τ)			
Model names	-	*-p*	`Cy-`, *-p*
`Siamese_LeNet5_var`	0.2153 ± 0.0129	0.5238 ± 0.0241	0.5199 ± 0.0178
`Siamese_MobileNetV3_var`	0.1974 ± 0.0171	0.0520 ± 0.0048	0.0886 ± 0.0084
`Siamese_ResNet18_var`	0.1174 ± 0.0085	0.2831 ± 0.0177	0.2788 ± 0.0184

Table 5: Statistics of the thresholds τ responsible for the validation TAR@FAR(0.01) performances represented as orange dots in Figure 43 are presented in this table in terms of the mean and standard deviation of the thresholds.

In reviewing the box-plots illustrated in Figures 42 and 43, and their corresponding threshold statistics represented in Tables 4 and 5, the best performing model on the real validation dataset

was `Cy-Siamese_LeNet5_var-p`. In regards to F1-score, the median validation performance was 60.9%. In regards to TAR@FAR(0.01), the median validation performance was 46.7%.

Cy-Siamese_LeNet5_var-p on the synthetic testing dataset

The synthetic dataset contains enough images to properly produce testing performances, which is listed in this paragraph:

Using the mean threshold value $\tau = 0.6747$ (see Table 2), `Cy-Siamese_LeNet5_var-p` acquired a testing F1-score performance of $72.5 \pm 2.6\%$. For the testing TAR@FAR(0.01) performance, the mean threshold value $\tau = 0.6938$ (see Table 3) is set. Note that the @FAR(0.01) part is only used for model selection, such that by choosing τ, the testing performance cannot be guaranteed to yield a FAR of 0.01. Regardless, the testing TAR performance was found to be $72.8 \pm 3.0\%$.

5.3 Result discussion (*RQ1*)

5.3.1 Synthetic dataset results vs. real dataset results

In reviewing the synthetic dataset performance results (Figures 40 and 41), an interesting observation is that the customized LeNet5-version containing `CyConv` layers fairly outperformed the rest of the models, even beating every `CyConv` version of the more modern off-the-shelf models by a significant margin. As for the results of the real dataset, one can observe more or less the same thing as for the synthetic dataset.

Another observation to note, is that the models were clearly better when trained on and evaluated with a synthetic dataset as opposed to the real dataset. This might be attributed to some image attributes not covered by the synthetic dataset, but could also come from the scarcity of the real dataset. In any case, the performances on the real dataset are underwhelming, and stands to show that straightforward transfer learning with hyperparameters that worked fairly well for the synthetic dataset is not guaranteed to work well for the case of a real dataset.

5.3.2 Performance measures and thresholds

By inspecting the thresholds presented in Tables 2 and 3 in regards to the best performing models, there is a tendency that is worth elaborating on. When reviewing the F1-scores on the synthetic dataset, the thresholds tend to be a bit higher than when reviewing the TAR@FAR(0.01) results on the synthetic dataset. This indicates a clear difference between the performances measures of F1-score and TAR@FAR(0.01).

Being more specific, the F1-score performance measure is designed to accomodate both *false rejects* (i.e. the model predicts impostor pair when the label is genuine pair) and *false accepts* in finding an optimal threshold, whereas TAR@FAR(0.01) only considers *false accepts* in its thresholding[29]. A consequence of this is that the TAR@FAR(0.01) performance measure cuts the model a total of 1% "slack" when it comes to having *false accepts* in the predicted similarity matrix (i.e. @FAR(0.01)).

[29]Revisit subsection 4.5.4 and note 4.10 for a reminder of the nature of TAR@FAR(0.01).

In any matter, although the TAR@FAR(*"requested-percentage-of-false-accepts-allowance"*) performance measure guarantees a much more confident model, such a performance measure does not take into account *false rejects*. Although this is by design, the model's capabilities are somewhat dampened by not letting the thresholds expand to possibly include those image pairs that are falsely rejected for relatively low threshold values. Increasing the threshold naturally increases the chance of falsely accepting image pairs, such that setting up both F1-score and TAR@FAR(0.01), and reviewing their overall differences brings insight into what performance measure could be suited for such a task.

Continuing further on the notion of thresholds, is the choice of the similarity measure $\mathcal{S}$. As the thesis utilizes the Triplet Loss function developed in the FaceNet-paper [40], many of the design choices in the training pipeline are inspired from their works. The authors of the FaceNet-paper thresholds the L2-distance (i.e. Euclidean distance) between image embeddings, where the image embeddings are normalized to have a length of 1. This effectively scatters the image embeddings on the $D-$dimensional hypersphere, in which a baseline distance function such as Euclidean distance might not be the most suitable similarity measure. For future investigations, it would be interesting to explore the pipeline with some cosine similarity measure, which can be argued to be more appropriate in that the embeddings live on a hypersphere.

5.3.3 Model architectures

By reviewing the results, it is somewhat surprising that the fairly simple model `Siamese_LeNet5_var` contests with and/or outperforms variations of the more modern off-the-shelf models. It is also noteworthy that cylindrical convolution *significantly* accelerates the performance of the `Siamese_LeNet5_var`, clearly outperforming polar convolution. Contrastively, the models `Siamese_MobileNetV3_var` and `Siamese_ResNet18_var` experience little to no performance boost by polar and/or cylindrical convolution, and in most cases the performance is in fact worse when compared to baseline convolution.

There could a plethora of things that could contribute in explaining this. One thing that needs to be addressed, is the activation functions and pooling functions that the models use. LeNet5 employs the tanh() activation function along with average pooling in the intermediate layers, whereas MobileNetV3 and ResNet18 employ the relu() activation function along with max pooling. Also worth mentioning, is that both MobileNetV3 and ResNet18 both employ residual connections. In regards to the model performances and their general differences, this is worth mentioning for future work.

5.3.4 Triplet selection

Crucial to deal with prolonged training times is undoubtedly performing careful triplet selection, but it is a multi-sided coin. On the upside of triplet selection, selecting only triplets that contribute to training will effectively filter out triplets that do not contribute to training, as the Triplet Loss for these triplets are 0. In the trade of deep learning, saving on computational resources is highly favorable.

A downside to triplet selection that requires elaboration, is that the hardware setup needs to be somewhat powerful in order for triplet selection to be effective. Setting the batch size to be 128 is an *upper bound* for how many training samples are selected to be fed into backpropagation and optimization, and is a consequence of selecting triplets online. This has the consequence of downsizing the mini-batch size when performing mini-batch gradient descent schemes, which effectively makes the gradient steps more noisy as the model gets trained. Remedying this requires a more powerful hardware setup in order to increase the apparent batch size, and some observations regarding increased batch size are discussed in subsection 5.3.8.

Nevertheless, choosing the triplets to fulfill the semi-hard constraint as illustrated by Figure 21 is not guaranteed to be the best solution, and is justified by FaceNet-paper [40] finding success with the semi-hard constraint. The notion of triplet selection is important to consider when drawing any conclusions on the training pipeline, and is subject to future investigation.

5.3.5 Mislabeling

In the real world, there is always an opportunity for the real dataset itself to not have completely correct labels. Because verifying the labels of the real dataset is difficult for even humans by inspecting the images, the possibility of it being mislabeled and its consequences must be properly addressed.

Referring back to section 4.3.6 in regards to selecting triplets from the real dataset, there are some details about triplets that needs attending. For starters, unlabeled lice are not allowed to exist as an anchor-positive pair, as there is a chance of selecting a randomly sampled negative image that possesses the same ID as the anchor and positive image. Further, unlabeled lice can theoretically not be set as negative images either, due to the risk of mislabeling. If a labeling chip were to fall of a louse, that louse would go on to be categorized as unlabeled. Accidentally selecting this louse as a negative image, when it in reality belongs to a labeled set of lice samples, will to some extent compromise training.

Without assurances from the dataset labeling procedure that these cases will not occur, the unlabeled lice are practically rendered useless in the case of Triplet Loss. This is because they cannot be guaranteed to exist exclusively (i.e. the ID of one unlabeled image could be the same ID of another unlabeled image), and because unlabeled lice cannot be guaranteed to not be a part of the labeled lice.

In any case, this restricts training by potentially excluding perfectly valid triplets. In the situation of a scarce dataset, studying the performance of an SNN with Triplet Loss with mislabeling can be considered a potential research question for further investigation.

5.3.6 Influence of rotation

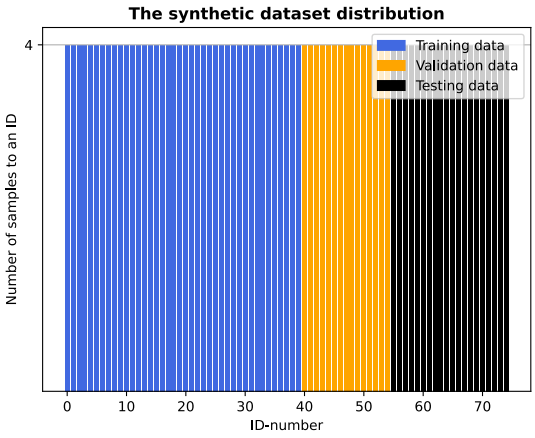
As the performance results in section 5.2 point out, the datasets are trained and evaluated on fully rotated images. In regards to the motivation to use polar convolution and cylindrical convolution, the point was to tackle rotated images. If one is to invalidate the variations of the modern off-the-shelf models as appropriate for salmon lice images due to their substandard results, some discussion can be made on the variation of the LeNet5-architecture.

Mainly, with the variation of LeNet5-architecture, the procedure of performing rotation augmentation along with convolution schemes that are designed to account for rotation, shows a significant increase in performance results. Due to this, rotation is deemed interesting to study further as part of ***RQ2***.

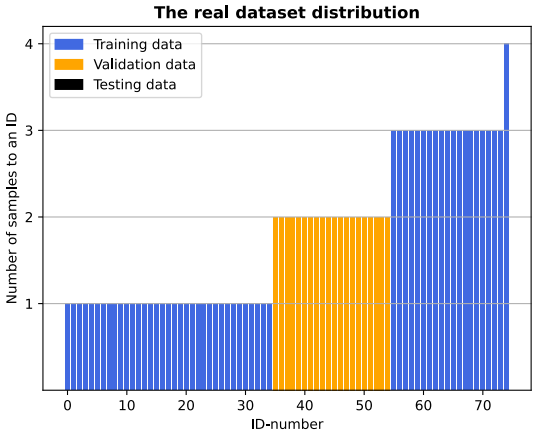
5.3.7 Regarding the real dataset

As alluded to in this thesis, the real dataset suffers from scarcity. This hurts both models' ability to train and how accurately their performances are. Training solely on the real dataset as described in the triplet cardinality section (section 4.3.6) yielded non-satisfactory results, and was attempted remedied by utilizing tricks such as image augmentations and transfer learning from the synthetic dataset.

An attempt to explain the disappointing results of the real dataset, can be made by looking at its distribution of IDs and samples, which is illustrated in Figure 44b alongside the distribution of the synthetic dataset (Figure 44a):



(a) Synthetic dataset distribution.



(b) Real dataset distribution. Note that images with unknown IDs are not plotted here for aesthetic purposes.

Figure 44: The distribution of IDs and samples of the synthetic and real distributions shown alongside each other.

Indicated by the legend labels in Figure 44, the synthetic dataset is far richer than the real dataset, both in terms of training and evaluation. In facilitating any model evaluation with the real dataset, a choice was made to let all IDs with 2 samples be a part of the validation dataset (as section 5.1 notes). Consequentially, a testing dataset is noticeably missing here, which has the implication of not yielding any definitive performance result of the best performing model on the real dataset.

Regardless, the performance results on the real validation dataset still provide a little bit of insight into what one can expect from the models. The distribution of the real dataset is deemed as an interesting attribute of salmon lice images, and will be investigated further as a part of ***RQ2***.

5.3.8 Inconclusive experimentations

While experimenting with the pipeline in this thesis, an immediate discovery was made in that the corresponding batch size set for mini-batch gradient descent mattered significantly in terms of performance. In some cases the performance was lifted up close to 10 percentage points by upping the batch size to 256 as opposed to the performance achieved through a batch size of 128. Experiments were also made by upping the batch size to 1024, which did not yield any significant performance boost as opposed to a batch size of 256.

Another observation that might be tied to the observations made by experimenting with the batch size, is that setting a different reproduction seed for the training and evaluation pipeline also made the performances vary. The variability of model evaluation is to be somewhat expected because of validation dataset augmentations and the scarcity of the datasets, and is implemented in the presentation of the performance results in the form of box-plots in Figures 40-43. Ideally, setting another pipeline seed should produce more or less the same results, but experimentation suggests otherwise, and could indicate that the current experimentation pipeline is sensitive in some way or another.[30]

In any event, future users of the experimentation pipeline will likely benefit from these experiences, but results here still remain inconclusive.

5.4 Model results (*RQ2*)

Referring back to the problem formulation (see section 2.1), this section addresses *RQ2*, which is restated here for the purpose of readability:

What attributes of salmon lice images have a significant effect on the performance of zero-shot classification?

Investigating *RQ2* is more tricky than *RQ1* because there is a vast landscape of image attributes that might affect the performance of zero-shot classification. As was made apparant from the resuts and discussion of *RQ1* in sections 5.2 and 5.3, the *influence of rotation* and the *dataset distribution* were regarded interesting to elaborate on, and will followingly be the topics of the results and discussions on *RQ2*.

Note 5.3. *The synthetic dataset will be utilized in addressing RQ2, as it can be readily altered, perturbed and otherwise tampered with.*

5.4.1 Influence of rotation

Baseline convolution models trained on $\mathcal{R} \in [-45°, 45°]$ ***and*** $\mathcal{R} \in [-5°, 5°]$

[30] Experimentation of the seed in the pipeline eventually uncovered a huge bug in the OpenCV implementation of polar transformation of images. As of writing, this is still an open issue at `https://github.com/opencv/opencv/issues/23701`, and is nothing more than a footnote in regards to this thesis.

A key dataset parameter is the images' ability to be rotated. As the synthetic dataset can be produced to be not rotated, a controlled performance analysis can be executed on this parameter (recall that section 5.2 contains representative results for fully rotated datasets where $\mathcal{R} \in [-180°, 180°]$). To get a sense of the implications of rotation, an experiment is performed on models employing *baseline convolution* (i.e. no polar transformation, and regular `Conv`-layers.), where the images are rotated in the ranges $\mathcal{R} \in [-45°, 45°]$ and $\mathcal{R} \in [-5°, 5°]$, with the results being presented by Table 6-9:

Performances on the synthetic dataset - $\mathcal{R} \in [-45°, 45°]$		
Model names	F1-scores	TAR@FAR(0.01)
`Siamese_LeNet5_var`	$45.6 \pm 4.3\%$	$26.8 \pm 6.4\%$
`Siamese_MobileNetV3_var`	$40.0 \pm 3.7\%$	$18.8 \pm 5.2\%$
`Siamese_ResNet18_var`	$58.6 \pm 3.46\%$	$38.2 \pm 6.5\%$

Table 6: Validation performance statistics of the models employing baseline convolutions (*mean* $\pm$ *std*). The rotation range $\mathcal{R} \in [-45°, 45°]$ indicates that these models are both trained on images rotated in this range, and evaluated on the validation dataset rotated in this range.

Thresholds τ ***on the synthetic dataset*** - $\mathcal{R} \in [-45°, 45°]$		
Model names	Threshold (F1-score)	Threshold (TAR@FAR(0.01))
`Siamese_LeNet5_var`	0.7104 ± 0.0504	0.5893 ± 0.0264
`Siamese_MobileNetV3_var`	0.4215 ± 0.0640	0.2433 ± 0.0251
`Siamese_ResNet18_var`	0.4187 ± 0.0440	0.3000 ± 0.0229

Table 7: Threshold statistics of the models employing baseline convolutions (*mean* $\pm$ *std*). The rotation range $\mathcal{R} \in [-45°, 45°]$ indicates that these models are both trained on images rotated in this range, and evaluated on the validation dataset rotated in this range.

Performances on the synthetic dataset - $\mathcal{R} \in [-5°, 5°]$		
Model names	F1-scores	TAR@FAR(0.01)
`Siamese_LeNet5_var`	$75.5 \pm 2.7\%$	$68.0 \pm 5.4\%$
`Siamese_MobileNetV3_var`	$38.4 \pm 4.1\%$	$17.0 \pm 5.4\%$
`Siamese_ResNet18_var`	$46.2 \pm 2.8\%$	$20.8 \pm 5.3\%$

Table 8: Validation performance statistics of the models employing baseline convolutions (*mean* $\pm$ *std*). The rotation range $\mathcal{R} \in [-5°, 5°]$ indicates that these models are both trained on images rotated in this range, and evaluated on the validation dataset rotated in this range.

Thresholds τ on the synthetic dataset - $\mathcal{R} \in [-5°, 5°]$		
Model names	Threshold (F1-score)	Threshold (TAR@FAR(0.01))
`Siamese_LeNet5_var`	0.7579 ± 0.0444	0.7234 ± 0.0233
`Siamese_MobileNetV3_var`	0.6423 ± 0.0784	0.3919 ± 0.0338
`Siamese_ResNet18_var`	0.4070 ± 0.0650	0.2064 ± 0.0224

Table 9: Threshold statistics of the models employing baseline convolutions (*mean* $\pm$ *std*). The rotation range $\mathcal{R} \in [-5°, 5°]$ indicates that these models are both trained on images rotated in this range, and evaluated on the validation dataset rotated in this range.

Models trained on $\mathcal{R} \in [-180°, 180°]$, evaluated on gradually rotated datasets

Another interesting result is how models that are trained on fully rotated datasets perform on a range of rotation intervals. The following results show all the models that performed the best on the synthetic dataset, employed on validation datasets that gradually rotate. The x-axis range will be from $\mathcal{R} \in [-0°, 0°]$, all the way up to $\mathcal{R} \in [-180°, 180°]$, with increments of 3 degrees. Similar to the results from *RQ1*, each model evaluates 100 augmented validation datasets at each rotation range. For clarity, that means each model performs $61 \cdot 100 = 6100$ evaluations of augmented validation datasets. Deviating from the tradition of box-plots in section 5.2 to plot varying performances, the range of performances are now depicted as a shaded region illustrating one standard deviation of validation performances. The performance measure is set to be F1-score in these results, and the results are shown in Figures 45-47:

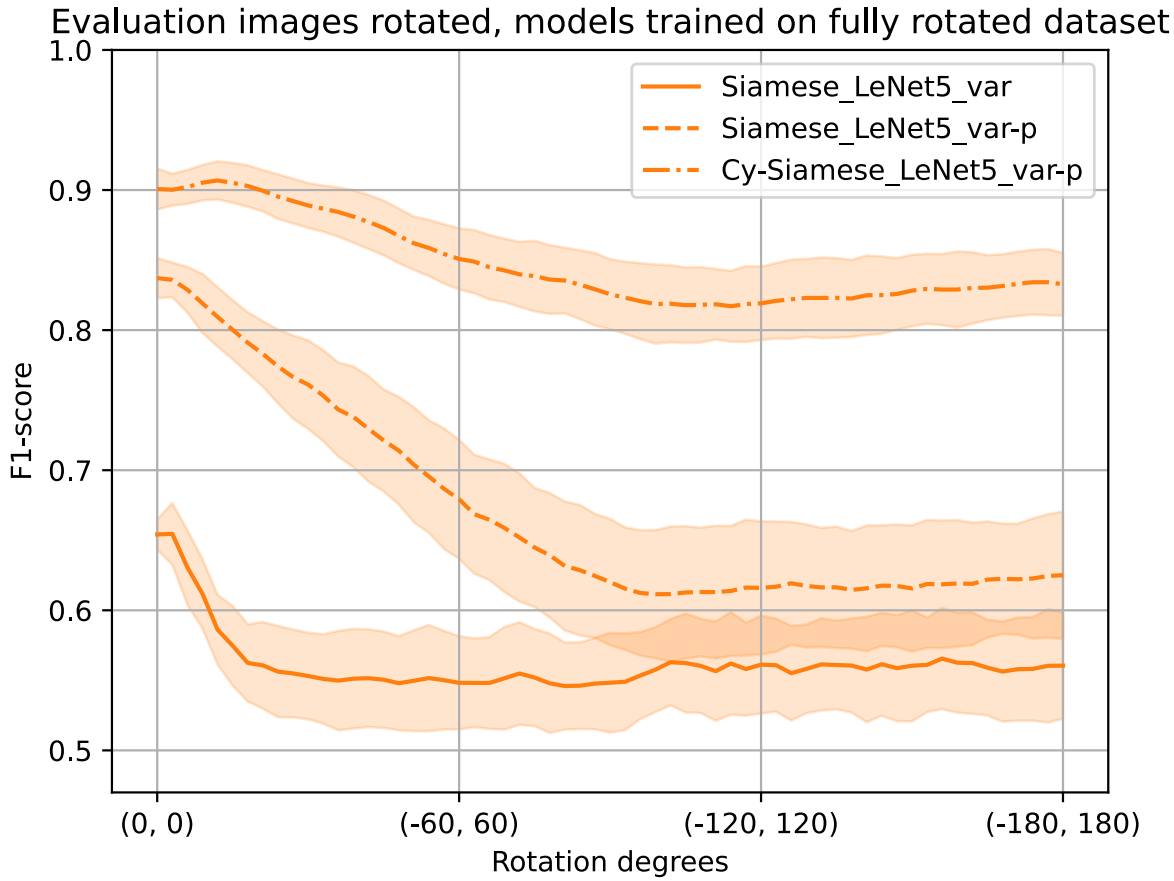


Figure 45: Model validation performances of the different convolutions of the LeNet5-variation architecture plotted against the rotation range dataset hyperparameter. The legends in the plot illustrates which model architecture is tied to which line in the plot, and the shaded area shows one standard deviation of the validation performances at each rotation range. NOTE: Pay attention to the y-axis scale in this graph.

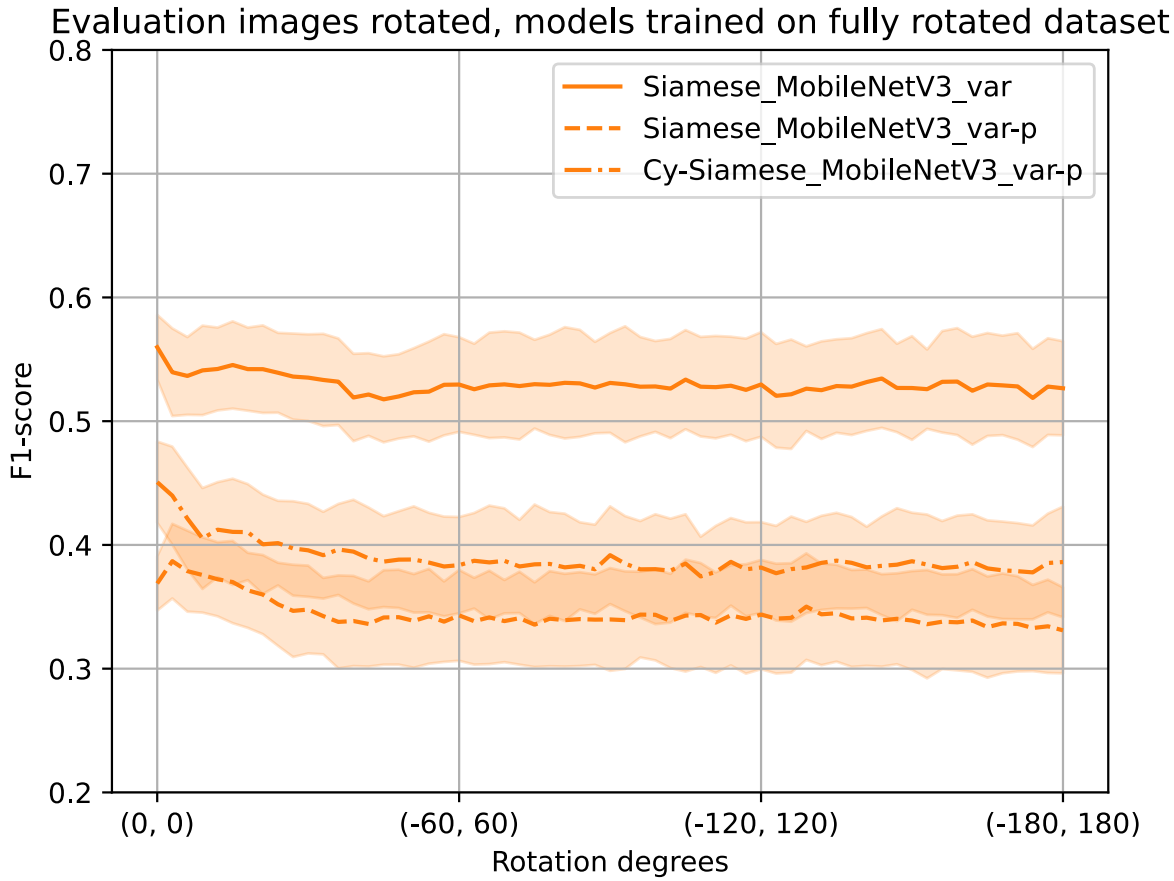


Figure 46: Model validation performances of the different convolutions of the MobileNetV3-variation architecture plotted against the rotation range dataset hyperparameter. The legends in the plot illustrates which model architecture is tied to which line in the plot, and the shaded area shows one standard deviation of the validation performances at each rotation range. NOTE: Pay attention to the y-axis scale in this graph.

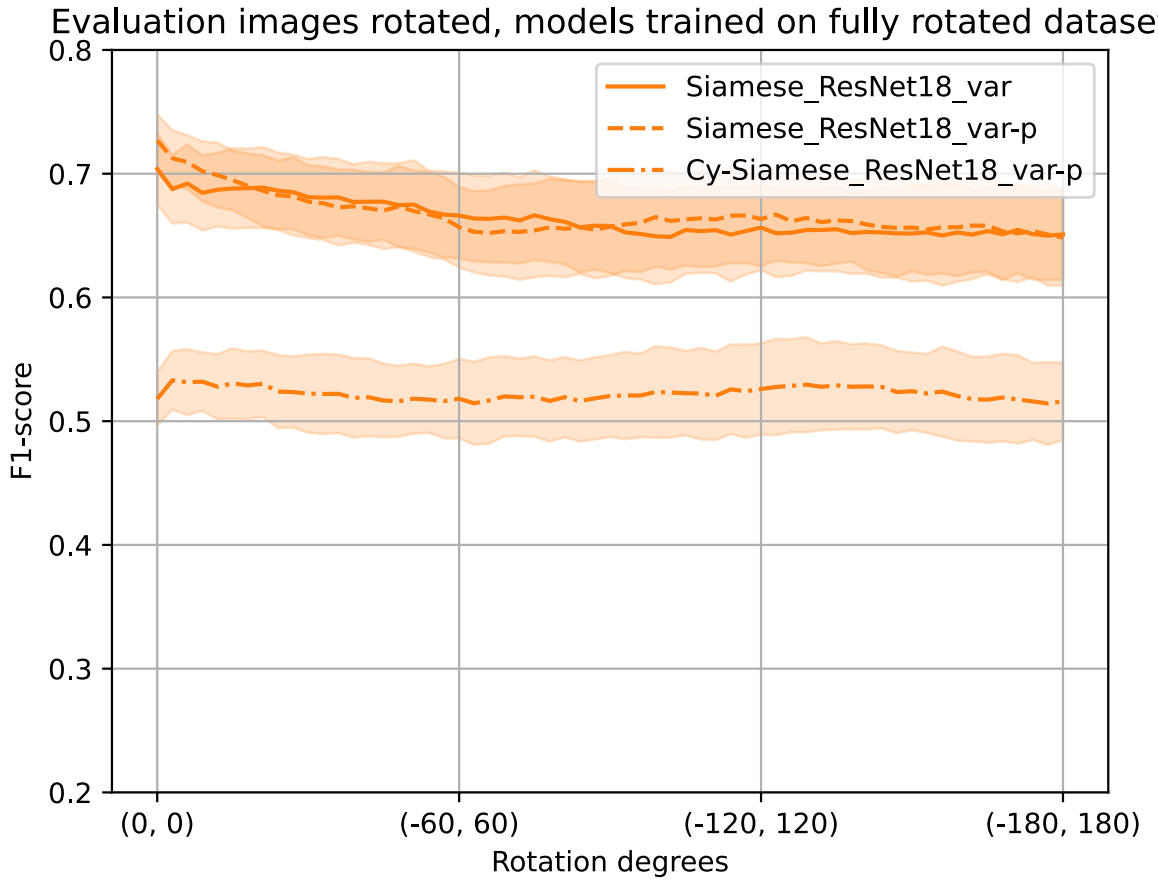


Figure 47: Model validation performances of the different convolutions of the ResNet18-variation architecture plotted against the rotation range dataset hyperparameter. The legends in the plot illustrates which model architecture is tied to which line in the plot, and the shaded area shows one standard deviation of the validation performances at each rotation range. NOTE: Pay attention to the y-axis scale in this graph.

5.4.2 Dataset distribution

The nature of the dataset is critical for the performance of any deep learning model. An interesting point to investigate is followingly how the images in the dataset are distributed, which is performed by letting the synthetic dataset adopt the distribution of the real dataset. Results of training models on the synthetic dataset and evaluating the validation performances with the distribution of the real dataset is demonstrated by box-plots in the same way section 5.2 demonstrates model performances[31]. The results are depicted in Figures 48-49, along with the corresponding threshold statistics responsible for these performances, which can be found in Tables 10-11:

[31]Note that the polar convolution model results are missing. These results might contribute in the discussion of these results, and was left out due to time constraints regarding computational burden.

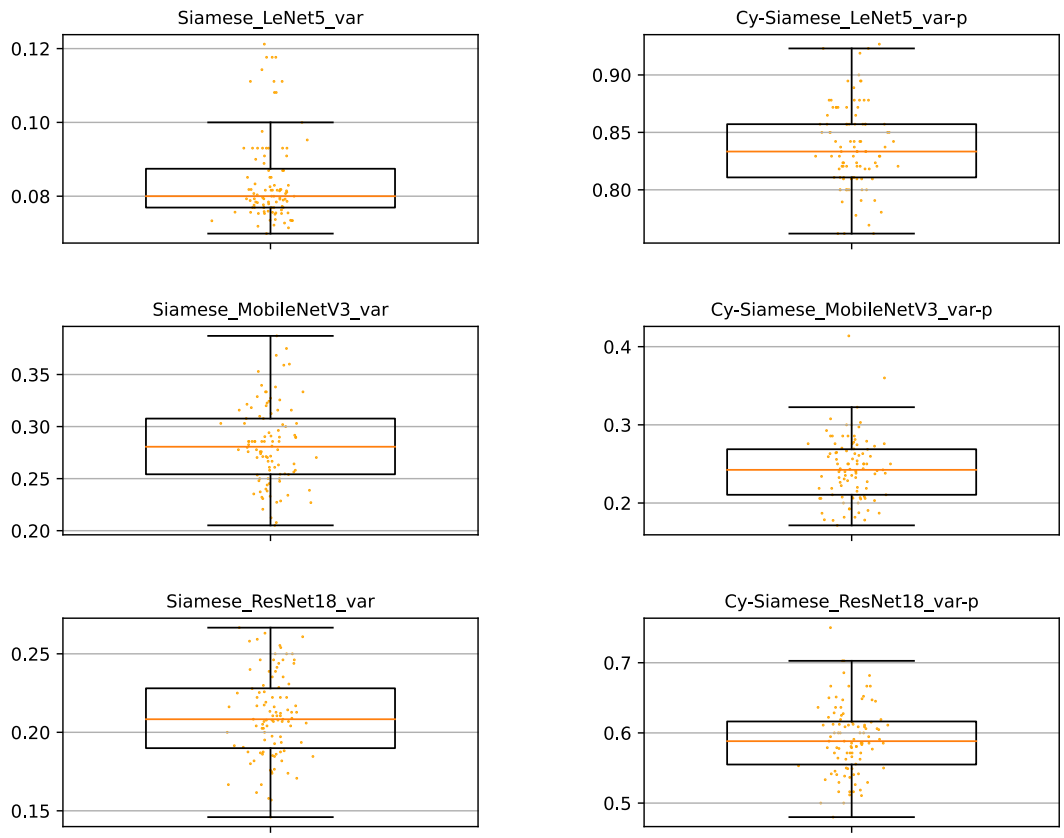


Figure 48: Box-plots of 100 validation F1-score performances on the synthetic dataset that has adopted the real dataset distribution, where the title of each box-plot indicates the responsible model. One orange dot corresponds to one validation performance acquired by randomly augmenting the validation data, and the orange line corresponds to the median of these validation performances. Note that the orange dots are horizontally displaced only for readability purposes. The reader is also reminded to pay attention to the y-axis scale in these plots.

Synthetic dataset - F1-score thresholds (τ)		
Model names	-	`Cy-`, *-p*
`Siamese_LeNet5_var`	0.4318 ± 0.2664	0.6598 ± 0.0364
`Siamese_MobileNetV3_var`	0.3423 ± 0.1255	0.3452 ± 0.1368
`Siamese_ResNet18_var`	0.5708 ± 0.1290	0.4699 ± 0.0670

Table 10: The thresholds τ responsible for the validation F1-score performances represented as orange dots in Figure 48 are presented in this table in terms of the mean and standard deviation of the thresholds.

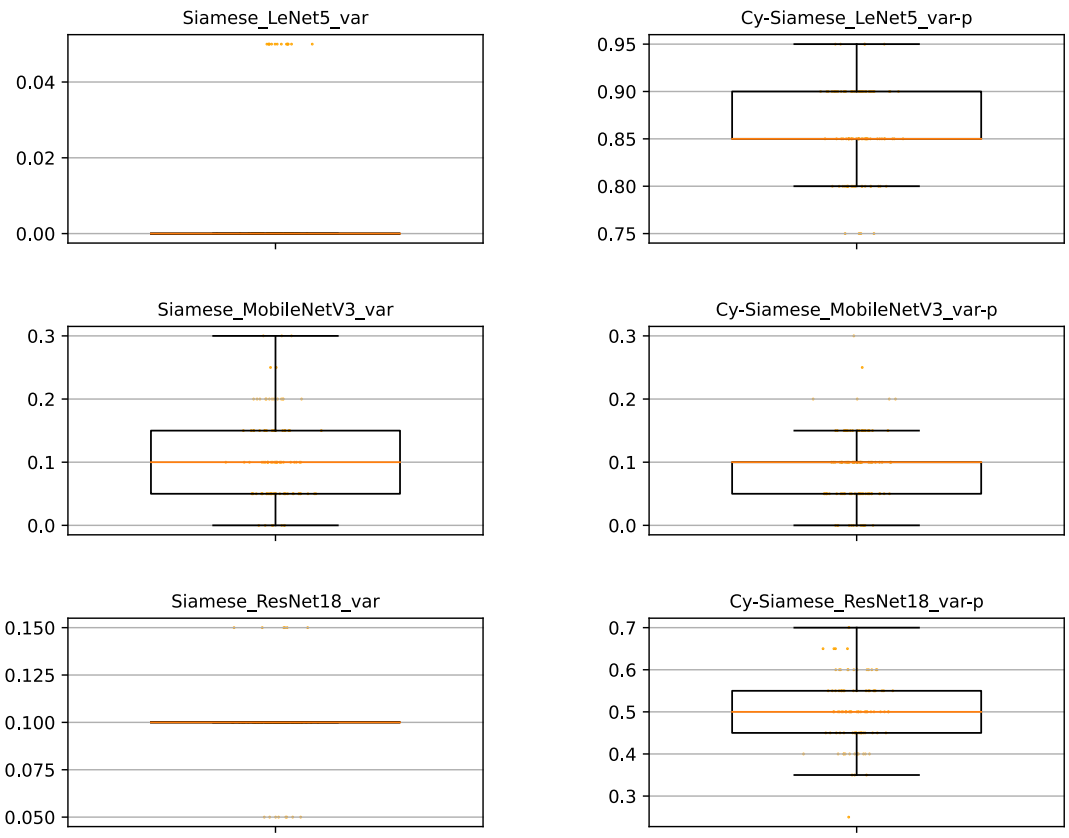


Figure 49: Box-plots of 100 validation TAR@FAR(0.01) performances on the synthetic dataset that has adopted the real dataset distribution, where the title of each box-plot indicates the responsible model. One orange dot corresponds to one validation performance acquired by randomly augmenting the validation data, and the orange line corresponds to the median of these validation performances. Note that the orange dots are horizontally displaced only for readability purposes. The reader is also reminded to pay attention to the y-axis scale in these plots.

Synthetic dataset - TAR@FAR(0.01) thresholds (τ)		
Model names	-	`Cy-`, *-p*
`Siamese_LeNet5_var`	0.0572 ± 0.0116	0.7231 ± 0.0189
`Siamese_MobileNetV3_var`	0.1631 ± 0.0238	0.1543 ± 0.0251
`Siamese_ResNet18_var`	0.3522 ± 0.0296	0.4562 ± 0.0289

Table 11: The thresholds τ responsible for the validation TAR@FAR(0.01) performances represented as orange dots in Figure 49 are presented in this table in terms of the mean and standard deviation of the thresholds.

In reviewing the box-plots illustrated in Figures 48 and 49, and their corresponding threshold statistics represented in Tables 10 and 11, the best performing model on the synthetic validation

dataset that has adopted the distribution of the real dataset, was `Cy-Siamese_LeNet5_var-p`. In regards to F1-score, the median validation performance was 83.3%. In regards to TAR@FAR(0.01), the median validation performance was 85.0%.

5.5 Result discussion (*RQ2*)

5.5.1 Influence of rotation (discussion)

An interesting observation arises from interpreting Table 6 in that the models trained on synthetic images rotated in the range $\mathcal{R} \in [-45°, 45°]$, perform significantly worse than models trained on synthetic images rotated in the range $\mathcal{R} \in [-180°, 180°]$. Onwards, by looking at Table 8, one can observe both better and worse performance results from training models with baseline convolution on images subject to even loss rotation $\mathcal{R} \in [-5°, 5°]$. Specifically, `Siamese_LeNet5_var` shows improvement, but the other models performs worse.

Although this seemingly contradicts the claim that rotated images are to blame for the performance results, there is a possible explanation found in the notion of triplet selection. One can argue that letting the models train on fully rotated images (i.e. $\mathcal{R} \in [-180°, 180°]$), such a dataset provides the model a number of more difficult triplets. In this case, the model optimizes its parameters based on mini-batches that are more rich with data, and performs possibly more optimization steps. Another explanation could be that the models trained on the fully rotated dataset are trained to be generally more robust towards rotation, because that is what they are trained on.

Another interesting result is how the models trained on the fully rotated dataset performs on a range of rotation intervals, illustrated by Figures 45-47. The variations of the more modern off-the-shelf models, that being `Siamese_MobileNetV3_var` and `Siamese_ResNet18_var`, perform more or less equally good/bad when presented with ranges of rotated images from the validation dataset. When looking at `Siamese_LeNet5_var`, some interesting results arise. Particularly, images that are more rotated gradually hurts model performances up until about $\mathcal{R} \in [-90°, 90°]$ for polar convolution and cylindrical convolution. Any more rotation of the validation dataset is demonstrated by Figure 45 to not hurt model performance any further. In regards to baseline convolution, Figure 45 makes it apparent that rotation of the validation dataset hurts model performance more "immediately", beforing settling when the image rotation is approximately $\mathcal{R} \in [-20°, 20°]$. Lastly, Figure 45 also demonstrates that polar transformation is generally a good idea, and that performing cylindrical convolution really helps, both in terms of overall performance and robustness towards the validation dataset being rotated.

5.5.2 Dataset distribution (discussion)

Surprisingly, letting models train on the synthetic dataset which has adopted the distribution of the real dataset shows polarizing results.

To be more specific, `Siamese_LeNet5_var` utilizing baseline convolutions does not learn to generalize at all, but that *really* changes when it uses cylindrical convolution instead. Comparing the performances of `Cy-Siamese_LeNet5_var-p` employed on the synthetic dataset distribution

(as the "***Synthetic dataset results***" paragraph in section 5.2 demonstrated), and the real dataset distribution (see section 5.4.2, Figures 48-49 and Tables 10-11), the results reveal little to no changes in validation performance. Assuming the results are not due to any bugs in either model training or evaluation, this discovery demonstrates an exceptional relationship: *Using cylindrical convolution on the model* ***`Siamese_LeNet5_var`*** *more or less solved the problem of a poorly distributed synthetic dataset.* This is still to be read with a bit of caution, as the hyperparameters of the training pipeline are not necessarily the most optimal ones for both problems. Another thing to point out, is that the polar convolution results are absent from both the results and therefore this discussion, which could provide more insight to this observation.

In terms of the model `Siamese_ResNet18_var` being trained on and evaluated on the synthetic dataset that has adopted the real dataset distribution, the results in section 5.4 show that turning to cylindrical convolutions also improves the model performance significantly. On the contrary, by looking back at the results in section 5.2 regarding the synthetic dataset that has *not* adopted the real dataset distribution, baseline convolutions on `Siamese_ResNet18_var` was in fact better than by using cylindrical convolution. A possible explanation to this is that the results from section 5.2 regarding `Siamese_ResNet18_var` with baseline convolutions has had many more training samples, because those results were from the synthetic dataset distribution. Furthermore, it could be that `Siamese_ResNet18_var` with cylindrical convolutions in section 5.2 were simply overfit due to the bigger dataset. When running this model on a poorer dataset, it might not get the chance to overfit.[32]

Comparing the results with the performances on the real dataset (see section 5.2 for results) however, shows that there are still some image attributes that requires attending in order lift the performance on the real dataset. The difference in performances on the synthetic and real datasets could also indicate that the real dataset trains on and/or evaluates mislabeled images.

5.5.3 Other image attributes

The image attributes presented in this section are merely excerpts from a larger space of tuneable dataset hyperparameters. For further investigation, it would be fruitful to perform similar analysis to other image attributes such as the number of speckles, the spatial distortion of the speckles, what number of IDs is required to significantly lift model performance, among many other possible attributes. For further work, the reader is referred to the Python file `DATASET_HPs.py` in the GitHub repository for a comprehensive overview of tuneable dataset hyperparameters, and the script named `datasetGenerator.py` inside the `dataset_generation` folder employs these hyperparameters to make the requested dataset.

[32]Remember, model selection is performed after each epoch, such that a model's most optimal parameters is likely to have been missed.

6

Conclusions and future work

6.1 Concluding *RQ1*

Reviewing the moments discussed in section 5.3, the thesis has evaluated the performance of models measuring similarity between images by the performance measures F1-score and TAR@FAR(0.01). On both the synthetic validation dataset and the real validation dataset, the model `Siamese_LeNet5_var` with cylindrical convolution performed the best. The synthetic testing dataset performances is augmented and evaluated a number of 100 times, where the testing F1-score was $72.5 \pm 2.6\%$ with the threshold $\tau = 0.6747$ and the testing TAR was $72.8 \pm 3.0\%$ with the threshold $\tau = 0.6938$. In the case of the real dataset, more data is needed to assess the true real testing dataset performances. Regardless, the real validation dataset performances with this model, evaluated on a number of 100 augmented real validation datasets revealed the median validation F1-score to be 60.9%, and the median validation TAR@FAR(0.01) to be 46.7%.

Whilst on the notion of performance measures, the overall differences in the performance measures F1-score and TAR@FAR(0.01) were elaborated. F1-score was shown to be less strict than TAR@FAR(0.01), something that was expected due to the design of the TAR@FAR performance measure. An upside to using TAR@FAR over F1-score is that its premise involves setting a prior boundary on how much of a false accept rate (FAR) one is willing to accept, but a caveat is that false rejects are not considered. F1-score on the other hand, considers false rejects and false accepts. However, a user does not have the luxury of setting a prior for FAR when evaluating with F1-score.

Another important conclusion to stress, is that the models performed better on the synthetic dataset than the real dataset. Although this was somewhat expected, the difference in performances illuminates the importance of investigating *RQ2*. In this thesis, the rotation dataset hyperparameter, and the distribution of the real dataset were deemed most critical to investigate in order to attempt to explain the suboptimal performances on the real dataset. Rotation was selected because the convolution versions of polar convolution and cylindrical convolution, which are designed to deal with rotation, showed promise in yielding better validation performance results as opposed to baseline convolution. The distribution of the real dataset was also selected because a prerequisite for deep learning models to perform well, is that they are rich with data.

Other *RQ2* possibilities such as investigating other similarity measures $\mathcal{S}$, mislabeling, possible problems arising from small batch sizes in conjunction with triplet selection, and other triplet selection schemes were deemed as interesting for further work on *RQ2*.

6.2 Concluding *RQ2*

In regards to training baseline convolution models by several rotation intervals (discussed in subsection 5.4.1), the results have shown that more rotation during training improves the generalizability of `Siamese_MobileNetV3_var` and `Siamese_ResNet18_var`, but decreases the generalizability of `Siamese_LeNet5_var`. A possible cause of this was proposed to be that triplet selection makes the models receive smaller mini-batches.

Moving on, the results of employing models trained on fully rotated datasets on gradually rotating validation images are appropriately wrapped up. All convolution versions of `Siamese_MobileNetV3_var` and `Siamese_ResNet18_var` showed little to no difference in val-

idation performance on F1-score. Turning to `Siamese_LeNet5_var` however, yielded results that show rotated images to be gradually hurtful for the model performance. For baseline convolution, all it took was a rotation range of about $\mathcal{R} \in [-20°, 20°]$ before the validation performance evened out. For polar convolution and cylindrical convolution, validation performance gradually decreased until the rotation range was about $\mathcal{R} \in [-90°, 90°]$, before leveling out. Moreover, the magnitudes of the validation performances achieved by the convolution versions of `Siamese_LeNet5_var` (see Figure 45) generally indicate that polar transforming the image is a good idea.

Lastly, looking at the results on letting the synthetic dataset adopt the distribution of the real dataset provided some interesting results. For starters, baseline convolution with `Siamese_LeNet5_var` was found to not generalize at all. However, when using cylindrical convolution with `Siamese_LeNet5_var`, the performance was more or less the same as when the model was trained on the synthetic dataset that had not adopted the distribution of the real dataset. The major difference in the presented performances warrants caution, as the hyperparameters are not necessarily optimal. Also, results of employing polar convolution with `Siamese_LeNet5_var` might also provide some insight.

In regards to `Siamese_ResNet18_var`, it could suffer from overfitting issues, and that the model selection procedure misses out on optimal paramters.

6.3 Contributions

This thesis has to an extent, uncovered polar and cylindrical convolution to be helpful in using siamese neural networks for classifying salmon lice images. To reiterate, cylindrical convolutions have, to the best of the author's knowledge, never been done in siamese neural networks.

What's more, the thesis has readily defined the notion of a triplet epoch in regards to Triplet Loss, with varying dataset circumstances.

Lastly, given the pipeline hyperparameters, it would seem that model architectures in the likes of LeNet5 outperforms more modern architectures such as MobileNetV3 and ResNet18.

6.4 Future work

In regards to the training pipeline, the works of the thesis and its results could benefit from analyzing training loss and validation loss as the model trains. Although code and plots were developed for the purpose of investigating this, they were down-prioritized to be a part of discussions due to time constraints based on computational burden.

Furthermore, other model architectures and hyperparameters, along with changing optimizer schemes could be fruitful to explore. The observation that more modern architectures such as ResNet18 and MobileNetV3 achieves substandard results is interesting to investigate in further work. To perform this, the reader is also advised to utilize a more powerful hardware setup.

Another interesting approach would be to maybe mix synthetic images with real images during training, instead of the traditional transfer learning apporach which was applied in this thesis.

Lastly, the thesis has mainly focused on the *zero-shot classification problem*. Code has been developed for investigating the *few-shot classification problem*, but requires datasets rich with samples.

7

Revision notes

Revision inquiries can be sent to molbachlian@gmail.com.

Revisions [19.08.2023]

- The triplet cardinality expressions in subsection 4.3.6 were wrong. Where it says $S_i\,(S_i - 1)$ or $f\,(S_i)\,f\,(S_i - 1)$, it previously said $S_i!$ or $f(S_i)!$ and was uncovered during examination. **NOTE:** This practically invalidates all of the results as early stopping had a wrong sense of epochs during pipeline training. Fixing this would however yield similar results, as only change would be that the model parameters are checked more frequently with this corrected cardinality expression. The code is *not* attuned to this as of writing this revision.

- As the presentation of the thesis did, *Research Question 2* is amended from *What attributes of salmon lice images have the biggest effect on the performance of zero-shot classification?* to *What attributes of salmon lice images have a significant effect on the performance of zero-shot classification?*

- Miscellaneous errors in grammar and other writing mistakes have been cleaned up.

References

[1] Google machine learning glossary. https://developers.google.com/machine-learning/glossary/. Accessed on [16.06.2023].

[2] Paul T. Aandahl and Eivind Hestvik Brækkan. Norge eksporterte sjømat for 151,4 milliarder kroner i 2022. URL: https://seafood.no/aktuelt/nyheter/norge-eksporterte-sjomat-for-1514-milliarder-kroner-i-2022/, 2022.

[3] Sanghyeon An, Minjun Lee, Sanglee Park, Heerin Yang, and Jungmin So. An ensemble of simple convolutional neural network models for mnist digit recognition, 2020.

[4] Pierre Baldi and Yves Chauvin. Neural networks for fingerprint recognition. *Neural Computation*, 5, 05 1993.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[7] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1993.

[8] Davide Chicco. *Siamese Neural Networks: An Overview*, pages 73–94. Springer US, New York, NY, 2021.

[9] Kristin Fagerli. *Statistics Norway - Norges viktigste handelspartnere*. 2022.

[10] Anjalie Field, Su Lin Blodgett, Zeerak Waseem, and Yulia Tsvetkov. A survey of race, racism, and anti-racism in nlp, 2021.

[11] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Pearson, 2018.

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

[13] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, 2006.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[15] Geoffrey Hinton. Neural networks for machine learning. Coursera, video lectures, 2012.

[16] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[17] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

[18] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[19] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks, 2016.

[20] Ruoqiao Jiang and Shaohui Mei. Polar coordinate convolutional neural network: From rotation-invariance to translation-invariance. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 355–359, 2019.

[21] Jinpyo Kim, Wooekun Jung, Hyungmo Kim, and Jaejin Lee. CyCNN: A Rotation Invariant CNN using Polar Mapping and Cylindrical Convolution Layers, 2020.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[23] Eirik B. Kvernevik. Identification of salmon lice based on image processing methods, 2021.

[24] Alex Labach, Hojjat Salehinejad, and Shahrokh Valaee. Survey of dropout methods for deep neural networks, 2019.

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[26] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In *NIPS*, 1989.

[27] Johannes Lederer. Activation functions in artificial neural networks: A systematic overview, 2021.

[28] Bjørn Magnus Mathisen, Kerstin Bach, Espen Meidell, Håkon Måløy, and Edvard Schreiner Sjøblom. Fishnet: A unified embedding for salmon recognition. *IOS Press*, pages 3001–3008, 2020.

[29] Adele Mennerat et al. Anthropogenic parasite evolution. `URL: https://paranthrope.w.uib.no/`, 2020.

[30] Joseph Merz, Paul Skvorc, Susan Sogard, Clark Watry, Scott Blankenship, and Erwin Nieuwenhuyse. Onset of melanophore patterns in the head region of chinook salmon: A natural marker for the reidentification of individual fish. *North American Journal of Fisheries Management - NORTH AM J FISH MANAGE*, 32:806–816, 08 2012.

[31] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[32] Ministry of Agriculture and Food. *Animal Welfare Act*, 2009.

[33] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. In Shang-Hong Lai, Vincent Lepetit, Ko Nishino, and Yoichi Sato, editors, *Computer Vision – ACCV 2016*, pages 189–204, Cham, 2017. Springer International Publishing.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[35] Ragnhild Pettersen, Håvard Lein Braa, Bartlomiej Adam Gawel, Paul Anton Letnes, Kristin Sæther, and Lars Martin Sandvik Aas. Detection and classification of lepeophterius salmonis (krøyer, 1837) using underwater hyperspectral imaging. *Aquacultural Engineering*, 87:102025, 2019.

[36] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[37] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[38] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021.

[39] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[40] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, jun 2015.

[41] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2015.

[42] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[43] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[44] Eva B Thorstad and Bengt Finstad. Impacts of salmon lice emanating from salmon farms on wild Atlantic salmon and sea trout. 2018.

[45] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christopher Bregler. Efficient object localization using convolutional networks, 2015.

[46] M S Ugelvik, A Skorping, and A Mennerat. Parasite fecundity decreases with increasing parasite load in the salmon louse lepeophtheirus salmonis infecting atlantic salmon salmo salar. *Journal of Fish Diseases*, 40(5):671–678, 2017.

[47] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning, 2021.