NTNU
Norwegian University of
Science and Technology

# A Dual-Stream Deep Learning Architecture for Action Recognition in Salmon from Underwater Video.

Håkon Måløy

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Agnar Aamodt, IDI
Co-supervisor: Ekrem Misimi, SINTEF Ocean

Norwegian University of Science and Technology
Department of Computer Science

# Problem Description

For this thesis I was given the following problem description:

SINTEF Ocean is leading a very interesting project, called Intelligent, aiming to develop novel concepts for salmon behaviour prediction in net cages. The prediction models are to be developed based on computer vision features and implementing prediction models based on Convolutional Neural Networks (CNNs). The current work in this MSc thesis is a continuation of the work that student, Håkon Måløy, has done in during summer job internship 2016, and his project assignment in TDT4501 – Computer Science, Specialization Project. During the summer job, student Håkon Måløy implemented a combination of CNNs in the role of feature extractor and used the output of the fully connected layers as the features for the training of Suport Vector Machines and generation of prediction models. In the project assignment the student has carried out a comprehensive and structured literature review on the use of CNN- and Recurrent Neural Network (RNN)-based architectures for human Action Recognition and the most promising architectures that can be used as an inspiration for Action Recognition for salmon in underwater videos.

In this MSc assignment, the student will carry on with the work based on the findings from the project assignment. The focus will be on investigating a Dual-Stream CNN approach to capture both spatial and motion data. One stream will use grayscale images as inputs while the other will use Optical Flow as input. The Optical Flow stream will also utilize 3D-Convolutions to further capture motion features. The two streams will be combined and used as input to a RNN for sequence processing before the final score is computed using e.g. a softmax layer.

# Preface

This thesis was prepared during the spring of 2017 at the Norwegian Institute of Science and Technology(NTNU), Faculty of Information Technology and Electrical Engineering, Department of Computer Science. The thesis was accomplished in cooperation with SINTEF Ocean AS.

I would like to thank my supervisors Ekrem Misimi, Agnar Aamodt and Bjørn Magnus Mathisen for their guidance and clarifying discussions through the work with this thesis. I would also like to thank my coworkers at the SINTEF Ocean Robotics Lab for their help and encouraging words during the implementation phase of the thesis. Lastly I would like to thank my family, and especially my Mother and Father for their support and hours spent reading my sketches to provide constructive feedback.

Trondheim, June 2017
*Håkon Måløy*

# Abstract

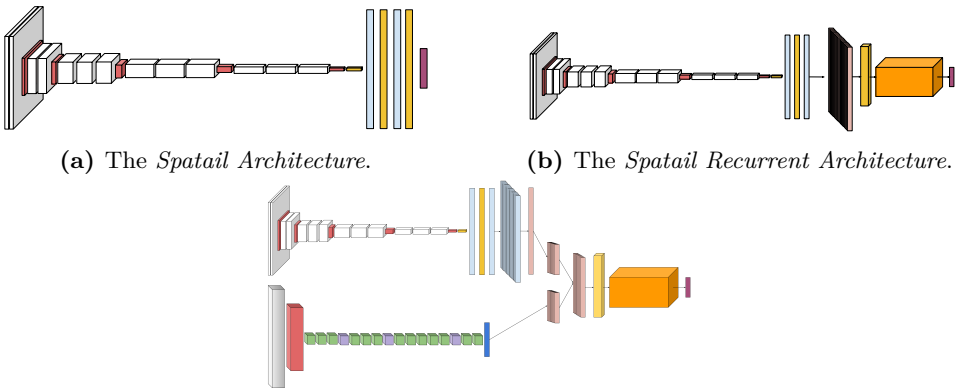
Over half of the costs from breeding salmon in the Norwegian salmon farming industry comes from feed usage[13]. Today the feeding process is largely a manual labor, requiring an operator to monitor the amount of feed sinking to the bottom of a breeding cage. When the amount of feed exceeds a certain threshold, the feeding process is terminated. Automation of this process and using salmon motion behavior instead of sinking feed to determine when to terminate feeding, could greatly reduce costs both in through the labor needed and amount of feed wasted.

Resent developments in Human Action Recognition have shown that Deep Learning approaches are well suited to perform Action Recognition[25, 31]. We therefore examine the feasibility of using Deep Learning approaches to automate the feeding process. We use 76 videos of salmon collected from within a breeding cage during the month of November 2016. Using these videos for training, validation and testing, we propose three approaches to automatically classify Feeding and NonFeeding behavior in salmon. The three approaches are a *Spatial Architecture*, a *Spatial Recurrent Architecture* and a *Dual-Stream Architecture* as seen in figure 1.

Our results show that all our proposed architectures are able to separate videos of Feeding and NonFeeding salmon with high accuracy. We also find that the *Dual-Stream Architecture* is the best performing architecture. It combines spatial and temporal information through the use of a *Spatial Stream*, a novel *Temporal Stream* and a Recurrent Neural Network (RNN). Our *Dual-Stream Architecture* is able to accurately classify 80.0% all of our testing videos, presenting state-of-the-art performance.

To the best of our knowledge, both our *Temporal Stream* and our *Dual-Stream Architecture* are original and novel architectures, as is the application of Deep Learning inference models for the Salmon Activity Recognition domain in optimization of feeding operation in Norwegian Aquaculture. We hope the results presented in this thesis will contribute to achieve a higher sustainability in Norwegian salmon aquaculture, optimize feeding operations, and consequently reduce potential waste.

Future work beyond the results presented in this thesis concerns research on understanding of what our the Deep Learning architecture have learned and visualizing this learning process.



**(a)** The *Spatail Architecture*.

**(b)** The *Spatail Recurrent Architecture*.

**(c)** The *Dual-Stream Architecture*.

**Figure 1:** We propose three architecture to automatically classify Feeding and NonFeeding videos of salmon.

# Sammendrag

Over halvparten av kostnadene i Norsk lakseoppdrett kommer fra fôrbruk[13]. I dag er fôringsprosessen en manuell prosess, som krever en operatør som overvåker mengden fôr som synker til bunnen av merden. Når mengden fôr overstiger en forhåndsbestemt grense, blir fôringen avsluttet. Automatisering av denne prosessen gjennom å bruke laksens bevegelsesmønstre for å avgjøre når fôringen skal avsluttes, kan redusere kostnader gjennom redusert tap av fôr og redusert behov for arbeidskraft.

Nyere forskning i Human Action Recognition har vist at Deep Learning tilnærminger er godt egnet for å utføre Action Recgonition[25, 31]. Vi utforsker derfor mulighetene for å bruke Deep Learning for å automatisere fôringsprosessen. Vi bruker 76 videoer av laks, hentet fra en oppdretsmerd i November måned i 2016. Ved å bruke disse videoene til trening, validering og testing produserer vi tre arkitekturer for å automatisk kunne klassifisere Spisende og IkkeSpisende oppførsel hos laksen. Disse arkitekturene består i en *Spatial Architecture*, en *Spatial Recurrent Architecture* og en *Dual-Stream Architecture*, som kan sees i figur 1.

Våre resultater viser at alle våre arkitekturer er i stand til å separere videoer av Spisende og IkkeSpisende laks med stor nøyaktighet. Vi kommer også frem til at vår *Dual-Stream Architecture* er den beste arkitekturen. Den kombinerer romlig og tidsmessig informasjon gjennom å bruke en *Spatial Stream* og en ny og orginal *Temporal Stream* sammen med et Recurrent Neural Network (RNN). Vår *Dual-Stream Architecture* er i stand til å klassifisere 80.0% av alle våre testvideoer, noe som tilsvarer en ny state-of-the-art.

Så vidt vi vet er både vår *Temporal Stream* og vår *Dual-Stream Architecture* nyskapende og orginale arkitekturer, som tidligere ikke har vært utforsket. Vår bruk av Deep Learning på Salmon Action Recognition domenet for optimering av fôringsporsessen i norsk akvakultur er også ny. Vi håper at resultatene presentert i denne oppgaven vil bidra til å oppnå større bærekraftighet i norsk oppdrettsnæring og redusere fôrspill.

Videre arbeid med våre resultater vil bestå i å forske på å forstå hva våre tilnærminger har lært gjennom visualisering av læringsprosessen.

# Contents

# List of Figures

# List of Tables

# Acronyms

**3D-CNN** 3D-Convolutional Neural Network. 5, 17, 44–46, 54, 70–72, 74, 83, 89

**CNN** Convolutional Neural Network. i, 3–5, 13, 15–17, 28–31, 37, 43–46, 54, 61, 70, 71, 76, 77, 114

**LSTM** Long Short-Term Memory. 4, 7, 8, 32, 34, 35, 43–46, 90–97, 104, 105, 115, 116

**RCNN** Recurrent Convolutional Neural Network. 5, 46

**RNN** Recurrent Neural Network. i, v, vii, 4, 5, 7, 8, 10, 15–17, 31–34, 37, 43–46, 54, 90, 92–95, 104, 105, 108, 114–116

**SVM** Support Vector Machine. 3, 13–15, 20

Chapter 1

# Introduction

This chapter presents the motivation behind this master's thesis. We present a brief summary of the work done by the author during a summer internship at SINTEF Ocean, as well as the work done in the Specialization Project. The findings and conclusions from the Specialization Project are used as guidelines and motivation for the hypotheses we present in this thesis. Finally, we give a brief overview of the thesis structure.

## 1.1 Motivation

With the rapid development in the aquaculture industry and the rise of salmon farming on the Norwegian sector, the need for automation and decision support systems has become apparent in recent years. The Norwegian salmon industry has become a huge business and the cost of farming has been steadily increasing with one of the biggest contributing factors being the cost of feed. Feed is approximated to be accounting for more than half of the total farming costs[13] and thus, big savings can be made through optimization of the feeding process. There is therefore much interest in automating this process.

During the summer of 2016, I worked at SINTEF Ocean as a Summer Intern in the Intelligent Project, where Convolutional Neural Networks (CNNs) were used as feature extractors for Support Vector Machines (SVMs)[2] to do salmon Action Recognition in videos. This work was then extended through the subject TDT4501 – Computer Science, Specialization Project, where a literature review was performed to summarize the current state-of-the-art in human Action Recognition. The main aim of the Specialization Project was to use the findings from the literature review as inspiration to design an architecture for Action Recognition in salmon. This Masters thesis is the natural extension of that work and will show the implementations, extensions and testing of the architecture proposed in the Specialization Project.

### 1.1.1 SINTEF Ocean Intelligent Project

During the summer of 2016, I worked at SINTEF Ocean as a summer intern. I worked on the Intelligent Project, which is aimed at predicting behavior of salmon in breeding cages.

**Intelligent Project**
SINTEF Ocean is leading a project called Intelligent, aiming to develop novel concepts for salmon behavior prediction in net cages.

1. Novel concepts and methods for processing of underwater image/video data.
2. Novel prediction models applicable to salmon behavior analysis and informed decision making.

The concepts and methods to be developed in Intelligent, are inspired by state-of-the-art BigData and Machine Learning concepts and will exploit Deep Learning and SVM algorithms to enable improved context awareness and intelligence for monitoring and control of operations in aquaculture industry. Intelligent will lay a foundation for the use of the Big Data concept in future project development stages. Technology concepts developed in Intelligent can enable the Norwegian aquaculture industry to better optimize operations, reduce waste, and increase the overall sustainability.

**Summer Internship**
During the summer internship work on the Intelligent Project, led by SINTEF Ocean, I developed a Twin-Stream architecture for Action Recognition in salmon. The architecture used two identical, pretrained, versions of the well known AlexNet Neural Network architecture[28]. The networks used plain video frames and Optical Flow frames as input respectively and produced high-level feature vectors, which were then fed into an SVM classifier at the end of each network. The architecture was then topped with a final SVM classifier, taking the outputs of the two SVM classifiers as its input, as seen in figure 1.1. This architecture was able to produced Action Recognition scores around the 65% accuracy mark on out testing videos.

Although these scores are significantly better than a random-guesser classifier, we believe that they can be significantly improved by including more of the temporal information contained within videos. The reason for this is that humans have great trouble classifying the content of a video based on a single frame. This performance is even further reduced if the frames are from a video where the context is not immediately clear, such as a darkened underwater frame or very close up frames of gravel. However, when presented with several seconds of video, humans are quickly able to classify the videos as videos of a shipwreck or of an ant colony on the move. Thus we intuitively see that the temporal information in the video is crucial for accurate classification by humans and that an increase in temporal information could also lead to even more accurate classifications. We believe that this observation also holds true for computer vision, thus we want to explore the effects of increasing the temporal information used by classification architectures.

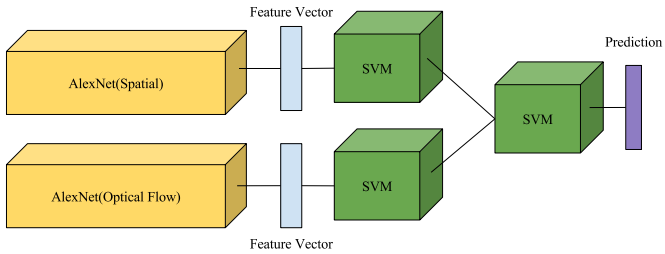



**Figure 1.1:** The figure shows the Twin-Stream architecture implemented during the summer internship at SINTEF Ocean. The architecture uses two AlexNet CNNs to produce feature vectors of the input. One AlexNet takes individual video frames as input, while the other takes Optical Flow video frames as input. The feature vectors were fed into separate SVMs which produced outputs, which were then fed into a final SVM for classification.

**Specialization Project**

In my Specialization Project, I did a literature review of the field of Human Action Recognition in videos. Deep Learning has recently demonstrated very promising results on image classification tasks through the use of CNNs[29]. This approach has also successfully been extended to Action Recognition in videos of humans through the addition of Recurrent Neural Networks (RNNs)[42, 51] and has lately seen the most prominent use for such tasks. It was therefore natural to focus on Deep Learning-based approaches for the literature review. The findings from the review were used as inspiration to propose several architectures aimed at Action Recognition in salmon. These architectures were then compared against each others based on a set of architecture requirements, specifically designed for underwater video classification. This comparison lead to a final architecture proposal, which will be extended and implemented in this master's thesis. The Specialization Project also lead to an article[1] set to be submitted to Computers and Electronics in Agriculture.

[1]Måløy H., Misimi E., Aamodt A., Mathisen B. M.; Deep learning architectures for human action recognition: Transfer learning for action recognition in biomarine environments.

## 1.2 Hypotheses

Today, the feeding process in the salmon industry is usually a manual process, based on the feed eaten rather than the salmon behavior. An operator is monitoring the feeding process through the use of underwater cameras with the aim to determine whether the salmon are eating the feed or not. When the operator can visually determine that the amount of feed sinking to the bottom of the cage has exceeded a certain threshold, the feeding process is terminated. Since this requires feed to already be wasted for the operator observe that the threshold has been exceeded, we can assume that this method results in a significant amount of feed being wasted. We therefore want to explore the possibilities of automating this process using Deep Learning approaches. Since we also aim to reduce the waste of feed during the process, we want to rely on the behavior of the salmon themselves rather than on the amount of feed sinking to the bottom of the cage. Since Deep Learning, through the use of CNNs and RNNs, has already been used to successfully perform Action Recognition in humans, we want to explore if this is also possible for salmon. However, human actions are often distinguishable from each other through the pose of the person, performing the action. If a person is playing baseball, we usually only need to see a single image to be able to classify the action. These poses are known as discriminative action poses and have been shown to be a large factor in Deep Learning Human Action Recognition performance[32]. For salmon, however, it is not clear that they possess such discriminative action poses. Thus our architectures might have to rely on the poses of multiple salmon and and how it changes over time to perform Action Recognition. This analysis therefore leads us to the following hypotheses for this thesis:

> **H1:** A system based on Convolutional Neural Networks can perform Action Recognition in videos of salmon with the goal of separating Feeding from NonFeeding behavior.

We focus on training Neural Networks that are able to separate videos into Feeding and NonFeeding categories with high accuracy, specifically through the use of CNNs.

Since this type of network uses single frames to do classification, we propose to expand the model to include the movements of the salmon, through the temporal aspects videos, utilizing the fact that videos consists of multiple frames through time. This will be done by treating the videos as sequences of frames and therefore using the sequence handling capabilities of RNNs, in conjunction with CNNs to increase performance. We also want to explore the use of 3D-Convolutional Neural Networks to treat video segments as cubes of video frames and then perform convolutions both spatially over individual frames and temporally over multiple frames. This leads us to our second hypothesis:

> **H2:** The system can be improved by including the temporal dimension contained within videos through the use of Recurrent Neural Networks or a combination of Recurrent Neural Networks and 3D-Convolutional Neural Networks.

To test the first hypothesis(**H1**) we have chosen to implement and test a CNN trained towards the task of Action Recognition in videos of salmon. This will be done by implementing different architectures, training them from pretrained checkpoints and testing them on a test set of videos.

The second hypothesis(**H2**) will be tested though the implementation of several different 3D-Convolutional Neural Network (3D-CNN) architectures and RNN architectures. These will then be used in conjunction with the best model from **H1**, to test whether the accuracy is increased on the test set. We will also be comparing different combinations of the models to each other. Finally a comparison between the best model in **H1** will be compared to the best models in **H2** on the test set to evaluate the possible increase in performance from **H1** to **H2**.

With this as a background, we produce the following scientific contributions in this thesis:

1. An approach for combining several Deep Learning techniques in a novel architecture to produce state-of-the-art Action Recognition results on salmon by:
   (a) Utilizing the temporal dimension through the use of a 3D-Residual Neural Network in combination with Optical Flow inputs for motion feature extraction.
   (b) Combining the temporal architecture with a spatial CNN for spatial feature extraction.
   (c) Treating videos as a sequence of frames by feeding the temporal and spatial features to a Long Short-Term Memory Recurrent Neural Network.

The results and architectures presented in this thesis also go beyond the current state-of-the-art and will therefore result in an article to be published in a high ranking journal. The thesis has therefore also received a delayed release date to accommodate for this publication.

## 1.3 Structure of the Thesis

This thesis is divided into four main chapters.

Chapter 2 gives a brief overview of the field of Deep Learning as well as an overview of the environment, from which the data used in this thesis is collected. It also gives and overview of the previous work in the Specialization Project.

Chapter 3 contains our method and experiments. We introduce our dataset and explain our approach for arriving at our final architectures for Action Recognition in salmon.

Chapter 4 presents our testing dataset as well as our results, using our architectures from chapter 3. We also give an in-depth analysis of our results to further explain them and enhance understanding.

In chapter 5, we give our final conclusion as well as presenting our vision for future work.

Chapter 2

# Theory and Background

This chapter presents the theoretical background of this Master's thesis. The goal is that readers who are unfamiliar with the topics presented, can learn what is needed to understand the later contents of the thesis. The first topic we present is the field of Deep Learning, presented in section 2.1. We then proceed to the subject of Activity Recognition in section 2.3, with special focus on the use of Deep Learning approaches. Then, in section 2.4, we give an introduction to Activity Recognition in underwater videos of salmon. Finally we present a brief summary of our previous work in section 2.5.

## 2.1 Deep Learning

This section gives a brief overview of the theory within the field of Deep Learning. It is intended to serve as an introduction to the field and to create a theoretical foundation on which the reader can rely for the rest of the thesis.

### 2.1.1 Historical Background

Since the start of the Internet, the amount of readily available general data has grown at an incredible phase. At first, this data mainly consisted of documents and web pages, but in the later years, this growth has expanded to include photos and even videos[11]. This has lead computer vision to become one of the biggest technological advances in the last decade. With a vast array of applications such as image recognition[38], self driving cars[23] and surveillance[25], computer vision has become an integral part of many business models. In many of these approaches, Image Classification plays a major role. This task is very demanding for computers, as images can contain multiple objects, be taken from different viewpoints and be occluded or severely cluttered. The goal has therefore been to develop agile algorithms capable of recognizing objects in complex scenes.

Traditionally this was done using hand-crafted approaches such as Bag of visual Words (BovW) topped with a classifier such as a Support Vector Machine (SVM)[6, 9]. These approaches produced the state-of-the-art results in image classification competitions such as ILSVRC [38] for several years. However, recent developments in Deep Learning has led to drastically increased performance and Deep Learning-based approaches have taken over as the new state-of-the-art performers[20, 28, 41, 47].

Deep Learning is a field of Machine Learning specializing in statistical models called Deep Neural Networks. These models can learn complex hierarchical representations that correspond to multiple levels of abstractions. This is done through the use of multiple layers of nonlinear processing units, called neurons, to transform data, where each layer takes the previous layers as input. This creates a flow of information, from the input through the network to the output. The way these models are able to learn such complex representations is through the use of the *Backpropagation Algorithm*[37]. This algorithm works in several steps. First, the error, or cost, between the model output and the true output is calculated through the use of a cost function. Then the cost for each neuron in the network is calculated and propagated back through the network. The model weights are then updated based on these cost calculations, resulting in a gradual increase in performance for each weight update.

Since 2010 we have seen a drastic improvement in both natural language processing[4] and image classification through the use of Deep Learning[29], producing results that far exceed the competition. In the last five years alone, Deep Learning has completely transformed the field of Computer Vision. This is not only due to the fact that these models learn so well, but also because of the introduction of modern GPUs and an exponential growth in available data[18]. Modern GPUs allow researchers to greatly parallelize the forward and backward passes through Neural Networks by utilizing the hugely parallel design of GPUs. This reduce training times by several time folds, leading to faster development and better models.

An inherent limitation of Deep Learning is the need for very large datasets for training. Since the weight update procedure has to be performed thousands if not millions of times for a even quite simple networks to converge on a good set of weights, the demand for large amounts of data is obvious. Thus, with more data, we are able to explore more complex models and achieve better performance. Recently, the use of pretrained models, already trained on large datasets have shown great results when used as a starting point for training models towards new tasks. This approach is known as *Transfer Learning*, where one can transfer many low-level representations learned on one dataset to another, drastically reducing the need for data. This has allowed for a much larger audience to acquire expertise and develop new models.

### 2.1.2 Dataset splits

The most common approach to training Deep Neural Networks is *Supervised Learning*. In supervised learning tasks, models learn features from labeled examples and try to approximate their predictions to the correct labels as much as possible. A common problem with this approach is a problem known as *Overfitting*. Overfitting happens when the model learns features that are not necessarily valid for real-world examples and become overfit to the training data. Such a model has not learned general concepts, but rather remembers the correct output for a given example in the training set. This results in poor performance in the real world. To combat the problem of overfitting it is common practice to divide the available data into three partitions, called the *training set*, the *validation set* and the *testing set*. It is then possible to check for overfitting during training, using the validation set performance as a guide. An important factor when this partitioning is done is to make sure that the test set is representative of the data the model will be working with when deployed. It is also important that the training set is representative of the validation set and the test set. There are many ways of separating the original dataset into training, validation and test sets, but a split of 60/20/20 or 50/25/25 are both quite common[19].

**Training set**
The training set is the partition used to train the model and is also, by far, the largest of the three partitions. This is a labeled set of data, containing the input data and the expected output. This expected output is then compared to the output of the model to calculate the cost for each example in the dataset during training.

**Validation set**
The validation set is used to validate and tune the model during the training phase. This is done by measuring the model's performance on the validation set, without allowing it to update its weights. This produces a good estimation for how well the model will perform on the test set. The performance on the validation set is also a good indication of when a model has become overfit to the training data. When the validation performance goes from increasing to decreasing during training, it usually indicates that the model has started to become overfit and further training will only further deteriorate model performance.

**Test set**
The test set is used for a final testing of the model. After the model has been tuned towards the optimal performance on the validation set, it is tested on the test set. This gives a good indication of how well the finished model will perform on new data and thus how well it will preform when deployed in the real world. It is very important that the test set is not used until the model has finished training and has been fully optimized towards the validation set. This is to avoid researcher bias and to ensure valid test results.

### 2.1.3 Neural Networks

Neural Networks are graphs that consist of one or more connected neurons, or nodes, with learnable weights $W$ on their connections, or edges, as seen in figure 2.1. Each neuron also has a learnable bias $b$, which enables the neuron to activate even for zero-valued inputs. This is critical for successful learning as it helps the network to converge on a good set of weights and biases. A neuron receives a set of inputs $x$ along its edges, computes the dot product over these inputs and its weights. It then follows it with an optional non-linear activation function $f$ to produce an output $y$ as shown in equation (2.1). Neural Networks are usually stacked in layers, where every layer in the network takes the previous layers as inputs. If the network consists of more layers than the input and output layers, the remaining layers are usually referred to as *hidden layers*, as we do not see either the input or outputs of these layers directly. An example of a simple Neural Network with two hidden layers is seen in figure 2.1

$$y = f(\sum_i W_i \bullet x_i + b) \tag{2.1}$$

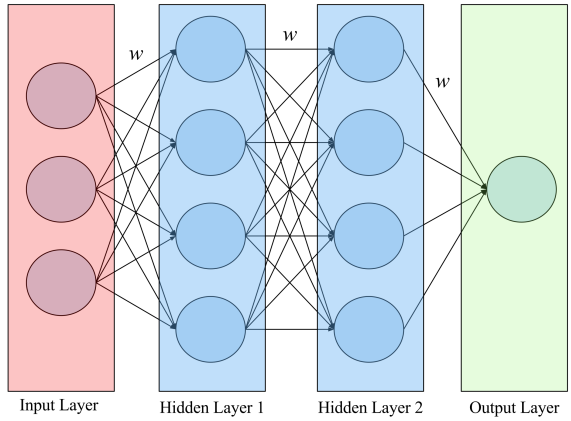



**Figure 2.1:** The figure shows a fully connected Neural Network with an input layer, two hidden layers and an output layer. This network is of the typical *feed forward* architecture, where all connections go forward through the network. Each of the connections between the neurons also have a weight $W$. Figure adapted from [30]

The use of a non-linear activation function allows Neural Networks to approximate any function, including non-convex functions. The activation function takes a number and does a fixed mathematical operation on it to squash it withing a well defined range. The three most common activation functions today are:

1. The Sigmoid activation function.
2. The Tanh activation function.
3. The ReLU activation function.

**Sigmoid**
The sigmoid activation function, shown in figure 2.2a, takes a real-valued number and squashes it to a range between 0 and 1. It has the mathematical form presented in equation (2.2). This results in large positive numbers becoming 1 and large negative numbers becoming 0. The sigmoid activation function was historically frequently used since it closely resembles the firing rates of real neurons in real brains. However, it has seen a decline in the resent years due to the fact that it can kill the gradients. Since the activations of the neuron can saturate at the tails of the activation function, the gradient in these regions become very close to zero and *vanish*. This leads to almost no signal flow during the backpropagation phase and hence only very small or no weight updates are being performed. This in turn leads to a network that stops learning.

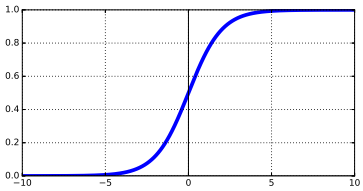
$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

**Tanh**
The tanh activation function, shown in figure 2.2b, squashes a real-valued number to a range between -1 and 1. Just like the Sigmoid, this activation function suffers from the same saturation problem at its tails. The mathematical expression for tanh is shown in equation (2.3)

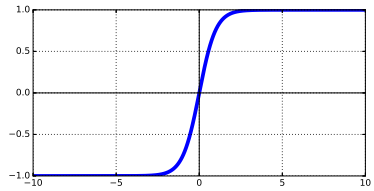
$$f(x) = Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.3}$$

**ReLU**
The most popular activation function in recent years is the Rectified Linear Unit activation function as seen in figure 2.2c. The activation is thresholded at zero and has the mathematical equation shown in equation (2.4). This activation function does not suffer from the saturation problem that both the sigmoid and tanh do. This is due to its linear form and the ReLU has been shown to significantly accelerate network convergence[29]. However, the ReLU activation function has one drawback. A large gradient flowing through a ReLU activated neuron can cause the weights to update in a way that results in the neuron never activating on a datapoint again, effectively resulting in a "dead" neuron. This is irreversible, but is somewhat avoidable by setting a good weight update parameters.

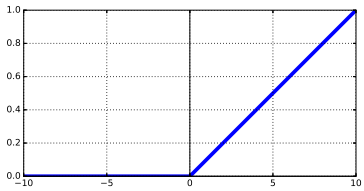
$$f(x) = ReLU(x) = max(0, x) \tag{2.4}$$




**(a)** Sigmoid activation function.




**(b)** Tanh activation function.




**(c)** ReLU activation function.

**Figure 2.2:** The three most commonly used activation functions in Deep Learning. We see that the ReLU can not get saturated, thus the gradients do not vanish.

#### 2.1.3.1 The Loss Function

The loss function can be defined as a function from a set of input values to class scores, parameterized by a set of weights $W$ and a set of biases $b$. It follows from this assumption that some sets of parameters are better than others. If a network is given an image of a ball, but gives the ball class a very low score, we can assume that this particular set of parameters are not good. The loss function is therefore a measure of the quality of a particular set of parameters based on how well the network scores align with the ground truth labels in our training data.

There are several types of loss functions from hinge loss[14] to cross-entropy loss[36], which all produce a loss function landscape, using all possible combinations of the parameters. This landscape can be traversed by changing the parameters of the network.

#### 2.1.3.2 Optimization

The goal of optimization is to find the set of parameters that minimizes the loss function. This can be viewed as traversing the loss landscape, by updating the parameters, in order to find the lowest valley. There are several ways of doing this, but the most common strategy is to follow the gradient through gradient descent. To follow the gradient, we first compute the gradient of the loss function with our current set of parameters and then perform a parameter update in the negative direction of the gradient. This is done iteratively for each example or, batch of examples, until the optimal set of parameters are found.

#### 2.1.3.3 The Training Process

Training Neural Networks usually follows a set structure in which the network is fed some training data, a loss is calculated based on the outputs of the network and the true value for the data. The network then uses the backpropagation algorithm, to perform a backward pass to find the appropriate weights adjustments for all the weights and update the weights with these adjustments.

**Epochs**
When the network has seen all the available training data it has finished one *Epoch* of training. A network usually requires several epochs of training before it converges on a good set of weights.

**Mini-Batches**
In the earlier days of Neural Networks it was common to feed the network an individual training example, calculate the loss for this example and updating the network weights for this example through gradient descent in the backward pass. In recent years, however, it has become common to compute the loss over several training examples before preforming the backward pass. This collection of training examples is called a *Mini-Batch*. When using mini-batches it is very important to

shuffle the training dataset at the start of each epoch. This ensures that the mini-batches do not contain the same examples for every epoch and avoids the network overfitting to individual mini-batches. Correctly using mini-batches usually results in smoother convergence as the gradients computed in the backward pass use more training examples.

#### 2.1.3.4 Data Preprocessing

Data collected in the real world is generally suffering from several drawbacks in relation to machine learning. It may be incomplete, thus lacking values or certain attributes. It may be noisy, containing errors or statistical outliers, skewing the data. And it may be inconsistent, containing discrepancies in codes or labels, such as mislabeled data. Data preprocessing is a commonly used step to combat these issues as it transforms the raw data into an understandable format. In Deep Learning, there are several types of data preprocessing schemes, but the two most common are:

**Zero-Centering**
This is the most common form of preprocessing. To zero-center data, the mean is subtracted across every individual feature in the dataset. This results in centering the datacloud around the origin along all dimensions. For images it is common to perform this step by subtracting the the dataset mean from all images.

**Normalization**
The normalization process involves normalizing the data dimensions in order to make them approximately the same scale. The most common way of doing this is to divide each dimension by its standard deviation.

#### 2.1.3.5 Regularization

As we described earlier, a common problem when training Neural Networks is *overfitting*. This happens when the network learns the details and noise in the training data to an extent that negatively impacts the model performance on the validation data. To avoid this problem, several ways of controlling the learning capacity of Neural Networks have been devised:

**Dropout**
Dropout[44] is a regularization technique which involves keeping a neuron active during training with some probability $p$, and otherwise turning it of by setting it to zero. This essentially trains an ensemble of networks, consisting of all sub-networks that can be formed by removing non-output units from an underlying network.

**Batch Normalization**
Batch Normalization[24] is a technique developed to tackle the problem of *internal covariate shift* in Deep Neural Networks. Internal covariate shift is the change in the distributions in network activations due to the change in network parameters during training. The Batch Normalization layer accounts for this problem through shifting its inputs to zero mean and unit variance for each mini-batch, resulting in a normalized input. The exact steps of the batch normalization transform applied to activation, $x$, over a mini-batch, $\mathcal{B}$, is given in equation (2.5) and was first presented by Ioffe and Szegedy in [24].

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$
**Output:** $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$
\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i && \text{// mini-batch mean} \\
\sigma^2_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 && \text{// mini-batch variance} \\
\widehat{x_i} &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} && \text{// normalize} \\
y_i &\leftarrow \gamma\widehat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i) && \text{// scale and shift}
\end{aligned}
$$

Here $\epsilon$ is a constant added to the mini-batch variance for numerical stability.

(2.5)

**Data Augmentation**
Data Augmentation is a method for boosting the size of the training set to help to avoid that the model memorizes it. There are many different ways to perform data augmentation, but it is most common to augment the data in the ways the model is supposed to be invariant to. If a model is supposed to be invariant to rotation, the data augmentation could include various forms of rotation to the original data. Data augmentation can also be preformed online, meaning that the data is augmented with a probability $p$ as it is being loaded, instead of having the augmented data stored. This reduces storage space and means that the model will be presented with differently augmented data every time.

### 2.1.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are very similar to regular Neural Networks as the same principles are being used and the network still expresses a single differentiable score function. The main difference lies within the fact that a CNN assumes that its inputs are matrices of numbers, such as images, for image classification, or sentence matrices, for natural language processing. This allows for the convolution operator to be encoded. CNNs consist of three main building blocks. Convolutional layers, pooling layers and fully connected layers. These layers are stacked on top of each other to form a finished CNN.

**The Convolution Operator** The convolution operator is the fundamental basis of the convolutional layers in CNNs. Given a two-dimensional input such as an image, $I$, it uses a small matrix of weights known as a filter, $F$, of size $h \times w$, to compute the convolved image or activation map of that filter as shown in figure 2.3. The activation map is computed using equation (2.6).

$$(I * F)_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} F_{ij} \bullet I_{x+i-1,y+j-1} \tag{2.6}$$

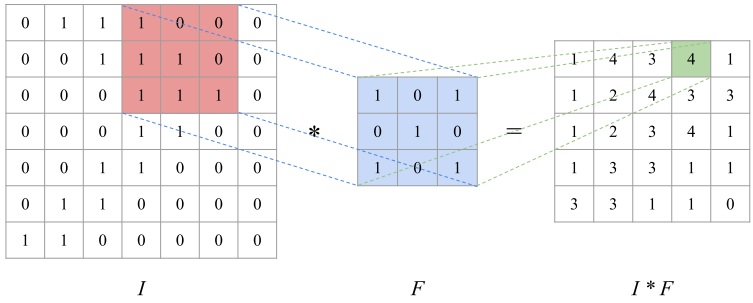
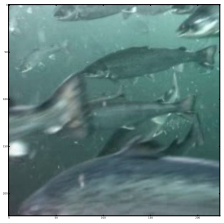



**Figure 2.3:** A figure showing an overview of the Convolution Operation. The activation map $I * F$ is computed by sliding(or convolving) the filter $F$ over the input image $I$ and computing the dot product between the filter and its current location on the input image. Figure adapted from [49].

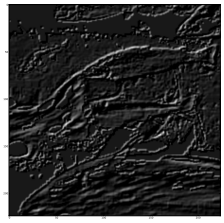
#### 2.1.4.1 Convolutional Layer

The convolutional layers are the main layers of CNNs. These layers consist of a set of several learnable filters. The filters are slid, or convolved, over the width and height of the input volume, computing the dot product. This produces that filter's 2D activation map of the input as seen in figure 2.4. The filters act as feature extractors and activate when they see a particular type of visual feature that excites them. In the first, most basic layers, this can be edges or blobs of colors and in the later layers, we see more advanced patterns such as circles or faces. The filters, together with individual neuron biases are what is learned in the learning process

for CNNs. A convolutional layer usually contains multiple different filters, which in turn produce multiple different activation maps. Thus, the convolutional layer produces a stack of these activation maps along the depth dimension called the output volume.



**(a)** Input image.



**(b)** 2D Activation map.

**Figure 2.4:** The figure shows an input image**(a)** and an example activation map**(b)** from the first Convolutional layer in VGG-16[41]. The activation map is produced by performing the convolution operation over the input image, using one of the 64 filters in the first Convolutional layer. We see that this particular filter seems to act like an edge detector.

It is important to note that the neurons in a convolutional layer are not connected to all the neurons in the previous layer's output volume. Instead each neuron is only connected to a local region of the volume as shown in figure 2.5, called the neuron's *receptive field*. The connections in the receptive field are local in the spatial dimensions, but always full through the depth dimension, meaning that the receptive field is full along the entire depth of the input volume.

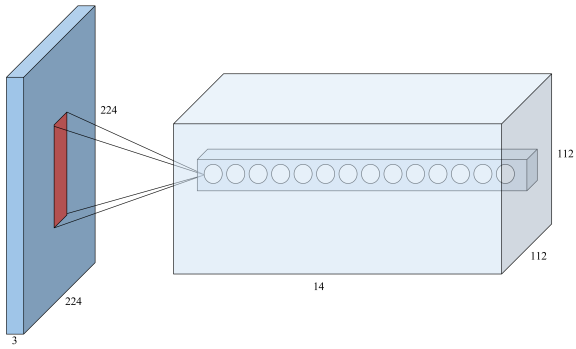



**Figure 2.5:** A figure showing the input volume(left) and an example output volume of neurons(right) in the first Convolutional layer. Each neuron is only connected to a spatial region, called the neurons Receptive Field, shown in red. Figure adapted from [30]

Another important property of the convolutional layer, is the use of *parameter sharing*. This assumes that if it is useful to compute a feature at one spatial position

of the input $(x, y)$, it should also be useful to compute this feature at a different position $(x_2, y_2)$. If we denote a 2D slice of depth as a *depth slice*, shown in figure 2.7, we say that all the neurons in a depth slice are sharing the same filter. This results in a dramatic reduction in parameters in each layer, as these parameters are shared within depth slices. This makes CNNs able to handle large image data inputs, without suffering from slow training times, as the amount of parameters in the network is still manageable.

#### 2.1.4.2 Pooling

It has become common practice to insert a *pooling layer* between a set of convolution layers in most CNNs. The pooling layer reduces the spatial size of the representation in order to reduce the number of parameters in the network. The pooling layer operates on each depth slice independently and resizes it in the spatial dimension. The most commonly used pooling version has a filter size of $2 \times 2$, a stride 2 as seen in figure 2.6. The most common pooling layer is the maxPool layer. The maxPool filter selects the maximum value over a square of $2 \times 2$ numbers and outputs that number. A stride of 2 corresponds to the filter being moved two steps to the side or down for each calculation. This results in the number of activations being decreased by 75% as seen in figure 2.7. There are also other functions such as AveragePooling and L2-normPooling. However, MaxPooling is the preferred pooling function, as it often performs better in practice.

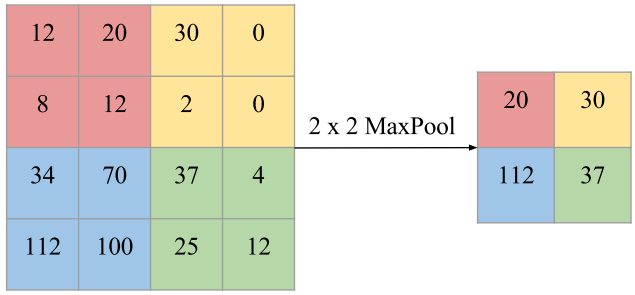



**Figure 2.6:** An illustration showing a $2 \times 2$ MaxPooling with stride 2. Each max is taken over a $2 \times 2$ square. The filter is then moved two squares for the next computation. Figure adapted from [30].

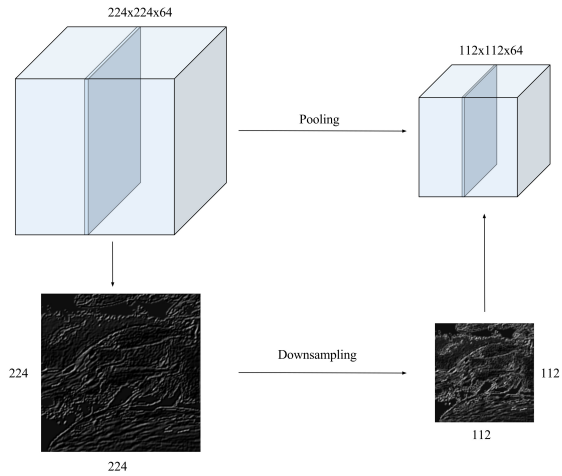



**Figure 2.7:** The pooling layer downsamples the volume in the spatial($224 \times 224$) dimension independently for each of the 64 depth slices of the volume. The volume is pooled with a filter size of $2 \times 2$ and a stride of 2, resulting in the spatial dimensions being halved. Note that the volume depth is preserved. Figure adapted from [30].

#### 2.1.4.3 3D-Convolutions

Traditional CNNs are two-dimensional CNNs. This means that they are using 2D filters and produce a 3D volume of 2D depth slices as their output. It is, however, very possible to extend this type of layer to become three-dimensional Convolutional Layers. This is done by increasing the dimensionality of the filters to 3D and increasing the dimensionality of the input. This results in a 4D volume of 3D depth cubes as the output. For videos, this can be done by stacking sequential video frames together, producing a cube of frames as the input. The 3D filters are then convoluted over this cube in both the spatial and depth dimensions. This produces depth slices that not only learn features in a single image, but also how they transform through time in a video. This results the network learning *spatio temporal* filters that are able to extract useful features in both space and time.

### 2.1.5 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) separate themselves from regular neural networks through their ability to handle sequences of input. Regular neural networks can only accept a fixed-sized input vector and only calculate a corresponding fixed-sized output vector. They also use the exact same steps for every input example to calculate their output. RNNs, on the other hand, are able to handle both single example input vectors and sequences of input vectors. They can also produce single outputs as well as sequences of outputs, making these networks very flexible. They do this, using loops to pass information from one time-step of the network to the next. This property enable RNNs to handle sequential input data, such

as sequences of words for natural language processing for transforming speech to text[17] or machine translation[46]. There are several different types of RNNs, but all of them can be visualized as a chain of repeating modules, known as cells, as seen in figure 2.8. The most common types are Vanilla RNNs and Long Short-Term Memory (LSTM) RNNs.

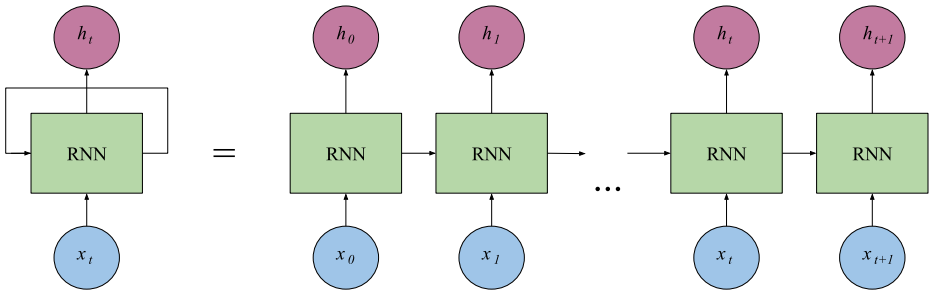



**Figure 2.8:** The figure shows a chunk of a RNN in the looped form(left) with the inputs $x_t$ and outputs $h_t$ at time-step $t$. The loop allows the network to pass information from one time-step to another. The the unrolled form(right) shows the same network, but with each time-step discretely visualized.

### Vanilla Recurrent Neural Networks

The Vanilla RNN cell architecture is very simple. It consists of an input vector, a hidden state and an output vector. The hidden state is calculated using the input vector and the previous state as shown in equation (2.7) and the looped form in figure 2.8, where $W_{hh}$ and $W_{xh}$ are learnable weight matrices. Then the output vector is calculated as shown in equation (2.8). In other words, the network uses the previous state to calculate the output at the current state and since the previous state was calculated the same way, the network uses the entire sequence to produce the current output.

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{2.7}$$

$$y_t = W_{hy} \bullet h_t \tag{2.8}$$

### Bidirectional Recurrent Neural Networks

Regular RNNs are only able to process information from the past, through previous examples in a sequence. They are therefore not able to use the entire input sequence, with examples from both the past and the future to produce their current-state output. Bidirectional RNNs[39], on the other hand, are able to use both future and past examples in an input sequence to produce its current-state output. They do this by processing the data in both directions with two separate hidden layers, which are then fed to the same output node, as seen in figure 2.9.

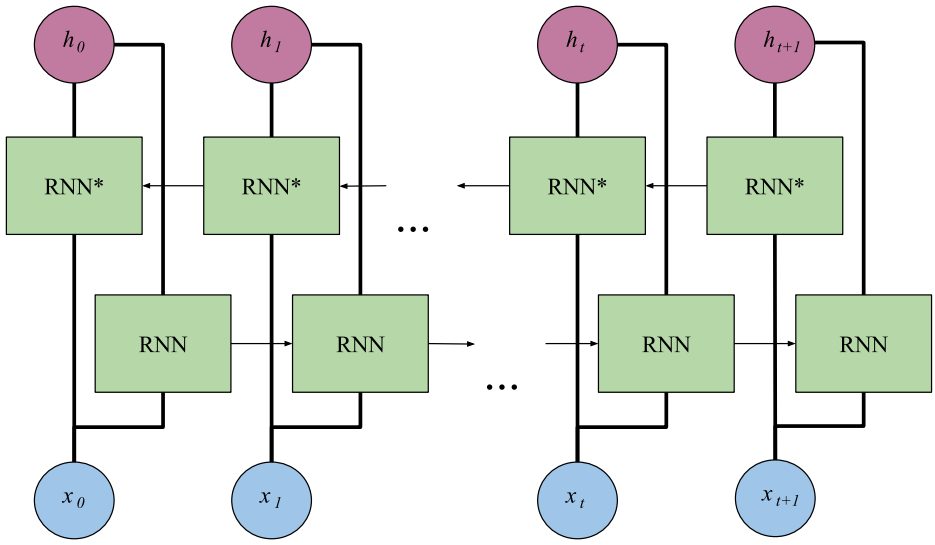



**Figure 2.9:** The figure shows a Bidirectional RNN with both the forward and backward layers. This enables the network to use both past($x_{t-n}$) and future($x_{t+n}$) inputs from the sequence to produce the current output $h_t$.

### Vanishing and exploding gradients

Vanilla RNNs commonly suffers from a problem of learning dependencies over longer sequences. Vanilla RNNs can be visualized as a deep feed forward neural network, where each timestep corresponds to one layer in the unrolled network as seen in figure 2.8. Since the derivative of the tanh function is close or equal to zero in the saturated areas, multiple matrix multiplications, as a result of multiple timesteps, can result in the gradient completely vanishing for distant timesteps. This then results in no weight updates being made and the network stops learning from these timesteps. Conversely, the gradient might also explode, resulting large updates being made to the weights and thus disrupting the learning process. In section 2.1.3, we saw that a common way to initially combat these problems was to use the ReLU activation function. However, for RNNs, the introduction of the Long Short-Term Memory network is the preferred solution.

**Long Short-Term Memory Recurrent Neural Network**
LSTM networks are specially designed to combat the problem of vanishing and exploding gradients. The way LSTM RNNs do this is through their special cell architecture. The LSTM architecture's main component is the cell state $C_t$ as seen in figure 2.10. The cell state is a highway for information with only two linear interactions. This enables information to flow through the cell largely unchanged. The LSTM cell is also capable of removing and adding carefully regulated information to the cell state through three gates. The three gates are the forget gate($f_t$), the input gate($i_t$) and the output gate($o_t$). Each gate has its own set of learnable weights $W_f, W_i, W_o$ and biases $b_f, b_i, b_o$.

**The forget gate** decides what information is going to be removed from the cell state. It takes $h_{t-1}$ and $x_t$ as input and calculates the output as shown in equation (2.9)

$$f_t = \sigma(W_f \bullet [h_{t-1}, x_t] + b_f) \tag{2.9}$$

**The input gate** decides which values in the cell state is going to be updated. It works in conjunction with a vector of candidate values $\tilde{C}_t$ to decide what information is going to be added to the cell state. This is done through the operation $i_t * \tilde{C}_t$, where $i_t$ and $\tilde{C}_t$ are given by (2.10) and (2.11) respectively.

$$i_t = \sigma(W_i \bullet [h_{t-1}, x_t] + b_i) \tag{2.10}$$

$$\tilde{C}_t = tanh(W_C \bullet [h_{t-1}, x_t] + b_C) \tag{2.11}$$

**The output gate** decides what parts of the cell state is going to be output from the cell. This is done through equation (2.12)

$$o_t = \sigma(W_o \bullet [h_{t-1}, x_t] + b_o) \tag{2.12}$$

The output of the LSTM cell is based on the current cell state and the output gate. The cell state is updated by the forget gate and the input gate as shown in (2.13) to produce the current cell state $C_t$, which is then squashed by a *tanh* and multiplied with $o_t$ to produce the final output $h_t$ of the LSTM cell, as shown in equation (2.14).

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.13}$$

$$h_t = o_t * tanh(C_t) \tag{2.14}$$

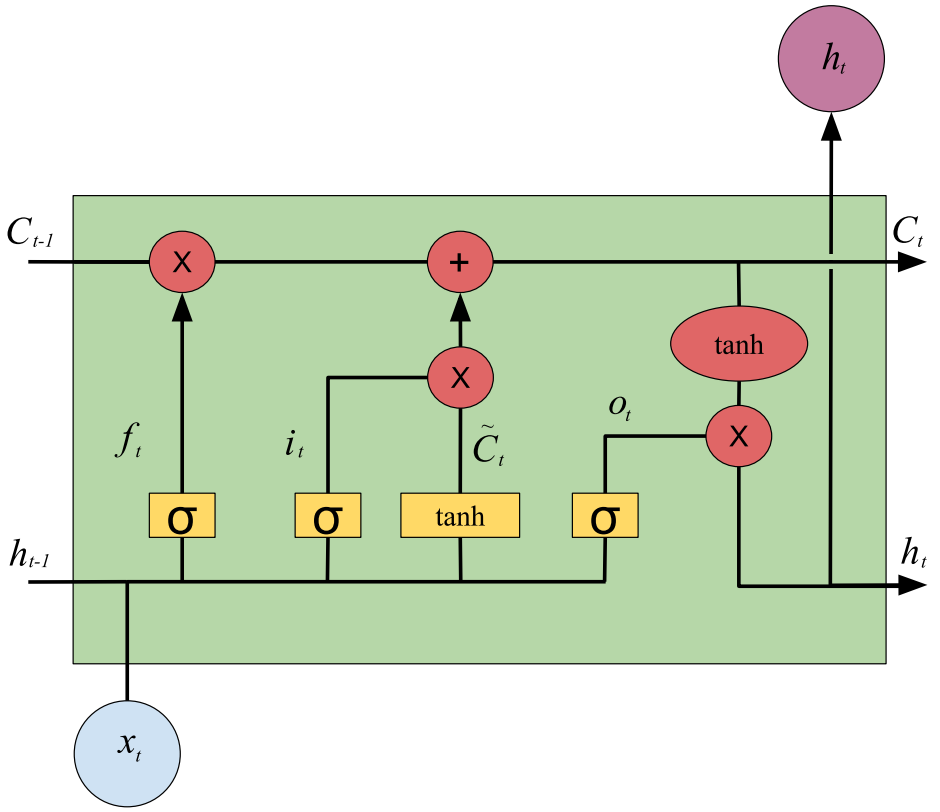



**Figure 2.10:** The LSTM takes the previous cell state, $C_{t-1}$, the previous output, $h_{t-1}$ and the current input, $x_t$ as input. The cell state of the LSTM is then modified by the forget gate, $f_t$, and the input gate, $i_t$ to produce the current cell state, $C_t$. The output, $h_t$ of the LSTM is then produced by feeding the cell state through a tanh activation and then the output gate $o_t$.

## 2.2 Optical Flow

Optical Flow is a pattern illustrating the motion of image objects between two or more consecutive frames. These motions are usually caused by the movement of the objects in the frame, but can also be caused by the movement of the camera. Optical Flow works on two main assumptions:

1. The pixel intensities of an object do not change between consecutive frames
2. Neighboring pixels have similar motion,

If we consider a pixel $P(x_0, y_0, t_0)$ in frame 0. This pixel is moved by a distance $(dx, dy)$ in frame 1, taken after $dt$ time. We therefore end up with equation (2.15)

$$P(x_1, y_1, t_1) = P(x_0 + dx, y_0 + dy, t_0 + dt) \tag{2.15}$$

We can then take the taylor series approximation of the right-hand side, remove the common terms and divide by $dt$ to get equation (2.16)

$$f_x u + f_y v + f_t = 0 \tag{2.16}$$

where $f_x$, $f_y$, $u$ and $v$ are given by the Optical Flow equation, shown in equation (2.17). We see that $f_x$ and $f_y$ are the image gradients and that $f_t$ is the gradient along the time dimension.

$$\begin{aligned} f_x = \frac{\delta f}{\delta x}; \quad f_y = \frac{\delta f}{\delta y} \\ u = \frac{dx}{dt}; \quad v = \frac{dy}{dt} \end{aligned} \tag{2.17}$$

There are several methods to find $u$ and $v$, but a common one is the Gunner Farneback's algorithm[12]. This algorithm produces a 2-channel array of Optical Flow vectors $(u, v)$ with magnitude and direction as shown in figure 2.11.

**(a)** Spatial image.

**(b)** The resulting Optical Flow image.

**Figure 2.11:** The conversion from spatial image to Optical Flow using Gunner Farneback's algorithm on our videos of salmon. The direction and magnitude corresponds to the hue and value planes respectively, resulting in different colors for different directions of movement.

## 2.3 Human Activity Recognition

This section will give a brief overview of the field of Human Activity Recognition.

### 2.3.1 Action Recognition and Action Detection

Human activity recognition has spiked interests in several industries involved with computer vision in recent years. Human activity recognition is a field concerned with classifying human actions performed in videos. It can be separated into two subtasks:

1. Human Action Recognition
2. Human Action Detection

**Human Action Recognition**
Human Action Recognition involves classifying individual videos. For this task, each video contains only one class of action, and the goal is for the system to accurately classify the action performed in the video.

**Human Action Detection**
Human Action Detection, on the other hand, is concerned with detecting actions through continuous videos. This means that any given video contains multiple classes of actions and the goal of the system is to accurately segment the video into correctly classified segments.

The field Human Activity Recognition has become an important research domain, spanning different applications, such as sport analysis [43], human computer interaction[35], and video surveillance[45]. It is also of general interest to the field of computer vision as it expands the ability of machines to understand the contents of video. There are several standardized datasets for human Activity Recognition, but the most commonly used are the UCF-101[43] and Sports1M[26] Action Recognition datasets. These datasets include videos of different humans, performing several classes of actions from several different angles under a variety of conditions.

Human Activity Recognition is considerably more challenging than regular image classification, as it relies on videos for inputs. This combines the challenges of both image recognition and sequence handling, as videos are constructed of sequences of single image frames. Since 2012, we have seen complete domination in both image recognition[38] tasks and natural language processing tasks[40] through the use of CNNs and RNNs. This has lead researchers to believe that a combination of these techniques could do the same for Human Action Recognition. Thus, recent years have seen a dramatic increase in use of Deep Learning architectures for Human Action Recognition tasks. In section 2.5, we show that the the current state-of-the-arts performance on both UCF-101 and Sports1M are driven by Deep Learning approaches.

## 2.4 The Salmon Activity Domain

This section will present the Salmon Activity Domain and give an overview of how it distinguishes itself from Human Activity.

### 2.4.1 Salmon in Videos

The Salmon Activity Domain distinguishes itself from the Human Activity in several ways. The most noticeable is that there are only two classes of actions in the salmon domain whereas the standard human Activity Recognition datasets often include over 100 classes. The two classes in Salmon Activity Recognition are:

1. Feeding
2. NonFeeding

The Feeding class contains videos of salmon that are feeding throughout the entire video. The NonFeeding class contains videos of salmon that are not feeding for the entire duration of each video. This makes the Salmon Activity Domain very suitable for Action Recognition. Another area where the two domains differ significantly are the surroundings in the videos. The human domain include a vast variety of surroundings from outdoor river rafting to indoor keyboard typing. This makes many of the classes immediately recognizable purely on the basis of their surroundings. Conversely, the salmon domain is limited to underwater video from within a breeding cage at sea. Thus the surroundings in the videos from both classes in the salmon videos are very similar.

The fact that the videos are captured under water also introduces several different challenges, as light behaves quite different in water, compared to air. When light hits the surface of the ocean it is reflected off the surface to a varying degree, depending on the state of the water. The rougher the surface, the more light is reflected, resulting in a dimmer scene for the camera to capture. The light that does penetrate the surface is also refracted as light travels at different speeds in air and water. Water also scatters and absorbs different wavelengths of light due to particles in the water, resulting in different shades of color based on the depth of the camera position. These factors may be further enhanced by the weather at the facility where the videos are captured. Direct sunlight produces very different lighting conditions compared to overcast weather. This is evident, not only in the shades of color and amount of light in the scene, but also in the amount of light reflected off of the fish themselves. As more direct sunlight is present in the scene, more light is also reflected off of the fish, producing very bright areas in the frames as seen in figure 2.12.

**(a)** A frame from a day with direct sunlight. The specular reflection off the salmon produces very bright areas in the frame.

**(b)** A frame from a day with overcast weather. There are very few bright areas in the image.

**Figure 2.12:** The figure shows the difference between a sunny day and an overcast day. In **(a)** the sunlight is reflected off of the salmon, producing very bright areas in the frame. In **(b)** there is no direct sunlight and therefore very few bright areas in the frame.

The final major difference between the salmon and human domain is that human activities often can be classified purely based on the pose of the human in the video. If a human is posed to kick a football, we might quite easily classify the actions as "playing soccer" just by observing the human in that defining pose. This has also been observed in Deep Learning approaches through the use of discriminative action poses to supplement videos during training[32]. Fish, on the other hand, might not possess such defining poses. They are also often occluded by other fish, thus it might not be possible to discern whether fish are eating or not based on their pose alone. We therefore believe we are more dependent on the motion of the fish through time in the Salmon Activity Domain. Another aspect of the Salmon Activity Domain is the fact that fish act together in a shoal, thus shoal behavior and pose might be just as indicative of Feeding or NonFeeding fish as individual fish behavior and should therefore also be considered when developing solutions for this domain.

## 2.5 Previous Work

During the fall semester of 2016, a Specialization Project was done in preparation for this master's thesis and as part of the subject TDT4501 – Computer Science, Specialization Project. Since research on Salmon Activity Recognition is limited at best, the goal of the Specialization Project was to create an overview of the state-of-the-art in Human Action Recognition approaches. The project was then aimed to conclude with proposing an architecture for Salmon Activity Recognition, using the findings in human action recognition as inspiration. To create an overview of the state-of-the-art in Human Action Recognition, a *systematic literature review* was performed.

### 2.5.1 Systematic Literature Review

A systematic literature review is a process in which the available primary studies on a subject are gathered and filtered through a series of well-defined steps. This filtration produces a final set of studies containing the most relevant research. This approach is most prevalent in medical research, but has seen increased use in other fields in recent years.

According to Kitchenham and Charters[27], a systematic literature differentiates itself from unsystematic reviews by using a strict framework of well-defined steps, which in turn are being performed according to a predefined protocol. This enables other researchers to reproduce the results from the literature review.

**The systematic literature review**
The systematic literature review was conducted through four main phases:

1. **The research question phase**
   This phase required a set of literature research questions to be formulated. The goal was for these questions neither to be too specific nor too vague, and to still clearly target the problem faced in the project. It resulted in the following research questions:

   **LRQ1:** How can a combination of CNNs and RNNs be used to do action recognition in videos?

   **LRQ2:** How do the different architectures found compare to each other?

   **LRQ3:** What implications do these approaches have for the domain of action recognition in salmon?

2. **The search phase**
   The goal of this stage in the review process is to find all the studies that are relevant to the stated literature research questions. We did this, using a two-step search strategy:

   (a) The first step was to identify which sources that should be searched. This was done by locating relevant online digital libraries and search engines.

(b) The second step was to create a structured way of searching the sources. This was done by building a search string through the use of key search terms and logical operations. This enabled us to search 48 search strings in one search through permutations of the search terms.

3. **The filtering phase**
   The goal of the filtering phase is to filter out the irrelevant articles and to create a final selection of articles. To achieve this, a set of inclusion and quality criteria were developed to help with the objectivity in the selection process. Articles were filtered through both the inclusion criteria and the quality criteria to create the final set of selected articles. This resulted in the articles seen in table 2.1 to be included in the review:

**Table 2.1:** The final selection of included articles along with their study IDs, authors and publish year.

| **ID:** | **Title:** | **Authors:** | **Published year:** |
|---|---|---|---|
| ST001 | Long-term Recurrent Convolutional Networks forVisual Recognition and Description | Donahue, J. et al.[10] | 2015 |
| ST002 | Recurrent Neural Networks and Transfer Learning for ActionRecognition | Giel, A. and Diaz, R.[15] | 2015 |
| ST003 | Beyond Short Snippets: Deep Networks for Video Classification | Yue-Hei Ng, Joe, et al.[52] | 2015 |
| ST004 | ARCH: Adaptive recurrent-convolutional hybrid networksfor long-term action recognition | Xin, Miao, et al.[50] | 2015 |
| ST005 | Multimedia Event Detection via Deep Spatial-Temporal NeuralNetworks | Hou, Jingyi, et al.[22] | 2016 |
| ST006 | A Very Deep Sequences Learning Approachfor Human Action Recognition | Lin, Zhihui, and Chun Yuan[31] | 2016 |
| ST007 | A Multi-Stream Bi-Directional Recurrent Neural Network forFine-Grained Action Detection | Singh, Bharat, and Ming Shao [42] | 2016 |
| ST008 | Recurrent Convolutional Neural Networks for VideoClassification | Xu, Zhenqi, Jiani Hu, and Weihong Deng[51] | 2016 |

4. **The analysis phase**
   The goal of the analysis phase was to perform an analysis of the final set of articles with regards to the literature research questions. This included a comparison of the different architectures and an analysis on how the architectures could be used as inspiration for the salmon action recognition domain.

### 2.5.2 Proposing an architecture

After we had concluded the systematic literature review, we used our findings to propose several different architectures aimed at performing Activity Recognition on salmon. The architectures were also developed with extra considerations to the Salmon Activity Recognition Domain. The proposed architectures were explained and compared to each other based on a set of *Architectural Requirements* developed from analyzing the Salmon Activity Recognition Domain.

**AR1:** The architecture for the Salmon Activity Recognition Domain should be robust to changes in light conditions.

**AR2:** The architecture for the Salmon Activity Recognition Domain should consider both individual fish movements as well as the movement of the entire shoal of fish.

**AR3:** The architecture for the Salmon Activity Recognition Domain should be able to handle longer sequences of video.

**AR4:** The architecture for the Salmon Activity Recognition Domain should be possible to implement and test for a single student as the work for a Master's thesis.

**The Dual-Stream Network(DSN)**
The Dual-Stream architecture features two streams of VGG-16[41] CNNs. The first stream handles still frames extracted from videos, while the second stream uses optical flow as its input. The feature vectors are extracted from fc6 in both networks and are then combined as input to a set of two fully connected layers. Sequences of feature vectors from fc8 are extracted and used as input to a bi-directional LSTM RNN for the final sequence processing. The LSTM network is topped with a softmax to produce the final score for a sequence. The design can be seen in figure 2.13.

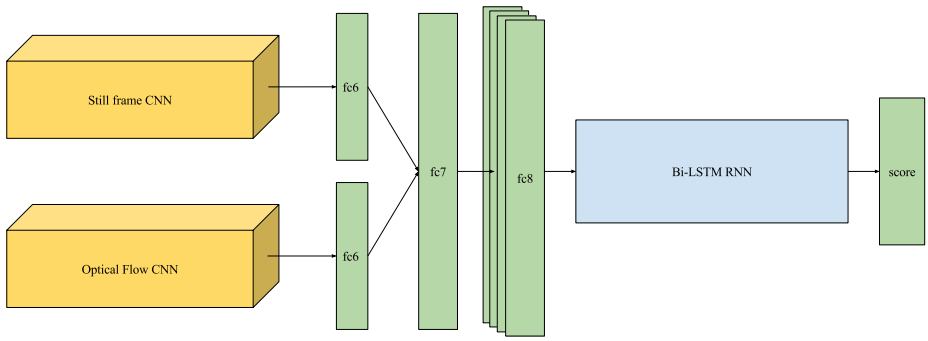



**Figure 2.13:** The Dual-Stream Network. This network uses two CNNs, one with regular video frames as input and one with Optical Flow frames as input. It then combines the feature vector outputs from the two networks to produce sequences of feature vectors, which are then used as input to a RNN.

**The 3D-Convolutional Dual-Stream(3D-DS)**
This approach is very similar to the Dual-Stream architecture. It uses one VGG-16 CNN for still frames and a network for optical flow. The difference comes in the optical flow network. Taking inspiration from ST004, we proposed a 3D-convolutional optical flow network in the hopes of capturing more temporal features when compared to the DSN. The rest of the architecture is exactly the same as the Dual-Stream and uses a bi-directional LSTM network with a softmax to produce the final scores for a sequence. The design of the network can be seen in figure 2.14

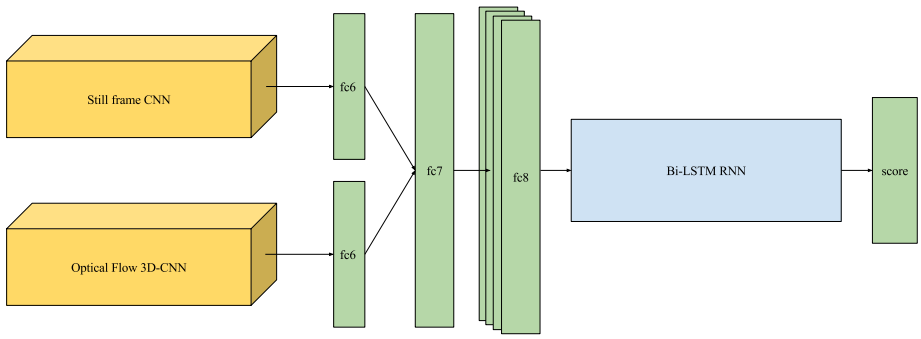



**Figure 2.14:** The 3D-Convolutional Dual-Stream Network. This network uses a regular CNNs taking video frames as input and one 3D-Convolutional Neural Network (3D-CNN), taking Optical Flow frames as input. It then combines the feature vector outputs from the two networks to produce sequences of feature vectors, which are then used as input to a RNN.

**The Multi-Stream 3D-Convolutional Network(MS3D)**
This architecture is heavily inspired by the architecture described in ST007. It consists of four CNNs, where two of them are taking still frames of video as input and two are using optical flow. The two different types of network both contain one network for handling full frames and one network for cropped action regions using bounding boxes around individual fish. This allows for motion from both individual fish and the shoal of fish to be captured explicitly. The Optical Flow networks are also inspired by ST004 and use 3D-CNNs to capture even more temporal information. The four networks are then fused using two fully connected layers before sequences of activations are fed into a bi-directional LSTM network which calculates the final score using a softmax. The design of the network can be seen in figure 2.15

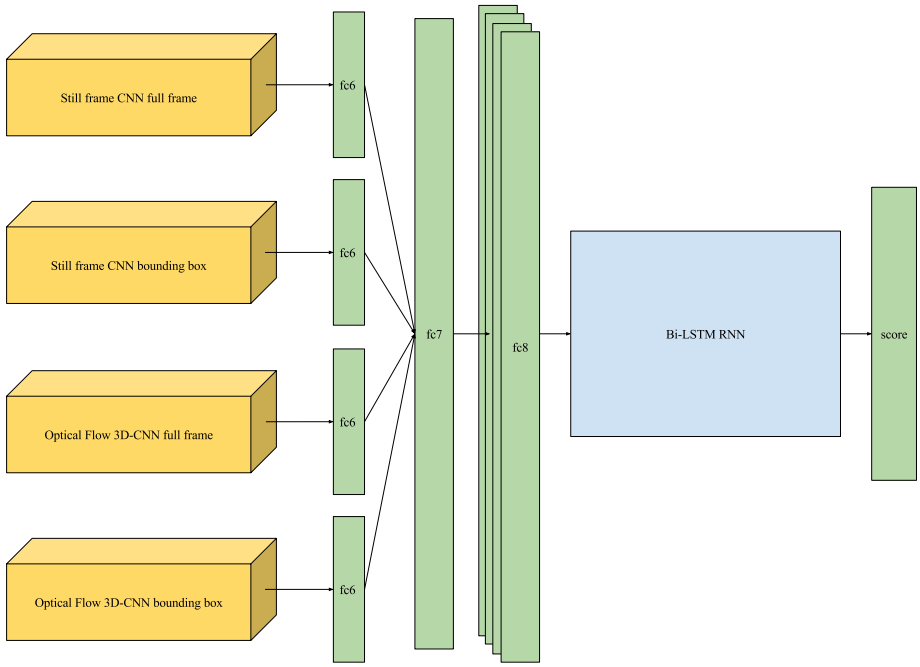



**Figure 2.15:** The Multi-Stream 3D-Convolutional Network. This network uses two regular CNNs and two 3D-CNNs. One of the regular CNNs takes full video frames as input, while the other takes cropped still frames, cropped around the action, as input. The 3D-Convolutional Networks use a similar approach, but takes Optical Flow frames as input instead. The Multi-Stream 3D-Convolutional Network then combines the feature vector outputs from the four networks to produce sequences of feature vectors, which are then used as input to a RNN.

**The Full Motion Network(FMN)**
This architecture takes inspiration from both ST008 and ST004 to produce an architecture that is highly focused around motion features. It contains two streams of two different types of CNNs. The first stream is a Recurrent Convolutional Neural Network (RCNN) where all the convolutional layers are replaced with Recurrent Convolutional layers as described in ST008. To further capture temporal and motion features, the fully connected layers of the CNN architecture is replaced with LSTM networks topped with a softmax classification layer. The second stream consists of a 3D-CNN which takes Optical Flow as input. The sequences of feature vectors outputted from this network are then stacked and fed into an LSTM network topped with a softmax classification layer. Finally the two streams are fused using a weighted average for the outputs, producing the final score. The design of the network can be seen in figure 2.16:

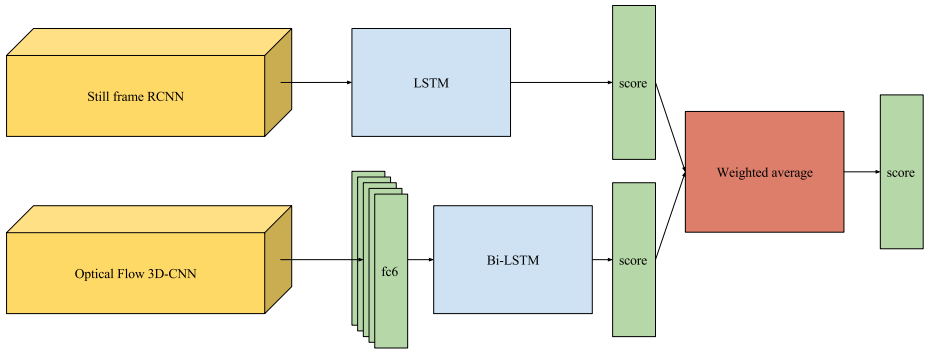



**Figure 2.16:** The Full Motion Network. The network uses a Recurrent-CNN and a 3D-CNN. The feature vectors from the 3D-Convolutional vectors are stacked to produce sequences of vectors which are used as input to a RNN. The outputs from the RCNN and the RNN are then combined through a weighted average to produce the final score.

The four architectures were then compared and ranked based on a weighted sum of *The Architectural Requirements* as shown in table 2.2, resulting in the final proposal of the 3D-Convolutional Dual-Stream. This architecture is, to the best of our knowledge, an original and novel architecture both for addressing the temporal aspects of motion and for the Salmon Activity Recognition Domain.

**Table 2.2:** Ranking of the proposed architectures based on *The Architectural Requirements*. The weights for each *Architectural Requirements* are shown in bold.

| **Architecture:** | **AR1:** | **AR2:** | **AR3:** | **AR4:** | **Total:** |
|---|---|---|---|---|---|
| 3D-DS | 1.0 | 1.0 | 1.0 | 1.0 | 7.0 |
| DSN | 1.0 | 0.0 | 1.0 | 1.0 | 6.0 |
| FMN | 1.0 | 1.0 | 1.0 | 0.5 | 5.5 |
| MS3D | 0.5 | 1.0 | 1.0 | 0.5 | 4.5 |
| **Weights** | **2.0** | **1.0** | **1.0** | **3.0** | **N/A** |

Chapter 3

# Method and Experiments

This chapter will first present the programming libraries used for this thesis. It will then give a description of our dataset and how we have processed it in our work. Next it presents the method and experiments we have used to arrive at the architectures used in the rest of this thesis.

## 3.1 Deep Learning Development Platforms

This section will give an overview of the Deep Learning libraries used during the implementation and testing phases of this thesis. We implemented all models using the Python APIs of the libraries presented.

### 3.1.1 TensorFlow™

TensorFlow™[1] is an open source Machine Learning library developed by Google to meet their needs for a system capable of developing and testing Neural Networks. It uses data flow graphs to do numerical computations, where nodes and edges in the graph represent mathematical operations and tensors respectively. It allows the user to run code on both CPU and GPU, enabling faster computations through parallelization. TensorFlow provides an extensive suite of functions and classes that allow users to build models from scratch with abundant customization options. TensorFlow also facilitate making checkpoints when performing experiments and an extensive amount of visualization options, making it a natural choice for research.

### 3.1.2 TfLearn

A downside of TensorFlow's widely available customization options is the amount of code needed to accurately specify a model. This problem is addressed by TfLearn[7]. TfLearn provides a modular and transparent Deep Learning library, built on top of TensorFlow with the intention to provide a high-level API to TensorFlow. It provides an easier-to-read code structure and requires fewer lines of code to get a model running, while still giving the option for full customization, as all functions are built over TensorFlow. This enables researchers to speed up the experimentation process through faster prototyping and testing.

## 3.2 The Salmon Activity Recognition Dataset

This section will introduce the reader to the dataset used for training, validating and testing of the code implemented during this thesis. It enables the reader to better understand the analysis given in chapter 4 and the design choices made during implementation.

### 3.2.1 Summer Internship Dataset

During the summer internship we were given a dataset consisting of 16 color videos of salmon taken from within a breeding cage. The videos were captured at 25 frames per second with a resolution of 640×197 pixels. Each video had a duration of 8 seconds, resulting in a total dataset size of 3200 frames. Given that standard Deep Learning datasets for image classification range from a few hundred thousand to several million images, this dataset is extremely small. Initial testing also revealed that the content of two of the videos differed significantly from the rest, in that these two videos contained salmon swimming very close to the camera compared to the others. We also found that the videos in the Feeding and NonFeeding classes were easily distinguishable based on the color tint in the videos as a result from the videos being captured during different light conditions.

This required both preprocessing and data augmentation in order to normalize the data as much as possible. We therefore turned all the videos into grayscale to account for the color tint. As described in section 1.1.1, we were able to produce an architecture with classification accuracies around the 65% mark. Given the nature of this dataset we concluded that the dataset was too small and that a new dataset was needed to improve this score.

### 3.2.2 The Master's Thesis Dataset

For this master's thesis we did an acquisition of a new dataset and much larger dataset. It consisted of videos were collected in northern Norway, during November of 2016. They were recorded, using a single breeding cage and a standardized procedure to produce the most consistent recordings possible. The camera was mounted looking inward towards the center of the cage during both Feeding and NonFeeding videos and captured at intervals of 2000, 5000 and 20000 frames throughout the day.

The total dataset consisted of 76 videos, taken at a resolution of 224×224 pixels with RGB color channels and at 24 frames per second. An illustration of the video capture rig is given in figure 3.1. The videos were then labeled according to the feeding times, provided by the feeding operator at the farming facility. However, from comparing the feeding schedule, and video capture times, we found that some were captured at the transition phase between Feeding and NonFeeding fish. These videos were therefore edited to remove this transition phase, resulting in correctly labeled videos. This editing involved removing all parts of a video closer than 3 minutes in time from the transition phases.

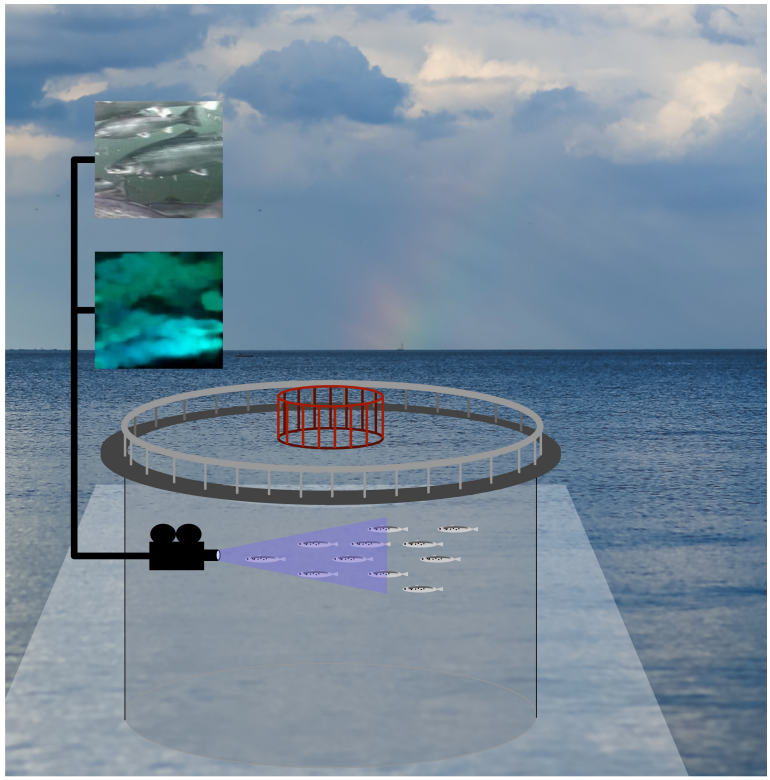


**Figure 3.1:** An illustration showing the camera placement within a breeding cage. The camera is looking in towards the center of the cage, recording the salmon swimming in front of it. The recordings are then processed into regular video and Optical Flow video.

When compared to the Summer Internship Dataset, this has resulted in a much improved dataset with smaller deviations between the videos. The size of the dataset was also significantly increased to around 8.6 hours of video, after it had gone through the labeling process and quality check. This gives us a much better foundation to base our research and we believe that the results presented, using this dataset, will be much more representative than the results from the Summer Internship.

#### 3.2.2.1 Dataset Split

When splitting the original dataset into training, validation and test sets, we decided to split the videos based on dates. This ensures that all videos from a particular date is only present in one of the three subsets. The reasoning behind this split was twofold:

1. The conditions for a particular day might enable the model to overfit on other factors than the behavior of the fish, such as how the camera moves or the light conditions for that day. Splitting on dates avoids this.

2. Splitting the dataset based on dates gives the best representation of the performance that can be expected if the model is deployed on a breeding site and starts receiving new video data.

However, splitting the data in this fashion also creates more challenging validation and test sets. Since we split the dataset on dates instead of individual videos, we increase the possibility of the validation and test sets containing conditions, not well represented in the training set. This split was chosen intentionally in order to give a good representation of performance on new videos and to prove the robustness of our architectures through a challenging dataset split.

#### 3.2.2.2 Dataset analysis

Although the new dataset is significantly larger than the dataset from the summer, it still pose some challenges. Most importantly, the distribution between Feeding and NonFeeding videos is not equal. Of the 8.56 hours of videos, 2.88 hours are Feeding and 5.68 hours are NonFeeding. This corresponds to approximately 249260 Feeding frames and 491540 NonFeeding frames. In order to avoid our models developing a learning bias towards NonFeeding behaviour, we are dependent on having an equal amount of both Feeding and NonFeeding frames in our training set. We also want an equal distribution of Feeding and NonFeeding frames in our validation set in order to get an accurate representation of both Feeding and NonFeeding performance. We are therefore limited to using a subset of the NonFeeding examples for these phases. The exact sampling procedure is described in Appendix A. Another factor is the length of each video. Since the videos ranges from 2000 to 20000 frames per video this produced some difficulties when splitting the data.

The size of the split was set to approximate a 60%, 20%, 20% split, as much as possible. However to ensure an accurate representation of the models performance on new data, we included videos from three separate dates, spread over the entire data collection period in our test set. To achieve the test set size we wanted, we were forced to use dates containing mostly videos of duration 5000 frames. Our validation set only contain videos from a single date to ensure that the training set would be big enough. Thus, the final dataset contains 5.77 hours of video, split over training, validation and test datasets of size 65.55%, 17.20%, 17.25% respectively. The dataset split is also seen in table 3.1.

**Table 3.1:** The table shows the three dataset splits with their respective amounts of frames and percentage of the total amount of data used in this thesis.

| **Dataset Name** | **Number of Frames** | **Hours of Video** | **% of Total Dataset** |
|---|---|---|---|
| Training Dataset | 326768 | 3.78 | 65.55% |
| Validation Dataset | 85760 | 0.99 | 17.20% |
| Testing Dataset | 86000 | 0.99 | 17.25% |
| **Total Dataset** | **498528** | **5.77** | **100.00%** |

**Cage Overfitting**
Since all the videos are collected from a single breeding cage, we note that the model might also become overfit toward this particular cage, learning cage specific fish behavior rather than general salmon behavior. This problem can only be overcome by collecting videos from multiple cages and multiple farming facilities. However, this is not viable for this master's thesis and we will therefore only present results valid for this specific cage. However, we believe that the approaches presented in this thesis are generally valid for all cages.

**Separating Data into Action Snippets**
Since the main measure of performance for this thesis is *Video Action Recognition Accuracy*, we should ideally report this measure of accuracy for each video in our validation set. However, our validation set only contains a total of 9 videos which is to few examples to give a good measure of performance. For training, the number of videos is 20 Feeding videos and 27 NonFeeding videos, giving us a total of 47 training examples for Video Action Recognition. Standard Deep Learning datasets for Action Recognition usually contain anywhere from several thousand to over one million videos in the training data[26]. We therefore decided to segment the videos into shorter Action Snippets for the training and validation phases to produce more examples in our datasets. These Action Snippets range in duration from single frames to 400 frames, depending on the model architecture. The validation results presented in this chapter are therefore reported using Action Snippets Action Recognition accuracies and losses instead of Video Action Recognition accuracies and losses. This gives us a more detailed presentation of architecture performance, thus giving us more data to work with. In our test results, we also provide the Action Snippet Action Recognition accuracies for better comparison with our validation results.

## 3.3 The Dual-Stream Approach

In section 2.5 we arrived at a 3D-Convolutional Dual-Stream as the best suited approach to explore in this master's thesis. This approach was designed to fully utilize the information contained within videos. The approach combines two different Neural Network streams to produce high-level feature vector representations of the videos and then feed these vectors into a Recurrent Neural Network (RNN) for sequence processing. The *Dual-Stream* architecture is shown in figure 3.2

**Spatial Stream**
The first stream is the *Spatial Stream*. It consists of a Convolutional Neural Network (CNN), taking individual video frames as input. By using individual video frames as input, the *Spatial Stream* could able to learn a correlation between the spatial properties of individual fish and the shoal, and which class of behavior is exhibited in the frame.

**Temporal Stream**
The second stream is the *Temporal Stream*. It consists of a 3D-Convolutional Neural Network (3D-CNN), taking Optical Flow, generated from the videos, as input. By using Optical Flow as input, the *Temporal Stream* could learn motion features, thus learning how the motion of the salmon and the shoal is related to Feeding and NonFeeding behavior.

**Recurrent Network**
The *Recurrent Network* takes sequences of high-level feature vectors from both the *Spatial Stream* and the *Temporal Stream* as input and process these as sequences of video frames. This enables the *Dual-Stream* to process both spatial and motion information through time. This could enable it to learn how changes in both the *Spatial Stream* and the *Temporal Stream*, over time, is related to Feeding and NonFeeding behavior.

**The Final Classification Layer**
Since we only have two classes in our dataset, Feeding and NonFeeding, we will be using the same final classification layer for all the architectures presented in the remainder of the thesis. Our classification layer consists of a single output, sigmoid activated, fully connected layer. The sigmoid activation squashes our output to a range between 0 and 1, thus enabling us to represent our two classes as 0 and 1 for NonFeeding and Feeding respectively. When calculating the model accuracies, we round the outputs to the nearest integers by forcing every number $< 0.50$ to 0 and every number $\geq 0.50$ to 1. Thus we have the following prediction intervals for our two classes:

1. **NonFeeding** $= [0.0, 0.50)$.
2. **Feeding** $= [0.50, 1.0]$.

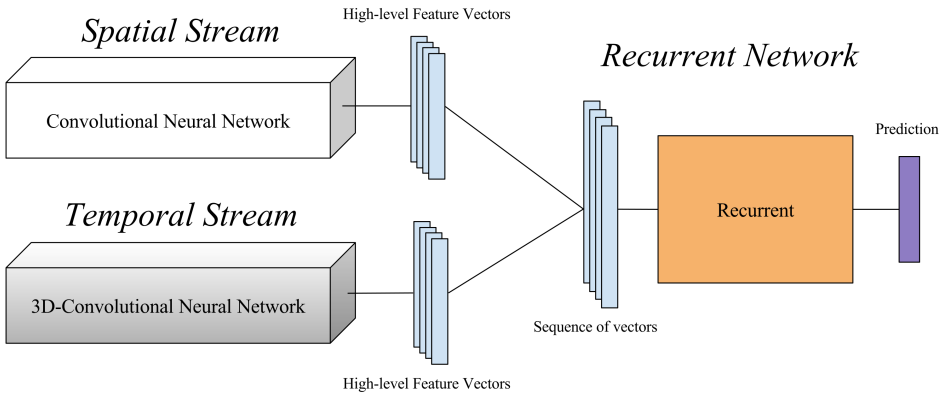



**Figure 3.2:** The *Dual-Stream* approach takes sequences of high-level feature vectors from both a *Spatial Stream* and a *Temporal Stream* as input to a *Recurrent Network* to fully utilize the spatial and motion information contained within videos over time.

## 3.4 Data Preparation

Since our Dual-Stream approach requires both spatial data and Optical Flow data, this section will give an overview of how we generated these two types of data.

### 3.4.1 Spatial Data

The spatial data consists of the individual image frames, extracted from the videos. Since it is very impractical and slow to extract the frames from each video each time they are needed, we created a tree folder structure and extracted all the frames for storage in this structure. The frames were extracted using the OpenCV library[3] and stored in a folder tree that separates test-, training- and validation sets as individual folders. The frames for Feeding and NonFeeding were also separated within each sub dataset, thus the folder structure consists of three sub dataset folders, each with their own Feeding and NonFeeding folders. This enables us to manage the data much more easily while training models as well as giving us a deterministic dataset, that produces the same results given the same initial parameters.

### 3.4.2 Optical Flow Data

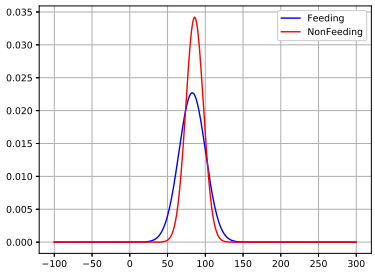
For convenience, we stored the Optical Flow frames in an identical tree folder structure as the spatial data. The Optical Flow dataset was generated using OpenCV's implementation of the Farneback's algorithm [12]. This produces a 2-channel array with Optical Flow vectors which are then stored as a Hue, Saturation and Value(HSV) image. Here, the Direction array is stored in the Hue channel and Magnitutde array is stored in the Value channel, the final Saturation channel is filled with the maximum pixel value of 255. This enables us to color code the

direction of movement between frames, by assigning different colors to different directions of movement.

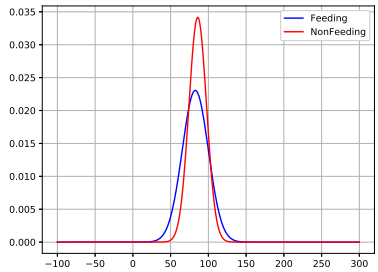
Since Optical Flow is generated from the relative motion of pixels between frames, we also wanted to explore the effects of generating Optical Flow datasets at different sampling rates from the videos. We therefore created three different datasets, where we sampled and generated Optical Flow from every frame, every other frame and every fifth frame. The reasoning behind this was that Optical Flow, generated from more distant frames, might contain more discriminative motion patterns, thus enabling the temporal models to learn better motion features. Figure 3.3 shows the difference in distributions of Hue values for both Feeding and NonFeeding, using the three sampling rates. From the figure we find that there is a significant difference in the swimming directions between the Feeeding and NonFeeding classes. And we can see that videos of feeding salmon contain a lot more variation, indicating that the salmon are not swimming as coherent when feeding. As we can also see from figure 3.3, the distributions for frames sampled at every frame and every other frame look very similar. Indeed a direct comparison, seen in figure 3.4, reveals that their distributions are virtually the same. This has two implications which will be further explored in section 3.6:

1. The two datasets are very similar and contain virtually the same motion information.

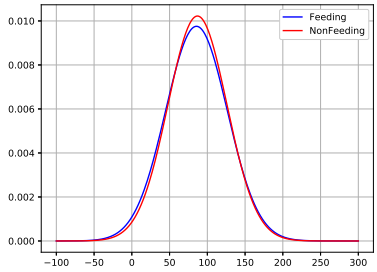
2. The dataset sampled at every other frame is half the size of the dataset sampled at every frame, while still containing the same motion information. This might enable us to produce models that learn the same motion features in half the training time by training on this dataset.




**(a)** Distribution for Optical Flow Hue values, sampled at every frame.




**(b)** Distribution for Optical Flow Hue values, sampled at every other frame.




**(c)** Distribution for Optical Flow Hue values, sampled at every fifth frame.

**Figure 3.3:** Distribution of Optical Flow Hue values, in Feeding and NonFeeding training videos for three different sampling rates. Since the Hue values correspond to the direction of movement, it is clear that the Feeding videos contain a lot more variation in the directions of movement than the NonFeeding videos. It is also clear that this difference is much more visible in the sampling rates using every frame or every other frame than it is in the every 5th sampling rate.

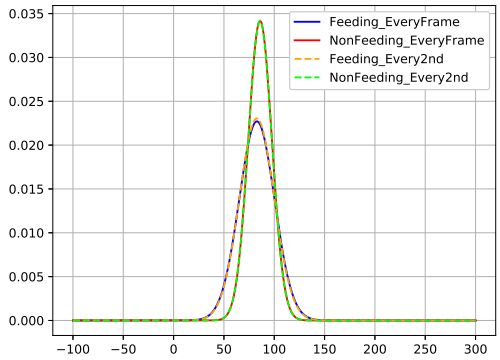



**Figure 3.4:** Feeding and NonFeeding distributions sampled at every frame and every other frame. The two distributions overlap almost perfectly, indicating that the distribution is very similar for both sampling rates.

## 3.5 The Spatial Stream

This section will present the work done while developing the *Spatial Stream* of the *Dual-Stream* approach. It will present the choice between using a pretrained network or training a network from scratch, compare several preprocessing strategies and the use of data augmentation. Finally it will present the final *Spatial Model* used in the rest of this thesis.

### 3.5.1 Transfer Learning

When approaching a new classification task, a common practice in Deep Learning is to use a pretrained model to perform *Transfer Learning*. Transfer learning assumes that a model trained on a similar classification task can be used as a starting point for a model performing a new classification task to shorten training times. However, using a pretrained model might also introduce some negative aspects. If the model is not capable of handling our dataset, we could be forced to process our dataset to meet the model's requirements, risking to loose valuable information in the process. Our dataset could also be very different than the data the model was trained on. This could mean that starting from a pretrained model could result in worse performance than starting from scratch.

For our task, however, we assume that a Neural Network trained on the ImageNet dataset[38] has learned good features for images and thus is a better starting point for our classification task than starting from scratch. We also believe that, since our dataset consists of videos of salmon, it is suitably similar to the ImageNet dataset, especially since images of fish are already part of the ImageNet dataset. Using a pretrained model also introduces two other benefits for our purpose:

1. We do not have to spend time developing and testing our own specialized network architecture.

2. Training time is greatly reduced as the network has already learned good filters and only has to fine-tune these to the new classification task.

We therefore conclude that we fine-tune our *Spatial Stream* from a pretrained model, rather than training it from scratch.

### 3.5.2 The Pretrained Model

For our implementation, we have chosen to use the well known VGG-16 architecture[41]. All our implementations using the VGG-16 architecture are fine-tuned networks, starting from a VGG-16 model trained on a subset of the ImageNet dataset, used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)[38]. This network is trained using single images as input, thus we use Action Snippets consisting of individual frames for the *Spatial Stream*. The specific checkpoint used as a starting point for our models is the checkpoint available through TfLearn, found at [8]. There are three main reasons for choosing VGG-16:

1. Many of the best performing architectures dealing with the Human Action Recognition problem use this network for feature extraction[22, 31, 42].

2. The pretrained weights for VGG-16 is readily available in the libraries we use for our implementation.

3. VGG-16 has a particularly easy to understand structure, which makes our implementation easier to understand and visualize.

**VGG-16**
VGG-16 is a Deep Covolutional Neural Network, designed by Karen Simonyan and Andrew Zisserman. Their goal was to uncover the effects of depth on a network's accuracy on large-scale image recognition tasks. They take a fixed size input of $224 \times 224$ pixel, 3-channel images, use filters with size $3 \times 3$ and maxPoolings with filter size $2 \times 2$ and a stride of 2. They also double the amount of filters after every maxPooling to maintain the amount of parameters. The network is topped with three fully connected layers, where the fist two are of size 4096 and use a ReLU activation function. The final classification layer is a softmax activated layer of size 1000, to reflect the number of classes in the ILSVRC dataset. During training, they also used two *Dropout* layers between the two final fully connected layers. This helped regularize the network and thus improved its accuracy. In their work, Simonyan and Zisserman found that accuracy increased with the network depth and proposed two new architectures in VGG-16, as seen in figure 3.5 and VGG-19. The VGG-19 architecture expands upon the VGG-16 architecture by adding an additional convolutional layer to the $56 \times 56$, $28 \times 28$ and $14 \times 14$ stacks in figure 3.5. Both these architectures significantly improved on the previous state-of-the-art on the ImageNet challenges, but they also suffer from having an enormous amount

of trainable parameters. This puts limitations on the memory and parallelization capabilities of even the newest GPUs. It is therefore common to use the slightly smaller VGG-16 instead of VGG-19 when fine-tuning from a pretrained model, even though VGG-19 actually performs slightly better in terms of accuracy.

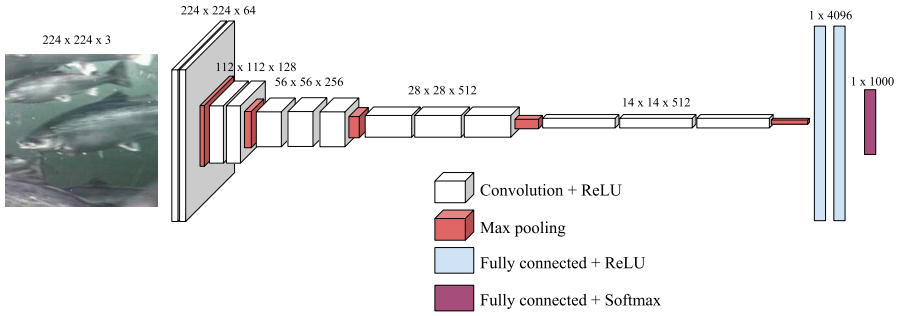



**Figure 3.5:** The VGG-16 architecture. Figure adapted from [5].

### 3.5.3 Using Every Frame

For video classification tasks, it is common practice to use every frame from the videos in the dataset for both training and testing of spatial networks. However, from visual inspection of the Salmon Dataset, we observe that all the videos look very similar, with up to several consecutive frames looking almost identical. We therefore explore the effects of using different sampling frequencies to train our *Spatial Stream* model. A subsampling of the training dataset could result in drastically reduced training times as each epoch would only contain a fraction of the original dataset. Given the limited amount of time for this master's thesis, this could enable us to explore a larger variety of model architectures. We therefore compared three different sampling rates during training:

1. Sampling every frame from the videos.
2. Sampling every other frame from the videos.
3. Sampling every fifth frame from the videos.

We compare models fine-tuned with each of the three sampling rates on the validation set to explore the effects of the sampling rates. The validation results are presented in figure 3.6. From this comparison we find that there is a real advantage to using every frame from the videos. We therefore use every single frame from the training dataset with our approach when training all the following models for the *Spatial Stream*.

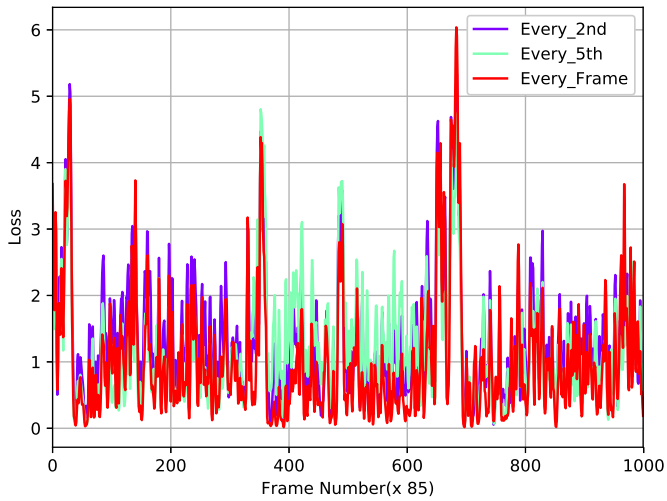



**Figure 3.6:** The figure shows comparison between three models, trained with different sampling rates. The performance is shown using each model's prediction loss for the validation set. It is clear that using every frame produces lower loss through most of the validation set, thus using every frame gives better performance.

### 3.5.4 Image Preprocessing

As we described in section 2.1.3.4, data collected from the real world might not be directly usable for machine learning. Thus a preprocessing step might be required. For the Salmon Activity Recognition Domain we outline two main factors of noise in the dataset:

1. **The lighting conditions.** Lighting conditions in different videos may vary significantly due to both time of day and weather conditions. This produces different color tints in the water, as well as different amounts of light in the frame, depending on the conditions during video capture. Thus a need for normalizing the data across these factors is identified.

2. **Specular reflections from salmon in the videos.** Since the salmon skin is very reflective, there are some conditions which produce very bright specular reflections in the videos. Since CNNs use matrix dot products as their main operation, a very bright area of an image could produce abnormal activation maps in the convolutional layers. Thus we identify the need to eliminate these specular reflections.

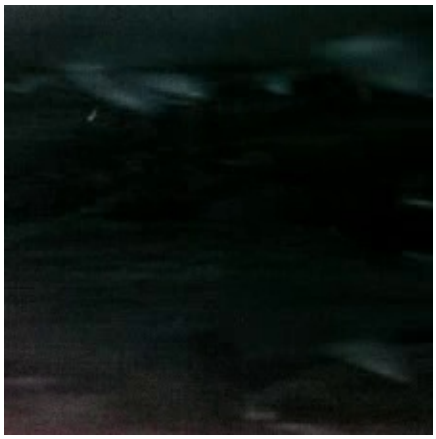
To handle the different lighting conditions in our dataset, we decide to gray scale all our videos, creating videos consisting of 3 identical grayscaled channels instead of the regular RGB channels. This is done to avoid our network overfitting to specific

color tints in the training set due to differences in lighting conditions. To further normalize and prepare our data we experiment with two different preprocessing strategies, explained in section 3.5.4.1 and section 3.5.4.2.

#### 3.5.4.1 Our Own Preprocessing Strategy

With the videos being gray scaled, we are able to deal with the color tint introduced by different lighting conditions. However, we do not address the different amounts of light as a result of the time of capture. If a video is captured late in the day, it may be significantly darker than a video captured near noon as a result of under exposure. This effect is seen in figure 3.7. We therefore equalize the histograms for every frame. This improves the contrast in underexposed images and thus helps normalizing our data over the different exposures we get from the different conditions during video capture.

**(a)** A video frame captured at 11:58 in late November.



**(b)** A video frame captured at 15:42 in late November.

**Figure 3.7:** Comparison between two frames captured at the same day, but at different times of the day. The changes in light conditions are clearly visible.

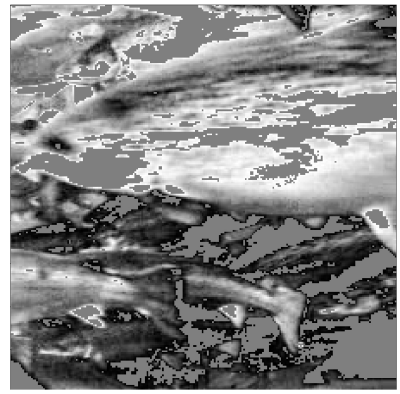
The next step in our preprocessing method is to zero-center and normalize the dataset. We calculated the mean and standard deviation of the entire training dataset and subtract this mean from every image to zero-center the data. We also divide every image by the standard deviation to create a unit variance dataset. This is common practice in many Deep Learning applications as it helps the models to learn from the content of the images rather than the lighting conditions or exposure values.

Our final preprocessing step is to set very bright and very dark pixels to zero. This is done to handle the specular reflections from the salmon skin in direct light. It also zeroes out very dark areas of the videos as these are not important for our task. Since zero-valued pixels do not affect the matrix dot products, this results in more appropriate activation maps in our model. We call our preprocessing strategy

the *Specular Removal Preprocessing Strategy*. A comparison between an input image and a preprocessed image can be seen in figure 3.8.

**Possible Limitations to our Preprocessing Strategy**
Since the pretrained model of VGG-16 was trained using a different preprocessing strategy than our *Specular Removal Preprocessing Strategy*, it is worth noting that our fine-tuned model could suffer from our preprocessing steps. The reason for this is that it places our data in another region, in relation to the ImageNet data, than it would be located had we used the original VGG-16 preprocessing strategy. This also means that the pretrained model is also trained on data with a different distribution than our own dataset. This could make it more difficult for the pretrained model to produce accurate predictions using this preprocessing strategy, thus resulting in the need for more fine-tuning or in worse results overall. In section 3.5.4.3 we therefore compare our *Specular Removal Preprocessing Strategy* to the original VGG-16 preprocessing strategy.

**(a)** The original input video frame



**(b)** The preprocessed video frame

**Figure 3.8:** A comparison showing an input image and the resulting *Specular Removal* image, processed using our *Specular Removal Preprocessing Strategy*

### 3.5.4.2 VGG-16 Preprocessing

When training the VGG-16 architecture for ILSVRC, Simonyan and Zisserman did not do much preprocessing of the ILSVRC dataset. In fact, the only preprocessing step they used was to subtract the pixel-wise mean image value of the entire training dataset, effectively zero-centering the dataset. We fine-tune a VGG-16 network from a pretrained model, subtracting the ILSVRC-mean from the Salmon Dataset. This does not zero-center our data, but rather makes our data fit better with the ILSVRC data. Since the pretrained model is trained on the ILSVRC dataset, this also makes our data fit better with what the pretrained model is used to seeing, thus it might converge on a better set of parameters.

**Table 3.2:** The final average validation accuracies for the three preprocessing strategies.

| **Preprocessing Strategy** | **Avg. Validation Accuracy** |
|---|---|
| VGG-16 subtracting ILSVRC-mean | 84.4% |
| VGG-16 using zero-centered + unit variance data | 80.5% |
| VGG-16 using *Specular Removal Preprocessing Strategy* | **86.7%** |

#### 3.5.4.3 Comparing the Preprocessing Strategies

To decide which preprocessing strategy to use, we compare three alternatives:

1. VGG-16 subtracting ILSVRC-mean
2. VGG-16 using zero-centered + unit variance data
3. VGG-16 using our *Specular Removal Preprocessing Strategy*

We fine-tune three models, each using only one of the three preprocessing strategies and compared them using the validation set. We include the zero-centered + unit variance data, preprocessing to see the added effects of histogram normalization and removal of specular reflections in our *Specular Removal Preprocessing Strategy*. The comparison results are shown in figure 3.9, as well as the final average validation accuracies in table 3.2. From these results, it becomes clear that the *Specular Removal Preprocessing Strategy* significantly outperforms the other strategies and improves upon the standard VGG-16 preprocessing strategy by 2.21%. We can also see from figure 3.9 that the *Specular Removal Preprocessing Strategy* on average, produces a slightly flatter curve, indicating that the model is less inclined to prefer a particular class over the other. Finally, we note that the model trained using only the zero-centered + unit variance data actually performs worse than the original VGG-16 model with the ILSVRC-mean, indicating that the extra preprocessing steps in the *Specular Removal Preprocessing Strategy* add a real benefit.

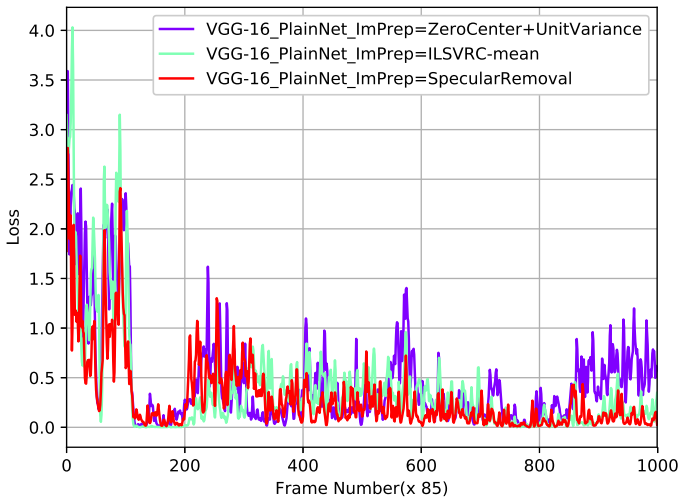



**Figure 3.9:** The figure shows the validation loss curves for three models, trained using different preprocessing strategies. The *Specular Removal Preprocessing Strategy* model has the lowest loss of all the models, while the *Zero-Center + Unit Variance* has the highest loss. We therefore conclude that the *Specular Removal Preprocessing Strategy* produces the best performing models.

### 3.5.5 Improving the VGG-16 Architecture

Since the salmon data is quite different from the ImageNet data, we believe we can enhance the performance of our model by expanding the VGG-16 architecture with *Batch Normalization* layers before the final classification layer. We also remove the two dropout layers between the fully connected layers since Batch Normalization has a regularizing effect on its own. As we described in section 2.1.3.5, Batch Normalization tackles the problem of internal covariate shift, by normalizing each mini-batch. We therefore believe that using a Batch Normalization layers while fine-tuning a model will help to account for the fact that our dataset is quite dissimilar to the ILSVRC dataset, by normalizing the salmon data for each mini-batch. We propose three variations of Batch Normalized VGG-16 networks:

1. A single Batch Normalization layer, before the final classification layer.
2. Three Batch Normalization layers. One before every fully connected layer.
3. A fully Batch Normalized network. Here, we use Batch Normalization layers before every layer in the network, except the pooling layers.

We compare these architecture improvements together with the original VGG-16 architecture in figure 3.10. All the architectures were fine-tuned from the pretrained

VGG-16 model, using the *Specular Removal Preprocessing Strategy* and compared on the validation set. From table 3.3 and figure 3.10, we see that using a Batch Normalization layer before every fully connected layer, improves the validation accuracy by 2.4% over the standard VGG-16 architecture and by 1.4% compared to the VGG-16 architecture with a single Batch Normalization layer. We also note that the fully Batch Normalized network is the model struggling the most, indicating that only Batch Normalizing the fully connected layers is beneficial. Thus, we can conclude that using Batch Normalization layers between fully connected layers increases performance by a significant margin.

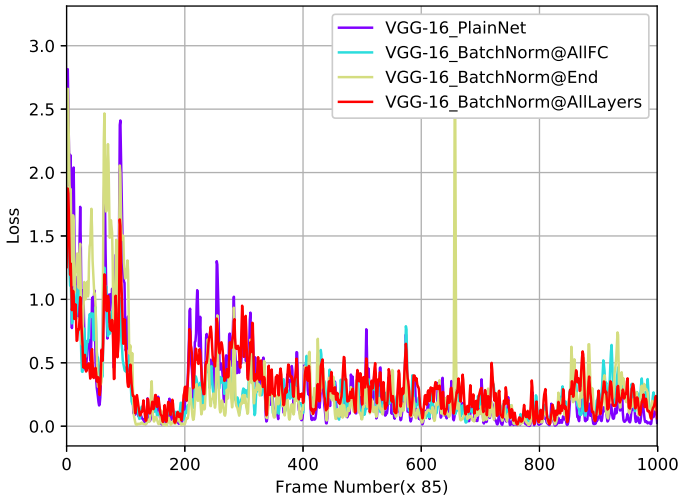



**Figure 3.10:** The figure shows the validation loss for the three proposed VGG-16 architecture improvements as well as the original VGG-16 architecture. It is clear that using Batch Normalization layers through the entire network deteriorates performance. We also see that the BatchNorm@AllFC and BatchNorm@End architectures both produce similar loss and seem to outperform the original VGG-16 architecture. We finally observe that the BatchNorm@AllFC architecture has fewer spikes than the BatchNorm@End, thus leading us to conclude that the BatchNorm@AllFC is the best architecture.

**Table 3.3:** The final average validation accuracies for the four VGG-16 architectures.

| **VGG-16 Architecture** | **Avg. Validation Accuracy** |
|---|---|
| VGG-16 Plain Model | 86.7% |
| VGG-16 Batch Normalized all layers | 82.4% |
| VGG-16 Batch Normalized all FC | **89.1%** |
| VGG-16 Batch Normalized final FC | 87.7% |

### 3.5.6 Data Augmentation

Data Augmentation is a commonly used strategy to increase the size of datasets by augmenting the original dataset to create additional data points. It is also used to produce more robust models in Machine Learning by introducing noise to the training data. For our purposes, we believe that our dataset is sufficiently large, but that we could see benefits in terms of a more robust model if we introduce noisy data through data augmentation. We propose five different data augmentation policies in order to enhance our model's performance:

1. Blur
2. Flip
3. Rotate
4. Flip+Blur
5. Flip+Rotate

As we mentioned in section 3.2.2, our data only comes from one breeding cage and is captured using a single camera, thus this camera only sees the salmon swimming from the same angle. As we saw from section 3.4.2, the Hue values are noteably different for Feeding and NonFeeding data. Since the Hue channel represents direction, it is clear that Feeding salmon have more erratic swimming patterns than NonFeeding salmon. We therefore propose to flip our training data in order to avoid our model overfitting to the direction the salmon are swimming. We also propose to rotate our training data by up to 5 degrees to account for the effects waves and underwater currents might have on the camera, making our model more robust to extreme weather conditions. The same reasoning is also used for the addition of blurred data, since camera movements produced by weather could result in blurry images. Finally we also explore the effects of combining the augmentation strategies to further improve model robustness. Figure 3.11 and table 3.4 show the results of using the different data augmentation strategies. All models were fine-tuned from the same VGG-16 checkpoint and using the *Specular Removal Preprocessing Strategy*, but with their respective data augmentation policies. From the results we see that data augmentation actually seems to deteriorate our architecture's performance, but that blurring the images does the least harm. We believe that this is because our un-augmented dataset contains enough variation to produce a sufficiently robust model. We also note that our validation data is captured from the same breeding cage and in similar conditions as the training data. This means that all the data will look very similar. However, had we used data from several cages, we might have seen a better effect from using the data augmentation policies. We therefore conclude that not using data augmentation produces the best model for our dataset.

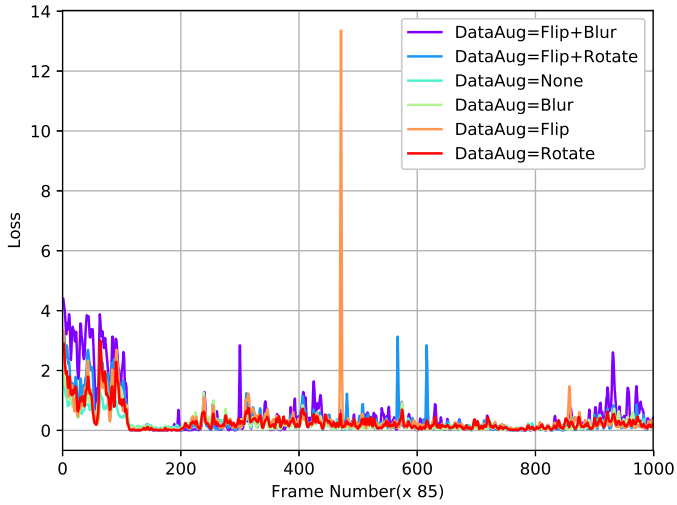



**Figure 3.11:** The figure shows the validation loss for six different models trained with six different data augmentation policies. It is clear that the *Flip* and the *Flip+Blur* policies produces the highest loss. It is also very hard to separate the *Blur* and *None* policies, as they both seem to outperform the other policies with similar margins.

**Table 3.4:** The average validation accuracies for the data augmentation policies.

| **Data Augmentation Strategy** | **Avg. Validation Accuracy** |
|---|---|
| VGG-16 DataAug=Blur | 88.5% |
| VGG-16 DataAug=Flip | 87.4% |
| VGG-16 DataAug=Rotate | 87.2% |
| VGG-16 DataAug=Flip+Blur | 85.3% |
| VGG-16 DataAug=Flip+Rotate | 87.6% |
| VGG-16 DataAug=None | **89.1%** |

### 3.5.7 The Final Spatial Stream

Using the results presented above, we can produce the final architecture to be used in our *Spatial Stream* and its training strategy. As we saw earlier, using every frame in the training dataset resulted in increased performance. We also found that using our *Specular Removal Preprocessing Strategy* significantly improved performance of the architecture. Using Batch Normalization layers before every fully connected layer also resulted in a notable performance increase, while Data Augmentation had a degrading effect. From these observations we conclude that our *Spatial Stream* will consist of an improved VGG-16 Architecture, using Batch Normalization layers before every fully connected layer. The architecture is also trained using:

1. Action Snippets of length one.
2. Every frame from the training data.
3. Our *Specular Removal Preprocessing Strategy.*
4. No Data Augmentation

Our final *Spatial Stream* is presented in figure 3.12.

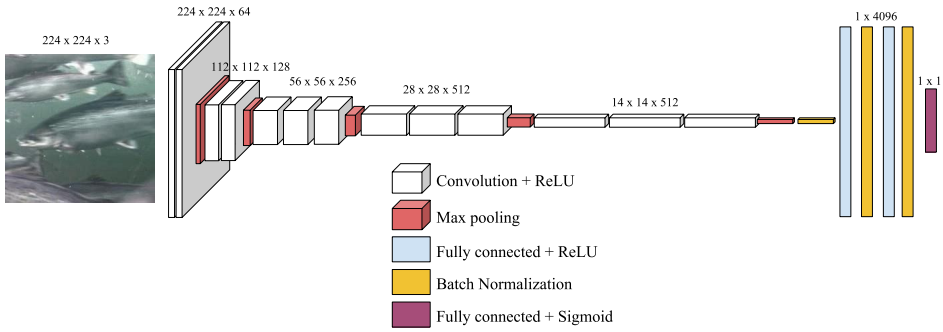



**Figure 3.12:** The final improved VGG-16 architecture used in the *Spatial Stream.* We use Batch Normalization layers before every fully connected layer to improve model performance.

## 3.6 Temporal Stream

This section will present the work done while developing the *Temporal Stream* in the *Dual-Stream* architecture. It will present and compare several drastically different network architectures. Finally it will present the final model used as the *Temporal Stream*.

### 3.6.1 Capturing Temporal Information

As we mentioned in section 1.1.1, we believe that more temporal information will result in better classification accuracy. Thus we propose using a 3D-CNN as our *Temporal Stream*. As described by Tran et al. [48], 3D-CNNs are better suited for learning *spatiotemporal* features compared to 2D-CNNs, therefore making them an obvious choice for our *Temporal Stream*. We also saw from [50] that combining 3D-CNNs with Optical Flow further enhances the ability of these networks to learn temporal information. We therefore use this as inspiration and use Optical Flow as input to our *Temporal Stream*.

**Optical Flow Input**
Since 3D-CNNs are able to use the convolution operation in three dimensions, we are able to learn correlation between pixels in a single frame as well as how they transform through several consecutive frames from a video. We therefore expand our input for the *Temporal Stream* to an image cube of several stacked consecutive video frames as seen in figure 3.13. This means that the Action Snippet length for the *Temporal Stream* is the same as the number of stacked consecutive video frames used in the image cube.

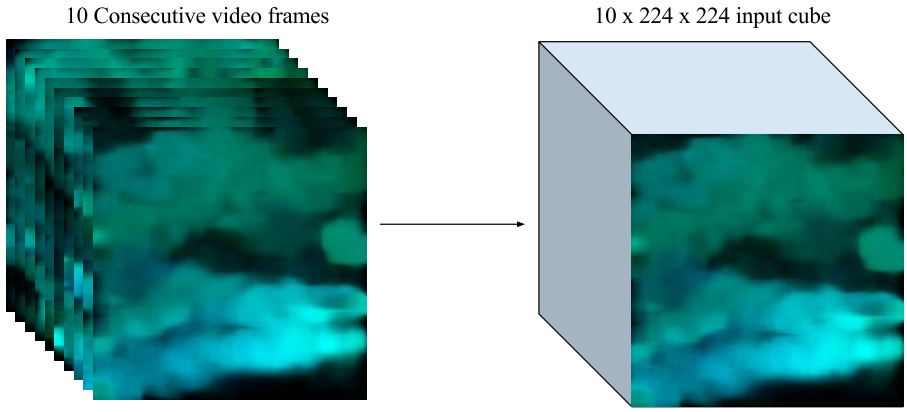



**Figure 3.13:** The transformation of 10 consecutive images into an image cube used as input for the *Temporal Stream*.

**Preprocessing**
For our Optical Flow data, the only preprocessing steps we apply are zero-centering and unit variancing the data. Optical Flow is generated from the movement of pixels through time and not the content of the video frames. it therefore does not suffer from problems such as specular reflection or poor lighting conditions. It therefore does not need to be processed further.

### 3.6.2 Developing a 3D-Convolutional Network Architecture

Since research on the use of 3D-CNNs for video classification is very limited, there are no available pretrained models for us to fine-tune. We are therefore forced to develop our own architecture and train it from scratch. When developing new architectures, it is often a good idea to look to what other researchers have done. In general, it seems that most 3D-CNNs used in research are not as deep as their 2D counterparts[25, 33, 48]. This is because 3D-CNNs often are used for different types of tasks than 2D-Convolutional Networks. The increase in dimensionality from 2D to 3D also brings with it an increase in the amount of trainable parameters and therefore also an increase in the demands on GPU memory. It also leads to increased training times, thus forcing researchers to stick to shallower architectures. However, since an increase in depth proved beneficial for 2D-CNNs on ILSVRC as seen in [20, 41], we also want to explore the effects of depth in our 3D-Convolutional architectures. We therefore propose three different 3D-CNN architectures:

1. Taking inspiration from Ji et al. [25], we propose a Multi-Stream 3D-Convolutional Network.

2. Taking inspiration from Tran et al. [48], we propose a Twin-Stream 3D-Convolutional Network.

3. Taking inspiration from He et al. [20], we extend the Residual Neural Network architectures to use 3D-Convolutional layers, producing a 3D-Convolutional ResNet.

**Using Every Other Frame for Faster Architecture Exploration**
To explore and compare multiple architectures for our *Temporal Stream* also requires us to train these architectures. In order to speed up training times, while still producing models that are comparable to each other, we decided to use a Optical Flow dataset sampled at every other frame. This halves the amount of data used for training and speeds up training times. In figure 3.4, we show that the distribution of Hue values are virtually identical for Optical Flow data sampled at every frame and every other frame. This indicates that we should be able to train models producing representative validation results using Optical Flow data, sampled at every other frame. When we have found our final 3D-Convolutional architecture, we will train that model, using the Optical Flow dataset sampled at every frame.

#### 3.6.2.1 Multi-Stream

For our *Multi-Stream* network architecutre, we take inspiration from Ji et al.[25] and their 3D-CNN for Human Action Recognition. In their work, Ji et al. take 7 consecutive video frames as input and applies a set of hardwired filters to produce multiple channels of information, resulting in 33 feature maps divided over 5 different channels. These channels are: gray, gradient_x, gradient_y, optflow_x and optflow_y. They then apply two different sets of convolutions with a filter size of $7 \times 7 \times 3$($7 \times 7$ in the spatial dimension and 3 in the temporal dimension) to each of the five channels separately. This creates two sets feature maps for each channel. However the difference in the two convolution sets is not specified by Ji et al. They then subsample their feature maps through a $2 \times 2$ subsampling to reduce the spatial resolution. Their next step is to further increase the amount of feature maps, by applying three different sets of convolutions with filter size $7 \times 6 \times 3$, to each of the channels in each of previous feature maps. They again fail to mention the specifics of the differences in their convolution kernels. This results in 6 different sets of feature maps for each channel. They again perform a subsampling, this time using a size of $3 \times 3$. They finally apply a convolution with a filter size of $1 \times 1$ to all the sets of feature maps, before they apply their classifier through a fully connected layer.

Since we only use Optical Flow as input to our architecture we do not apply any hardwired filters to produce additional channels of information. However we are inspired by the approach used by Ji et al., to apply different convolutions to the input to produce more information. In our approach we aim to create multiple representations of the temporal information in order to better represent the motion of salmon in our architecture. We therefore apply two different sets of convolutions to our input to produce two streams. Both convolutions use spatial filter of size $7 \times 7$, but we use different filter sizes for the temporal dimension in the two streams. The first stream uses a filter of size 3, while the other uses a filter of size 5. This creates two temporal representations of our input data. We then use a maxPooling layers with filter size $2 \times 2 \times 2$ to reduce the size of our feature maps. We next apply three different convolutions with filter sizes: $7 \times 7 \times 3$, $5 \times 5 \times 3$ and $3 \times 3 \times 3$ to each of the two streams. This creates different spatial interpretations of the two streams and results in 6 streams with different feature maps. These feature maps are then downsampled through maxPooling layers, using a filter size of $2 \times 2 \times 2$, to further reduce the size of each feature map. We then apply a fully connected layer with 256 units, before we apply our classification layer. Our *Multi-Stream* architecture is shown in figure 3.14.

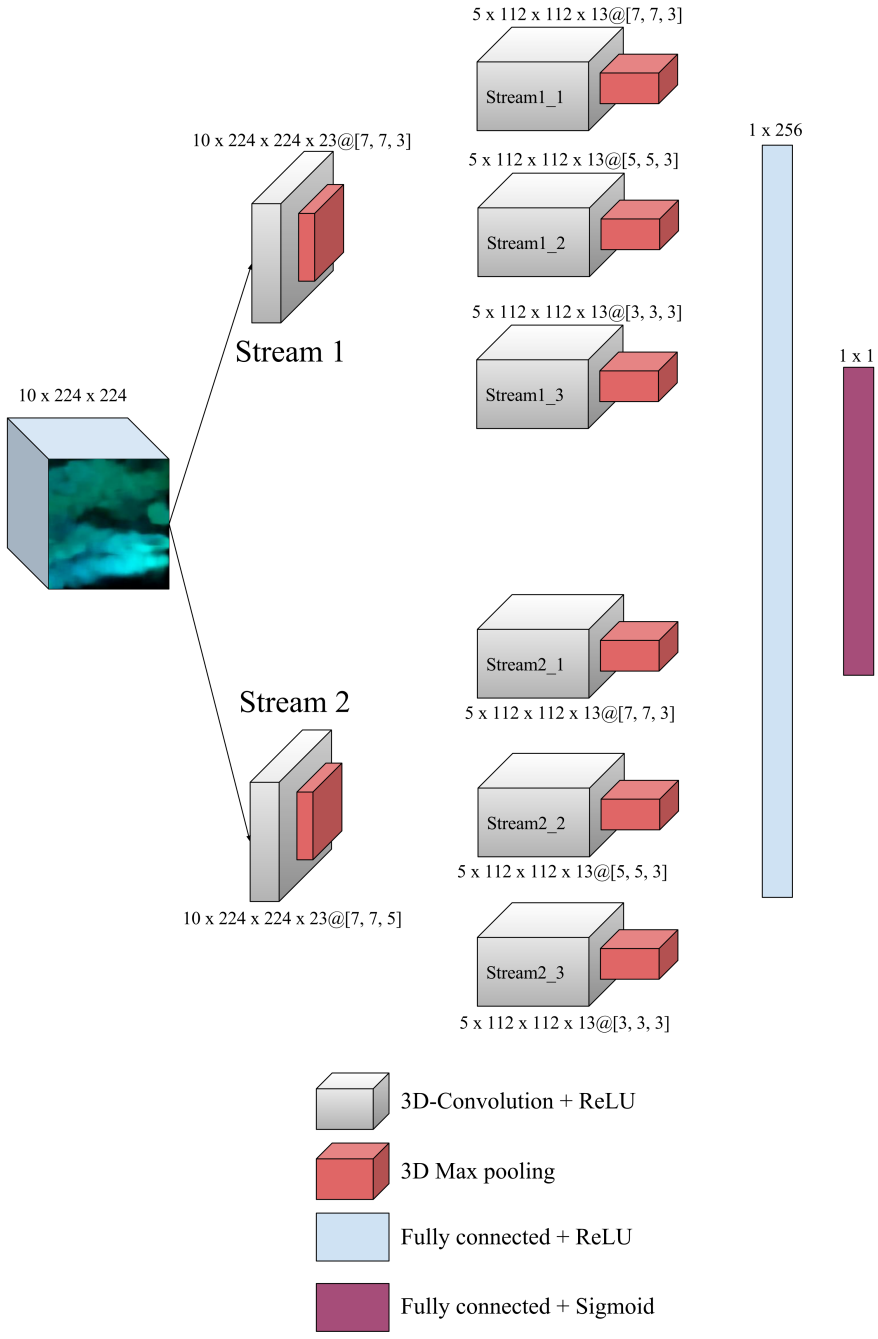



**Figure 3.14:** The Multi-Stream Architecture. The convolutions and poolings are shown in 3D, instead of 4D, to enhance visualization.

#### 3.6.2.2 Twin-Stream

For the *Twin-Stream* network architecture, we look to the findings of Tran et al. [48]. They experimented with 3D-CNN and the effect of different filter sizes in the temporal dimension. They compared six different networks, all with the same architecture, but using a different strategy for the temporal filter depth. The network architecture consisted of five convolutional layers, each immediately followed by a pooling layer with filter size $2 \times 2 \times 2$. The exception to this was the first layer, which used a pooling of size $2 \times 2 \times 1$ to avoid downsampling the temporal dimension too early. They topped the architecture with two fully connected layers with 2048 units and a final softmax classifier. The number of filters for the convolutional layers were 64, 128, 256, 256 and 256 from the first to last layer. They compared architectures using filters with temporal depths of either 1, 3, 5 or 7, as well as *increasing* and *decreasing* depth. The increasing filter strategy used temporal depths of 3-3-5-5-7 and the decreasing used 7-5-5-3-3. In their work, Tran et al. found that a filter size of $3 \times 3 \times 3$ seemed to be the best performing filter, shortly followed by the increasing strategy.

Inspired by these findings, we propose a *Twin-Stream* architecture, consisting of the two best performing architectures found by Tran et al. Our *Twin-Stream* architecture uses two streams, with a temporal depth of 3 for Stream 1 and the *increasing* temporal depth proposed by Tran et al. for Stream 2. However, due to hardware limitations, we decided to decrease the amount of filters in each of the convolutional layers, resulting in 32, 64, 128, 128 and 128 filters from the first to last layer. We then apply two fully connected layers with 2048 units, before we apply our final classification layer. We also use a Batch Normalization layer before the first convolutional layers in both streams, as well as Dropout between the fully connected layers during training. Using this approach, we aim to combine and improve the best features from the two best performing architectures proposed by Tran et al. Our final *Twin-Stream* architecture is shown in figure 3.15

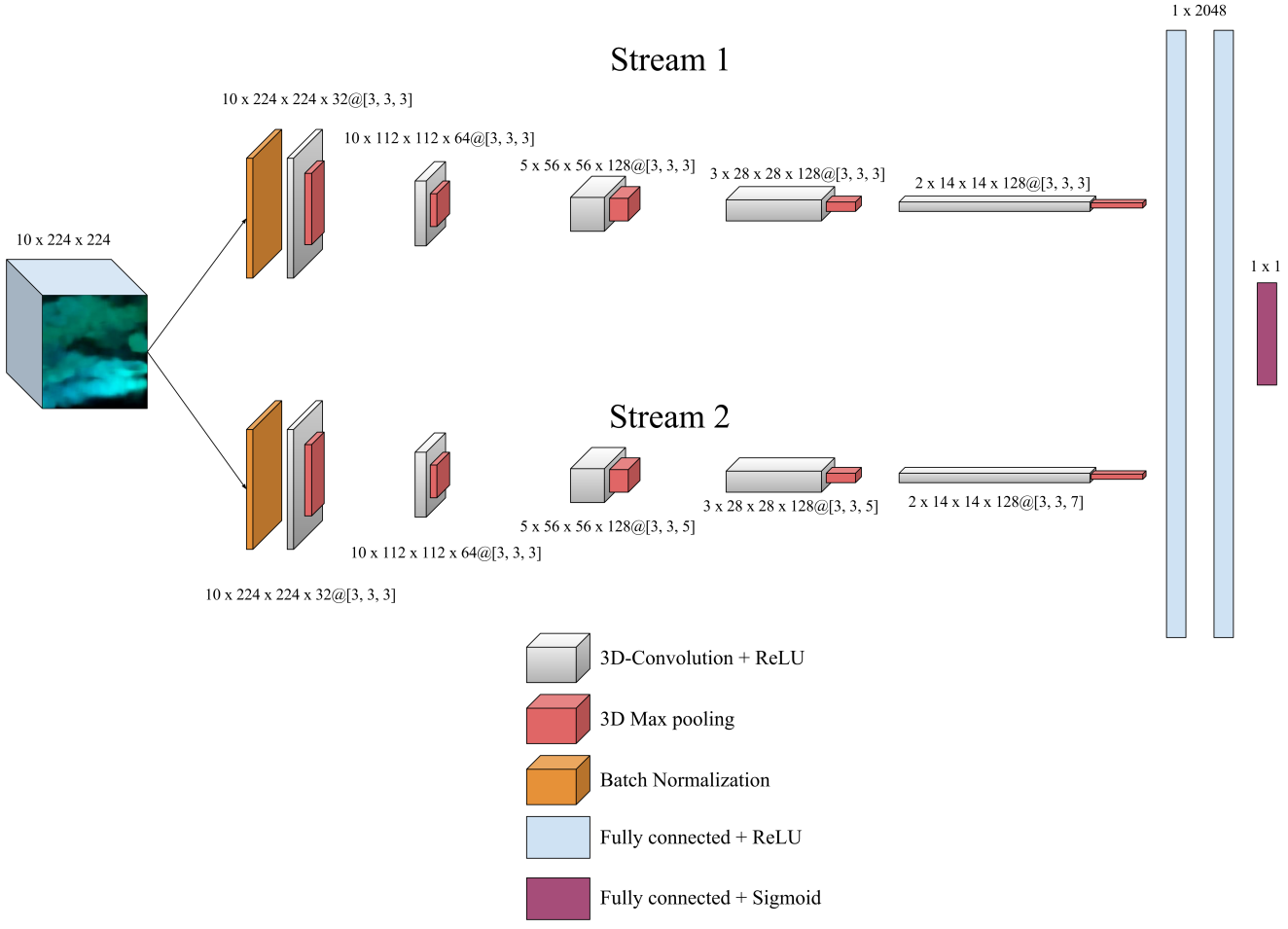



**Figure 3.15:** The Twin-Stream Architecture. The convolutions and poolings are shown in 3D, instead of 4D, to enhance visualization.

#### 3.6.2.3 Deep Network

As has been shown through the work of Simonyan and Zisserman[41], deeper network architectures improves classification accuracies for 2D-CNN. This evidence is further strengthened by the work of He et al. and their Residual Neural Networks[20, 21]. In their work He et al. explored the problem of degrading training accuracies in deeper networks. This problem occurs when more layers are added to suitable deep models and manifests itself in increased training error as models become deeper. They argued that this is not due to the models overfitting, but rather because deeper models are harder to optimize. To overcome the degradation problem, He et al. proposed a *deep residual learning* framework in which they explicitly let stacked layers fit a residual mapping, instead of hoping that they directly fit a desired underlying mapping. Formally they denoted the desired underlying mapping as $H(x)$ and let the stacked layers fit another mapping of $F(x) := H(x) - x$. The original mapping was recast into $F(x) + x$. They hypothesized that it is easier to optimize the residual mapping than the original mapping, thus avoiding the degradation problem. To realize the $F(x) + x$ mapping, they used short cut connections. These connections skip over some convolutional layers, to create what they called residual blocks, shown in figure 3.16. Using these residual blocks, He et al. were able to train several networks with depths of 18, 34, 50, 101 and 152, shown in table 3.5, and showed that deeper residual networks did not suffer from degrading training accuracies.

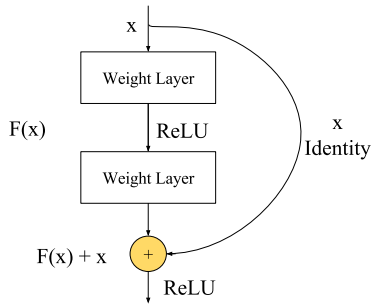



**Figure 3.16:** The shortcut used in Residual Neural Networks. Figure adapted from [20].

All architectures presented by He et al. started with a convolutional layer with filter size $7 \times 7$, followed by a max pooling layer. Then they followed the same principle as the VGG architectures by using filters of size $3 \times 3$ and doubling the number of filters each time they performed a downsamping, to preserve time complexity in each layer. The downsamplings were performed directly by applying convolutional layers using filters of size $1 \times 1$ and a stride of 2. They initially made use of Batch Normalization layers after every convolutional layer and before every activation. However, in [21], He et al. proposed an improved residual block, where the Batch Normalization were applied along with a ReLU activation before the convolutional layers. This improved configuration was called full pre-activation. He et al. continued to show that this configuration further eased optimization.

Finally, to end their network they used an average pooling layer followed by a fully connected softmax layer. For the networks of depths 50, 101 and 152 they expanded their residual block to what they call a residual bottleneck. The bottleneck used 3 convolutional layers instead of 2, where they sandwiched a layer with filter size $3 \times 3$ between layers with filter size $1 \times 1$ to reduce and increase dimensions respectively. This was done to decrease training time of the deeper models. He et al. concluded that their Residual Neural Networks avoided the problem of degrading training accuracies and were able to outperform the state-of-the-art networks on ILSVRC by a significant margin.

Encouraged by these findings, we propose a 3D-Residual Neural Network by expanding the original architectures, developed by He et al., from 2D-CNNs to their 3D counterparts. To do this, we increase the dimensionality of every convolutional layer from 2D to 3D. We use the same strategy as He et al., with the full pre-activation configuration [21], when creating our networks. We use 3D-residual blocks, shown in figure 3.17a and 3D-residual bottlenecks, shown in figure 3.17b to build networks of varying depths. To the best of our knowledge, this is a *novel network architecture*, that has not previously been explored. Thus our results will be the state-of-the-art results using these architectures.

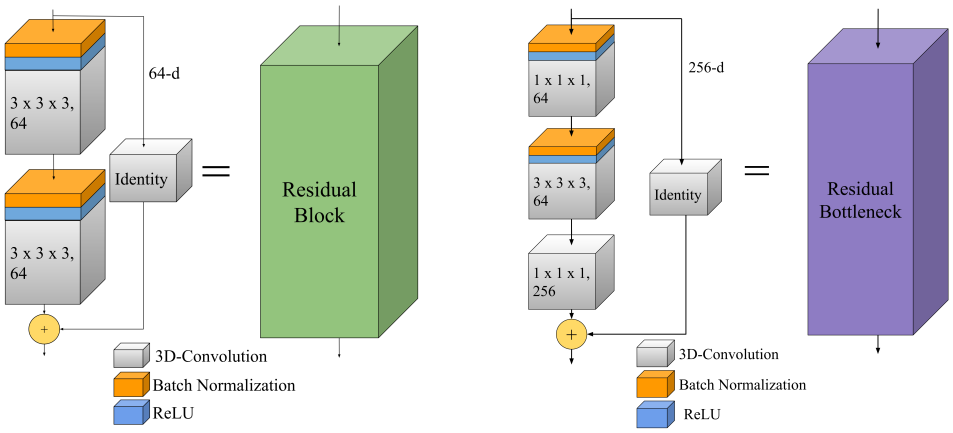
We explore four different network depths:

1. 18-layer 3D-residual block.
2. 34-layer 3D-residual block.
3. 50-layer 3D-residual bottleneck.
4. 101-layer 3D-residual bottleneck.

The four architectures were then compared, using the validation data as shown in figure 3.18 and table 3.6. From this comparison, we observe that both the 34-layer and the 50-layer networks outperform the shallower 18-layer network as is expected considering the results presented by He et al. in [20]. However, we also observe that the 34-layer network outperforms the 50-layer network, which is contrary to what He et al. find using their 2D-Networks. We also observe that the 101-layer network does not seem to converge at all. We hypothesize that this is due to the degradation problem described earlier. Deeper networks are harder to optimize and we suspect that the highway approach, used by He et al. is still not enough when it comes to very deep networks using 3D-Convolutional building blocks.

**Table 3.5:** Architectures for 2D-ResNets. Building blocks are shown with (input, output) number of filters and the numbers of blocks stacked in each stack. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2. Table adapted from [20].

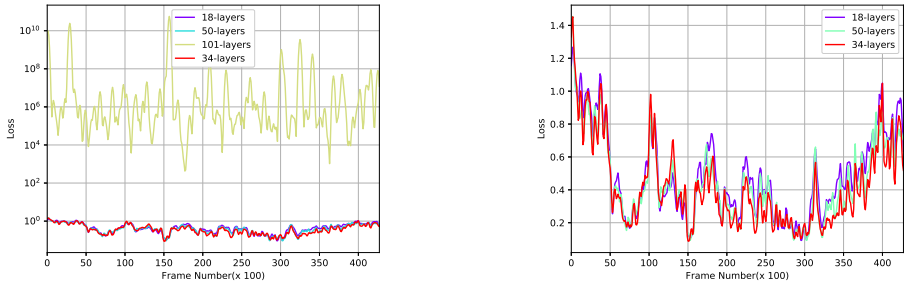
| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112 x 112 | 7 x 7, 64 stride 2 | | | | |
| conv2_x | 56 x 56 | 3 x 3 max pool, stride 2 | | | | |
| | | resBlock(64, 64) x 2 | resBlock(64, 64) x 3 | resBottleneck(64, 256) x 3 | resBottleneck(64, 256) x 3 | resBottleneck(64, 256) x 3 |
| conv3_x | 28 x 28 | resBlock(64, 128) x 2 | resBlock(64, 128) x 4 | resBottleneck(128, 512) x 3 | resBottleneck(128, 512) x 4 | resBottleneck(128, 512) x 8 |
| conv4_x | 14 x 14 | resBlock(128, 256) x 2 | resBlock(128, 256) x 6 | resBottleneck(256, 1024) x 3 | resBottleneck(256, 1024) x 23 | resBottleneck(256, 1024) x 36 |
| conv5_x | 7 x 7 | resBlock(256, 512) x 2 | resBlock(256, 512) x 3 | resBottleneck(512, 2048) x 3 | resBottleneck(512, 2048) x 3 | resBottleneck(512, 2048) x 3 |
| | 1 x 1 | average pool, 1000-d fc, softmax | | | | |




**(a)** 3D-residual block.

**(b)** 3D-residual bottleneck.

**Figure 3.17:** The building blocks of our 3D-Residual Neural Networks. They are the 3D-Convolutional equivalents to the original building blocks, presented in [20] with the modifications from [21]. This enables them to handle *Spatio Temporal* input, such as our input cubes, since they can perform the convolution operation over both space and time.




**(a)** The 101-layer network does not seem to converge and therefore produces very poor loss curves.

**(b)** The validation loss curves for the 18-, 34- and 50-layer 3D-Residual Network architectures.

**Figure 3.18:** The figure shows the validation loss curves for four different 3D-Residual Network architectures, using four different depths. The 101-layer network does not seem to converge and therefore produces very poor loss curves. The 34-layer architecture seems to be the optimal depth and produces the lowest loss.

**Long 3D-Residual Networks**

To explore whether the lack of performance in our deeper architectures is due to the degradation problem, we propose to expand our 3D-residual networks to an architecture we call *Long 3D-Residual Networks*. This architecture does not only use highways internally, in each residual block or bottleneck, but also adds a highway through each stack in the network, as shown in figure 3.19. This was done to see if it could further ease optimization of deep networks. We traine four Long Residual counterparts to our regular 3D-residual networks and compare them directly as shown in figure 3.20 and table 3.6. From the results we see that the addition of the long highway to the residual networks does not seem to significantly improve accuracies for the shallow networks, and in one case, it even deteriorates it. However, for the deeper models, we observe some improvement and for the 101-layer network we see significant performance gains in the long version. This further strengthens our belief that the deeper networks benefit from the longer highways. However, our deeper models still do not outperform the shallower 34-layer model, thus we conclude that the deeper 3D-residual networks are not worth exploring further in this thesis.

**Table 3.6:** The average validation accuracies for both our regular and long 3D-residual networks.

| **Depth** | **Regular** | **Long** |
|---|---|---|
| 18-layer | 77.2% | 77.1% |
| 34-layer | **80.7%** | 78.4% |
| 50-layer | 79.0% | 80.0% |
| 101-layer | 51.0% | 75.3% |

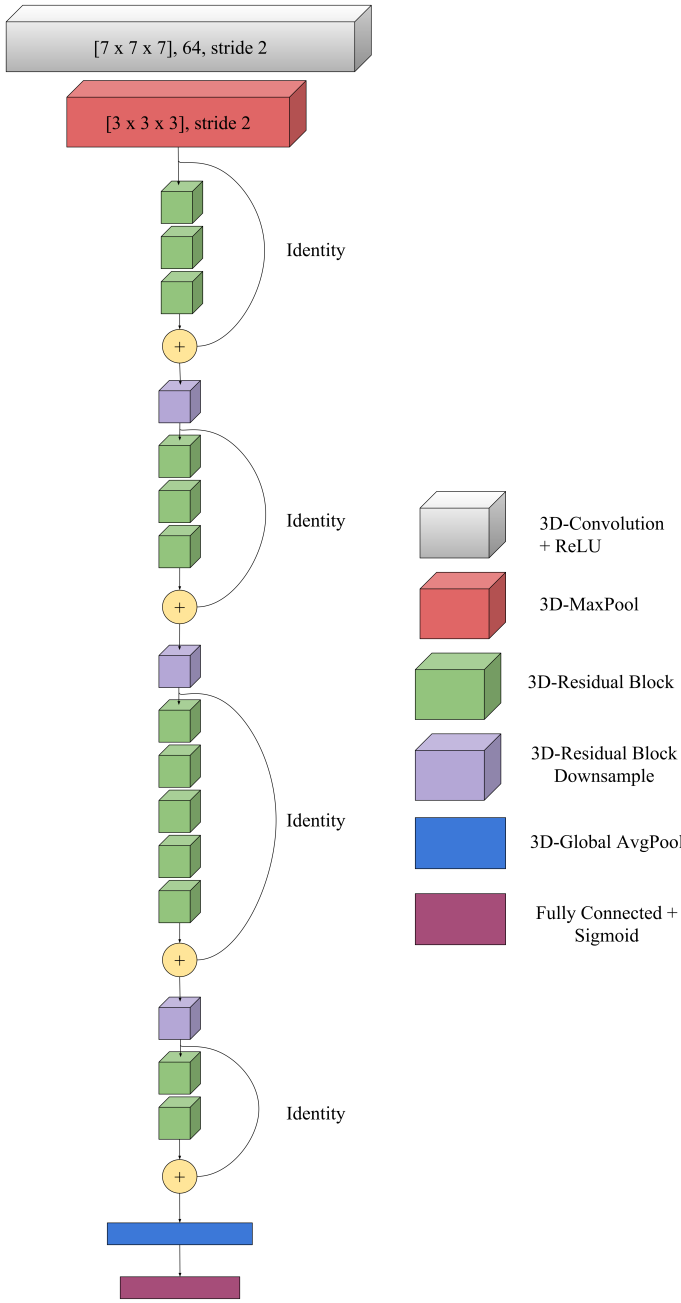
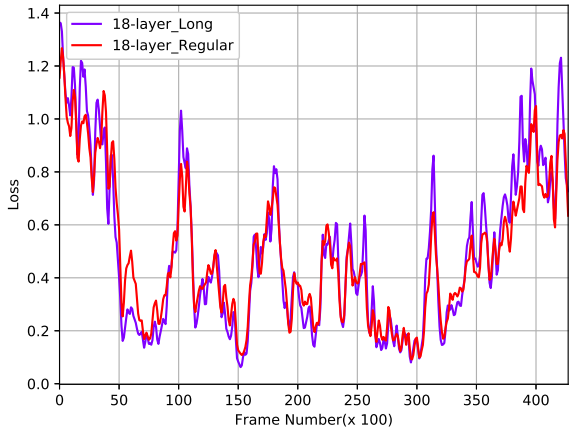
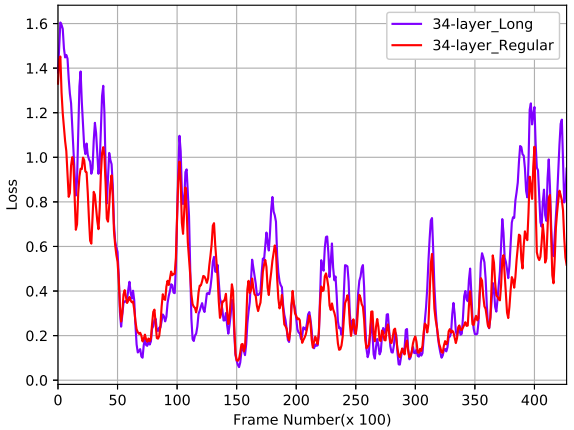


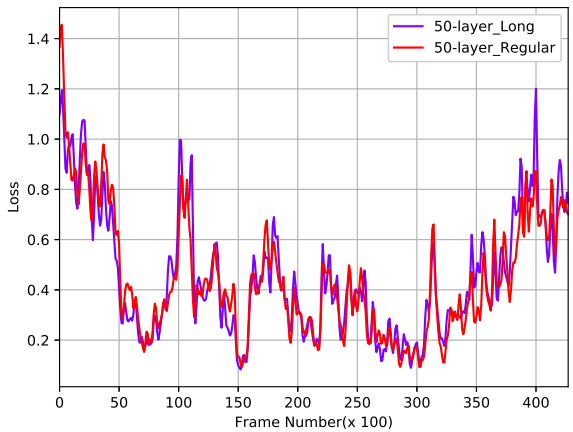

**Figure 3.19:** The *Long 3D-Residual Network* architecture, here shown on the 34-layer version. We add shortcut connections to each stack of building blocks to ease model optimization during training.



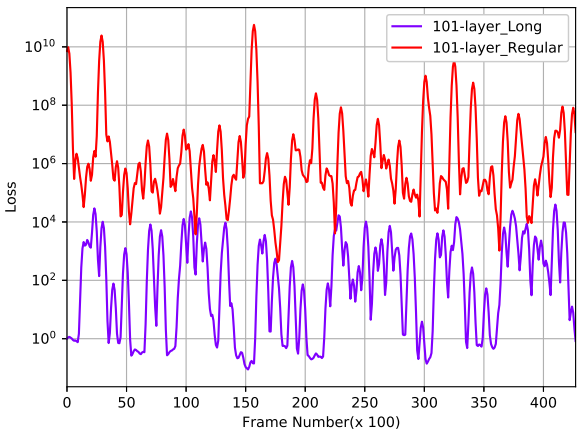

**(a)** 18-layer.




**(b)** 34-layer.




**(c)** 50-layer.




**(d)** 101-layer.

**Figure 3.20:** The figure shows the validation loss for each 3D-Residual Network architecture and their Long Residual counterparts. Only the 101-layer architecture seems to benefit from the Long Residual expansion, thus we do not explore this architecture further.

**Larger Input Sizes**

Since we have only used input cubes of size $10 \times 224 \times 224$ up until now, we explore the effects of doubling the amount frames to create a cube of size $20 \times 224 \times 224$. Since 3D-CNNs convolve both spatially and temporally, we believe that we can further enhance the performance of our 3D-residual network by using more frames for each input, thus increasing the amount of temporal information available. Since the 34-layer version seems to be the most promising architecture, we train a 34-layer 3D-residual network, using an input cube of size $20 \times 224 \times 224$ and compare it to the 34-layer network trained with an input cube of size $10 \times 224 \times 224$. The results are seen in figure 3.21 and in table 3.7. We find that we are able to further improve model performance by using the larger input data. We therefore use $20 \times 224 \times 224$ input cubes as we continue.

**Table 3.7:** The average validation accuracies for the two input cube dimensions.

| Input dimensions | Accuracy |
|---|---|
| $10 \times 224 \times 224$ | 80.7% |
| $20 \times 224 \times 224$ | **81.4%** |

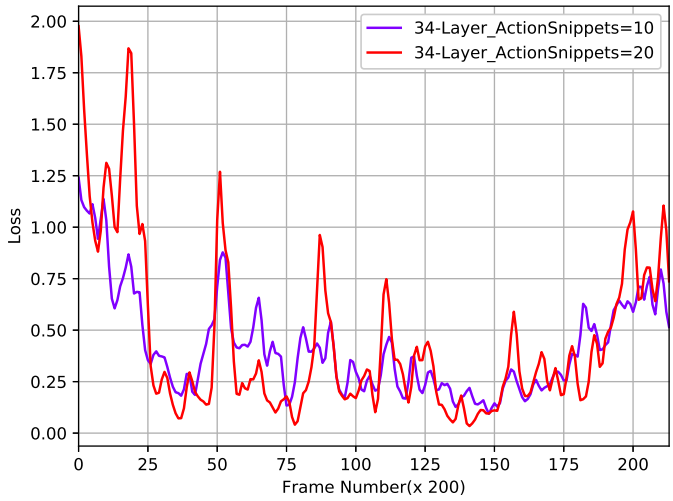



**Figure 3.21:** The figure shows the validation loss for 34-layer 3D-Residual Networks trained with two input dimensions(10Frames = $10 \times 224 \times 224$ and 20Frames = $20 \times 224 \times 224$). We see that using larger input dimensions improves model performance.

**Downsampling Strategies**
We now explore the possibilities of different downsampling strategies for our network. From He et al. we saw that using direct downsampling through the use of $1 \times 1$ convolutional layers with a stride of 2 produced good results. However, we believe that it is possible to improve upon this architecture through the use of pooling layers for downsampling instead. This is because pooling layers consider a larger region of the input before the downsample is performed, thus considering more information when the downsampling is applied. We compare two different pooling layers to the original $1 \times 1 \times 1$ convolution strategy for a total of three different downsampling strategies:

1. Type A: $2 \times 2 \times 2$ max pooling layer with stride 2.
2. Type B: $2 \times 2 \times 2$ average pooling layer with stride 2.
3. Type C: $1 \times 1 \times 1$ convolutional layer with stride 2.

From the results, seen in figure 3.22 and table 3.8, we find that downsampling using pooling layers indeed does produce better accuracies. We also see that Type A seems to be the most promising downsampling strategy, however the difference between Type A and Type B are so small that we consider then negligible. We therefore decide to go with the most commonly used pooling strategy in Deep Learning and choose Type A.

**Table 3.8:** The average validation accuracies for the three downsampling strategies.

| **Downsample strategy** | **Accuracy** |
|---|---|
| Type A | **83.3%** |
| Type B | 83.2% |
| Type C | 81.4% |

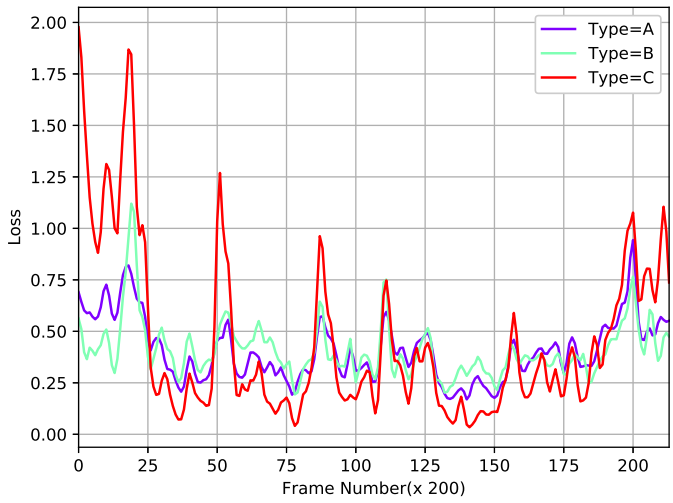



**Figure 3.22:** The figure shows the validation loss for three 34-layer 3D-Residual Networks using three different downsampling strategies. Type C has the lowest loss valleys, but also the highest peaks. We also see that the loss at the start and end of the plot is significantly higher for Type C, thus the average performance is the worst. We also see that Type A slightly outperforms Type B in most of the plot, indicating that Type A is the best downsampling strategy.

### Keep Temporal Information

We now explore the effects of keeping the full temporal depth of the input further through the network. Since we only use 20 consecutive frames as input, we completely collapse the temporal signal at the downsampling layer in conv5_1 for the original 3D-residual network architecture, as seen in table 3.9. This means that the residual blocks following this layer only receive an input with temporal depth 1. Since these blocks still apply 3D-convolutional layers with filter sizes $3 \times 3 \times 3$, we hypothesize that we can further improve model performance by retaining the temporal dimension further through our network. This would give these final residual blocks more temporal information to work with, thus enabling them to develop better features. To keep the temporal information intact we use strides of length 1 in the temporal dimension for both the conv1 and maxPool layers. We call this network architecture the *Keep Temporal Dimension Network*. This results in the final residual blocks receiving a signal with temporal depth 3 as opposed to 1 in the Original network architecture as seen in table 3.9. We compare these two networks on our validation set in figure 3.23 and table 3.10. We find that there is a slight increase in performance in our *Keep Temporal Dimension Network*.

**Table 3.9:** The output sizes of the different layers in our 3D-Residual Networks. Downsampling is performed in layers conv1, maxPool, conv3_1, conv4_1 and conv5_1.

| layer name | output size | |
|---|---|---|
| | Original | *Keep Temporal Dimension* |
| input | 20 x 224 x 224 | 20 x 224 x 224 |
| conv1 | 10 x 112 x 112 | 20 x 112 x 112 |
| maxPool | 5 x 56 x 56 | 20 x 56 x 56 |
| conv2_x | 5 x 56 x 56 | 20 x 56 x 56 |
| conv3_1 | 3 x 28 x 28 | 10 x 28 x 28 |
| conv3_x | 3 x 28 x 28 | 10 x 28 x 28 |
| conv4_1 | 2 x 14 x 14 | 5 x 14 x 14 |
| conv4_x | 2 x 14 x 14 | 5 x 14 x 14 |
| conv5_1 | 1 x 7 x 7 | 3 x 7 x 7 |
| conv5_x | 1 x 7 x 7 | 3 x 7 x 7 |

**Table 3.10:** The average validation accuracies for the Original and the *Keep Temporal Dimension* 3D-residual networks.

| **Architecture** | **Accuracy** |
|---|---|
| Original | 83.3% |
| *Keep Temporal Dimension* | **83.6%** |

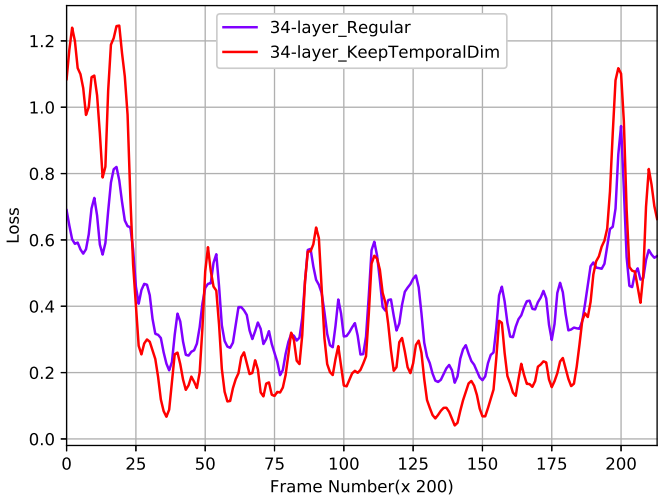



**Figure 3.23:** The figure shows the validation loss for a regular 34-layer 3D-Residual Network and a *Keep Temporal Dimension* 3D-Residual Network. We see that the *Keep Temporal Dimension* network significantly outperforms the regular network through most of the plot, indicating that the *Keep Temporal Dimension* network is the superior architecture.

**Final Residual Network Architecture**

Based on the findings described above, we now present the final 3D-Residual Network architecture. We found that a 34-layer 3D-residual block architecture proved to be the best performing network depth. We then went on to show that classification accuracies could be further improved by increasing the input size from $10 \times 224 \times 224$ to $20 \times 224 \times 224$. We also compared three different downsampling strategies for the downsampling residual blocks and found that a pooling strategy resulted in the best performance. Finally we showed that keeping the temporal dimension size for longer through the network also improved performance. We therefore conclude that the final 3D-Residual Network architecture should be a 34-layer 3D-residual block network, using Action Snippets of length 20 and taking input cubes of size $20 \times 224 \times 224$. It also uses the Type A downsampling strategy for the downsampling residual blocks and the *Keep Temporal Dimension* approach to maintain the temporal dimension depth for longer. The final architecture is shown in figure 3.24

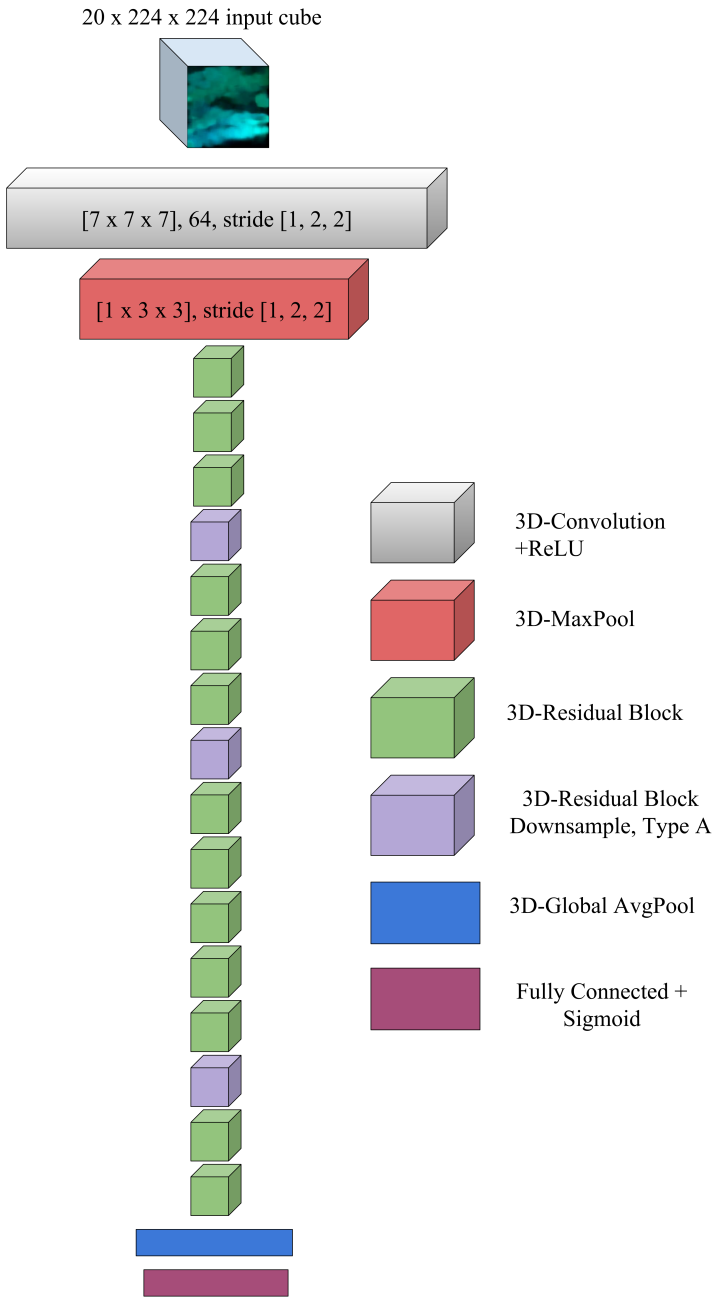



**Figure 3.24:** The final 3D-Residual Network Architecture. We use an input cube, consisting of 20 consecutive stacked Optical Flow frames, the Type A downsampling strategy and the *Keep Temporal Dimension* strategy for retaining more temporal dimension through the network.

#### 3.6.2.4 Final Temporal Stream Architecture

We now compare the three 3D-CNNs we have explored on our validation set. The results are shown in figure 3.25 and in table 3.11. It is clear that the 3D-Residual Network significantly outperforms the two other architectures. This indicates to us that we have found a novel network architecture, which *improves* upon the state-of-the-art architectures for 3D-CNNs on our data. We therefore choose the 3D-Residual Network as our *Temporal Stream*.

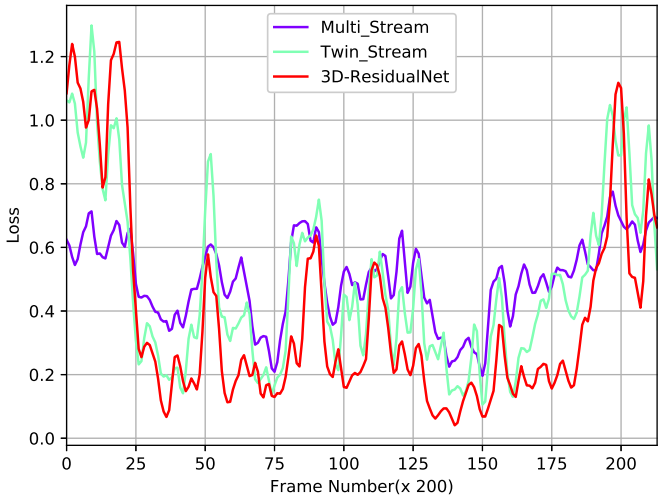



**Figure 3.25:** A comparison of the three 3D-Convolutional Network architectures we have explored. The plot is smoothed to enhance visualization.

**Table 3.11:** The figure shows the validation loss for the three proposed 3D-Convolutional Network architectures. We see that the 3D-Residual Network significantly outperforms the other models.

| **Network Architecture** | **Accuracy** |
|---|---|
| MultiStream | 74.7% |
| TwinStream | 76.5% |
| 3D-Residual Network | **83.6%** |

## 3.7 Recurrent

This section will present the work done while developing the RNN models to test hypothesis **H2** from section 1.2. It presents the development of a *Spatial RNN* as well as the *Recurrent Network* used for our *Dual-Stream* approach.

### 3.7.1 Spatial Recurrent Network

To test hypothesis **H2** from 1.2, we extend the *Spatial Stream* from section 3.5 with a RNN. We use sequences of high-level feature vectors from the *Spatial Stream* as input. These feature vectors come from the final fully connected layer in our *Spatial Stream* as this is the highest level representation for each image in the network, apart from the classification layer. Since the *Spatial Stream* only uses Action Snippets of length 1, the Action Snippet length for the *Spatial RNN* corresponds to the lengthe of the input sequences. We compare three different recurrent models:

1. *Spatial Stream* + Vanilla RNN.
2. *Spatial Stream* + Long Short-Term Memory (LSTM) RNN.
3. *Spatial Stream* + Bidirectional LSTM RNN.

All networks were trained using one layer of 256 of their respective recurrent cells, taking 10 vectors as a sequence input. We also Batch Normalize the inputs and use dropout before the classification layer during training. The RNN architecture is shown in 3.26. We compare the three models on our validation set and the results are presented in figure 3.27 and table 3.12.

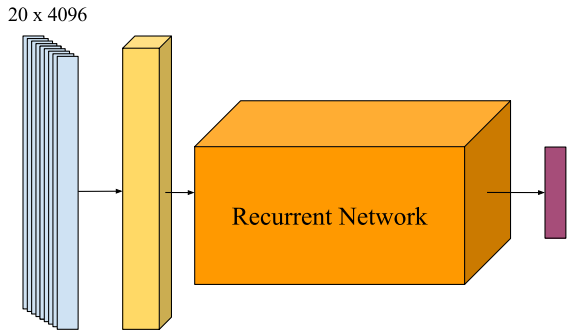



**Figure 3.26:** The Spatial RNN Architecture. We use stacks high-level feature vectors from the final fully connected layer in our *Spatial Stream* to create sequence inputs for the recurernt model. We also use a Batch Normalization layer before the RNN to improve performance.

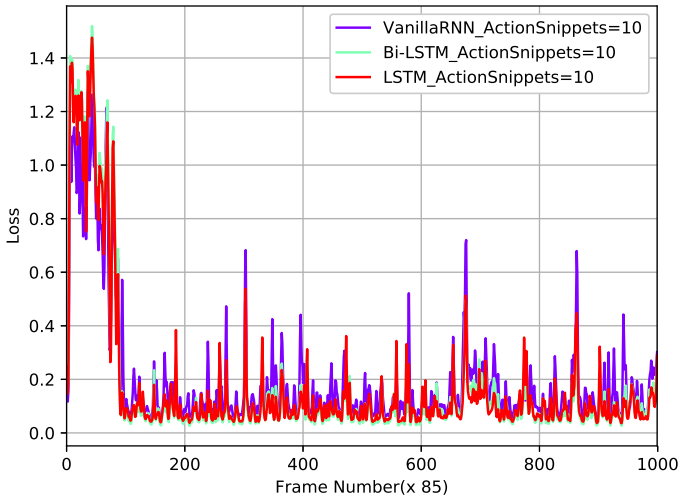



**Figure 3.27:** The validation loss for three Recurrent Architectures using Action Snippets of length 10. We see that both the LSTM and the Bidirectional LSTM outperform the Vanilla RNN. We also observe that the Bidirectional LSTM seems to slightly outperform the LSTM architecture.

**Table 3.12:** The average validation accuracies for the three Spatial RNNs.

| Network Architecture | Sequence = 10 | Sequence = 20 |
|---|---|---|
| Vanilla RNN | 93.2% | N/A |
| LSTM | 94.0% | **95.0%** |
| Bi-LSTM | 94.0% | 94.9% |

From figure 3.27 and table 3.12, we find that the LSTM and the Bidirectional LSTM models outperform the Vanilla RNN with a slight margin. We believe this is due to the Vanilla RNN being prone to suffer from the vanishing and exploding gradients problem described in section 2.1.5. We therefore exclude the Vanilla RNN model from the rest of our explorations.

We now explore the effects of doubling the number of input vectors for the LSTM and Bidirectional LSTM models. Since these models are specifically designed to handle longer sequences of input, we believe that this will increase performances of both models. The results are shown in figure 3.28 and table 3.12.

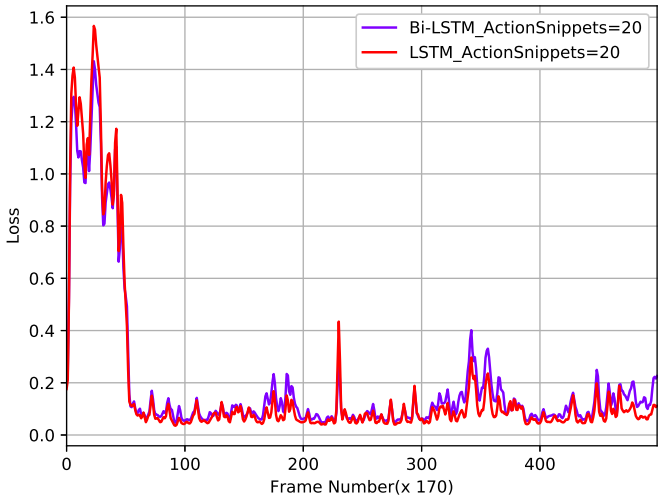



**Figure 3.28:** The validation loss curves for the Bidirectional LSTM and the LSTM using Action Snippets of length 20(Lower is better). The LSTM architecture outperforms the Bidirectional LSTM architecture with a small margin.

From table 3.12 we see that doubling the input size leads to performance gains for both models. In figure 3.28 we further observe that the LSTM network actually slightly outperforms the bidirectional LSTM network, in spite of the Bidirectional LSTMs ability to process examples from both past and future. This is however, by such a small margin that we do not consider it significant. Thus we conclude that both the regular LSTM or the Bidirectional LSTM network architecture, consisting

of 256 cells and using sequence lengths of 20 vectors are well suited models. For our final model we choose to use the LSTM version. This is because we were not able to show a significant improvement by using the more advanced Bidirectional LSTM. The final *Spatial RNN* is shown in figure 3.29

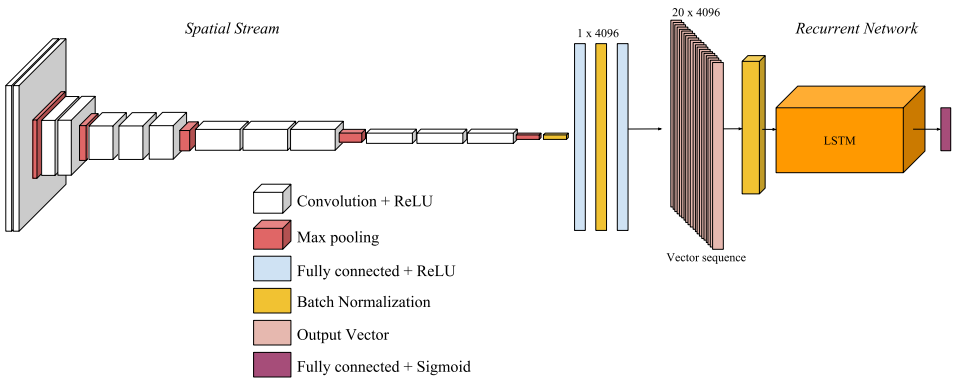



**Figure 3.29:** The final Spatial RNN. It takes stacks of 20 high-level feature vectors from the *Spatial Stream* as sequence input to the *Recurrent Network*, which consists of a Batch Normalized LSTM RNN with 256 cells.

## 3.7.2 Dual-Stream Recurrent

We now move on to our *Dual-Stream* approach and present the *Recurrent Network* developed for the *Dual-Stream*.

### 3.7.2.1 The Input Data

As we stated in section 3.3, we use feature vectors from both the *Spatial Stream* and the *Temporal Stream* as input to the *Dual-Stream RNN*. From the *Spatial Stream*, we use the final fully connected layer as our vector generator, giving us 4096 values. For the *Temporal Stream*, we use the Global Average Pool, seen in figure 3.24, as our vector generator. This gives us 512 values and was selected since it is the highest level representation of the input cube available in the *Temporal Stream*, apart from the classification layer. However, since the *Temporal Stream* takes input cubes containing 20 consecutive frames, it only produces one output for every 20th frame of video. The *Spatial Stream*, on the other hand, produces an output for every frame. Thus we have twenty times more data from the *Spatial Stream* than from the *Temporal Stream*. To handle this discrepancy in data set sizes, we produce element wise average vectors from 20 consecutive feature vectors in the *Spatial Stream* data set. This gives us the same amount of data from both the *Temporal Stream* and the *Spatial Stream* to use for our *Dual-Stream* approach.

**Table 3.13:** The average validation accuracies for the two *Dual-Stream RNN* architectures.

| Network Architecture | Accuracy |
|---|---|
| LSTM | **95.2%** |
| Bi-LSTM | 94.8% |

#### 3.7.2.2 Network Architecture

Using our findings from section 3.7.1, we train RNNs, using 256 recurrent cells and input sequences of length 20. This results in an Action Snippet length of 400 frames for the *Dual-Stream RNN*, since each vector in the input sequence is produced using 20 frames. We also use a Batch Normalization layer to Batch Normalize the input, as well as a dropout layer after the recurrent cells, during training. Since we use vectors from both the *Spatial Stream* and the *Temporal Stream* as input, the corresponding input examples in the sequences are concatenated to produce one input example for the network as shown in figure 3.30. We train two RNNs:

1. *Dual-Stream* + LSTM RNN.
2. *Dual-Stream* + Bidirectional LSTM RNN.

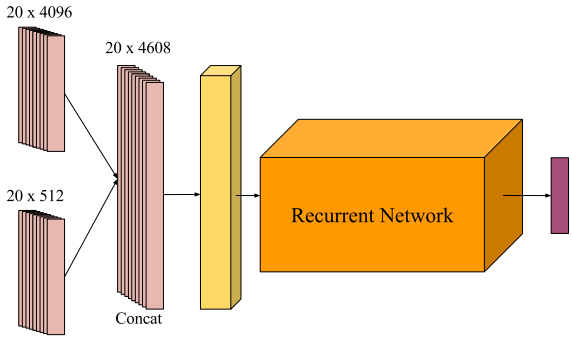



**Figure 3.30:** The *Dual-Stream RNN*. The network concatenates 20 high-level feature vectors from both the *Spatial Stream* and the *Temporal Stream* to produce a sequence of 20 feature vectors of size 4608 as input to a RNN.

We then compare both models on our validation set. The results are presented in figure 3.31 and table 3.13. We find that the best performing model is the *Dual-Stream* LSTM. Since no performance gains can be seen from using the more advanced Bidirectional LSTM network, we conclude that our *Dual-Stream* approach will use a Batch Nomalized LSTM network with 256 cells as its *Recurrent Network*.

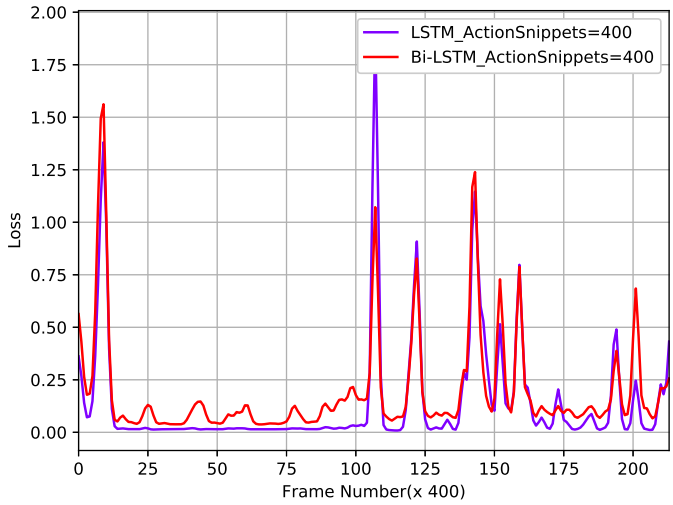



**Figure 3.31:** The validation loss for the LSTM and the Bidirectional LSTM *Dual-Stream RNNs* using Action Snippets of lenght 400. We see that the LSTM *Dual-Stream RNN* outperforms the Bidirectional LSTM Network.

### 3.7.3 The Final Dual-Stream

We now present our final *Dual-Stream Approach*. It consists of three parts as shown in figure 3.32.

**Spatial Stream**
The *Spatial Stream* consists of an improved VGG-16 Architecture, using Batch Normalization layers before every fully connected layer.

**Temporal Stream**
The *Temporal Stream* uses a 34-layer 3D-residual block network, taking input cubes of size $20 \times 224 \times 224$. It also uses the Type A downsampling strategy for the downsampling residual blocks and the *Keep Temporal Dimension* approach to maintain the temporal dimension size for longer.

**Recurrent Network**
The *Recurrent Network* consists of a Batch Nomalized LSTM model with 256 cells. It takes input sequences consisting of 20 high-level feature vectors from both the *Spatial Stream* and the *Temporal Stream*. The vectors from the *Spatial Stream* are created by performing element wise averaging over 20 consecutive feature vectors to reflect the output size of the *Temporal Stream*.

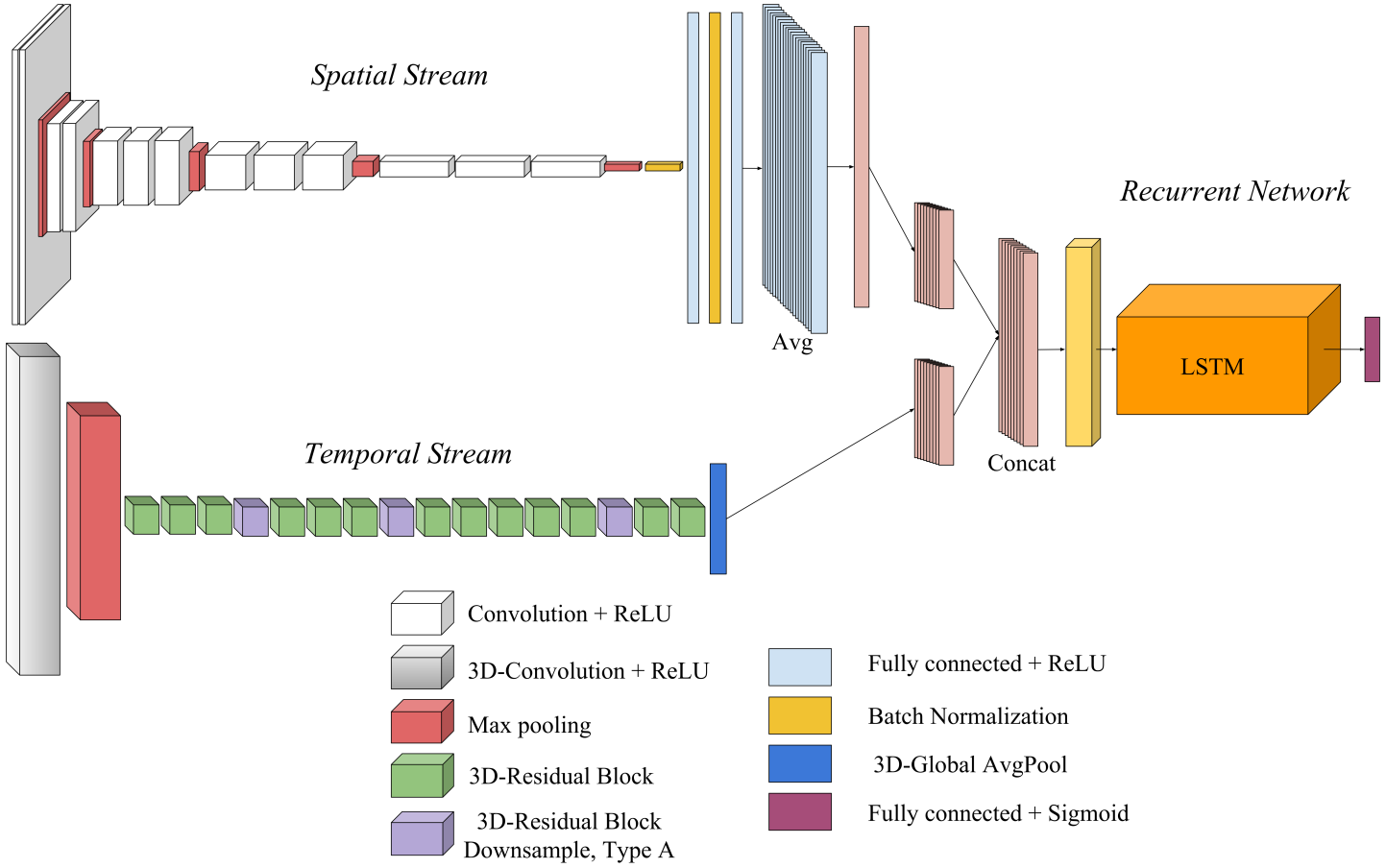
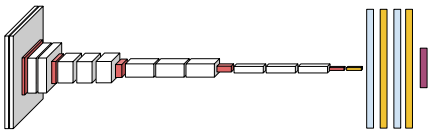



**Figure 3.32:** The final *Dual-Stream Approach*. It averages 20 high-level feature vectors from final fully connected layer in the *Spatial Stream* to produce one high-level average vector. 20 of these vectors are then concatenated with 20 high-level feature vectors, using the 3D-Global AvgPool in the *Temporal Stream* to produce a sequence of 20 input vectors for the 256-cell, Batch Normalized LSTM *Recurrent Network*. This corresponds to an Action Snippet length of 400 frames.

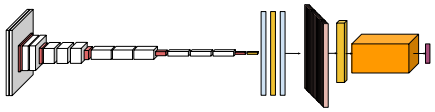
Chapter 4

# Testing and Analysis

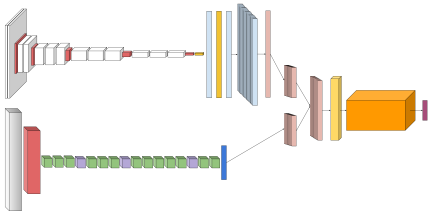
In chapter 3 we experimented with several Neural Networks for Activity Recognition in Salmon. By comparing their performance on our validation dataset, we arrived at three different network architectures, shown in figure 4.1. This chapter will present empirical testing of each of the architectures using our testing dataset, in an attempt to quantify the effects of increasing the amount of temporal information used for prediction and to test our hypotheses from section 1.2.



**(a)** Spatial Network(Detailed figure shown in figure 3.12).



**(b)** Spatial Recurrent Network(Detailed figure shown in figure 3.29).



**(c)** Dual-Stream Network(Detailed figure shown in figure 3.32).

**Figure 4.1:** The three network architectures which will be compared in this chapter.

## 4.1 Testing Dataset Overview

Our testing dataset consists of 20 videos, distributed over 11 Feeding videos and 9 NonFeeding videos. These videos have not been seen by neither us nor our architectures at any previous point in our work. This was done to avoid researcher bias and to follow the standard Machine Learning testing procedure[16]. As we mentioned in section 3.2.2, there are different video durations in our dataset. This is also the case in our testing dataset. However, the testing set was constructed to contain as equal a distribution from both classes as possible, without editing the individual videos. In table 4.1, the total distribution in our testing set is given.

**Table 4.1:** The distribution for the number of videos and number of frames in our testing dataset.

| **Video Class** | **Number of Videos** | **Number of Frames** | **Size Percentage** |
|---|---|---|---|
| Feeding | 11 | 43000 | 45.74% |
| NonFeeding | 9 | 51000 | 54.26% |
| **Total** | **20** | **94000** | **100.00%** |

To make referring to individual test videos easier, we introduce unique video IDs for each video. In table 4.2 we give an overview of all the testing videos, using video ID, video class and number of frames. A full table, showing the videos with their original video names as well as video IDs is available in Appendix B.

**Table 4.2:** Overview of the testing videos showing video ID, video class and number of frames.

| Video ID | Video Class | Number of Frames |
|---|---|---|
| Video 0 | Feeding | 2000 |
| Video 1 | Feeding | 2000 |
| Video 2 | Feeding | 2000 |
| Video 3 | Feeding | 2000 |
| Video 4 | Feeding | 5000 |
| Video 5 | Feeding | 5000 |
| Video 6 | Feeding | 5000 |
| Video 7 | Feeding | 5000 |
| Video 8 | Feeding | 5000 |
| Video 9 | Feeding | 5000 |
| Video 10 | Feeding | 5000 |
| Video 11 | NonFeeding | 2000 |
| Video 12 | NonFeeding | 2000 |
| Video 13 | NonFeeding | 2000 |
| Video 14 | NonFeeding | 5000 |
| Video 15 | NonFeeding | 5000 |
| Video 16 | NonFeeding | 5000 |
| Video 17 | NonFeeding | 5000 |
| Video 18 | NonFeeding | 5000 |
| Video 19 | NonFeeding | 20000 |

## 4.2 Testing Procedure

The main measure of performance for this master's thesis is Video Action Recognition Accuracy, seen in Measure 2 below. However, since we introduced Action Snippets in section 3.2.2.2, we have included performances on Action Snippet Action Recognition Accuracy as well. We give four performance measures in our testing:

**Measure 1:** Individual Video Action Recognition accuracy. This measure gives one binary classification for the entire duration of the video, where 1 means correctly classified and 0 means incorrectly classified. The classify the video using the same class as the majority of predictions for that video. If the amount is the same for both classes, we say the video is incorrectly classified.

**Measure 2:** Average Video Action Recognition accuracy for the entire test set. This measure gives the total percentage of correctly classified videos using Video Action Recondition accuracy. This is the main performance measure for this master's thesis.

**Measure 3:** Average Action Snippet accuracy for individual videos. This measure gives the the average accuracy for all the Action Snippets in that video.

**Measure 4:** Average Action Snippet Action Recognition accuracy over entire test set. This measure gives the average accuracy over the entire test dataset using Action Snippets. This is the measure used in chapter 3 and is included for better comparison between the validation results and our test results.

We use four measures so that we are able to assess architecture performance on individual videos as well as the entire dataset. Using Action Snippets also makes it easier to compare our test results to the validation results, reported in chapter 3, since these results are only reported for the entire validation dataset and not individual videos. Finally, testing individual video accuracies, using both binary and Action Snippet measures, also enable us to determine if there are some videos that stand out as either harder or easier to classify, helping us better understand the overall architecture performance.

## 4.3 The Spatial Architecture

The *Spatial Architecture* consists of the final *Spatial Stream* architecture, described in section 3.5.7. It takes single frames as input, corresponding to Action Snippets of length 1. In table 4.3 we show the results on our test dataset using the four measures described in section 4.2.

**Table 4.3:** Testing results for the *Spatial Architecture*

| | **Performance Measure:** | | | |
|---|---|---|---|---|
| **Video ID:** | **Measure 1** | **Measure 2** | **Measure 3** | **Measure 4** |
| Video 0 | 0 | 65.0% | 49.0% | 72.3% |
| Video 1 | 1 | | 65.1% | |
| Video 2 | 0 | | 2.3% | |
| Video 3 | 0 | | 0.0% | |
| Video 4 | 1 | | 71.9% | |
| Video 5 | 1 | | 90.1% | |
| Video 6 | 1 | | 95.1% | |
| Video 7 | 1 | | 71.8% | |
| Video 8 | 0 | | 11.6% | |
| Video 9 | 0 | | 32.6% | |
| Video 10 | 0 | | 9.1% | |
| Video 11 | 1 | | 99.9% | |
| Video 12 | 1 | | 100.0% | |
| Video 13 | 1 | | 100.0% | |
| Video 14 | 0 | | 12.6% | |
| Video 15 | 1 | | 99.6% | |
| Video 16 | 1 | | 99.9% | |
| Video 17 | 1 | | 99.3% | |
| Video 18 | 1 | | 99.6% | |
| Video 19 | 1 | | 100.0% | |

Using Measure 1, we see that the *Spatial Architecture* is able to correctly classify 13 of the 20 test videos, giving it a Video Action Recognition accuracy of 65.0% as seen in Measure 2. Further, using Measure 4, we note that 72.3% of all Action Snippets in the entire test dataset are correctly classified. From Measure 3 we see that this is largely due to the network's ability to accurately classify the NonFeeding videos. Only one of the NonFeeding videos is wrongly classified, while six of the Feeding videos are wrongly classified. This indicates that Feeding videos are harder to recognize, which also becomes clear by comparing the accuracies for Feeding and NonFeeding videos using Measure 3.

## 4.4 The Spatial Recurrent Architecture

The *Spatial Recurrent Architecture* consists of the final *Spatial Recurrent Network* architecture, described in section 3.7.1. It uses 20 consecutive frames to produce a sequence of 20 vectors which are fed into a Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN). This corresponds to using Action Snippets of length 20. In table 4.4 we show the results on our test dataset using our four measures from section 4.2.

**Table 4.4:** Testing results for the *Spatial Recurrent Architecture*

| | **Performance Measure:** | | | |
|---|---|---|---|---|
| **Video ID:** | **Measure 1** | **Measure 2** | **Measure 3** | **Measure 4** |
| Video 0 | 1 | 75.0% | 97.0% | 80.9% |
| Video 1 | 1 | | 74.0% | |
| Video 2 | 0 | | 0.6% | |
| Video 3 | 0 | | 0.0% | |
| Video 4 | 1 | | 76.4% | |
| Video 5 | 1 | | 100.0% | |
| Video 6 | 1 | | 100.0% | |
| Video 7 | 1 | | 99.6% | |
| Video 8 | 0 | | 16.4% | |
| Video 9 | 1 | | 91.2% | |
| Video 10 | 0 | | 44.0% | |
| Video 11 | 1 | | 100.0% | |
| Video 12 | 1 | | 100.0% | |
| Video 13 | 1 | | 100.0% | |
| Video 14 | 0 | | 3.2% | |
| Video 15 | 1 | | 100.0% | |
| Video 16 | 1 | | 100.0% | |
| Video 17 | 1 | | 100.0% | |
| Video 18 | 1 | | 98.8% | |
| Video 19 | 1 | | 100.0% | |

We see that the *Spatial Recurrent Architecture* is able to correctly classify 15 of the 20 test videos. This results in a Video Action Recognition accuracy of 75.0%. We also observe that 80.9% of all Action Snippets in the testing dataset are correctly classified by this network. Using Measure 1 and Measure 3, we find that the reason for this increase in performance comes from an improved ability to correctly classify Feeding videos. This indicate that temporal information could be very important for correctly classifying Feeding behavior.

## 4.5 The Dual-Stream Architecture

The *Dual-Stream Architecture* consists of the final *Dual-Stream* approach, described in section 3.7.2. It uses 400 consecutive Frames to produce a sequence of 20 spatial vectors and 20 temporal vectors which are concatenated and fed into a LSTM RNN. This corresponds to using Action Snippets of length 400. In table 4.5 we show the results on our test dataset using the measures from section 4.2.

**Table 4.5:** Testing results for the *Dual-Stream Architecture*

| | **Performance Measure:** | | | |
|---|---|---|---|---|
| **Video ID:** | **Measure 1** | **Measure 2** | **Measure 3** | **Measure 4** |
| Video 0 | 1 | 80.0% | 100.0% | 86.4% |
| Video 1 | 1 | | 100.0% | |
| Video 2 | 1 | | 100.0% | |
| Video 3 | 0 | | 0.0% | |
| Video 4 | 1 | | 100.0% | |
| Video 5 | 1 | | 100.0% | |
| Video 6 | 1 | | 100.0% | |
| Video 7 | 1 | | 100.0% | |
| Video 8 | 0 | | 16.7% | |
| Video 9 | 1 | | 100.0% | |
| Video 10 | 1 | | 91.7% | |
| Video 11 | 0 | | 50.0% | |
| Video 12 | 1 | | 100.0% | |
| Video 13 | 1 | | 100.0% | |
| Video 14 | 0 | | 0.0% | |
| Video 15 | 1 | | 100.0% | |
| Video 16 | 1 | | 100.0% | |
| Video 17 | 1 | | 100.0% | |
| Video 18 | 1 | | 91.7% | |
| Video 19 | 1 | | 100.0% | |

From table 4.5 we find that the *Dual-Stream Architecture* correctly classifies 16 of the 20 test videos, which results in an Video Action Recognition accuracy of 80.0%. From Measure 4 we can see that 86.4% of all Action Snippets in the testing dataset are correctly classified by this architecture. Using Measure 1 and Measure 3, we again find that the increased performance comes largely from the ability to correctly classify Feeding videos. This further strengthens our belief that Feeding videos require more temporal data to be accurately classified than NonFeeding videos do.

## 4.6 Analysis

We now give a comparison between the three architectures. Table 4.6 gives the Video Action Recognition results on the testing dataset using Measure 2 from section 4.2. We find that the *Dual-Stream* architecture outperforms both the other architectures, while the *Spatial* architecture struggles the most. This indicates that utilizing more temporal information in an architecture results in notable performance gains.

**Table 4.6:** A comparison of the test results between the *Spatial Architecture*, the *Spatial Recurrent Architecture* and the *Dual-Stream Architecture* using Performance Measure 2.

| **Architecture:** | **Performance Measure 2:** |
|---|---|
| Spatial | 65.0% |
| Spatial Recurrent | 75.0% |
| Dual-Stream | **80.0%** |

Table 4.7 gives the average Action Snippet Action Recognition results for the testing dataset, using Measure 4. We find that the *Dual-Stream* architecture significantly outperforms all the other architectures and that the *Spatial* architecture is the struggling the most. This further strengthens the evidence suggesting that more temporal data leads to better architecture performance.

**Table 4.7:** A comparison of the test results between the *Spatial Architecture*, the *Spatial Recurrent Architecture* and the *Dual-Stream Architecture* using Performance Measure 4.

| **Architecture:** | **Performance Measure 4:** |
|---|---|
| Spatial | 72.3% |
| Spatial Recurrent | 80.9% |
| Dual-Stream | **86.4%** |

In table 4.8, we present the average Action Snippet Action Recognition results for the individual videos, using Measure 3. We find that the *Dual-Stream* architecture outperforms the two other architectures by a significant margin on multiple videos. We also observe that the *Spatial Recurrent* outperforms the *Spatial* architecture on almost every video. Furthermore, we note that both the *Spatial Recurrent* and the *Dual-Stream* are much more consistent in their classification. Producing either very high or very low accuracies, indicating that the architectures produce very similar predictions throughout the entire video.

This is further shown in the loss curves for the three architectures, given in figure 4.2. The loss curves clearly show that both the *Spatial Recurrent* and the *Dual-Stream* architecture are generally very accurate in their predictions, as is seen by the low loss for most of the videos. We also see that even when the architectures wrongly classify Action Snippets, they are very wrong, as is indicated by the high loss spikes for both architectures. This is to be expected from Recurrent Networks, as they use previous predictions when producing their current predictions. Moreover, we see that the *Dual-Stream* loss is generally lower than the *Spatial Recurrent* loss, indicating better performance.

Using tables 4.8, we also note that more temporal information mostly seem to influence the performance on the Feeding videos. This is mainly because performance on NonFeeding videos is already really good, even for the *Spatial* architecture. However, it also suggests that Feeding videos require more temporal information for accurate classification. We hypothesize that this is because of the much higher variance in swimming directions in Feeding videos, as we saw in figure 3.3. The different swimming directions might seem "confusing" to the architectures at first glance, but given more temporal data, the architecture can make sense of the swimming patterns, resulting in a Feeding classification.

**Table 4.8:** A comparison of the test results between the *Spatial Architecture*, the *Spatial Recurrent Architecture* and the *Dual-Stream Architecture* using Performance Measure 2.

| **Video ID:** | **Performance Measure 3:** | | |
|---|---|---|---|
| | **Spatial** | **Spatial Recurrent** | **Dual-Stream** |
| Video 0 | 49.0% | 97.0% | **100.0%** |
| Video 1 | 65.1% | 74.0% | **100.0%** |
| Video 2 | 2.3% | 0.6% | **100.0%** |
| Video 3 | **0.0%** | **0.0%** | **0.0%** |
| Video 4 | 71.9% | 76.4% | **100.0%** |
| Video 5 | 90.0% | **100.0%** | **100.0%** |
| Video 6 | 95.1% | **100.0%** | **100.0%** |
| Video 7 | 71.8% | 99.6% | **100.0%** |
| Video 8 | 11.6% | 16.4% | **16.7%** |
| Video 9 | 32.6% | 91.2% | **100.0%** |
| Video 10 | 9.1% | 44.0% | **91.7%** |
| Video 11 | 99.9% | **100.0%** | 50.0% |
| Video 12 | **100.0%** | **100.0%** | **100.0%** |
| Video 13 | **100.0%** | **100.0%** | **100.0%** |
| Video 14 | **12.557%** | 3.2% | 0.0% |
| Video 15 | 99.6% | **100.0%** | **100.0%** |
| Video 16 | 100.0% | **100.0%** | **100.0%** |
| Video 17 | 99.3% | **100.0%** | **100.0%** |
| Video 18 | **99.6%** | 98.8% | 91.7% |
| Video 19 | **100.0%** | **100.0%** | **100.0%** |

These result suggest that the 3D-Convolutions combined with Optical Flow input provide important motion information for the *Dual-Stream* architecture, which ultimately leads to it being the best performing architecture.

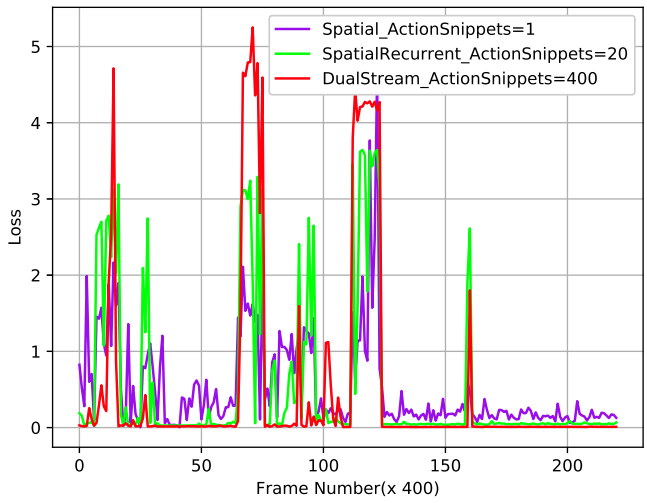



**Figure 4.2:** The loss of the three architectures for the entire testing set. Low loss indicates very good predictions, while high loss indicates very poor predictions. We see that the *Dual-Stream* architecture is very stable through each video, as is indicated by the flat loss curve. This is also somewhat observable when the it is wrongly classifying videos. This indicates very similar predictions for the entire duration of each video.

### 4.6.1 Testing the Hypotheses

In section 1.2, we introduced hypotheses **H1** and **H2**, which stated:

> **H1:** A system based on Convolutional Neural Networks can perform Action Recognition in videos of salmon with the goal of separating Feeding and NonFeeding behavior.
>
> **H2:** The system can be improved by including the temporal dimension contained within videos through the use of Recurrent Neural Networks or a combination of Recurrent Neural Networks and 3D-Convolutional Neural Networks.

In our tests we show that the *Spatial* architecture achieves a Video Action Recognition score of 65% and an Action Snippet Action Recognition score of 72.3% as seen in tables 4.6 and 4.7. This is well above the 50% mark which is to be expected from a random guesser. Thus we consider these results strong evidence for hypothesis **H1**.

Our test results also show that including the temporal dimension from videos through the use of RNNs, in the *Spatial Recurrent Architecture*, further improve performance. In tables 4.6 and 4.7, we see that we were able to produce a video Action Recognition score of 75.0% and an Action Snippet Action Recognition score

of 80.9% using our *Spatial Recurrent Architecture*. This is a significant improvement on the *Spatial Architecture* and gives strong indications for **H2**. Finally, our test results also show that utilizing even more temporal information further improves performance. By combining our *Temporal Stream* and our *Spatial Stream* to produce input to a Recurrent network, our *Dual-Stream Architecture* produces a video Action Recognition score of 80.0% and an Action Snippet Action Recognition score of 86.4% as seen in tables 4.6 and 4.7. This shows that we can improve further upon the *Spatial Recurrent Architecture* by including even more temporal information through motion, with our *Temporal Stream*. We therefore consider these results to be very good evidence for hypothesis **H2**.

### 4.6.2 Test Video Analysis

When we compare our test results to our validation results using Measure 3 for the three architectures, we would expect that they should be very similar. However, we note a significant reduction in performance on our test results compared to the validation results as seen in table 4.9. To find the reason for this, we analyze the individual videos of our testing dataset. Although the architectures produce different result on the different test videos, there are three videos that are consistently misclassified by all the architectures. These videos are:

**Video 3**

**Video 8**

**Video 14**

**Table 4.9:** The validation results compared to the test results for our three architectures.

| **Architecture:** | **Validation:** | **Test(Measure 3)** |
|---|---|---|
| Spatial | 89.1% | 72.3% |
| Spatial Recurrent | 94.9% | 80.9% |
| Dual-Stream | **95.2%** | 86.4% |

To investigate why these videos are much harder for the architectures to classify, we perform a visual inspection of each video. Since there are multiple correctly classified videos from the same class and dates as our three hard videos, we only compare our hard videos to these videos.

**Video 3**
When comparing Video 3 to the three other videos from the same date(Video 0-2), we notice that the fish density seems higher in Video 3 compared to the other. We also notice that the fish seem to be swimming closer to the camera than in the other videos. More importantly, we also notice what looks like a piece seaweed floating in and out of view as shown in figure 4.3. Although this seaweed is visible in the other videos from that date, it covers a lot more of the view in Video 3 than it does

in the other videos. It also moves around much more and with quicker motions in Video 3 compared to the others. This might produce very strange images for the *Spatial Stream* in the three architectures, as the seaweed covers large parts of the frame. This could make it very hard for the *Spatial Stream* to produce good activation maps. The fact that the seaweed also moves a lot more in Video 3 than the other videos, makes it very likely to produce very different Optical Flow frames than what the *Temporal Stream* is trained for. Thus the *Temporal Stream* of the *Dual-Stream* architecture might also be affected negatively.

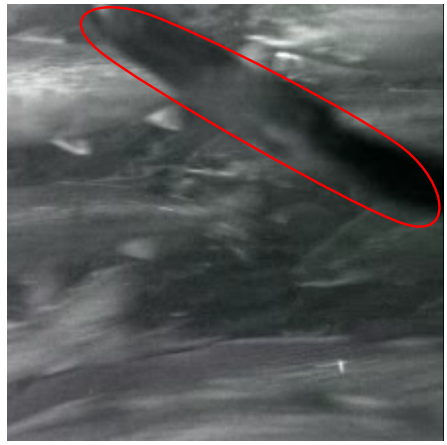


**Figure 4.3:** The seaweed floating in and out of view from Video 3. The seaweed moves a quite a lot and covers large portions of the frame for long durations. This could produce strange activations for the *Spatial Stream* and very abnormal Optical Flow frames, thus also negatively affecting the *Temporal Stream*. We believe that this is the reason for the poor performance in Video 3 by all architectures.

### Video 8

When we compare Video 8 to the other videos from the same date(Video 6-10) the only noticeable difference between the videos is that the direction of fish movement in Video 8 seems much more uniform than in the other videos. In all the other videos there are fish moving in different directions than the fish in the foreground. This will produce very different Optical Flow frames for Video 8, compared to the other videos, explaining the poor performance for the *Dual-Stream Architecture*. If the *Spatial Stream* uses the the way the fish are facing as a measure for its predictions, this could also explain why the *Spatial Architecture* and the *Spatial Recurrent Architecture* struggle with this video.

**Video 14**
When comparing Video 14 to the other videos from the same date(Video 15-17) the only thing we notice is that the fish in Video 14 seem to be further away from the camera. If the *Spatial Stream* uses distance from camera as a measure for its predictions, this might produce abnormal activations in this network.

#### 4.6.2.1 Adjusted Test Results

Since we found plausible reasons for why the hard test videos are difficult to classify, we present an adjusted comparison between our validation results and our test results. In this adjusted test set, we removed the hard videos from the test dataset. The results for all three architectures on the adjusted test set are given in table 4.10. From the table we see that all architectures had significant performance increases with the adjusted test set. We also note that the *Dual-Stream Architecture* was the architecture with the most performance gain using the adjusted test set. This indicate that the seaweed in Video 3 is making that video very hard for the *Temporal Stream*. We also present the adjusted Video Action Recognition accuracies in table 4.11 and find significant increases in Video Action Recognition accuracies in the adjusted results for all architectures.

**Table 4.10:** The validation results compared to the test results for our three architectures.

| **Architecture:** | **Validation:** | **Test(Measure 3)** |
|---|---|---|
| Spatial | 89.1% | 85.8% |
| Spatial Recurrent | 94.9% | 91.5% |
| Dual-Stream | 95.2% | **97.9%** |

**Table 4.11:** The adjusted test video Action Recognition accuracies for the three models.

| **Architecture:** | **Test(Measure 2)** |
|---|---|
| Spatial | 76.5% |
| Spatial Recurrent | 88.2% |
| Dual-Stream | **94.1%** |

### 4.6.3 Dataset Split Analysis

As we mentioned in section 3.2.2 our dataset split, splits our training, validation and test data based on dates. This was done to give the best representation of the architectures performances given new data. However, this also makes our task much more challenging. Regular Machine Learning dataset splits aim to create as representative datasets as possible, taking great care to ensure that both the training and validation datasets are representative of the test set. This is done because Machine Learning approaches are highly dependent on this property in order to produce good results. By splitting our dataset on dates, as opposed to

performing the split within each date, we completely remove all information from those dates from our training data. This means that our architectures have not been exposed to the conditions of those dates. Since salmon behavior could also be affected by the weather conditions, this makes our validation and test datasets much harder for the architectures to classify. However we are still able to produce very good testing and validation results despite these challenges. We therefore believe that the robustness of our architectures to noise is extremely good and that we could produce even better results, given a more representative dataset split. This also strengthens our belief in that we have developed a novel architecture well suited for general Action Recognition tasks.

### 4.6.4 Representativeness of the Dataset

Since our dataset is collected using a single camera rig and a single breeding cage, our architectures are likely biased towards that particular cage. Since there is a possibility that salmon behave differently under different conditions, our dataset does not contain these conditions. Thus the architectures presented in this thesis will not perform at the reported level without being trained on a dataset representative of these conditions. The salmon in Chile might behave very differently than the salmon in the north of Norway. There are also several species of salmon, which might also display very different patterns of behavior. Thus our architectures will have to be trained on a Chile dataset in order to produce similar results. Furthermore, our dataset was collected during the month of November and in the north of Norway. This means that our dataset might not be properly representative of the salmon behavior in that cage for an entire year, as salmon behavior might change based on water temperature and lighting conditions. Lastly, we note that the videos, on average, also contain less light than what is representative for an entire years worth. This is because the amount of light in the north of Norway during November is very limited. Since darker videos provide less contrast and spatial information, this makes the dataset used in this thesis more challenging to work with than if it was representative for an entire year. This makes us very confident in the results we have presented as a proof of concept for our architectures.

Chapter 5

# Conclusion and Future Work

This chapter will present the conclusions from our work presented in the previous chapters. We will also present how our work might impact the salmon breeding industry, before we present our vision for future work.

## 5.1 Conclusion

In this master's thesis we have developed and tested three architectures to perform Action Recognition on videos of salmon and to test two hypotheses:

> **H1:** A system based on Convolutional Neural Networks can perform Action Recognition in videos of salmon with the goal of separating Feeding and NonFeeding behavior.
>
> **H2:** The system can be improved by including the temporal dimension contained within videos through the use of Recurrent Neural Networks or a combination of Recurrent Neural Networks and 3D-Convolutional Neural Networks.

We used a dataset consisting of 76 videos of salmon, recorded from within a breeding cage. The videos were captured in the north of Norway during the month of November. We created our own dataset split, separating the dataset into training, validation and testing videos. The datasets contained an imbalanced distribution of Feeding and NonFeeding data. A subsampling method was therefore used to create datasets with equal amounts of Feeding and NonFeeding data for our training and validation data. Our test set was not subsampled and consisted of 20 videos, divided over 11 Feeding and 9 NonFeeding videos. The resulting total dataset consisted of 5.77 hours of videos. The percentage-wise size of the different splits were: 65.55%, 17.20% and 17.25% for training, validation and testing sets respectively. The split was done by separating the videos based on the date they were recorded. This was done to give a representative presentation of the performance of the three

architectures, given new video data, and to prove the robustness of our architectures through a challenging dataset split.

Since we had a limited amount of videos available, we separated the individual videos into Action Snippets for the training and validation phases, in order to produce more examples. The Action Snippets ranged in duration from a single frame to 400 frames of video, depending on the architecture used.

To test hypothesis **H1**, we implemented a Convolutional Neural Network (CNN), called the *Spatial Architecture*. We based our network on the VGG-16 model, proposed by Karen Simonyan and Andrew Zisserman[41]. This network takes single video frames as input and gives an output for every frame. Using our validation set as a guide, we made several improvements to the architecture, the main ones being summarized as:

1. Our Specular Removal Preprocessing Strategy for normalizing the dataset and removing specular reflections coming from the fish skin.

2. The use of Batch Normalization layers between every fully connected layer for faster convergence during training and improved performance.

Using the *Spatial Architecture* we were able to produce test Video Action Recognition Accuracy of 65.0% and Action Snippet Action Recognition Accuracy of 72.3%. This is much better than a random guesser, which would produce an accuracy of 50% in both measures, thus providing strong evidence for hypothesis **H1**.

Since humans usually require more than a single frame of video to accurately classify it, we explored whether the performance of the *Spatial Architecture* could be improved by including temporal information from the videos. We therefore tested hypothesis **H2** in two ways:

1. We extended the *Spatial Architecture* with a Recurrent Neural Network (RNN) to utilize the temporal information contained within videos, through treating videos as sequences of individual frames. We called this network the *Spatial Recurrent Architecture* and it can be seen in figure 3.29.

2. We combined the *Spatial Architecture* with a 3D-Convolutional Network taking Optical Flow frames as input, for motion feature detection. We called this network the *Temporal Stream*. The *Spatial Architecture* and the *Temporal Stream* were set up in a dual-stream configuration, processing input frames in parallel. The high-level feature vectors from these two networks were then used as input to a RNN to utilize the temporal information contained within the videos. We called this network the *Dual-Stream Architecture* and it can be seen in figure 3.32.

In the *Spatial Recurrent Architecture* we experimented with several types of recurrent networks as well as sequence lengths and found that a Long Short-Term Memory (LSTM) RNN, using 256 recurrent cells and a sequence length of 20 vectors gave the best performance. The *Spatial Recurrent Architecture* was compared to the *Spatial Architecture*, using our testing dataset. We found that the *Spatial Recurrent Architecture* outperformed the *Spatial Architecture* with a significant margin, producing a test Video Action Recognition Accuracy of 75.0% and Action Snippet Action Recognition Accuracy of 80.9%, giving strong evidence for hypothesis **H2**.

The *Temporal Stream* of the *Dual-Stream Architeture* was implemented, using a 3D-Convolutional Neural Network, taking input cubes consisting of several stacked consecutive Optical Flow frames. We explored multiple 3D-Convolutional Network architectures and found that a 34-layer 3D-Residual Network, using input cubes made from 20 consecutive Optical Flow frames, was the best performing architecture. This architecture was developed by extending the Residual Networks presented by He et al. [20, 21], to use 3D-Convolutions instead of 2D-Convolutions. We also explored the effects of using pooling layers instead of $1 \times 1 \times 1$ convolutional layers in the downsampling blocks. We found that maxPooling layers gave a significant increase in performance over the $1 \times 1 \times 1$ convolutional layers. This *Temporal Stream* presents, to the best of our knowledge, a *novel network architecture*, extending the Residual Neural Network architecture to be able to handle *Spatio Temporal* input through the use of 3D-Convolutional layers.

Using the findings from the *Spatial Recurrent Architecture* we used sequences of 20 vectors from both the *Spatial Architecture* and the *Temporal Stream* as input to the *Dual-Stream* Recurrent Network, resulting a total input of 400 frames for each output. For the recurrent network we found that a LSTM RNN, using 256 cells was the best performing architecture. In our test results we found that the *Dual-Stream Architecture* outperformed both the *Spatial Architecture* and the *Spatial Recurrent Architecture*, producing a test Video Action Recognition Accuracy of 80.0% and Action Snippet Action Recognition Accuracy of 86.4%. We therefore concluded that the *Dual-Stream Architecture* was the best suited architecture for Video Action Recognition in salmon. These findings suggests that extending the *Spatial Recurrent Architecture* with a 3D-Convolutional Network to capture even more temporal information through motion, is beneficial for performance. We considered these results as very strong evidence for hypothesis **H2**.

As we stated in the section 1.1, over half of the cost of breeding salmon in the Norwegian salmon farming industry comes from feed usage[13]. Since today's feeding process is a largely manual one, based on the amount of feed sinking to the bottom of the breeding cage, automation of this process through the use of salmon behavior could greatly reduce costs both in terms of labor needed and amount of feed wasted.

In this thesis, we have examined the feasibility of using Deep Learning approaches to automate the feeding process. Using videos of salmon from within a breeding cage as the input, we proposed several viable approaches to be able to automatically separate Feeding and NonFeeding behavior in the salmon.

We presented results showing that we were able to separate videos of Feeding and NonFeeding salmon with high accuracy. Through combining both spatial and temporal information in a Dual-Stream approach, we were able to accurately classify 80.0% of all testing videos using Video Action Recognition accuracy as our measure. Furthermore, we were able to accurately classify 86.4% of all frames in our testing videos through the use of Action Snippets. The practical implications from these results is that our architectures could be extended to develop robust models capable of continuous monitoring of salmon behavior and to perform on-line decision making on when to start and stop the feeding process.

The research contributions form our work can therefore be summarized in the following way:

1. A novel 3D-Residual Deep Neural Network architecture approach. By extending the existing residual networks proposed by He et al. [20, 21] through the use of 3D-Convolutions, we achieve state-of-the-arts performance using our Optical Flow Salmon Dataset.

2. A novel *Dual-Stream* Deep Neural Network architecture. By combining both *Spatial Stream* and *Temporal Stream* feature vectors as input to a LSTM RNN, we achieve state-of-the-arts performance on our Salmon Dataset.

## 5.2 Future Work

Expanding upon our current work we hope to evaluate the effects of using different *Spatial Stream* architectures, such as GoogLeNet[47] or 2D-Convolutional Residual Networks[20] in both their pretrained forms and training them from scratch. These networks report better performances on ILSVRC[38] and we want to explore if this could also lead to better performance on our data.

We believe that the techniques we have implemented would also show noticeable improvement by using longer input sequencers for our *Dual-Stream Architecture*. We therefore need more continuous data in order to validate this intuition.

We also believe that our *Dual-Stream Architecture* is a general approach, capable of producing good Action Recognition results for several action domains as well as other salmon cages. We therefore would like to collect data from several other cages over an entire year to validate our approach as a general approach to Salmon Activity Recognition. We also would like to test our approach on some of the existing Action Recognition Datasets to validate our approach as a general Action Recognition approach.

Finally, we believe that there are still large holes in the scientific understanding of what Deep Learning architectures are actually learning. To improve upon this, we want to explore and visualize what our architectures are learning in order to better understand Deep Neural Networks and to solidify and better explain our results through further understanding.

# Bibliography


[1] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dan ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. `http://tensorflow.org/`. Version: 2015. – Software available from tensorflow.org

[2] Boser, Bernhard E. ; Guyon, Isabelle M. ; Vapnik, Vladimir N.: A Training Algorithm for Optimal Margin Classifiers. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. New York, NY, USA : ACM, 1992 (COLT '92). – ISBN 0–89791–497–X, 144–152

[3] Bradski, G.: In: *Dr. Dobb's Journal of Software Tools*

[4] Collobert, Ronan ; Weston, Jason ; Bottou, Léon ; Karlen, Michael ; Kavukcuoglu, Koray ; Kuksa, Pavel: Natural language processing (almost) from scratch. In: *Journal of Machine Learning Research* 12 (2011), Nr. Aug, S. 2493–2537

[5] Cord, Matthieu: *Deep CNN and Weak Supervision Learning for visual recognition*. `https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/`. Version: 2016

[6] Csurka, Gabriella ; Dance, Christopher ; Fan, Lixin ; Willamowski, Jutta ; Bray, Cédric: Visual categorization with bags of keypoints. In:

*Workshop on statistical learning in computer vision, ECCV* Bd. 1 Prague, 2004, S. 1–2

[7] Damien, Aymeric u. a.: *TFLearn*. https://github.com/tflearn/tflearn, 2016

[8] Damien, Aymeric u. a.: *TfLearn VGG-16 pretrained model*. https://github.com/tflearn/models. Version: 2016

[9] Deselaers, Thomas ; Pimenidis, Lexi ; Ney, Hermann: Bag-of-visual-words models for adult image classification and filtering. In: *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on* IEEE, 2008, S. 1–4

[10] Donahue, Jeff ; Hendricks, Lisa A. ; Guadarrama, Sergio ; Rohrbach, Marcus ; Venugopalan, Subhashini ; Saenko, Kate ; Darrell, Trevor: Long-term Recurrent Convolutional Networks for Visual Recognition and Description. In: *CoRR* abs/1411.4389 (2014). http://arxiv.org/abs/1411.4389

[11] Duggan, Maeve: Photo and video sharing grow online. In: *Pew Research Internet Project* (2013)

[12] Farnebäck, Gunnar: Two-frame motion estimation based on polynomial expansion. In: *Image analysis* (2003), S. 363–370

[13] Fiskeridirektoratet: *Profitability survey on the production of Atlantic salmon and rainbow trout 2015*. http://www.fiskeridir.no/content/download/17237/244931/version/3/file/rap-lonnsomhet-akvakultur-2015.pdf. http://www.fiskeridir.no/content/download/17237/244931/version/3/file/rap-lonnsomhet-akvakultur-2015.pdf. Version: 2015. – [Online; accessed 19-March-2017]

[14] Gentile, Claudio ; Warmuth, Manfred K.: Linear hinge loss and average margin. In: *Advances in Neural Information Processing Systems*, 1999, S. 225–231

[15] Giel, Andrew ; Diaz, Ryan: Recurrent Neural Networks and Transfer Learning for Action Recognition. (2015). http://cs231n.stanford.edu/reports/giel_diaz.pdf

[16] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press. – 110–111 S. – http://www.deeplearningbook.org

[17] Graves, Alex ; Jaitly, Navdeep: Towards End-To-End Speech Recognition with Recurrent Neural Networks. In: *ICML* Bd. 14, 2014, S. 1764–1772

[18] Hashem, Ibrahim Abaker T. ; Yaqoob, Ibrar ; Anuar, Nor B. ; Mokhtar, Salimah ; Gani, Abdullah ; Khan, Samee U.: The rise of "big data" on cloud computing: Review and open research issues. In: *Information Systems* 47 (2015), S. 98–115

[19] Hastie, Trevor ; Tibshirani, Robert ; Friedman, Jerome: *The Elements of Statistical Learning*. 2. Springer. – 222 S.

[20] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep Residual Learning for Image Recognition. In: *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[21] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Identity mappings in deep residual networks. In: *European Conference on Computer Vision* Springer, 2016, S. 630–645

[22] Hou, Jingyi ; Wu, Xinxiao ; Yu, Feiwu ; Jia, Yunde ; undefined ; undefined ; undefined ; undefined: Multimedia event detection via deep spatial-temporal neural networks. In: *2016 IEEE International Conference on Multimedia and Expo (ICME)* 00 (2016), S. 1–6. http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/ICME.2016.7552981. – DOI doi.ieeecomputersociety.org/10.1109/ICME.2016.7552981. – ISSN 1945–788X

[23] Huval, Brody ; Wang, Tao ; Tandon, Sameep ; Kiske, Jeff ; Song, Will ; Pazhayampallil, Joel ; Andriluka, Mykhaylo ; Rajpurkar, Pranav ; Migimatsu, Toki ; Cheng-Yue, Royce ; Mujica, Fernando ; Coates, Adam ; Ng, Andrew Y.: An Empirical Evaluation of Deep Learning on Highway Driving. In: *CoRR* abs/1504.01716 (2015). http://arxiv.org/abs/1504.01716

[24] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *CoRR* abs/1502.03167 (2015). http://arxiv.org/abs/1502.03167

[25] Ji, Shuiwang ; Xu, Wei ; Yang, Ming ; Yu, Kai: 3D convolutional neural networks for human action recognition. In: *IEEE transactions on pattern analysis and machine intelligence* 35 (2013), Nr. 1, S. 221–231

[26] Karpathy, Andrej ; Toderici, George ; Shetty, Sanketh ; Leung, Thomas ; Sukthankar, Rahul ; Fei-Fei, Li: Large-scale Video Classification with Convolutional Neural Networks. In: *CVPR*, 2014

[27] Kitchenham, B. ; Charters, S: *Guidelines for performing Systematic Literature Reviews in Software Engineering*. 2007

[28] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. Version: 2012. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

pdf. In: Pereira, F. (Hrsg.) ; Burges, C. J. C. (Hrsg.) ; Bottou, L. (Hrsg.) ; Weinberger, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, 1097–1105

[29] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. Version: 2012. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf. In: Pereira, F. (Hrsg.) ; Burges, C. J. C. (Hrsg.) ; Bottou, L. (Hrsg.) ; Weinberger, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, 1097–1105

[30] Li, Fei-Fei ; Johnson, Justin ; Yeung, Serena: *CS231n: Convolutional Neural Networks for Visual Recognition, Lecture: Convolutional Neural Networks (CNNs / ConvNets)*. https://cs231n.github.io/convolutional-networks/. Version: 2017

[31] Lin, Zhihui ; Yuan, Chun: A Very Deep Sequences Learning Approach for Human Action Recognition. In: *International Conference on Multimedia Modeling* Springer, 2016, S. 256–267

[32] Ma, Shugao ; Bargal, Sarah A. ; Zhang, Jianming ; Sigal, Leonid ; Sclaroff, Stan: Do Less and Achieve More: Training CNNs for Action Recognition Utilizing Action Images from the Web. In: *CoRR* abs/1512.07155 (2015). http://arxiv.org/abs/1512.07155

[33] Maturana, Daniel ; Scherer, Sebastian: Voxnet: A 3d convolutional neural network for real-time object recognition. In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on* IEEE, 2015, S. 922–928

[34] Pedregosa, F. ; Varoquaux, G. ; Gramfort, A. ; Michel, V. ; Thirion, B. ; Grisel, O. ; Blondel, M. ; Prettenhofer, P. ; Weiss, R. ; Dubourg, V. ; Vanderplas, J. ; Passos, A. ; Cournapeau, D. ; Brucher, M. ; Perrot, M. ; Duchesnay, E.: Scikit-learn: Machine Learning in Python. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830

[35] Rautaray, Siddharth S. ; Agrawal, Anupam: Vision based hand gesture recognition for human computer interaction: a survey. In: *Artificial Intelligence Review* 43 (2015), Nr. 1, S. 1–54

[36] Rubinstein, Reuven Y. ; Kroese, Dirk P.: *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013

[37] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning representations by back-propagating errors. In: *Cognitive modeling* 5 (1988), Nr. 3, S. 1

[38] Russakovsky, Olga ; Deng, Jia ; Su, Hao ; Krause, Jonathan ; Satheesh, Sanjeev ; Ma, Sean ; Huang, Zhiheng ; Karpathy, Andrej ; Khosla, Aditya ; Bernstein, Michael ; Berg, Alexander C. ; Fei-Fei, Li: ImageNet Large Scale Visual Recognition Challenge. In: *International Journal of Computer Vision (IJCV)* 115 (2015), Nr. 3, S. 211–252. `http://dx.doi.org/10.1007/s11263-015-0816-y`. – DOI 10.1007/s11263–015–0816–y

[39] Schuster, Mike ; Paliwal, Kuldip K.: Bidirectional recurrent neural networks. In: *IEEE Transactions on Signal Processing* 45 (1997), Nr. 11, S. 2673–2681

[40] Shen, Lei ; Zhang, Junlin: Empirical Evaluation of RNN Architectures on Sentence Classification Task. In: *arXiv preprint arXiv:1609.09171* (2016)

[41] Simonyan, Karen ; Zisserman, Andrew: Two-stream convolutional networks for action recognition in videos. In: *Advances in neural information processing systems*, 2014, S. 568–576

[42] Singh, Bharat ; Marks, Tim K. ; Jones, Michael ; Tuzel, Oncel ; Shao, Ming: A multi-stream bi-directional recurrent neural network for fine-grained action detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, S. 1961–1970

[43] Soomro, Khurram ; Zamir, Amir R. ; Shah, Mubarak: UCF101: A dataset of 101 human actions classes from videos in the wild. In: *arXiv preprint arXiv:1212.0402* (2012)

[44] Srivastava, Nitish ; Hinton, Geoffrey E. ; Krizhevsky, Alex ; Sutskever, Ilya ; Salakhutdinov, Ruslan: Dropout: a simple way to prevent neural networks from overfitting. In: *Journal of Machine Learning Research* 15 (2014), Nr. 1, S. 1929–1958

[45] Subetha, T. ; Chitrakala, S.: A survey on human activity recognition from videos. In: *2016 International Conference on Information Communication and Embedded Systems (ICICES)*, 2016, S. 1–7

[46] Sutskever, Ilya ; Vinyals, Oriol ; Le, Quoc V.: Sequence to sequence learning with neural networks. In: *Advances in neural information processing systems*, 2014, S. 3104–3112

[47] Szegedy, Christian ; Liu, Wei ; Jia, Yangqing ; Sermanet, Pierre ; Reed, Scott ; Anguelov, Dragomir ; Erhan, Dumitru ; Vanhoucke, Vincent ; Rabinovich, Andrew: Going deeper with convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, S. 1–9

[48] Tran, Du ; Bourdev, Lubomir ; Fergus, Rob ; Torresani, Lorenzo ; Paluri, Manohar: Learning Spatiotemporal Features With 3D Convolutional

Networks. In: *The IEEE International Conference on Computer Vision (ICCV)*, 2015

[49] Veličković, Petar: *Deep learning for complete beginners: convolutional neural networks with keras*. `https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html`. Version: 2017

[50] Xin, Miao ; Zhang, Hong ; Wang, Helong ; Sun, Mingui ; Yuan, Ding: Arch: Adaptive recurrent-convolutional hybrid networks for long-term action recognition. In: *Neurocomputing* 178 (2016), S. 87–102

[51] Xu, Zhenqi ; Hu, Jiani ; Deng, Weihong: Recurrent convolutional neural network for video classification. In: *Multimedia and Expo (ICME), 2016 IEEE International Conference on* IEEE, 2016, S. 1–6

[52] Yue-Hei Ng, Joe ; Hausknecht, Matthew ; Vijayanarasimhan, Sudheendra ; Vinyals, Oriol ; Monga, Rajat ; Toderici, George: Beyond short snippets: Deep networks for video classification. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, S. 4694–4702

# Appendices

Appendix A

# Subsampling the Dataset

We store our datasets as individual frames in Feeding and NonFeeding folders so that the subsampling is easier to perform.

We only subsample the class with the most data examples in order to create an equal distribution between Feeding and NonFeeding examples. We use two different subsampling methods during our training, depending on which architecture is being trained:

**Method 1:** Action Snippets length = 1.

**Method 2:** Action Snippets length > 1.

Method 1 is used when training the *Spatial Stream*. This method only selects as many NonFeeding frames as Feeding frames during each training epoch. We shuffle the training data before each epoch and create mini-batches with an equal amount of Feeding and NonFeeding examples. We end the epoch when there are no more Feeding examples left in the epoch, thus making sure that we have trained on an equal amount of Feeding and NonFeeding frames.

Method 2 is used when training the *Temporal Stream* and the *Recurrent Networks*. These architectures require several consecutive frames as their Action Snippet input. For example, given an architecture that require $n$ consecutive frames, we sort our data and select every $n$th frame and add this frame to a new *n-frames* list of frames. The *n-frames* list is then shuffled and a representative selection is selected using the train_test_split utility from scikit-learn[34], using 42 as our random state. We then make sure that we have an equal amount of frames in both our Feeding *n-frames* list and our NonFeeding *n-frames* list. During training we use this the *n-frames* lists to build our training data. We do this by selecting a frame from our *n-frames* list and then add the $n$ consecutive frames from our original list, to create one Action Snippet of length $n$.

Appendix B

# List of Test Videos

**Table B.1:** Overview of the testing videos showing video names, video ID, video class and number of frames.

| Video Name | Video ID | Video Class | Number of Frames |
|---|---|---|---|
| 20161117_102001_cron2000frames_singlenotrigger_out1_224x224_from0930to1030_20kgmin | Video 0 | Feeding | 2000 |
| 20161117_105001_cron2000frames_singlenotrigger_out1_224x224_from1030to1200_10kgmin | Video 1 | Feeding | 2000 |
| 20161117_112001_cron2000frames_singlenotrigger_out1_224x224_from1030to1200_10kgmin | Video 2 | Feeding | 2000 |
| 20161117_115001_cron2000frames_singlenotrigger_out1_224x224_from1030to1200_10kgmin | Video 3 | Feeding | 2000 |
| 20161125_111501_cron5000frames_singlenotrigger_out1_224x224_from1115to1200_15kgmin | Video 4 | Feeding | 5000 |
| 20161125_114501_cron5000frames_singlenotrigger_out1_224x224_from1115to1200_15kgmin | Video 5 | Feeding | 5000 |
| 20161127_111501_cron5000frames_singlenotrigger_out1_224x224_from1021to1125_16kgmin | Video 6 | Feeding | 5000 |
| 20161127_114501_cron5000frames_singlenotrigger_out1_224x224_from1125to1325_12kgmin | Video 7 | Feeding | 5000 |
| 20161127_121501_cron5000frames_singlenotrigger_out1_224x224_from1125to1325_12kgmin | Video 8 | Feeding | 5000 |
| 20161127_124501_cron5000frames_singlenotrigger_out1_224x224_from1125to1325_12kgmin | Video 9 | Feeding | 5000 |
| 20161127_131501_cron5000frames_singlenotrigger_out1_224x224_from1125to1325_12kgmin | Video 10 | Feeding | 5000 |
| 20161117_122001_cron2000frames_singlenotrigger_out1_224x224 | Video 11 | NonFeeding | 2000 |
| 20161117_125001_cron2000frames_singlenotrigger_out1_224x224 | Video 12 | NonFeeding | 2000 |
| 20161117_132001_cron2000frames_singlenotrigger_out1_224x224 | Video 13 | NonFeeding | 2000 |
| 20161125_121501_cron5000frames_singlenotrigger_out1_224x224 | Video 14 | NonFeeding | 5000 |
| 20161125_124501_cron5000frames_singlenotrigger_out1_224x224 | Video 15 | NonFeeding | 5000 |
| 20161125_131501_cron5000frames_singlenotrigger_out1_224x224 | Video 16 | NonFeeding | 5000 |
| 20161125_134501_cron5000frames_singlenotrigger_out1_224x224 | Video 17 | NonFeeding | 5000 |
| 20161127_134502_cron5000frames_singlenotrigger_out1_224x224 | Video 18 | NonFeeding | 5000 |
| 20161117_134501_cron20000frames_singlenotriggerCAM2_out2_224x224 | Video 19 | NonFeeding | 20000 |

Appendix C

# Dataset Availability

The dataset used in this thesis was not openly available at the time the thesis was written. However, we plan to release the full dataset together with the article resulting from this thesis. This to encourage research on the subject and to contribute to the research community.

Appendix D

# Code Documentation

In the attachments for this thesis, we have included the commented source code, used for implementation and training of the architectures presented in this thesis. This source code, seen in Attachments/MastersCode, is only included for documentation purposes as it is not readily runnable in its delivered state. To verify the source code and our reported results, our trained models as well as our dataset is needed. This constitutes several gigabytes of data and is thus not practical to attach for the thesis. The source code also requires a powerful GPU to be able to run at a reasonable phase. This is because the architectures presented in this thesis consist of a large amount of parameters, which greatly benefit from the parallelization abilities of a GPU. It therefore also requires several parallel computing platforms for optimized GPU utilization. An example screenshot of the source code running on one of our computers is given in figure D.1.

We have also attached a short documentation of the source code, seen in Attachments/html. It can be viewed by opening the index.html file in an Internet browser. This documentation gives an overview of the code included in our attachments for easier visualization and understanding of the source code.

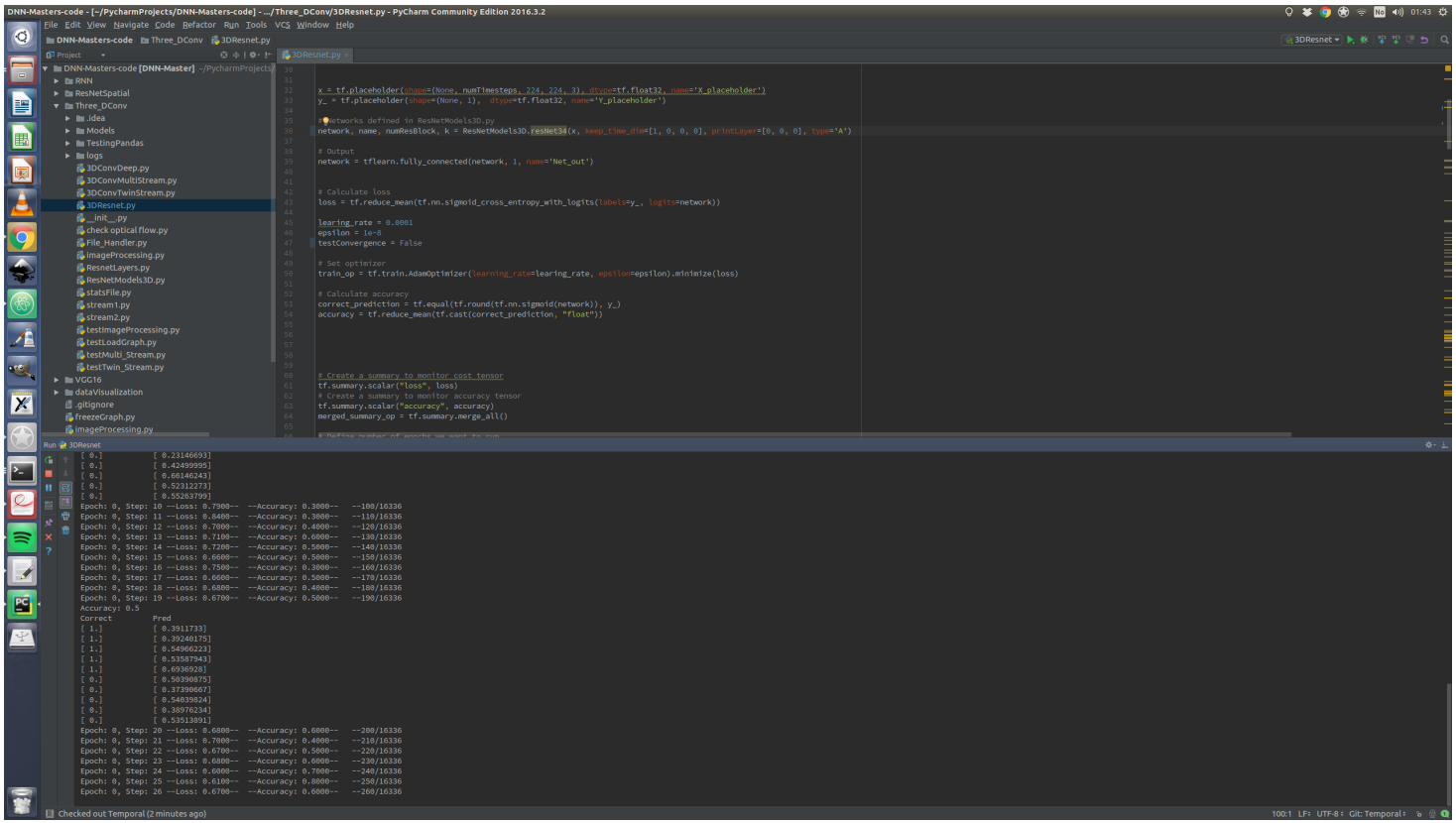



**Figure D.1:** A screenshot of the code running on one of the computers used in this thesis. We see the early training stages of a 34-layer 3D-Residual Network.