MASTER'S THESIS | LUND UNIVERSITY 2016

# Distributing a Neural Network on Axis Cameras

Anton Jakobsson, Axel Ahlbeck

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-41

# Distributing a Neural Network on Axis Cameras

Anton Jakobsson
dat11aj1@student.lu.se

Axel Ahlbeck
dat11aah@student.lu.se

August 25, 2016

Master's thesis work carried out at Axis Communications AB.

Supervisors: Mikael Lindberg, mikael.lindberg@axis.com
Niclas Danielsson, niclas.danielsson@axis.com
Patrik Persson, patrik.persson@cs.lth.se

Examiner: Jörn Janneck, jorn.janneck@cs.lth.se

## Abstract

This document describes the methods and results of our Master's Thesis, carried out at Axis Communications AB.

A central problem with deep neural networks is that they contain a large number of parameters and heavy computations. To cope with this, our idea was to split the network into chunks large enough that they require their own core, yet small enough to not violate our memory constraints.

The goal of the thesis is to investigate whether it is feasible to distribute and run a deep neural network on a network of cameras with tight constraints such as bandwidth and memory capacity. This is done by performing experiments on existing cameras as well as Raspberry Pi's as an assumption of how the next generation of cameras might perform.

The first part of the thesis discusses how a neural network can be partitioned, and describes the problems that may occur while doing so. The second part of the thesis presents results and measurements when run on cameras and Raspberry Pi's. The results and measurements are then discussed.

Optimizations and bottlenecks are then described and discussed. In this part, the thesis discusses how the application benefits from hardware acceleration. Conclusively a few unsolved problems are identified and presented as future work.

**Keywords**: Machine learning, computer vision, neural networks, deep learning, embedded systems, distributed systems

# Acknowledgements

We would like to thank the analytics and systems group at Axis for providing us with an excellent place to work with an inspiring atmosphere. We also direct a special thanks towards our supervisors - Mikael, Niclas and Patrik. For without whom this thesis would not exist.

# Contents

# Chapter 1
# Introduction

The field of Machine Learning and more specifically Neural Networks have exploded over the last few years. While machine learning and neural networks is not a new concept, the progress within neural networks which today all state of the art solutions depend on, has exploded due to recent discoveries made possible by increased computational power. This recent discovery kick-started a field which we now refer to as *deep learning* which includes algorithms that learns a high-level abstraction of data using multiple non-linear transformations. In 2012, the introduction of AlexNet hit a big milestone in the field of visual recognition using a convolutional neural network, CNN. Winning the ILSVRC2012 by a large margin, it showcased a portion of the power of neural networks[1].

This chapter serves as an introduction to our Master Thesis Project and to present some related research that touches on methods and results used by us.

## 1.1 Research question

Because of the composition and architecture of Neural Networks, the amount of parameters and computational power needed turns out to be considerable. In the case of AlexNet, the ~61 million parameters alone amounts up to roughly 285 MB. Add to that the memory footprint once AlexNet is run. Because of it's theoretical and complex nature, most applications running CNNs are developed in a high level language using frameworks abstracting the details.

In an embedded environment, there might not be enough memory or processing power to use neural networks without issue. However, if there are enough available computational units, so that their combined power and memory could sustain a neural network, it might be possible to perform classifications. In a scenario where the hardware already exists, it could be possible to use neural networks without the addition of specific hardware.

At Axis Communications, such a scenario exists.

Say a building has a set of Axis Network Cameras. Some of the cameras might not see much of significance on a typical day. However, when a camera does see something of interest, it might want to determine what it is looking at. It can then start a classification using a neural network, but because the neural network is so computation-heavy the camera can not perform the classification on its own. The cameras that are being idle could then help with the classification. Idle cameras are not utilizing their CPUs' full potential, so having them perform calculations is a way to tap into their unused computational power.

This is what we want to investigate. Could a neural network be distributed and run on Axis Network Cameras without the addition of specific hardware?

Our main research question is as follows:

- **Is it feasible to run a deep neural network by using distribution on embedded systems?**

Questions that are linked to this are:

- **How can we partition a neural network so that each node will receive a reasonable amount of data to process?**
- **How can we communicate data between the nodes in a satisfactory way?**
- **What kind of other bottlenecks exist when distributing neural networks?**

## 1.2 Terminology

Throughout this thesis, we will use some terminology that needs brief explaining:

- **CNN** - Convolutional Neural Network.
- **Pipeline Distribution** - Refers to the pipeline-like distribution described in section 4.3.
- **Iterative Reloading Distribution** - Refers to the distribution described in section 4.4.

The term *network* sometimes refers to a CNN and other times an IP-network. It should be apparent through context which meaning is intended.

## 1.3 Previous work

Most distributed and parallel neural network solutions as of today focus only on parallelizing the training phase. This is because when training a CNN, one needs to run through millions of training examples, backpropagate gradients and update corresponding weights with respect to the backpropagated gradients. The training phase may thus take up to several weeks of continuous computation. The parallellization is in most cases based around parallelizing over an input batch and averaging the weights update[2].

AlexNet is the neural network architecture that we base our implementation on. This is our baseline in which we compare correctness. AlexNet is described more detail in section 2.3.

The DarkNet framework (not to be confused with the underbelly of the searchable web) was developed as a way to implement deep neural networks in C. DarkNet is described in section 2.4. The framework has support for many different layers, which allows us to create neural networks freely.

Furthermore, some research has been done on neural network deployments fit for embedded systems. Some of these articles are summarized in 7.8.

## 1.3.1 Parallelizing Training

This section covers some work done by Krizhevsky[2]. The focus of this article is the parallelizing of the training phase. While training is not part of our thesis, it is still relevant since the article also discusses the parallelizing of the forward pass.

Krizhevsky mentions how the fully connected and convolutional layers have to be parallelized differently, because of the significant difference in number of parameters between them. Two methods are described, one for each type of layer, and they are called *data parallelism* and *model parallelism*. In data parallelism, the workers (nodes) train the same layer with different input, while in model parallelism the workers train different parts of the layer with the same input. The convolutional layers are trained using data parallelism and the fully connected layers using model parallelism. These concepts are not used by us, since we only care about the forward pass.

Most distributed machine learning pipelines are focused around scaling the training phase to a cluster of nodes before deploying it in the cloud. As far as we know no public papers present at this time discuss our topic.

# Chapter 2
# Background

This chapter contains a number of summaries of material and concepts relevant to this Master's Thesis. We direct our focus towards the application of neural networks, more specifically convolutional neural networks, in the field of computer vision. If you are familiar with the basic concepts of machine learning, we suggest you jump to section 2.2, otherwise continue reading.

## 2.1 Machine learning

Machine learning refers to the broad field of making a computer learn from data. In machine learning problems are differentiated into two subcategories, *supervised learning* and *unsupervised learning*.

### 2.1.1 Supervised learning

Supervised learning is the subset of machine learning algorithms in where the goal is to infer a function describing the relation between example input and its given ground truth output.

There is a discrimination between two types of sub problems in supervised learning, *regression* and *classification*. Problems where the goal is to predict a continuous value fall into the first category and problems where the goal lies in predicting a discrete value fall into the latter. Supervised learning is the most common category and includes popular algorithms such as *neural networks*, *support vector machines* as well as plain *linear regression*.

### 2.1.2 Unsupervised learning

In unsupervised learning the challenge at hand is the task of learning the underlying distribution of data, the hidden structure. The main difference from supervised learning is that no ground truth output exists to steer our solution by. In unsupervised learning the aim is therefore to discover groupings of similar input examples which is called *clustering*, or to learn more about the distribution of the data given input examples, called *density estimation*. Typical algorithms in the field of unsupervised learning are *k-means* and *mixture models*.

### 2.1.3 Training

With supervised learning there exists a need to make sure that the algorithm learns to reproduce the correct output corresponding to the given input. This is done in a training-phase where the input is fed through the algorithm, and then a cost-function is minimized in order to learn a representation which maximizes the number of correct classifications. Basically all machine learning algorithms boil down to an optimization problem.

In unsupervised learning, no notion of correctness exist so an algorithm cannot be trained for predictions. However, it is common procedure to manually inspect the output clusters or densities and tweak the hyperparameters associated with the used algorithm in order to try to get meaningful information.

A basic but nonetheless vital insight is that the goal of training is not to achieve 100% accuracy on the training data, but to maximize the precision when generalizing to previously unseen data. Tuning hyperparameters in order to achieve perfect scores on all of the data would lead to what in machine learning terms is called *overfitting*. This would be the result of trying to to fit a complex function on simple data. The converse of overfitting is called *underfitting* and follows from when a simple function is fit on complex data. Balancing over- and underfitting in order to maximize the generalization of the algorithm is tied to the *bias-variance tradeoff*.

When training a machine learning model it is therefore standard practice to divide the data up into three different sets - *training set*, *validation set*, *test set*. The model is trained on the training set, later on further optimized by tuning of hyperparameters on the validation set and eventually the generalization error is measured on the so far unseen test set.

Over- and underfitting will cause poor performance of an algorithm. There are several methods in order to combat this. Regularization is a method in where a penalty is introduced in the cost function in order to discourage an overly complex model. A typical regularization scheme seen in machine learning is L2-regularization, which augments the error function with the squared magnitude of all weights. This is preferably applied to data that cannot be sparsely represented. If the data instead can be well represented by a sparse solution, one would apply L1-regularization which augments the error function by a constant times the magnitude of the weights. Selecting wrong regularization can prove detrimental for the recovery of representation, L1 would select one signal as representation while neglecting the other, where as L2 would keep both but jointly shrink the weights.

Further regularization schemes used in neural networks are introduced in section 2.2.6.

### 2.1.4 Feature engineering

Feature engineering is a fundamental principle to machine learning and is critical to how well a model will perform. It is about designing the features, determining what the input into the algorithm should be for a specific problem. This can easily be the most expensive part of applying machine learning in practice, since it most likely requires expert domain knowledge in order to maximize performance.

In the case of neural networks manual selection of features is not needed. Neural networks itself learns to map features, this is called *feature learning*. The features learned by a neural network can be visualized by projecting the neuron activations associated with each layer back on to the pixel input space. Simply put, in the first few layers neuron activations are based on features similar to a gabor-filter, edge detection, where as deeper in the network more abstract visualizations and patterns surface since later activations is a combination of the earlier ones[3].

Machine learning and deep learning is heavily dependent on large training sets in order to get good enough performance. By performing data augmentation on the training data the number of training samples can be increased which will lead to better precision. Typical examples of data augmentation is to generate horizontal reflections and adding small image translations to the training data.

### 2.1.5 Neural networks

A neural network, or artificial neural network, can be described as a network of non-linear elements which are interconnected through adjustable weights. The neuron output is the sum over the inputs multiplied by its weights. A bias, constant value, is also added resulting in the equation $y = WX + b$. The non-linearity is then produced by feeding the neuron output through a threshold-function also called an activation function. Neurons are combined into *layers*, which themselves are combined to form a network-like structure. The background behind the name artificial neural network is that it is a mathematical construct designed to approximate a biological neural network. It is modeled after the brain, a network of neurons interconnected by synapses[4].

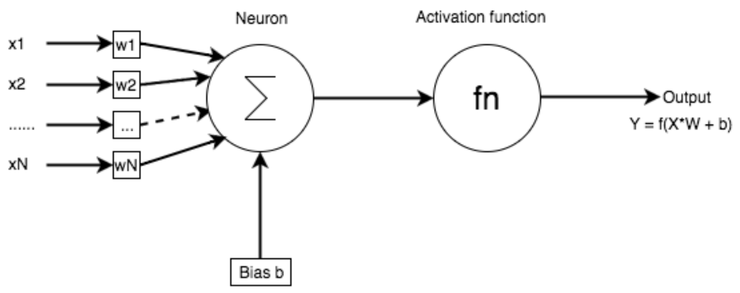

**Figure 2.1:** A single neuron with N inputs.

## 2.2 Convolutional Neural Networks

*Convolutional neural networks*, CNNs, refers to a feedforward neural network that makes use of convolutions forming a new buildingblock, the convolutional layer. This allows the network to keep and learn from important spatial information by definition of how a convolution works. Thus it makes intuitive sense that this kind of neural network would do well in practice in where a spatial dependency exist, such as computer vision[5]. In addition to the convolutional layer, most CNN architectures make use of more operations such as the original connected neurons(fully connected layer) and other operations such as *pooling* which is described in section 2.2.2.

### 2.2.1 Convolutional Layer

The convolutional layer is the core building block in CNNs and its output can be seen as a three dimensional tensor of neuron activations. As the name entails the layer basically applies the convolution operation over the input. Associated with the layer comes four degrees of freedom: number of *filters*, spatial extent of filters, *stride* and *padding*. Filter and kernel are two words that are used interchangeably and refers to a sliding window of weights. Each filter also has one bias term which is shared spatially for the filter. The filter is run over the input, calculating a dot product with its weights and what it covers in the input image. The dot product is the neuron's result before going through an activation function. After calculating the dot product the window shifts by the rate of the stride parameter, in terms of pixels. A layer consists of multiple filters, each with its own set of weights and bias term. Each multiplication when convolving, maps to an output neuron meaning that the resulting dimension when sliding a filter over the input will be:
Width = $(\frac{W-F_w}{Stride} + 1)$ and height = $(\frac{H-F_h}{Stride} + 1)$ with a depth of one. The output of each filter is referred to as an activation map. Each activation map is then stacked together meaning that the third dimension of the tensor will total the number of filters in the layer. If padding is used, a number of zeros is appended on all sides of the input. A pad of $\frac{F-1}{2}$ with stride of one would keep the original width and height. This is useful for keeping a reasonable dimension, otherwise depth grows while height and width decreases which could lead to important spatial information being lost. A side note is that the statistics on the border might be slightly different when padding.

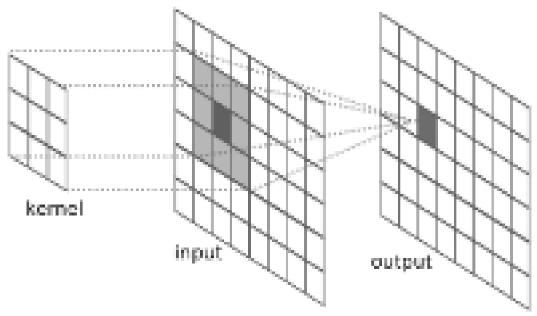

**Figure 2.2:** A 3x3 kernel resulting in one activation. Picture taken from [6].

The reason why convolutions work so well with image classification as opposed to a regular feedforward neural network is because of the weight sharing and receptive fields associated with the convolutional layer. A receptive field is the region of a kernel and the name originates from its biological counterpart in the retina. Each kernel has its own set of weights, and is run over the whole image. Therefore it will react and detect the same thing regardless of where in the image it is, *local invariance*, with some limitations concerning kernel size. When convolutional layers are stacked after each other compositionality of features is gained which means that lower level features from early layers will be combined into more complex features resembling real objects. As comparison, consider a network with only fully connected layers2.2.3. Where each neuron in the previous layer is connected with its own weight to all the neurons in the next layer. This would mean that detecting two of the exact same things in different parts of the input image would amount to two sets of completely independent weights.

### 2.2.2 Pooling Layer

A pooling layer is often used as a compliment in-between successive convolutional layers. Pooling works by partitioning the input data into a set of sub-regions, and outputs the maximum of each subregion. One could think of it as a sliding window, but instead of doing a convolution with regular multiplications it applies a function such as max, average or L2 norm over the receptive field. It is used as a dimension reduction by downsampling the output of a convolutional layer, reducing the number of activations and helps prevent overfitting.

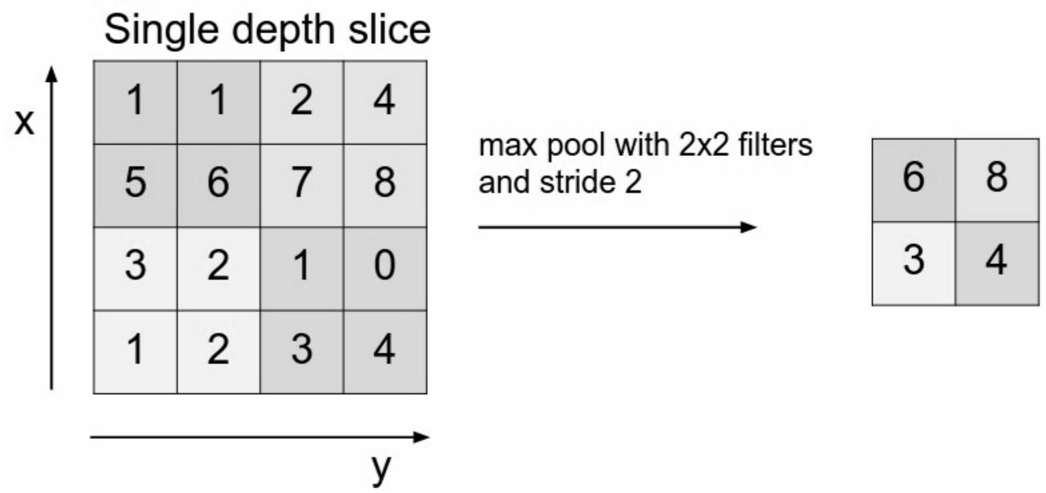

**Figure 2.3:** Maxpool operation with 2x2 kernel using stride 2 over a 4x4 input space. Picture retrieved from [7].

Pooling also provides a form of translation invariance to features. I.e a small translation of the input would have no noticeable effect on the pooled output. This is due to that the pooling works by throwing away a tiny bit of spatial information. E.g by using maxpooling the information thrown away would be the weaker neuron activations, by the definition of the argmax function. The weaker neuron activations would map to insignificant information in the input data. Feature invariance is particularly useful when we want to work with object recognition. The intuition behind it would be along the lines of that we do not really care whether a car wheel is in the upper right corner of the picture or in the bottom left. There still is a car wheel in the picture which should contribute to the probability that maps to the class car.
Traditionally, pooling is appled with stride equalling the kernel size. If instead a pooling layer with stride smaller than kernel size the kernel will overlap. This is called *overlapping pooling* and helps with robustness. Some reports have shown reduced error rates by using overlapping pooling[8].

### 2.2.3 Fully Connected Layer

The fully connected layer is the standard layer used in regular neural networks. Each output neuron depends on all the activations from the previous layer. This is where the largest amount of the weights are located. Recent studies show that as much as 95% of the weights in typical CNN architectures are located in the fully connected layers [2]. Computing the activations for a fully connected layer is done by a dot product with an added bias term. In convolutional neural networks, fully connected layers are usually placed as the last layers for when the input has already been transformed by stacked convolutions and pooling. Fully connected layers can be seen as the decision making layers. This is because when a tensor is fed through a fully connected layer the output is a one dimensional tensor, hence the spatial arrangement in the featuremap is lost.

### 2.2.4 Activation function

All neurons in a layer pass their output through a nonlinear activation function before passing the output to the next layer. The reason why a nonlinear function is almost always used is because that it increases the expressiveness of the algorithm. If a linear activation function were to be used in the intermediate layers the layers could be superposed via the superposition principle to just one layer resulting in a shallow network with small expressiveness. In previous years either a sigmoid or a tanh function were used as activation functions because of their nice derivatives, which reduced computational load. The main problem with these is that they eventually saturate. This means that the gradients associated with the signal would be small for either large or small values and we would not be able to propagate a weight update through the network which ultimately would lead to networks converging slow or not even not converging at all. in 2012, Krizhevsky et al. proposed a new activation function called Rectified Linear Unit, ReLU for short[1]. This function greatly increased the convergence rate of stochastic gradient descent as opposed to using sigmoid or tanh functions. The function is implemented by just a threshold at zero, $f(x) = max(0, x)$. By this it follows that the gradient is always one for signals larger than zero and hence it does not suffer from saturation that can occur in other activation functions.

### 2.2.5 Prediction

The end product of a CNN is its last layer's output. For a K-classifier its a K-dimensional vector which is fed through a logistic function which normalizes the values to be in the interval [0,1] and sum up to 1. This is called the *softmax*, or normalized exponential, function and the output is interpreted as the probability distribution over the K output classes given the input.

$$P(y = j|x) = \frac{e^{x^T w_j}}{\sum\limits_{k=1}^{K} e^{x^T w_k}}$$

### 2.2.6 Training a neural network

Training of a neural network might at first seem more complex than regular machine learning algorithms, however it uses the same standard practices. The main difference is that the output error needs to be corrected by backpropagating gradients through the whole network and update accordingly. This is done with an algorithm called *backpropagation*. The most common choice for the error function is the sum of squared errors, sse for short, defined as:

$$\sum_{i=1}^{n}(y_i - f(x_i))^2$$

An error that can occur when training deep neural networks is the *vanishing gradient* problem. To put it short, when using a tanh or sigmoid activation function the gradient decays exponentially with each layer when training, causing the gradient in the first layers to be very small which might lead to slow convergence. Using a ReLU activation function solves this problem since the function does not saturate[9].

As regularization in neural networks, *dropout* was a technique introduced in 2012 by Hinton et al. [10] which was used to prevent overfitting and speed up training. It works by randomly omitting neurons in the fully connected layers during training time, forcing the network to learn a representation not based on just a few neurons. Thus it makes the network more robust, since the output will not be based on a sparse internal representation. More recently *batch normalization* has conquered state of the art, reducing the need for regularization through dropout[11]. Batch normalization is implemented by bundling training data into small batches and then doing an averaged parameter update after each batch.

When training neural networks it is common to train in *epochs*. An epoch refers to an iteration over the complete training set. Training for several epochs increases the risk of overfitting, since the model will see each training example more than once.

### 2.2.7 Transfer learning

As previously stated, training a neural network to be precise requires a large amount of training data. However if only a smaller dataset for the task is available, a technique called *transfer learning* can be used. The technique involves retraining a proportional amount of an already pretrained network using the new training data. The new model is trained down-up meaning that the last few layers are retrained. How many layers that should be retrained is tied to how small the new dataset is, and the training is done as usual with the addition of freezing the learning rate on the layers in the beginning. The reason why this works is that the first layers have neuron activations responding to generic features, whilst the layers deeper down respond to more domain-specific features.

## 2.3 AlexNet

An important breakthrough in deep learning was the AlexNet architecture[1].The paper, first released 2012, showcased several new ideas and broke all previous records in the imageNet large scale visual recognition challenge, ILSVRC, which is seen as *the* competition to win when it comes to computer vision[12].

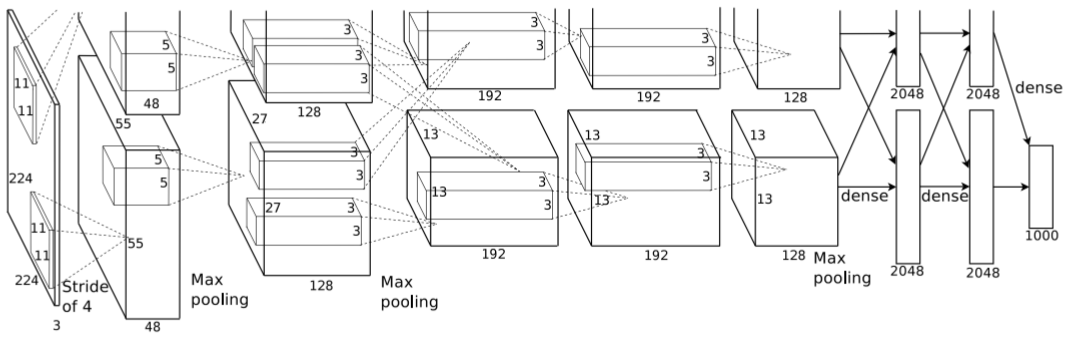

**Figure 2.4:** An illustration of AlexNet's architecture[1]

Depicted in figure 3.1 it is clear how the original architecture is split into two streams. This was a solution to GPU's not having enough memory capacity during training. It was not possible to fit the whole network on 3 GB of memory which was the limit at the time. This was the first architecture which introduced the technique called dropout as well as the ReLU activation function. We redirect the interested reader to the original paper[1].

Later on, in 2014, another paper was published by Alex Krizhevsky[2]. In this paper AlexNet was remodeled to follow a single column model using only one GPU instead of the original two. By utilizing the one-column model the goal was to experiment with parallelizing the training phase on an eight GPU setup. It was found that parallelizing the training works well for existing neural network models. The main architectural difference between the models is that there now is unrestricted layer to layer connectivity which leads to a needed small adjustment in layer parameters in order to fit dimensions. The final softmax layer was also changed to 1000 independent logistic units, each corresponding to one class and trained to minimize cross-entropy, instead of the original multinomial logistic regression. This was mainly because utilizing a multinomial logistic regression enforces the need to normalize over the classes which can not be parallelized and hence would make the cost of normalizing noticeable when scaling beyond 1000 classes.

## 2.4 DarkNet Framework

Darknet is an open-source neural network framework written in C and CUDA (for use with GPU)[13]. It supports both CPU and GPU computation, but for our purposes, only the CPU computations were used. Darknet comes with a pre-trained model of AlexNet, based on the original AlexNet[1], but adapted to modern machines according to[2].

Darknet was developed by Joseph Redmon in 2015. It comes with many features, including YOLO (You Only Look Once), Nightmare (simililar to Google's Deep Dream) and Imagenet Classification. Since AlexNet was developed to classify images using the Imagenet dataset, this feature is of significant relevance to us.

We use this framework as the basis for our C-library. By that we mean that all features not directly related to the forward pass have been removed, also some features have been added. For example the possibility of running partial layers. Because it is written in C, it can run on Axis Network Cameras. Since it also has a working implementation of AlexNet, we save time by not having to implement that as well. The AlexNet model used in this thesis is however slightly different from the one described in section 2.3. The new model is improved according to architectures developed in recent years, but still behaves the same. For a full presentation of how the models differ, see [13].

## 2.5 Node.js and FFI

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine[14]. It is event-driven and asynchronous, and designed to build scalable network applications. With Node.js, handling incoming connections can be done with little effort from the programmer. Node.js has many libraries which simplify the way connections are opened and used. In this thesis, we use the built-in net-module to administrate connections between the different cameras.

Since communication can be implemented in a short time in JavaScript, we can get a basic skeleton of a distributed application up and running much faster than we would have in C. JavaScript does however come with limitations such as performance and it being single-threaded, which makes it unfavourable when performing many heavy calculations concurrently. This is why we use the FFI-module (Foreign Function Interface), which allows us to call C-functions from Node.js. This way we can achieve the performance of C for the most part, while communicating with the simplicity of JavaScript. Although, there is some non-trivial overhead to FFI-calls[15], which will be addressed in section 3.1 and also in chapter 5.

## 2.6 Raspberry Pi 2 and 3 Model B

Raspberry Pi 2 Model B is a second generation Raspberry Pi released in February 2015[?]. It has a 900MHz quad-core ARM Cortex-A7 CPU and 1 GB of RAM. Because its CPU is an ARMv7 processor, we have access to the NEON hardware. NEON hardware allows for some acceleration of certain operations. This can be done in many ways, the most notable of which is the use of SIMD. With SIMD, multiplications can be vectorized and performed in parallel.

Raspberry Pi 3 Model B is the next generation Raspberry Pi, released in February 2016. It has a stronger CPU, a 1.2GHz 64-bit ARMv8 processor, also with access to NEON hardware acceleration. This processor also has a NEON accelerator, with a few other optimizations to go with it. The Raspberry Pi 3 was added in the late stages of this project. As a result, no optimizations unique to the Raspberry Pi 3 are used.

Due to the increased memory compared to the cameras, there is less constraints present when working with these platforms. The Raspberry Pi is used in this thesis as an example of how future camera hardware might perform.

## 2.7 Axis Q1615 Network Camera

The Q1615 is a network camera with an ARTPEC-5 System-on-chip, developed by Axis Communications. ARTPEC-5 has limited on-board memory and no writable disk. A portion of the RAM is instead dedicated as disk. To store larger amounts of data, an SD card can be used. In our case, the weights (parameters) will be stored on the SD card, and loaded into memory when needed. Of course, due to the limited memory, only a portion of the weights can be kept in memory at any one particular time.

The camera's CPU is a MIPS 1004kc that has a total of four cores. The processor has no on-chip floating point unit, and therefore uses soft floats which means it has to emulate the floating point unit in software. This poses somewhat of a problem for us given that our neural network implementation perform all of the computations in floating point arithmetic.

Many other features come with the Q1615 network camera. For more information on the camera, please have a look at the product specification: [16]

# Chapter 3
# Challenges

This section presents the most significant challenges associated with this thesis, in no particular order.

## 3.1 Reducing FFI-overhead

Because of the overhead associated with FFI-calls[15], we want to structure our code so that it gets maximum utilization of our C-library with as few FFI-calls as possible. That is why it is favorable to perform all the large scale multiplications within the C-code and only letting the JavaScript-code handle the result.

A not so obvious downside to using the FFI this way is that while the FFI-call is executing, all other execution halts. This includes the processing of incoming events, meaning that while the program is performing tasks in C, Node.js' queue of events will grow in size.

The FFI-overhead to each call on a Q1615 Camera is on average 5 ms (See chapter 6). While that is not such a long time by itself (in the context of our application), it would quickly add up if there were many FFI-calls in a row (in a loop for instance). Then the accumulated overhead would amount to several seconds for the computation-heavy layers.

## 3.2 Partitioning Convolutional Layers

Convolutional layers are inherently computation-heavy, and are therefore prone to be bottlenecks in the network. By splitting a convolutional layer over several nodes, the load on each node can be smaller, while the throughput of the network increases.

Partitioning a convolutional layer can be in one of two ways; either the input to the layer is partitioned, or the filters which are run over the image. Choosing one over the other has no significant impact on performance or throughput. However, partitioning the input means that spatial information in the image (or tensor) will be lost. To maintain accuracy, the network would need logic to compensate (See section 7.3 for more on this). If the filters are instead partitioned, no such logic is required since all spatial information is kept. The downside with this approach is that all the nodes need all of the input, which results in *N* times more input data being sent over the network where *N* is the number of nodes the layer is composed of. A positive effect of this is that the nodes only need a part of the weights which are associated with their assigned filters. This also means that there is a need to keep track of the filter order used when stitching together the output from each node.

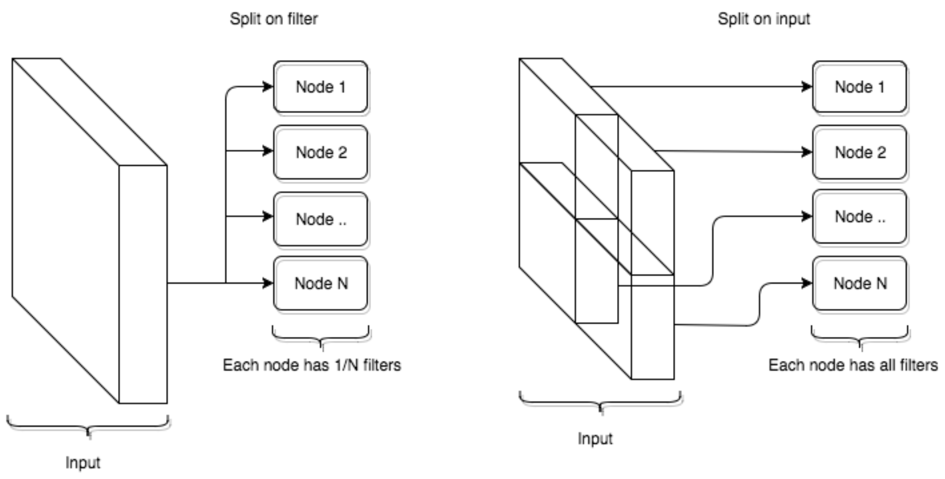

**Figure 3.1:** figure depicting how the input can be partitioned. Split on filter is the approach used in this thesis.

## 3.3 Workload on Nodes

The lab setup we use contains 24 raspberry Pi 3's, eight raspberry Pi 2's and eight Q1615 network cameras. With 32 Raspberry Pi's, there is no easy way to decide how the different nodes should be distributed. A straight-forward choice would be to have one layer distributed over four PI's and have the nodes never change their roles. Distributing this way could lead to congestion in the pipeline, if one layer is computed slower than the other. While this works when memory is not an issue, a more complex solution will be required when running on cameras.

With 8 cameras, in order for us to be able to complete a forward pass of AlexNet without too much hassle, at least 1/8 of the parameters for each layer must be present on each camera. Layer 6 in the CNN, which is the first fully connected layer, is the largest of the layers. During testing, we found that the cameras can, just barely, fit 1/8 of that layer in memory during runtime. However, no other layer can be placed in memory at that time, so all the cameras will need to be used to compute the forward pass for this particular layer.

## 3.4 Synchronizing Communication

A problem that presents itself in distributed applications is stalls and waiting periods. With no synchronization in place, the nodes would need to always have an equal workload, and the same execution time, if we want no deadlocks or unnecessary stalls. This is of course not likely to happen in a real-life situation.

In our application when run on cameras (see section 4.4), what would sometimes happen without synchronization is that many nodes have to wait for one node who is behind the others. However, often the nodes have to wait for node who is actually *ahead* of the rest. This might seem counter intuitive, but it has a perfectly logical explanation.

In section 3.1, we described how the execution can halt if a time consuming FFI-call is being executed. Also, because of the asynchronous nature of Node.js, there is no way to guarantee (without synchronization) that callbacks will be executed in the order presented in the program. Sending data over a socket is an asynchronous operation and can therefore execute immediately or in the future. What can happen is that one node can perform a forward pass and have all it's data sent to the next layer, but because of the asynchronous call, the data might not be sent before the node receives new data and starts computing the next layer. Because the forward pass is a time consuming FFI-call, the result of the *previous* computations can not be sent until the FFI-call has been completed.

## 3.5 Size of Communicated Data

With the distributed solution when using only cameras, the amount of data that needs to be communicated to the other nodes increases, because all of the nodes need all of the data. In the pipeline-distribution the layers are not distributed over all nodes, just a few nodes, so the data from each layer does not need to be broadcast.

Node.js has multiple ways of sending data over sockets, the easiest of which is to just send a string containing all the data. Obviously, there is some overhead to this approach, given that all floating point values have to be stringified.

In C, a number of type `float` is represented in the IEEE-754 format[17], which is a 32 bit representation. With this representation, each number can have a maximum of 9 significant digits[18]. When a number like that is stringified by Node.js, it is instead represented as a string of length 10 (9 numbers + 1 decimal point). This means that the initial 32 bit number is now an 80 bit string.

It gets worse, however. Node.js uses double precision floating point instead of single precision, which means that every floating point returned by the C-library is converted, and *then* stringified. In the double precision, a number can have up to 17 significant digits[18]. Now, the 32 bit number has turned into a string of length 18 (17 digits + 1 decimal point) represented in 144 bits. In this scheme, every number that needs to be sent is 4.5 times larger than its original representation. This implies that a lot of time will be spent communicating which will be a central problem for this thesis. This problem is tackled in 5.1.1.

# Chapter 4
# Method

Neural networks in an embedded environment face several problems. Due to the nature of Axis' cameras we had to limit ourselves in respect to what programming languages we may use. For instance, we cannot run python programs without difficulty since even a slimmed down interpreter would be large and limited. A working, cross-compiled, Node.js binary already exists in the firmware for the cameras, which we can use to run our application.

We chose to implement a distributed version of AlexNet since it is a well studied architecture. Our approach does not have support for backpropagation and thus has no support for training. This is simply a restriction we placed on our implementation since it is not computationally feasible to train a deep neural network on weak nodes running only with a CPU. Training is done offline with as powerful hardware as available. In order to run a neural network without training one needs to have access to a pre-trained set of weights corresponding to the chosen architecture. As mentioned in section 2.4, the Darknet-framework comes with pre-trained weights for AlexNet for us to use. Furthermore it is assumed that during runtime, all the nodes have 100% capacity available for us, and therefore we do not investigate how other running applications are affected.

In order to answer our research question, the investigation will be carried out according to the following steps:

1. Split network into separate layers and run locally on desktop
2. Split individual layers and parameters and run locally on desktop
3. Run distributed neural network on 8 separate Raspberry Pi's
4. Cross compile libraries and modules for Axis' cameras
5. Run distributed neural network on separate Q1615 Network Cameras
6. Run distributed neural network on 32 separate Raspberry Pi's

## 4.1 Distributing the Neural Network

Each layer in a Neural Network can be viewed as an independent part of a larger construction. The independent parts only performs computations on their input, and sends their output on before disregarding it.

This property can be used to split the network into smaller parts, that can fit on weaker computational nodes. Simply put, each layer will function as its own neural network. The individual nodes then carry a smaller number of parameters, but the output from each layer has to be sent over the network, resulting in increased latency. See chapter 6 for results and measurements.

However, splitting the neural network this way is not sufficient in order for us to make it run on the Q1615 cameras. The parameters associated with each of the fully connected layers are still much too large to fit within memory margin of the cameras. To further decrease the size of the parameters, the individual layers need to be split as well. While each convolutional layer may be able to run in its entirety on a single camera, it is still desirable to distribute them because of the computational complexity. Since the convolutional layers have the majority of their execution time made up of performing multiplications, they can benefit a lot from parallelism.

Given stronger computational nodes with less strict memory constraint and more compute, a pipelined distribution can be built. This makes it possible to achieve larger throughput of predictions, albeit still with an initial delay. A subproblem associated with this approach is that we need to do analysis of intra-layer computations, to compare execution time per layer, in order to find a distribution scheme which will even out the execution steps in the pipeline. Since the maximum throughput cannot exceed the throughput of the slowest stage in the pipeline[19].

When distributing, several problems can occur, such as synchronization and load balancing as well as lost data due to network issues. In this thesis we limit ourselves to just handle synchronization, and not dynamic load balancing or self-healing. More of this can be read in 9.

As for our model, we use a pretrained AlexNet network based on [2]. By model we mean all the weights and biases associated with an architecture. The weights and biases have been updated through backpropagation over 112640000 training samples from the imagenet dataset used for the ILSVRC[20].

## 4.2 Partitioning the Layers

As discussed in section 3.2, convolutional layers can be partitioned in one of two ways. We chose to partition the filters, because it was the most straight forward way and the other way would still lead to similar results. The effects of partitioning the layer spatially are discussed in section 7.3. Our method means that we let a set of nodes make up the convolutional layer, and each of the nodes handles an equal share of the filters. More often than not, an intermediate pooling layer is used after a convolutional layer. Since each activation map is independent we bundle our pooling-function with each node. This also means that we get a dimension reduction of the output before sending it over network to the next computational nodes corresponding to the next layer, reducing inter-layer communication. Considering AlexNet's configuration each pooling layer applies maxpool with size 3 and stride 2. This amounts to a factor 4 reduction of the output tensor. A nice attribute when partitioning over the filters, is that each filter is independent and increases the depth of the outgoing tensor. This makes it easy and fast to reconstruct the correct output from the nodes, since we can just append values in the correct order as depicted in 4.1.

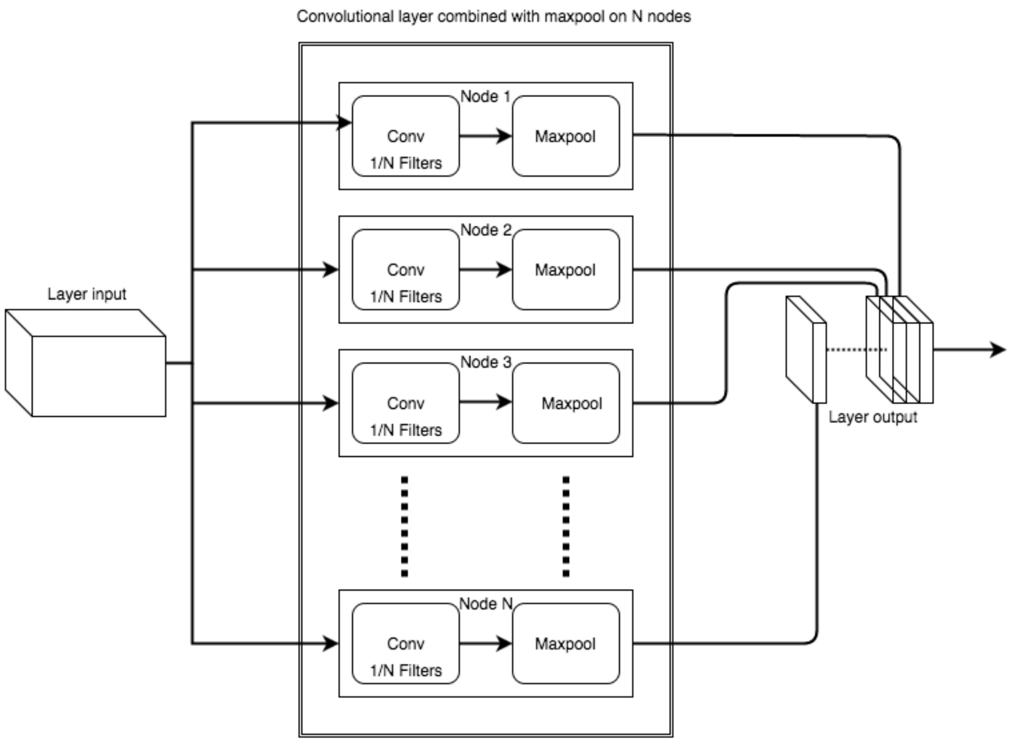

**Figure 4.1:** Visualization of the distributed implementation of a convolutional layer paired with a succeeding pooling layer

Fully connected layers are partitioned by its output neurons. We simply let each node handle uniform part of the output neurons. As described in 2.2.3, each single neuron in the fully connected layer is dependent on all of the output activation from the previous layer. Because the nodes all need the entire input, the communication overhead is somewhat increased. This is however not something we can avoid, since the memory constraints force us to distribute this way.

The final step in the algorithm is that a node, in our case we chose the node initiating the predictin, receives the neuron activations of the last fully connected layer from all nodes, a vector summing up to 1000 floats. Then it performs the softmax function to normalize the values to a probability distribution and presents the results. In order to guarantee a one-to-one mapping between the distributed model and the original model we compare the output vectors of both implementations, correctness would imply the last softmax layer outputs the exact same values.

## 4.3 CNN on Raspberry Pi 2 Model B

A Raspberry Pi is somewhat representative of a camera. The memory constraint is however non-present. In fact, a single Raspberry Pi can run AlexNet in its entirety without problem, other than the obvious increase in execution time.

With the Raspberry Pi's we can implement the ideal case, which is when none of the nodes need to change their role. To do this, we let each layer of AlexNet reside on its own node (apart from crop layers and pooling layers). This distribution creates a pipeline, where we can feed multiple images in a sequence through the network without having to wait for the previous one to finish.

Further optimizations can be made with the Raspberry Pi's. Since they all have their own quad core ARM processor, each Pi can run multiple Node.JS instances because it is single-treaded. This not only utilizies the CPU more efficiently, but it also reduces communication overhead between the nodes residing on the same Pi. However this could lead to congestion on the network bus since each Node.JS, run on the same PI, instance shares the same physical interface.

The setup of Raspberry Pi's serves as a testing ground for different distributions of the CNN. This platform does not require us to cross-compile any modules, and it does not require us to be as careful with memory management. This means that we can try to find a good distribution before porting our solution to be run on the cameras and future embedded platforms.

## 4.4 CNN on Q1615 Network Camera

Because the cameras have a MIPS-processor and no native compiler, everything needs to be cross-compiled and uploaded. The firmware on the cameras already contains a cross-compiled working Node.js runtime (version 0.12.7), so we only need to cross-compile the C-library and the third-party node-module ffi, used to handle the FFI-calls.

Unlike the case with the Raspberry Pi's, 8 Cameras is simply not enough to implement the case where no node need to switch its role. In fact, in order for us to perform a single forward pass, all 8 nodes will at some point need to switch roles (see section 3.3).

To complete a forward pass with just 8 cameras, we let each node handle 1/8 of each layer at a time, and then proceed to flush out the parameters before loading the weights for the next layer from the SD card. This is where problems with synchronization mentioned in section 3.4 start to occur. Because loading large amounts of parameters is also done using FFI-calls, there is even more risk of stalls than before. Henceforth this distribution scheme will be referred to as *iterative reloading*.

## 4.5 CNN on Raspberry Pi 2 and Q1615

We can simulate the scenario where we have enough cameras to never let any of the nodes switch roles, by using both the cameras and the Raspberry Pi's. The cameras can then have a fixed role that requires less parameters and calculations than before, while the Raspberry Pi's handle the rest. This way, we let the 8 cameras be part of a much larger network.

In this larger network, the nodes have different amounts of computational power. If we have a node that has twice the power of another, the logical thing to do would be to have the stronger node handle twice the amount of work that the weaker one does, decreasing total execution time by 33%. If we know how the different nodes perform in relation to each other, we can find a distribution where each node is responsible for a part of the algorithm proportional to its computational power. Load balancing is discussed in chapter 9. Impacts of using nodes with different computational power is discussed in section 7.5.1.

When distributing beyond eight nodes, the final fully connected layer will need to be distributed in a different way, since its 1000 neurons are not evenly divisible with any power of two larger than eight. How this affects the solution is discussed in section 7.1.1.

## 4.6 CNN on Raspberry Pi 2 and 3

With 32 Raspberry Pi's, we can get a better understanding of what happens when the work of each node becomes small and the amount of communication becomes large. For example, in the first convolutional layer there are 64 filters, so when distributing this layer over 32 Raspberry Pi's each Raspberry Pi will hold only two filters. The communication still increases, which suggests that when distributing over increasing number of nodes, there should exist some number of nodes where the cost of communication is too high and no speedup is achieved by distributing further. This is discussed in section 7.1.

# Chapter 5
# Software Architecture

This section provides a deeper look into the software architecture of this project.

## 5.1 JavaScript Application

As touched on in section 2.5, the communication part of the application is handled in JavaScript by Node.js. Additionally, the Javascript portion of the application is responsible for a number of things:

- The initialization of the FFI-module and the C-library
- The discovery of all available nodes that can participate in a classification.
- The initialization of a node's part of the neural network.
- The reinitialization of a node's part of the network when needed.
- The synchronization between nodes.

In order to discover how many computational nodes that are on the network we use an MDNS-discovery wrapper that listens on a gateway port and broadcasts when it's available. All available devices are recorded in a hashtable, and the devices present in the table are considered as discovered. When enough devices have been discovered, all the nodes connect to each other. This way we are able to connect to an arbitrary number of devices running our JavaScript program.

Each node is responsible for computing a different part of the neural network, specifically a layer, or part of a layer. When a node wants to classify an image, it starts by cropping the image. The node then starts the forward pass on the cropped image, before sending the result to the nodes that make up the next layer. When implementing this, we assume the scenario where all nodes have already received the image, so they can crop the image locally. This heavily reduces traffic on the network, since the image represented in floating points amounts to quite a few MB of data. In truth, pushing out large amounts of data on the network is not an issue since we have quite a forgiving bandwidth of 100 MB/s.

To perform the forward pass, the node uses the C-library via the FFI-module described in section 2.5. After having sent the data, the parameters for the next layer are loaded into memory. This again via an FFI-call. In summary, the JavaScript portion of the application can be seen as the orchestrating part of the application, while the C-library is responsible for all the heavy computations.

### 5.1.1 Representing the Data

Due to the structure of AlexNet, the output from the different layers can be quite large. And because data is sent as strings, the amount of data to be sent can be even larger. Output from the forward pass is a large vector of floating points, which have to be converted to a strings to be sent. As discussed in section 3.5, this leads to unecessary amounts of data on the network. To avoid this, we use the fact that the JavaScript code never has to be aware of the type or internal representation of the output from the C-library. What this means is that we never handle the output as floating point. Instead, we just wrap the output in a `Buffer`-object, and send that. The `Buffer`-class does not care about the internal representation of the data within it. It is a raw memory allocation outside the v8 heap. By representing the data this way, the size of the sent data will be the same before and after conversion. In addition to this, the data is also parsed faster on both ends.

## 5.2 C-library and `predictor.c`

Comprehensive functionality exists in the C-library, which we can easily use. However, it would not be preferable to use the FFI-module to make every single function accessible to the JavaScript code, because of the overhead associated with FFI-calls described in section 3.1. Instead, only a few functions are made available via `predictor.c`.

`predictor.c` is written by us, and it is a high-level abstraction of functions that make up the forward pass. More precisely, it handles all the library calls resulting in the forward pass. This way, the inner complexity can be hidden within the C-code rather than having extensive subroutines in JavaScript with multiple calls through the FFI.

The library is loaded into memory upon starting the application. Functions intended to be called in `predictor.c` have to be declared in the JavaScript code as well, in order for the FFI-module to understand what functions to call and to know what return values to expect. In order for the FFI-module to be able to execute functions in the library, the library has to be compiled as a shared library, i.e. with dynamic linking.
As can be seen in figure 5.1, `server.js` is completely cut-off from the library in that it simply calls a function using the FFI and awaits a response. The arrow pointing from the FFI-module to the `server.js` simply means a result is returned. Actually, `predic-tor.c` is not aware of the fact that the call is coming from a JavaScript program.

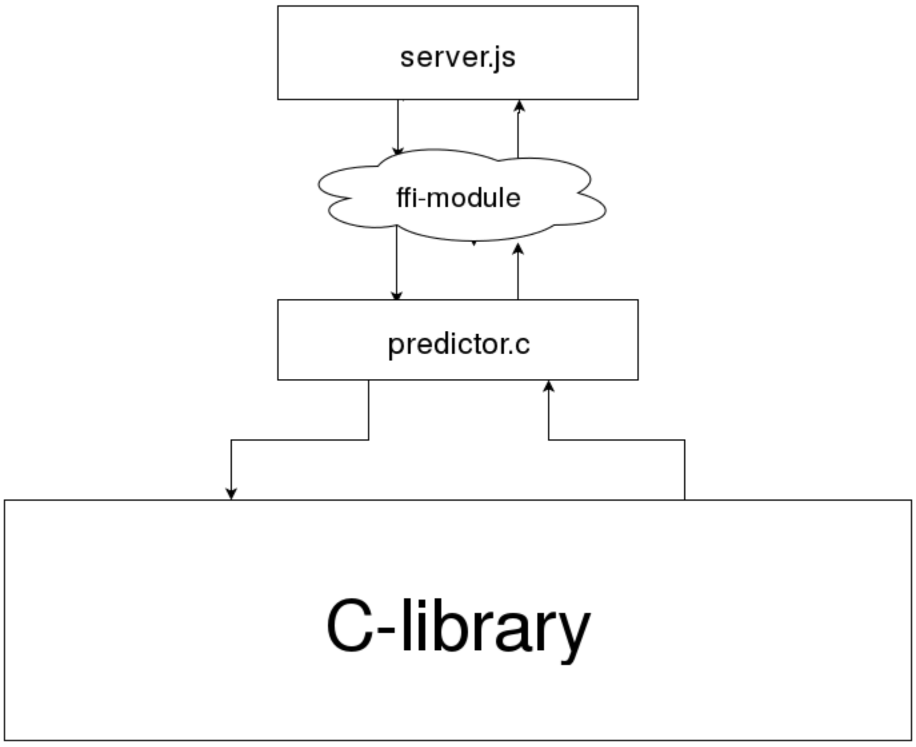

**Figure 5.1:** A basic visualization of how an FFI-call is carried out, when called from the JavaScript code.

This structure is not only easier on the programmer, but it also achieves better performance because less calls has to go through the FFI, as discussed in section 3.1. However, by having all of the resource consuming computations within the library, each FFI-call completely locks the program while executing, forcing us to introduce synchronization to avoid long stalls in the application (see section 3.4).

Another motivation for structuring the software this way is that all of the nodes are performing the same sequence of library calls to execute their forward pass. By only letting them call a few specific functions, we ensure that everything will be executed in the desired order.

### 5.2.1 CNN implementation details in C

Our end implementation is a modification of the darknet framework[13]. We have cut out unnecessary parts considering we only need to do a forward pass and we have built extra support for partial layers.

We represent a network using a C-struct, which holds parameters associated with input and output of the network as well as a pointer to a vector of layer-structs. Each layer-struct has a type associated to it and its layerspecific parameter which defines how it should perform its forward-pass. It also holds a pointer to its weights and biases. In order to make sense of the final network output, the output vector of the last layer, a softmax-layer, is sorted and mapped to a textfile containing indexed classnames.

Most of the math in a neural network can be described as a general scalar product between two arrays of floats. In order to implement a convolution as a scalar product, the data must first be transformed. This is done with an `im2col` operation, which rearranges a block patch from a given matrix and transforms it to a column. After this operation, the scalar product can be performed. A positive effect of this is that most system architectures have optimized subroutines for scalar products. After the scalar product, a simple addition of the bias is done on the output before finally being fed through a ReLU activation function which is implemented as a $max(0, output)$. This also holds for the fully connected neurons.

Unfortunately Node.js can not handle pointers the same way that C does. When a pointer is returned to the JavaScript application, it can not be dereferenced. That is of some significance to us, since the library can not simply return a pointer to the first element in an array that can then be sent over a socket. Instead, the array needs to be copied into a different array, allocated in the JavaScript portion of the application. This operation takes very little time compared to the multiplications in the forward pass, so overall performance is barely affected.

## 5.3 Orchestration

Unsurprisingly, as the nodes in the network become greater in numbers, the amount of coordination needed to perform the forward pass increases. To avoid the problems with stalls mentioned in section 3.4, we introduce a simple and well known synchronization technique: *acknowledgements*.

In the ideal case, all of the nodes would have something to execute all of the time. However, in the real world, this is difficult to achieve. Instead we try to synchronize so that every node is executing something *most* of the time. Since we do not want any node to lag behind or end up ahead of the rest, each node makes sure that all of the sent data is acknowledged before proceeding with its own execution. This does not completely eliminate stalls, as there is still some waiting. However, the execution runs more smoothly and performs better overall. Interestingly, with the performance increase due to the `Buffer`-representation of floats mentioned in section 5.1.1, combined with synchronization, the overhead associated with sending and receiving data decreases with the number of nodes (see chapter 6 for results and measurements). This suggests that while more coordination is required, the now smaller amounts of data require less parsing.

# Chapter 6
# Results

In order to determine how feasible our solution is, we wanted to measure the following things:

- Execution time of a forward pass when run on 8 Q1615 Cameras.
- Execution time of a forward pass when run on 1, 2, 4, 8, 16 and 32 Raspberry Pi's.
- The active and idle time of a camera's CPU when forward pass is run on 8 cameras.
- The active and idle time of a Raspberry Pi's CPU when forward pass is run on 8 Raspberry Pi's.
- How much of the execution time is spent on loading weights, forwarding and communication.

The Raspberry Pi's have more memory than we could ever hope to consume with our application, even when run on just one. Also, the application is not multithreaded at the time of taking these measurements. We leave this as future work.

## 6.1 Execution Times per Layer

In figure 6.1 the average execution time per layer in the AlexNet architecture, when run on 8 Q1615 Cameras. As can be seen in the graph, the forward pass is what make up most of the execution for the first five layers. This is perhaps not surprising, given that they are all convolutional layers. They have less weights and more multiplications by definition than the fully connected layers. The opposite is true for the fully connected layers, where loading weights takes up most of the time. It appears that the time spent loading weights per layer corresponds well with how many weights are associated with each layer. It is of course a trivial result, however it shows that if the loading of weights could be avoided the gain in execution time would be quite large for the fully connected layers.

Notice also the FFI-overhead displayed in the graph. By constructing our FFI-calls as described in this thesis, we have managed to minimize the overhead so much that it can be omitted. In fact, in most of the layers it is barely even visible.

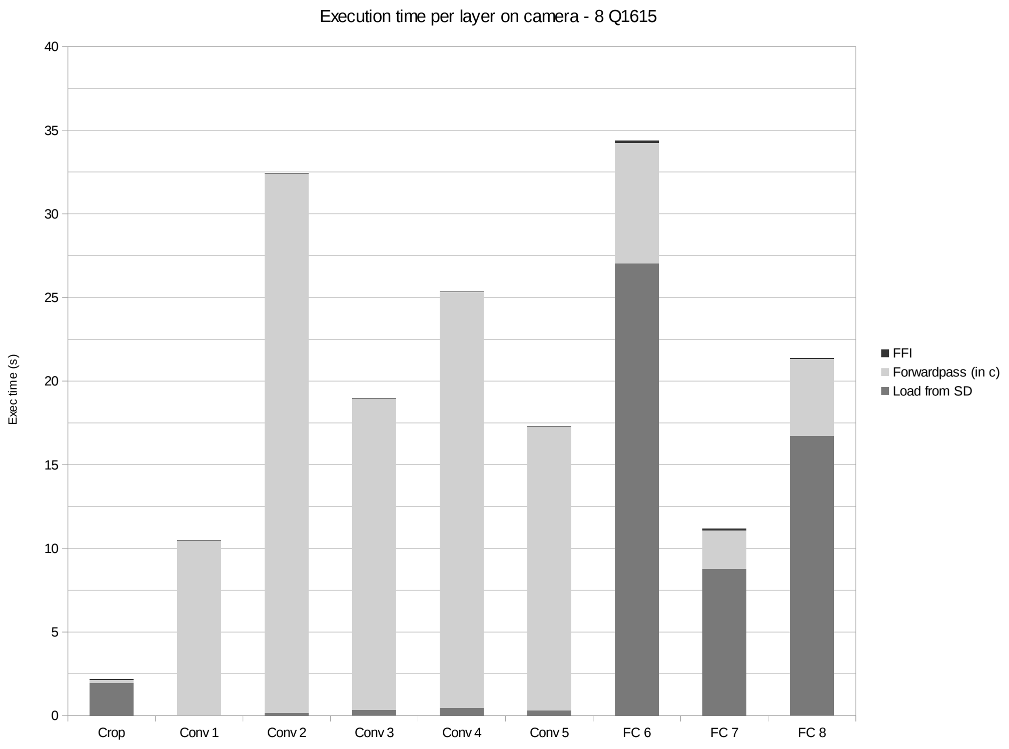

**Figure 6.1:** Measured execution time per layer in AlexNet when distributed over eight Axis Q1615 cameras. Load from SD is the time cost when loading in the weight parameters, forward pass is the total time of a complete pass over the layer and FFI is the added cost of calling a C-function from Node.js. This diagram does not include cost of inter-layer communication and orchestration.

In the pipeline distribution described in section 4.3, the goal is to eliminate the loading of weights. When distributed this way, the nodes only have to load their weights from memory once, and then keep them there. This can be done prior to starting the forward pass, so that the measured time does not include any reads from disk. Looking at figure 6.1, this means that the load portion of the bars would be removed. In this scenario, the throughput of the pipeline will be the execution time of the second convolutional layer, since it now is the most time consuming, given the fact that we do not perform load balancing. Impacts of how the CNN is distributed are discussed in section 7.2

## 6.2 Varying Number of Nodes

figure 6.2 shows the trend of decreasing execution time with an increasing number of nodes. This trend can not be seen with the Q1615 cameras, since the application requires all available cameras to run. Visible in the figure is the decrease in execution time for all the different aspects of the program. Because the amount of weights and multiplications halves with each step in the figure, it is logical that the execution times of those parts roughly halves with each step.

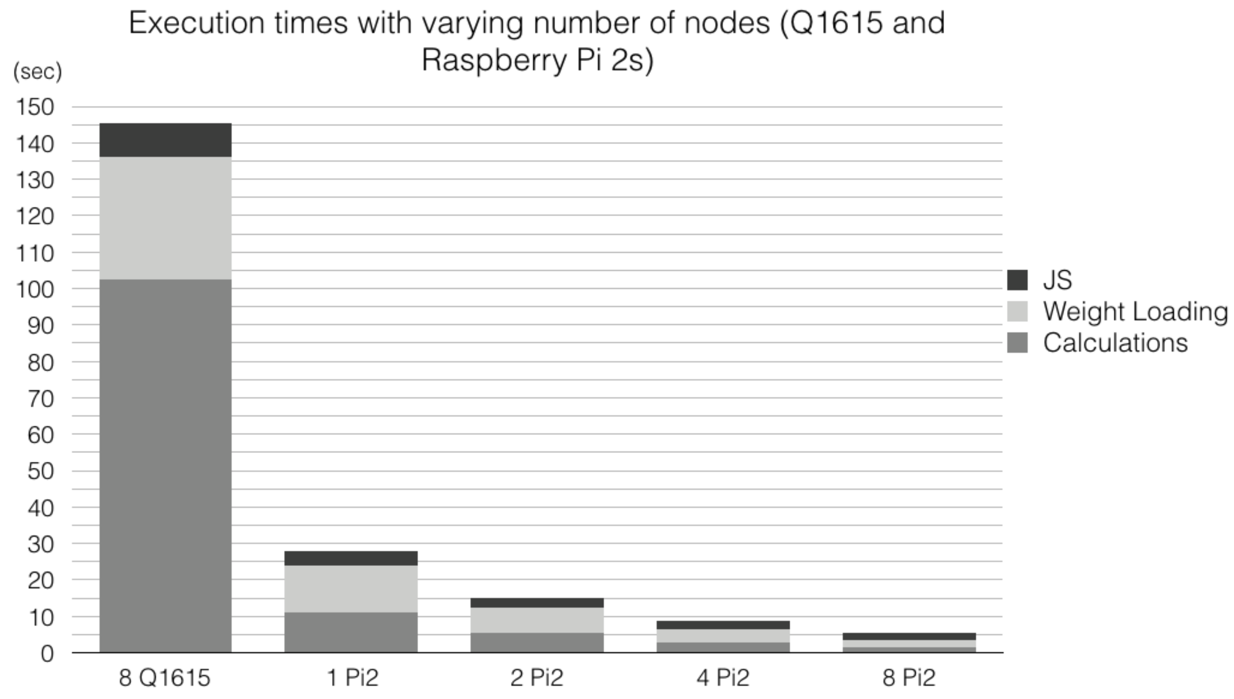

**Figure 6.2:** Mean execution time over 10 runs of AlexNet on 8 Q1615 cameras and increasing numbers of Raspberry Pi 2s. Shows relation between time spent in C environment (Weight Loading and Calculations) and JavaScript.

Just barely noticeable is also the decrease in execution time of the JavaScript portion of the application. This is somewhat counter intuitive, as it should be more difficult to schedule many nodes rather than few. For a more thorough discussion on this, see section 7.6.2. In figure 6.3, this trend seems to stop after 16 nodes, but this might be the result of using both Raspberry Pi 2 and 3 together. This special case is discussed in section 7.5.1. The exact execution times for the different amount of nodes can be found in appendix B.

When comparing figures 6.2 and 6.3 (excluding the the column representing the Q1615 cameras), it is visible that the Raspberry Pi 3 is quite a bit faster than the previous model.

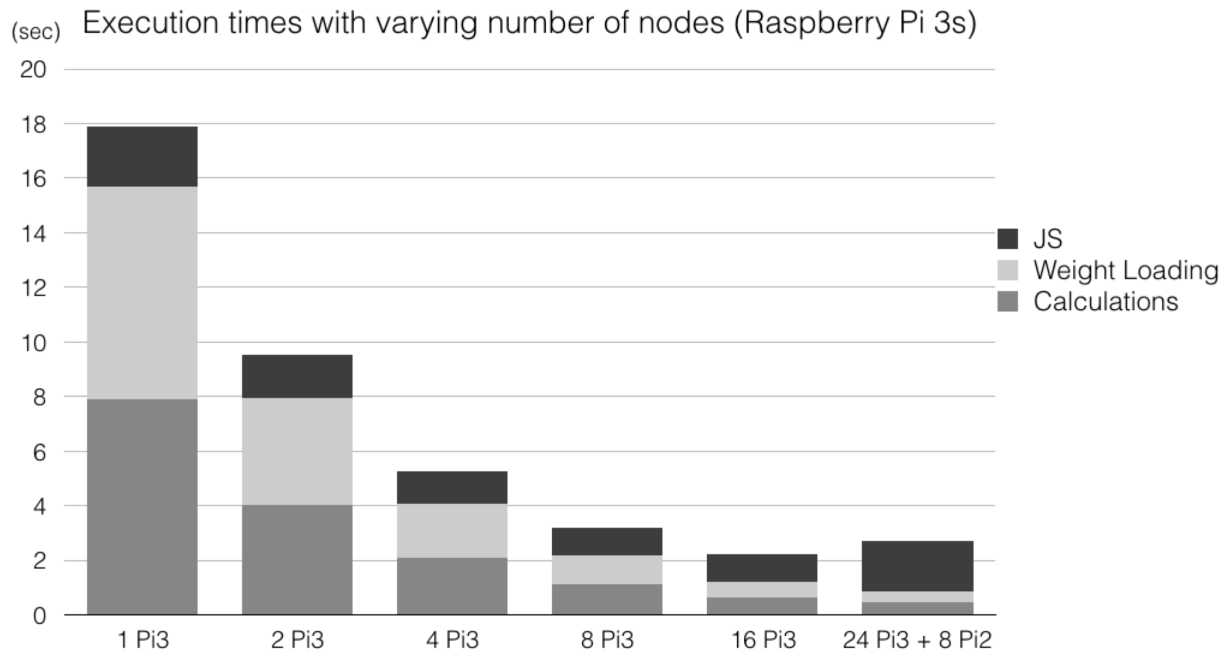

**Figure 6.3:** Mean execution time over 10 runs of AlexNet on increasing numbers of Raspberry Pi 3s. Shows relation between time spent in C environment (Weight Loading and Calculations) and JavaScript.

## 6.3 Work Distribution on Cameras

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 102.57 | 1.46 |
| Weight Loading | 33.60 | 0.76 |
| Orchestration (JS) | 9.16 | 1.83 |
| **Total** | **145.34** | **1.04** |

**Table 6.1:** Execution times for the different parts of the application when run on 8 Q1615 Cameras

figure 6.4 shows what each camera is executing and how much of the time each part is responsible for. Measurements are taken over the entire course of the forward pass, meaning all layers are at some point present on the camera. The forwarding cost is higher in the convolutional layers than in the fully connected ones, however the graph shows the work distribution on camera's CPU over the entirety of AlexNet. With the improved orchestration, the CPU is only idle 4% of the time. This happens when the varying nodes have differing execution times for the different layers. Each of the faster nodes will then have to wait for the slower ones.

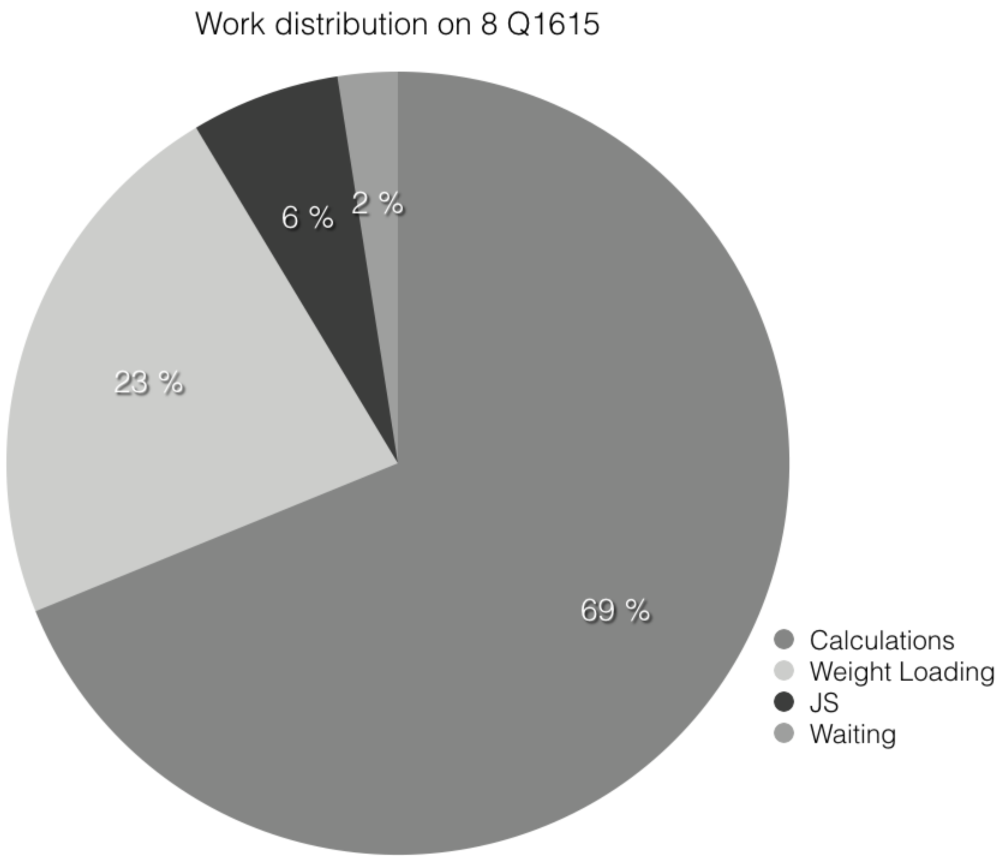

**Figure 6.4:** Pie chart showing what a camera is executing during a forward pass of AlexNet. On a camera, the computations in C are responsible for most of the execution time.

Note that the CPU utilization of 98% is only on one core. If the three other cores were to be included in the calculations, the utilization would be a mere 24.5% (98% of 25% is 24.5%), due to the fact that our application can only utilize one core. Performance could be improved by using all cores. See section 7.7.3 for discussion on this topic.

## 6.4 Work Distribution on Raspberry Pis

When the forward pass is instead run on 8 Raspberry Pi's, the work distribution appears slightly different, except for the idle time. This is shown in figure 6.5. Multiplications can be sped up quite a lot when performed on a CPU which supports SIMD. See section 7.6.1 for elaboration on this.

Forwarding is the the only aspect of the application that has been optimized in comparison to the scenario depicted in figure 6.4. Notable is the fact that the majority of the execution is now the JavaScript portion of the program. This is due to the decrease in execution time for the forwarding, relative to the JavaScript portion. In figure 6.2 we can see that the JavaScript portion scales the least when increasing the number of nodes. This suggests that there is less performance to gain by splitting the network, the stronger CPU's available. This topic is examined deeper in section 7.5.2.

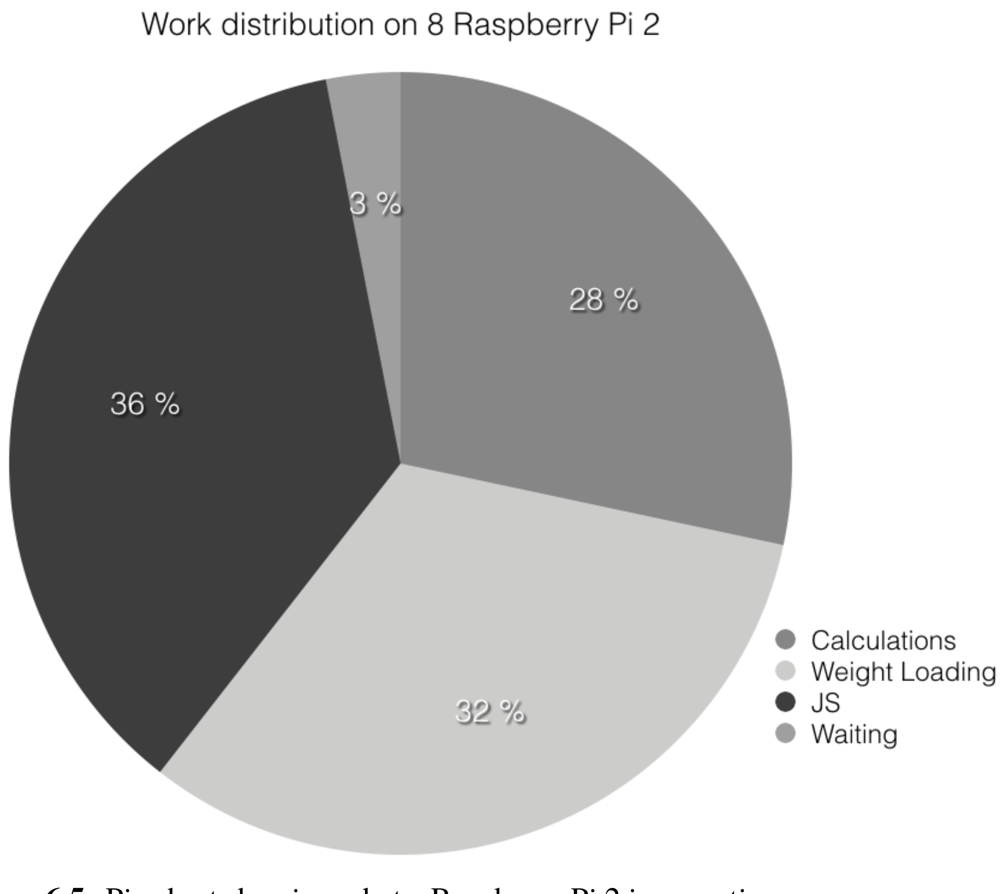

**Figure 6.5:** Pie chart showing what a Raspberry Pi 2 is executing during a forward pass of AlexNet. Notice that on the Raspberry Pi 2, the largest porting of the chart is made up of parsing and execution in JavaScript. When using Raspberry Pis, we were able to hardware accelerate most of the computations in C by using the Neon Hardware available in the CPU of the Raspberry Pi.

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 1.57 | 0.01 |
| Weight Loading | 1.78 | 0.02 |
| Orchestration (JS) | 2.01 | 0.06 |
| **Total** | **5.36** | **0.06** |

**Table 6.2:** Mean execution times for the different parts of the application over 10 runs on 8 Raspberry Pi 2's

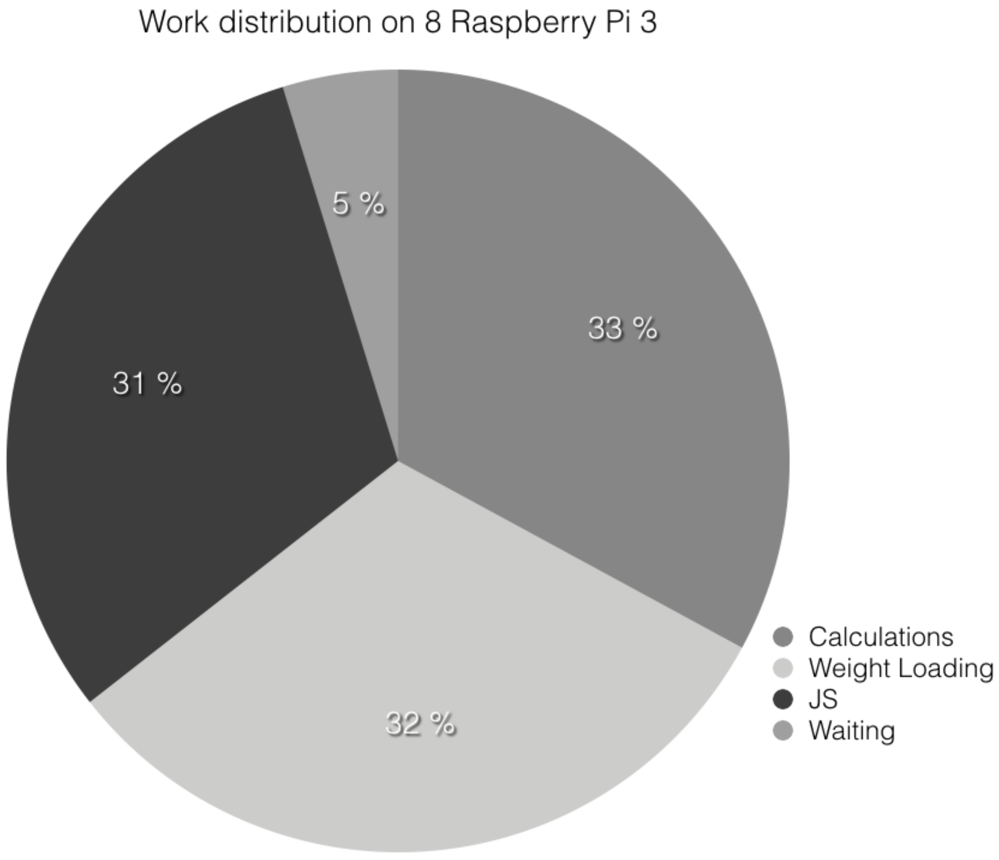

**Figure 6.6:** Pie chart showing what a Raspberry Pi 3 is executing during a forward pass of AlexNet. As with the Raspberry Pi 2, the largest porting of the chart is made up of parsing and execution in JavaScript.

In figure 6.6, the work distribution on a Raspberry Pi 3 is displayed. Not much is different when compared to figure 6.5. This is most likely because there are no new optimizations when running on Raspberry Pi 3. However, note that the time spent waiting is increased compared to before. It seems that the stronger CPU available, the more time is spent waiting for other nodes to complete. What might cause this effect is discussed in section 7.5.2.

# Chapter 7
# Discussion

We were able to successfully run a forward pass of AlexNet in a distributed environment of 8 Q1615 Network Cameras. However, using more cameras would have been ideal. Had there been more available nodes, the need for a node to change its role would have been significantly reduced or even removed. If the constant flushing and reloading of weights can be avoided, execution time will only include the actual forward pass and the communication cost. For nodes handling the fully connected layers, this is especially desirable, since they contain large amounts of parameters.

## 7.1 Desirable Number of Nodes

Of course, distributing the CNN is only desirable while there is still performance to be gained by doing so. The decrease in execution time is visible in figure 6.2. Also visible in the figure is the fact that not all aspects of the application speeds up as more nodes are available. Specifically, the JavaScript portion seems to not decrease more when there are more nodes than 16. Since we can not observe an actual trend of the scaling of the Javascript portion of the application, this part is considered as not parallelizable and all calculations below are performed according to this. However, the two parts which execute in C (Forward pass and Loading weights) do still halve with each doubling of nodes.

Using the charts displayed in figures 6.4 and 6.5, we can estimate how much of the application would benefit from increasing the number of nodes. In the case of using only cameras, at least 92% of the execution can benefit from more distribution. When using Raspberry Pi's, at least 63% of the application would benefit from more distribution. This is the case for both the Raspberry Pi 2 and 3, since they have roughly the same workload distribution. It is clear that in the case of using just cameras, there is more performance to be gained by distributing more. Amdahl's law states that we can achieve a maximum speedup of $\frac{1}{1-p}$ where $p$ is the portion of the application that can benefit from parallelism[21]. Using this formula, we get a theoretical maximum speedup of 16 times compared to running on eight cameras. This does however not include the cost of sending data over the network. With this speedup, we can expect the entire forward pass to be executed in 9 seconds when using the iterative reloading distribution.

A speedup of 16 is of course the theoretical max, and can only be achieved with an infinite number of processors.

This estimation is valid when scaling with Raspberry Pi 2. In the late stages of the project we have had access to several more Raspberry Pi 3. When evaluating the correctness of this estimation we can see that it shows a realistic scaling. Looking at appendix B showing execution times for Raspberry Pi's, we have for 32 Pi's a mean execution time of 2.69 seconds which is in line with the estimation of 2.85 seconds. Since the 24 Raspberry Pi 3's are bottlenecked by the eight Raspberry Pi 2's this equals the execution time of a system containing only Raspberry Pi 2's which is later elaborated in 7.5.1.

When comparing the real execution times to our expected ones, the real times seem to be a bit faster than the expected ones. This is due to the fact that the communication also scales somewhat with the number of nodes. This means that more of the application is parallelized than we accounted for. To determine how the communication scales with the number of nodes, in-depth analysis of how communication is carried out in Node.js would be required.

| Number of nodes | Speedup compared to 8 Q1615 | Expected execution time (s) |
|---|---|---|
| 16 | 1.9 | 77.2 |
| 32 | 3.4 | 43.2 |
| 64 | 5.6 | 26.2 |
| 128 | 8.2 | 17.6 |
| 1000 | 14.2 | 10.2 |
| $\infty$ | 16.0 | 9.1 |

**Table 7.1:** Estimated speedup and execution times for the iterative reloading distribution when increasing the number of cameras

Since there is a relatively large part of the application that does not fully benefit from further parallelism when running on Raspberry Pis, we gain quite little compared to when using only cameras by adding more nodes. Should the communication part be fully parallelizable as well, we could parallelize about 97% of the application, both on cameras and Raspberry Pis. The other 3% is the average idle time during a forward pass. With 97% of the application parallelizable, we can achieve an execution time of 22 seconds with 64 cameras and less than a second with 64 Raspberry Pis. The possibilities of parallelizing the communication part of the application is discussed in section 7.7.2.

| Number of nodes | Speedup compared to 8 Pi 2s | Expected execution time (s) |
|---|---|---|
| 16 | 1.5 | 3.69 |
| 32 | 1.9 | 2.85 |
| 64 | 2.2 | 2.42 |
| 128 | 2.4 | 2.22 |
| 1000 | 2.6 | 2.03 |
| $\infty$ | 2.7 | 2.01 |

**Table 7.2:** Estimated speedup and execution times for the iterative reloading distribution when increasing the number of Raspberry Pi 2s

Since the Raspberry Pi 3 has the same theoretical maximum speedup as the Raspberry Pi 2, we can expect an execution time of 1.16 seconds with a very large amount of Raspberry Pi 3s.

### 7.1.1 Scaling beyond eight nodes

When scaling beyond eight nodes, another problem occurs. The last fully connected layer consists of 1000 output neurons, each tied to one output class. Seeing as we want to distribute these neurons and we want an even divisor, we face a problem when scaling to 16 and 32 nodes (1000 is not divisible by 16 or 32). Our solution to this was to treat this layer as a special case, and simply use only eight nodes. The reason behind our choice was that since the layer only contains 1000 neurons, dividing it up further into very small parts would end up with an increased cost since the amount of work would be insignificant when compared to the added communication cost.

## 7.2 Distribution Impacts

Constant reloading of parameters is of course not desirable, as it just stalls the entire application. This is however only a problem when there are less nodes than what is needed to represent all parts of a CNN simultaneously. Should there be a fixed number of nodes, such that both distributions presented in this thesis can be implemented, the circumstances dictate which method is preferred.

The two different distributions are optimized for different things. The iterative reloading distribution is optimized for latency, while the pipeline distribution is optimized for throughput.

If the situation requires a single classification to be done as quickly as possible, the iterative reloading distribution performs slightly better. In this distribution, each of the layers are distributed to the fullest extent, meaning each layer is distributed over all the nodes. This way, there is more processing power since all the nodes are used for just one classification.

Should the situation instead require multiple classifications, but none of them are particularly time-critical, the pipeline-distribution should be considered. With it, all classifications are delayed the amount of time it takes to perform a single classifications, but they can be started independently of each other. After having completed the first, the next classifications will be separated by the time it takes to compute the heaviest layer. In this distribution there is less computational power per layer, because the nodes have fixed roles. However, classifications can be performed within less time of each other than compared to the iterative reloading distribution. In order to achieve optimal performance with this pipelined distribution scheme, a load-balancing approach should be used as described in 4.3, where the goal is to even out the execution time for each layer to be as uniform as possible.

Since the nodes never have to reload their weights, only the actual forward pass and the orchestration remains in terms of execution. This can be seen in figure 6.1 where the loading part would be removed after first setting up the neural network, since the weights would already be in memory. In this case, the fully connected layers amount to very little time, compared to the convolutional layers. Therefore, it is not so likely that each layer will be handled by just one node. Rather, the convolutional layers would be distributed since they are many times slower than the fully connected ones.

## 7.3 Splitting Layers Spatially

When considering the convolutional layers, we opted for a split in the convolutional layer's output depth, the filters. This implies that each machine needs all of the input in order to finish its computations. However another partitioning scheme would be possible. By letting each machine running the convolutional layer have all the weights, we could split the input spatially and distribute on the input instead of the filters. This would lead to a small increase in memory capacity needed, but would decrease the inter-layer communication. Instead of sending the complete input to all nodes we would need to send a part of the input plus a maximum of $\frac{(\text{kernelsize}-1)}{2}$ in pixels vertically and horizontally in order to be able to perform the convolution operation. The total communication would instead of $\text{n} \cdot \text{output}$ be upper bounded by $\text{output} + \frac{\text{kernelsize}-1}{2} \cdot 4\text{n} = \text{output} + 2\text{n}(\text{kernelsize} - 1)$. We give an upper bound since theoretically there is no need to send extra pixels horizontally and vertically for neurons along the edges.

## 7.4 Tradeoffs

A clear tradeoff is parallelism versus network overhead. High parallelism leads to more communication needed which is inherently expensive. Given a fast computational node, parallelizing would lead to increased overall execution time and therefore prove not to be worth it. This insight can also be applied when reducing the amount of work per node during distribution. Eventually the work per node will be disproportional to the orchestration time.

By investigating 6.2 we estimate that we will reach a saturation point of nodes given our current hardware and neural network model. In order to gain performance beyond the saturated number of nodes performance can be scaled further by pipelining.

The memory footprint when running the algorithm also comes with a tradeoff. The weights can be preloaded and kept in memory in order to remove time taken for the initialization of layers. This however requires $\frac{285}{\text{number of nodes}}$ MB memory in RAM at all times, which has proven not to be feasible for embedded systems with tight memory constraints such as Artpec-5. Instead, each layer can be initialized independently when needed and the memory be freed after use. This leads to a reduction in static memory to $\max_i \frac{\text{Memory of layer}_i}{\text{number of nodes}}$. In addition, memory for the input is also needed.

## 7.5 Bottlenecks

When analyzing our solution further we notice some bottlenecks occurring. Some of these can be minimized further by implementing the optimizations as suggested in section 7.7 where we discuss further optimization. With respect to our current hardware, some noticeable costs such as load from SD and multiplications using soft-floats cannot be solved without changing platforms.

Node is single-threaded, however this can be solved by either threading the C-library or clustering the Node.js instance. See section 7.7.3.

If the application were to be run on fast nodes using dedicated hardware or expensive GPU's the major bottleneck would lie in cost of communication.

### 7.5.1 Non-uniform computational nodes

As seen in figure 6.3 the result shows that when scaling to 32 nodes,eight Raspberry Pi 2's and 24 Raspberry Pi 3's, we get a slower total runtime when compared to using 16 Raspberry Pi 3's because of increased Javascript time, mainly spent as being idle on the Raspberry Pi 3's. We suspect this is a result of using different computational nodes without applying load-balancing. With our setup of 24 Raspberry Pi 3's and eight Raspberry Pi 2's it ultimately led to the newer generation Raspberry Pi 3's to be stalled in between each layer. However the time spent executing and loading is still linearly scaled with number of nodes.

### 7.5.2 Implications of Using Faster CPUs

While it might generally be a good idea to use the strongest CPU available at all times, it seems that the performance gain might not live up to what was expected. For instance, stronger CPUs can in general optimize parts of code to a great deal. In the case of our application, this means that our calculations can be optimized a lot more than the communication can. The result of this is that the parts of the program that normally benefit from parallelism are affected a lot less by the parallelism.

When using stronger CPUs, the time spent waiting seems to increase. We believe this is because while the computational power increases, the total communication cost remains constant. Hence the synchronization penalty increases, resulting in more frequent stalls.

## 7.6 Optimizations

Over the course of the thesis, some optimizations have been made in order to speed up execution. Some of the sections contain optimizations that can only be performed on the Raspberry Pi's, because they have looser memory constraints and more modern CPU's. All optimizations that are possible on the cameras are possible on the Raspberry Pi', but not necessarily the other way around.

| **Optimization** | **Exec time unoptimized** | **Exec time optimized** | **Decrease** |
|---|---|---|---|
| Synchronization (Pi 2) | 26.44 | 15.07 | 43% |
| SIMD (Pi 2) | 26.44 | 17.98 | 32% |
| Buffers (Pi 2) | 26.44 | 7.93 | 48% |
| **Total (Pi 2)** | **26.44** | **5.36** | **80%** |
| Synchronization (Q1615) | 474.12 | 274.99 | 42% |
| Buffers (Q1615) | 474.12 | 251.28 | 47% |
| **Total (Q1615)** | **474.12** | **145.36** | **69%** |

**Table 7.3:** The impacts of different optimizations on the cameras and Raspberry Pi 2.

### 7.6.1 Vectorization of multiplications

Because the Raspberry Pi's all have an ARM Cortex A7 processor, we can use its built-in NEON module to accelerate certain operations in the application. Specifically, we are interested in accelerating the matrix multiplications, since they make up the majority of the execution. The NEON module present in the A7, has support for VFP (Vector Floating Point), which allows us to use SIMD to great extent.

When using SIMD the 32-bit floats are placed two by two in 64-bit registers, allowing for parallel multiplications. Since there are no data dependencies between the multiplications, parallelizing the multiplications can be done without restrictions.

This optimization made all of the multiplication execute almost twice as fast, for a total decrease in execution time by 32%. However, this can not be performed on the cameras.

### 7.6.2 Communication with `Buffer`

Instead of sending data as strings, the application now uses buffers to encapsulate the data provided by the library. As mentioned in sections 3.5 and 5.1.1, this reduces the amount of data sent by 4.5 times. When using this approach, the execution time for the JavaScript part of the program also decreases with the number of nodes. We can not for certain say that this was not the case before optimizing communication, since we at the time did not have enough nodes to observe such a trend.

Parsing the data is now much faster, because the results provided by the library never have to be parsed by the JavaScript code. This means that the parsing no longer scales with the size of the data packets. Of course it still takes longer to receive larger amounts of data, but decoding the information does not.

## 7.7 Further possible optimizations

Here we suggest a few improvements to our final version of the application for running a forward pass of AlexNet. This section differs from future work in that it only states improvements for this particular problem. Some of the improvements suggested here are again only possible on the ARM Cortex-series CPU architecture, present on the Raspberry Pi's.

### 7.7.1 Half Precision Floating Point

SIMD is responsible for a large amount of the speedup in our application, due to the fact that there are so many multiplications that can be executed in parallel. However, we are limited by the fact that that the largest registers available to us are 64-bit registers. To get maximum utilization out of those registers, we would want our data to be represented with as few bits as possible.

Half precision floating point numbers are represented by just 16 bits, without a significant loss in accuracy (in the context of deep neural networks). With just 16 bits per number, we could fit four numbers in a 64-bit register, doubling the amount of parallel multiplications. In addition to this, the size of the data and weights would halve, letting us send more data and store more weights per node.

### 7.7.2 Complete C-application

While much of the calculations in the application can be parallelized, it is more difficult to parallelize the communication written in JavaScript. As mentioned in section 2.5, JavaScript is single threaded. Because it is single threaded, it is only ever able to utilize one core of the CPU. Should the application be implemented entirely in C, the application could utilize more of the CPU, if written correctly.

While it is possible to spawn threads in the library (see section 7.7.3), the communication is still limited to its one thread. Should the communication also be threaded, there is some room for optimizations. For example, loading weights could be done without delaying the parsing of incoming data, when a node needs to switch its role. This means that the application will not lock, as it does today when an FFI-call is being executed.

Another option would be to use a module called *cluster* in Node.js which spawns workers (child processes), by forking the node instance. This might have lead to issues when calling the FFI-library from different thread contexts. There could exist a scenario where duplicate copies of the parameters exists in memory, or access to the correct C environment might not be available to the correct layer.

### 7.7.3 Threading in the Library

In order to achieve better resource utilization, a good choice would be to use threading to handle the independent multiplications executed on each computational unit.

While it might not be possible to utilize all cores of the CPU with Node.js, it is still possible to use threads within the C-portion of the program. Since the multiplications performed are completely independant, they can be executed in parallel. To some extent this is already implemented via the use of SIMD, however we could further utilize the processor by distributing the multiplications to the individual cores and *then* utilize SIMD.

Furthermore, the multiple cores could contain different parameters, meaning different parts of a layer could be executed at the same time. Essentially, since we only use one core per CPU, we could reduce the number of nodes by a factor of four given that all the parameters would fit in memory.

## 7.8 Neural network architecture improvements for embedded systems

In this thesis we have based our research and results on the well studied AlexNet-model. However this model was not initially designed to run on embedded platforms. Recently a few studies have been made on how to optimize neural network models for embedded systems.

Fully connected layers take up a notorious amount of weights in comparison to a convolutional layer. In a network architecture such as AlexNet, the weights associated with the fully connected layers sum to roughly 95% of the total weights, while the convolutional layers have the remaining 5%. However, the converse holds for FLOPs during execution[2]. Recent research suggests different architectural approaches when applying deep learning on embedded systems.

Network-in-network, a paper released in 2013 by Min lin et al.[22], suggests a structure of nested networks. It also shows that fully connected layers can be removed in favor of average pooling. This structure can be seen in the famous inception module, used in GoogleNet. Recall that fully connected layers are associated with 95% of the weights, this shows that one can choose networks without fully connected layers and hence decrease the memory requirements of the model.

K. He and J. Sun investigated the accuracy of CNN's under constrained time cost[23]. They reached the conclusion that late downsampling leads to higher classification accuracy. This result speaks for not using a stride parameter larger than 1 in the early layers and instead opt for reducing the horizontal and vertical span of the activation maps later on in the network.

C. Szegedy and V. Vanhoucke et al.[24], pioneers of the inception architecture, did some rethinking in how similar receptive fields can be created by using asymmetric convolutional kernels. They applied spatial factorizing of larger convolutional operations into smaller asymmetric ones. Basically a NxN filter can be replaced by applying a Nx1 plus

1xN in conjunction which will lead to a weight reduction of $\frac{N}{2}$ but still generate a similar receptive field. The research in the article suggests using this method only for medium grid sizes and not in the early layers where horizontal and vertical dependency still is important.

Given a complete network model, further optimizations can be done in order to compress the model which is especially interesting for embedded systems due to its constrained nature. The first approaches to model compression has been based around pruning of weights under a certain threshold which showed promise in the 90's[25].
Later on network pruning was combined with quantization of weights and huffman encoding which resulted in *Deep compression* as coined by Song han et al[26]. This method of model compression showcased great results on existing CNN's, reaching a compression of 35x on AlexNet. The original network with ~250MB of parameters could now be pruned and compressed to ~7MB. Another nice effect of the compression was that since the weights are quantizied together it leads to weight sharing, which means that one weight represents several previous weights and a speedup is gained due to better cache utilization.

In a paper by researchers from Berkeley and Stanford [27], they provide a different neural net architecture that reaches the same accuracy as AlexNet. Their model is called SqueezeNet, in where they introduce a new buildingblock called fire-module. The fire-module is based on the network-in-network method introduced in [22], and makes use of three convolutional layers. First a "squeezelayer" of 1x1 convolutional filters and eventually an "expansionlayer" based on 1x1 along with 3x3 convolutional filters. Some conclusions from this paper is that smaller deep neural network architectures offer several key advantages when deployed. Less communication required when distributing, faster model update and better feasibility in FPGA or other hardware deployment. They finish the paper by compressing the model using deep compression and saying that small network models are also amenable to compression without losing accuracy. This shows promise for smaller models being compressed, fit for embedded systems.

# Chapter 8
# Conclusions

This chapter brings up our conclusions and insights that we have come across during our thesis. All partitioning schemes for a neural network comes with a communication overhead due to the nature of operations. Our partitioning scheme sees a linear increase w.r.t number of nodes.

Calling FFI-functions comes with an additional overhead which we measured to be around 5ms. While being small, this cost adds up when being heavily used. Functions called via FFI should be few and provide comprehensive functionality.

Node.js is event-driven and single threaded. Calling FFI-functions inside node will block the event-handling until completion. This calls for a synchronization between computational units for each layer, since we want the blocking execution to occur simultaneously. Otherwise a node could start the blocking execution before sending its data to rest of the nodes leading to a stall.

Benefits of our approach includes that anything being able to run Node.js and C, will be able to run our application. However when mixing machines of different endianness we suspect that unexpected results may occur.

The memory footprint will also be reduced by using a distribution scheme, which proved to be of great significance when running on constrained devices such as the Axis Q1615 camera. The reduction in memory requirements is a prerequisite to even be able to run a neural network on this camera. Since the total available RAM was in the order of ~40MB.

When optimizing communication, especially with nodeJS and C combined we noticed that sending data without explicitly specifying type in Javascript allows for faster communication since it is represented as a buffer-object. There is no overhead of transforming the data to an intermediate representation in between the execution in C. This is doable since we have made sure to have all of our neural network logic separated and kept in C.

The change that optimized our solution the most was using NEON SIMD (applicable for raspberry PIs). When applying deep learning on embedded systems hardware acceleration is almost a prerequisite in order to keep the time for execution reasonable.

Another conclusion is the impact of uneven computational units. As can be seen in the rightmost column of figure 6.3, scaling with different nodes amounts to a bottleneck effect by the slower nodes. This leads to a final execution time similar to that of 32 Raspberry Pi 2's. Therefore when scaling with uneven nodes, the total execution time will amount to a linear scaling of the slowest node because of synchronization.

Conclusively we would like to end by saying that it is possible to apply deep learning on weaker devices using distribution. However an important fact that should not be neglected is that communication overhead is expensive and reading from any memory source slower than RAM is expensive. To reduce time spent communicating and loading weights more neural network architectures need to be investigated as described in 7.8 and 9. All in all, Optimal applications of deep learning, such as real time classification, in embedded environments require dedicated hardware as well as a node need to single-handedly be able to run the network as a whole in order to remove communication overhead completely.

# Chapter 9
# Future work

As we have seen deploying deep neural networks on embedded systems requires a lot of computation. The best possible way of dealing with this problem is to have dedicated hardware, like the tensor processing unit google uses[28].

Currently, our solution does not handle dynamic load balancing. In the case of non-uniform computational nodes, analysis and prepartitioning of the network is still needed. It would be favorable to dynamically let nodes with greater computational capabilities handle larger portions of the CNN. Our distributed solution does not handle self-healing in case of node failure, this is also left as future work.

Another problem area that opens up when using nodes without enough memory is the time taken to load data from disk. Currently the Q1615 camera as well as both generation 2 and 3 of the raspberry PI's use microSD cards. A possible approach for investigation would be to measure if there exists a speedup to be gained when sending weights directly over network instead of loading them from the microSD cards.

In order for a distributed solution to be feasible the computational nodes available would need to be weak, unfit for a prediction to be done standalone. A non-negligible communication cost will always exist, therefore our solution cannot be used in time-critical applications such as real-time identification. Future work would also investigate a distributed approach on models fit for embedded systems, such as pruned networks and more ingenious network architectures as described 7.8.

More use cases involve the possibility of using specialized and/or parallel regressor heads. By regressor head we simply mean the bundle of last fully connected layers that has the responsibility of decision making. By retraining the last fully connected layers with transfer learning, as explained in 2.2.7, one could therefore reuse the computations made in the convolutional layers, since neuron-activations early on can be seen as more general features. A use-case for this could be that cameras mounted in certain settings might need different decision making in order to achieve high precision. Also one could experiment whether it is beneficiary to run different regressor heads in parallel and merge the result later on in order to get higher precision, similar to how an ensemble of networks work.

# Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. 2012.

[2] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. April 29, 2014.

[3] Matthew D. Zeiler, Rob Fergus. Visualizing and Understanding Convolutional Networks. 2014.

[4] Andrej Karpathy. Neural Networks. `http://cs231n.github.io/neural-networks-1/`.

[5] Jonathan Long, Ning Zhang, Trevor Darrell. Do Convnets Learn Correspondence? 2014.

[6] River trail project. . `http://intellabs.github.io/RiverTrail/tutorial/`.

[7] Andrej Karpathy. Convolutional Neural Networks (CNNs / ConvNets). `http://cs231n.github.io/convolutional-networks/`.

[8] A. Krizhevsky. Carnegie Mellon University Deep Learning, 2015. `http://deeplearning.cs.cmu.edu/slides.2015/24.krizhevsky.pdf`.

[9] Yoshua Bengio, Paolo Frasconi, Jurgen Schmidhuber. Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. 2001.

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

[11] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015.

[12] Stanford Vision Lab. Imagenet competition. http://www.image-net.org/challenges/LSVRC/.

[13] Joseph Redmon. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[14] Node.js Contributors. Node.js. https://nodejs.org/en/about/.

[15] Nathan Rajlich, Richard "Rick" W. Branson, Gabor Mezo. Node-ffi. https://github.com/node-ffi/node-ffi.

[16] Axis Communications AB. Axis Q1615 Network Camera. http://www.axis.com/nz/en/products/axis-q1615.

[17] David Goldberg. What Every Computer Scientist Should Know about Floating-Point Arithmetic. 1991.

[18] W. Kahan. Lecture Notes on the Status of IEEE 754. 1997.

[19] Rajesh Mansharamani. Little's Law Bottleneck Law, 2011. http://www.softwareperformanceengineering.com/uploads/1/2/6/6/12667295/littleslawandbottlenecklaw.pdf.

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. 2015.

[21] David A. Patterson, John L. Hennessy. *Computer Organization and Design*. 2009.

[22] Min Lin, Qiang Chen, Shuicheng Yanr. Network In Network. 2014.

[23] Kaiming He, Jian Sun. Convolutional Neural Networks at Constrained Time Cost. 2014.

[24] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. 2015.

[25] G. Castellano, A. M. Fanelli, and M. Pelillo. An iterative pruning algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, 8(3):519–531, May 1997.

[26] Song Han, Huizi Mao, William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. 2015.

[27] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. 2016.

[28] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip, 2016. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

# Appendices

# Appendix A
# Division of work

In this thesis much of the work was done together. Below we list the individual responsibilities for both students.

**Axel's main responsibilities were:**

- Creating the library based on DarkNet
- Implementing the weight parser
- Implementing functions in predictor.c
- Implementing functionality to automatically split the network depending on the number of nodes
- Initial communication using buffers
- Synchronization

**Anton's main responsibilities were:**

- Splitting CNN into separate layers
- Splitting convolutional and fully connected layers
- Combining JavaScript and C via FFI
- Implementing initial connection scheme with MDNS
- Initial reloading of weights
- Optimizations using NEON

# Appendix B

# Execution on Many Raspberry Pis

Below we have tables of execution times with varying number of nodes. All measurements are averaged over 10 runs with given standard deviation.

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 10.90 | 0.03 |
| Weight Loading | 13.17 | 0.06 |
| Orchestration (JS) | 3.76 | 0.02 |
| **Total** | **27.82** | **0.09** |

**Table B.1:** Execution times for the different parts of the application when run on 1 Raspberry Pi 2

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 7.91 | 0.03 |
| Weight Loading | 7.79 | 0.01 |
| Orchestration (JS) | 2.184 | 0.02 |
| **Total** | **17.88** | **0.04** |

**Table B.2:** Execution times for the different parts of the application when run on 1 Raspberry Pi 3

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 5.60 | 0.01 |
| Weight Loading | 6.67 | 0.02 |
| Orchestration (JS) | 2.90 | 0.11 |
| **Total** | **15.17** | **0.10** |

**Table B.3:** Execution times for the different parts of the application when run on 2 Raspberry Pi 2's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 4.02 | 0.02 |
| Weight Loading | 3.93 | 0.01 |
| Orchestration (JS) | 1.55 | 0.03 |
| **Total** | **9.51** | **0.03** |

**Table B.4:** Execution times for the different parts of the application when run on 2 Raspberry Pi 3's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 2.92 | 0.01 |
| Weight Loading | 3.40 | 0.02 |
| Orchestration (JS) | 2.28 | 0.03 |
| **Total** | **8.60** | **0.02** |

**Table B.5:** Execution times for the different parts of the application when run on 4 Raspberry Pi 2's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 2.08 | 0.02 |
| Weight Loading | 2.00 | 0.01 |
| Orchestration (JS) | 1.16 | 0.05 |
| **Total** | **5.24** | **0.05** |

**Table B.6:** Execution times for the different parts of the application when run on 4 Raspberry Pi 3's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 1.57 | 0.01 |
| Weight Loading | 1.78 | 0.02 |
| Orchestration (JS) | 2.01 | 0.06 |
| **Total** | **5.36** | **0.06** |

**Table B.7:** Execution times for the different parts of the application when run on 8 Raspberry Pi 2's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 1.11 | 0.01 |
| Weight Loading | 1.06 | 0.02 |
| Orchestration (JS) | 1.00 | 0.04 |
| **Total** | **3.20** | **0.06** |

**Table B.8:** Execution times for the different parts of the application when run on 8 Raspberry Pi 3's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 0.63 | 0.03 |
| Weight Loading | 0.59 | 0.02 |
| Orchestration (JS) | 1.01 | 0.06 |
| **Total** | **2.24** | **0.04** |

**Table B.9:** Execution times for the different parts of the application when run on 16 Raspberry Pi 3's

| | Mean Execution Time (s) | Standard Deviation |
|---|---|---|
| Forward Pass | 0.44 | 0.08 |
| Weight Loading | 0.40 | 0.11 |
| Orchestration (JS) | 1.85 | 0.21 |
| **Total** | **2.69** | **0.22** |

**Table B.10:** Execution times for the different parts of the application when run on 32 Raspberry Pi's, 24 3's and eight 2's

| | IDLE |
|---|---|
| 8 Q1615 Cameras | 2.54 % |
| 1 Raspberry Pi 2 | 1.01 % |
| 1 Raspberry Pi 3 | 0.78 % |
| 2 Raspberry Pi 2's | 2.18 % |
| 2 Raspberry Pi 3's | 0.95 % |
| 4 Raspberry Pi 2's | 0.70 % |
| 4 Raspberry Pi 3's | 3.24 % |
| 8 Raspberry Pi 2's | 3.17 % |
| 8 Raspberry Pi 3's | 5.00 % |
| 16 Raspberry Pi 3's | 5.81 % |
| 32 Raspberry Pi (24 3's + 8 2's) | 19.67 % |

**Table B.11:** Percentage of how much of the execution is spent being idle

**EXAMENSARBETE** Distributing a neural network on embedded systems
**STUDENT** Axel Ahlbeck, Anton Jakobsson
**HANDLEDARE** Patrik Persson (LTH), Mikael Lindberg (Axis), Niclas Danielsson (Axis)
**EXAMINATOR** Jörn Janneck (LTH)

# Kameror som tänker tillsammans

POPULÄRVETENSKAPLIG SAMMANFATTNING **Axel Ahlbeck, Anton Jakobsson**

Datorseende blir allt bättre, men med dess precision tillkommer en beräkningskostnad. Detta arbete undersöker hur man kan använda svagare men fler datorer, exempelvis kameror, för att implementera dagens kraftfulla algoritmer.

Dagens teknik tillåter datorer att utföra alltmer komplicerade uppgifter, som tidigare krävde en människa. Detta kallas för maskininlärning, och kortfattat så handlar det om att lära en dator att känna igen mönster av olika slag. En algoritm som ofta används är neurala nätverk, vilket är produkten av ett försök att återskapa människans tankesätt i en dator. Självklart så kommer inte Terminator att knacka på imorgon, utan det är fortfarande en relativt naiv algoritm. Användningsområdena hos neurala nätverk är stora, och de kan lösa många problem - så länge det finns rätt och fel att lära sig från.

Vi har utforskat möjligheterna för att få in stöd för detta direkt i dagens och gårdagens kameror, som är små datorer som i sin ensamhet är okapabla att köra neurala nätverk.

En ensam kamera är såklart inte lika snabb som en vanlig dator, och kan därför inte bestämma vad en bild föreställer lika snabbt. Den kan då istället be om hjälp från andra kameror som är lediga och inte tittar på något speciellt. Om de olika kamerorna tittar på olika mindre detaljer i bilden kan de tillsammans snabbare komma fram till vad bilden egentligen föreställer. Varje kamera behöver då tänka mindre och kan koncentera sig på en enda sak. Till exempel kan en kamera leta efter däck, en annan efter dörrar och en tredje efter fönster. Finns alla dessa i samma bild kan bilden föreställa en bil, men den kan också föreställa en hög med däck utanför ett hus.

Flera små datorer tänker i regel lika snabbt som en lite större, om varje dator kan vara ansvarig för en tillräckligt stor del av uppgiften. Om varje kamera ansvarar för en väldigt liten del av beräkningarna kommer kommunikationskostnaden bli för stor.

Att få flera datorer att samarbeta på ett smidigt sätt är en långt ifrån trivial uppgift, med många potentiella svårigheter. Under vårt examensarbete har vi utvecklat en modell för att applicera neurala nätverk i ett system av många svagare datorer.

Våra resultat visar på att ju fler kameror man har desto snabbare kan man de ta reda på vad en bild föreställer, fram tills den punkt då det blir en för stor kommunkations- och synkroniseringskostnad.

Slutligen så innebär detta att en människa inte längre kommer behöva titta genom kamerans ögon, utan kameran ser själv vad som finns i bilden. I den närmre framtiden så kommer det manuella arbetet som nuförtiden görs av människor, exempelvis att gå igenom övervakningsvideor, bli automatiserat och göras av kamerorna själva.