Increasing Spacecraft Autonomy through Embedded Neural Networks for Semantic Image Analysis

Andreas Schartel

Space Engineering, masters level (120 credits)
2017

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Master Thesis

Increasing Spacecraft Autonomy through Embedded Neural Networks for Semantic Image Analysis

submitted by
Andreas Schartel

Enrolment No.: 1807554
Submission Date: 15.11.2016
Supervisor (JMUW): Prof. Dr.-Ing. Hakan Kayal
Supervisor (LTU): Dr. Anita Enmark
This page is intentionally left blank.
I’d like to thank my supervisors Prof. Dr.-Ing. Hakan Kayal from Würzburg University as well as Dr. Anita Enmark from Luleå Tekniska Universitet for their great assistance and advice. They both gave me the freedom to pursue own ideas but also helped me out with valuable support when necessary. Furthermore, I’d like to thank Mr. Oleksii Balagurin for his great support. In his role as supervisor for my part time job here at Würzburg University, he did not only give me the freedom and flexibility to reconcile my master thesis with the job but also helped me out with new ideas when I was stuck. Thank you all for making the work on my master thesis an enjoyable experience.

Andreas Schartel
This page is intentionally left blank.
In the scope of this thesis, a possible usage of embedded artificial neural networks for on-board image analysis is investigated. After an introduction which imparts basic knowledge about artificial neural network and autonomy in spaceflight, a possible system design is elaborated based on previously defined reference scenarios. The reference scenarios are based on two projects that are currently under development at the professorship of Space Technology at the University of Würzburg: ASMET and SONATE. As base for the neural network architecture, a convolutional neural network called SqueezeNet was chosen, since it was developed for similar input data and performs sufficiently well. In addition, the SqueezeNet architecture requires extremely little memory for the trained-in model compared to other architectures which makes in-orbit updates of the model feasible. The system concept in this thesis is designed for offline learning, i.e. the training phase will be done on an ordinary computer. The resulting, trained-in model is then transferred to the embedded system. On the embedded side, a FPGA-based approach was chosen since FPGAs allow to parallelize the neural network execution and therefore accelerate it significantly. Even though not all components of the designed concept could be implemented in the scope of this thesis, all key elements were implemented and tested, either on real hardware or by using testbenches. Especially the tests conducted with the convolution unit of the embedded system went well and allowed to make a quite promising assessment of the expected execution speed. In addition, a tool with graphical user interface was developed to guide a potential user of the system through the steps of training-in and setting up the system. For the training process, a neural network framework called Caffe was used within this tool. In summary, this thesis provides as intended a profound starting point for further research on artificial neural network for space applications at the professorship of Space Technology of the University of Würzburg.
This page is intentionally left blank.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Single-Layer Perceptron</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Multi-Layer Perceptron</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of how a Convolution Layer works</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustration of a 2 × 2 Pooling Layer</td>
<td>24</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of the Layers and Dimensionality within a Convolutional Neural Network</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Overall Workflow Structure</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Screenshot of the Create Database tab in the Offline Learning Toolchain</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Screenshot of the Define and Train Network tab in the Offline Learning Toolchain</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Screenshot of the Test Neural Network tab in the Offline Learning Toolchain</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>TerasIC DE2-115 Development Board with Altera Cyclone IV FPGA</td>
<td>42</td>
</tr>
<tr>
<td>4.5</td>
<td>System Overview</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Illustration of the Pseudo-Code in Listing 4.1</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>Illustration of the Convolver Unit</td>
<td>48</td>
</tr>
<tr>
<td>4.8</td>
<td>Illustration of the Convolver Cluster</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Current Status of Work</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Illustration of Movement to Image Mapping using a Sequence of Images</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Screenshot with Annotations of the Waveforms of One of the Convolver Cluster Testbenches in ModelSim</td>
<td>55</td>
</tr>
<tr>
<td>C.1</td>
<td>Enlarged Screenshot of the Waveforms of One of the Convolver Cluster Testbenches in ModelSim</td>
<td>84</td>
</tr>
</tbody>
</table>
4.1 Data Sharing Relations (Source: [1]). . . . . . . . . . . . . . . . . . . . . 46
5.1 Runtime Estimation Of SqueezeNet. . . . . . . . . . . . . . . . . . . . . 58
A.1 User Requirements. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
A.2 System Requirements - Requirements on the Embedded System. . . . . 67
A.3 System Requirements - Requirements on the Neural Network Architecture. 67
A.4 Other Requirements. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
B.1 SqueezeNet Layers and Resource Utilization. . . . . . . . . . . . . . . . . 73
LISTINGS

4.1 Convolution Pseudo Code (Source: [1]) . . . . . . . . . . . . . . . . . . . . 44
4.2 Proposed Convolution Architecture by Zhang et al. (Source: [1]) . . . . 47
B.1 SqueezeNet Network Definition . . . . . . . . . . . . . . . . . . . . . . . 74
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>ASE</td>
<td>Autonomous Sciencecraft Experiment</td>
</tr>
<tr>
<td>BVLC</td>
<td>Berkeley Vision and Learning Center</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>Concat</td>
<td>Concatenation (Layer)</td>
</tr>
<tr>
<td>DLR</td>
<td>German Aerospace Center</td>
</tr>
<tr>
<td>EO-1</td>
<td>Earth Observing-1 Mission</td>
</tr>
<tr>
<td>FC</td>
<td>Fully-Connected (Layer)</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out Datastructure</td>
</tr>
<tr>
<td>FM</td>
<td>Feature Map</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>ILSVRC</td>
<td>ImageNet Large Scale Visual Recognition Challenge</td>
</tr>
<tr>
<td>IP Core</td>
<td>Intellectual Property Core</td>
</tr>
<tr>
<td>MLP</td>
<td>Multi-Layer Perceptron</td>
</tr>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>NN</td>
<td>Neural Network</td>
</tr>
<tr>
<td>RBFN</td>
<td>Radial Basis Function Network</td>
</tr>
<tr>
<td>RCNN</td>
<td>Recurrent Convolutional Neural Network</td>
</tr>
<tr>
<td>ReLu</td>
<td>Rectified Linear Unit (Layer)</td>
</tr>
<tr>
<td>SCL</td>
<td>Spacecraft Command Language</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous Dynamic Random Access Memory</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
</tbody>
</table>
Autonomy in space flight is up to now a rather underexplored topic. From an algorithmic point of view, present day spacecraft are quite simple machines: They execute one command after another, either immediately or time tagged. This has two main reasons: First of all, space flight is conservative. Due to the high costs of a mission, one does not want to risk a failure of the vehicle and thus resorts to flight proven and well-known components and concepts. Furthermore, space environment is hostile to electronics, which is why many components used on earth are not suitable for space application. As a result, present day spacecraft’s on-board computers are relatively weak compared to ordinary computers or even small machines like smartphones. This lack of computational power is in turn another reason why many sophisticated algorithms and techniques used in Earth-based applications have not been adapted for usage in space, even though they have an enormous potential.

One of those technologies are artificial neural networks, which are one of the most promising approaches for machine learning, a sub-branch of artificial intelligence. Artificial neural networks are inspired by the way the human brain works. They consist of many small, interconnected nodes (neurons), each having an input and an output with a certain relation defined by a mathematical function. The way neurons are interconnected and the mathematical relation between input and output differs between different flavors of artificial neural networks. What they all have in common is that they can be trained, i.e. the parameters of the mathematical function of each neuron can be automatically adapted by showing sample inputs to the neural network together with the desired outputs. A well-trained neural network is then able to provide the right output to a given input, even if this input is completely new, i.e. the neural network has never seen it
introduction during training. This works especially well for tasks where there is a known relation between input and expected output, but creating a rule for this relation in a closed form is hard or even impossible. An excellent example is the classification of images or semantic understanding of scenes on pictures. Modern neural networks are for instance capable of recognizing objects on pictures, e.g. whether the picture shows a dog, cat, car or plane no matter what breed of cat or dog is shown or under which angle and illumination conditions the picture was taken. This is something that is nearly impossible with traditional approaches since it is almost impossible to define rules on how to recognize a cat and how to distinguish a cat from a dog.

This kind of classification is also very interesting for space applications, for example to autonomously detect and delete images that are unusable due to cloud occlusion of the target area in order to safe memory and bandwidth or to detect and store transient, non-schedulable events like unstudied weather phenomena, meteors or geyers. It is also possible to use neural networks on other input data than images. It has been shown that a (ground-based) neural network is capable of identifying possible exo-planet candidates in the data of NASA’s Kepler spacecraft with a similar or even better reliability than humans (cf. [2]). If this data could be analyzed on board the spacecraft rather than having to download all the raw data to Earth, the necessary memory and communication infrastructure could be reduced significantly, enabling a new type of smaller and cheaper scientific space probes.

Even though image analysis through artificial neural networks has recently gained lots of attention, transferring those technologies to spaceflight has up to now been understudied. On the other hand, reducing costs of technology demonstration missions through pico and nano-satellites as well as recent technology improvements like high-performance low-power FPGAs make such embedded artificial neural networks for space applications feasible for the first time.

As a result, this work investigates possible implementations for embedded artificial neural networks which are suitable for usage aboard a small spacecraft in more detail. After a survey about existing approaches, frameworks and hardware, the requirements for an embedded neural network on small spacecraft are determined. With these requirements, a possible design for a neuro-processor is elaborated and key elements of this system are implemented to demonstrate their feasibility.
Chapter 2

The Basics

2.1 Artificial Neural Networks and Autonomy in Space-flight

The idea of artificial neural networks is as old as the computer. In 1943, McCulloch and Pitts published a paper describing how they believed neural networks work \[3\]. Their model, which was based on simple electrical circuits is still a basis for many modern models. Later on, the development of artificial neural networks drifted in two different directions: While biologists wanted to stick as close as possible to real neural networks in order to study their behavior, computer scientists and engineers simplified and optimized their models for usage in real world applications rather than to mind correctness regarding the biological example. After an initial boom, the excitement about artificial neural networks bottomed out in the mid 70s. A first renaissance occurred in 1985 when John Hopfield published a solution to the “traveling salesman problem” using a Hopfield-Network, a special form of an artificial neural network (cf. \[4\]). In 2012, artificial neural networks gained a lot of public attention through a Google X Lab experiment on unsupervised learning. Unsupervised learning networks are a special branch of artificial neural networks which – in contrast to ordinary supervised learning techniques – do not need any example input/output datasets for training but learn during operation on unlabeled input data by automatically forming output classes based on similarities. The large-scale neural network developed by Google was trained on unlabeled video data fetched from Google’s Youtube platform and learned in this way how to distinguish objects on its own. After one week of training, one neuron for instance reacted strongly to images of cats, which means the network learned what cats look like (cf. \[5\]).
It is important to note that there are lots of different flavors of neural networks with completely different structures and applications. While unsupervised learning techniques like the one used in Google’s X Lab experiment have a comparatively low technology readiness level and are almost exclusively used in research, there are also artificial neural networks that are even today reliable enough to be used in end-consumer products. For instance, almost all modern speech recognition systems like Apple’s Siri or Google Now use some form of artificial neural networks. Other popular examples include handwritten text recognition or autonomous labeling of images like in Google’s ”Photos” app. The latter two examples both have in common that they operate on images as input data. This is one of the strong points of artificial neural networks since they allow machines to abstract and detect similarities in pictures in a similar way humans do.

This ability to abstract and detect patterns is also a feature with great potential to increase autonomy in spaceflight. Autonomy in spaceflight is a rather new topic. As indicated before, spacecraft are usually very simple state machines up to now, which execute one command after another. All data evaluation and planning of upcoming actions is done on ground. While this works acceptably well for single spacecraft which are near to Earth and have a schedulable mission, one needs some kind of autonomy when the targets to observe are short-termed and unschedulable, the distance between Earth and spacecraft is too large to allow fast enough reaction (e.g. deep space missions) or the number of spacecraft is too large to be controlled by a reasonable amount of ground operators by hand like the planned OneWeb mega-constellation with about 650 satellites. Other reasons which militate for more autonomy could be reducing operational costs due to less required human interaction or less required on-board memory and bandwidth due to intelligent pre-processing and selection of data to store and send. Those factors are especially relevant for the advancing pico- and nano-satellite market, where low costs and compact hardware are crucial. Nevertheless, due to the high overall costs of a space mission and the potential risk of a loss of the spacecraft in case of an anomaly, autonomous and partly autonomous systems in spacecraft are up to now very rare and have only been studied in few missions.

One of the biggest missions studying autonomy in spaceflight is the Earth Observing-1 (EO-1) mission by NASA. Besides Earth observation, its purpose is the verification of different new technologies regarding instruments, spacecraft bus and software. One of
these experiments is the Autonomous Sciencecraft Experiment (ASE), which consists of an onboard image processor to autonomously detect "interesting" features, changes to previous observations and clouds. Furthermore, it makes use of a dedicated language called Spacecraft Command Language (SCL) to enable event-driven processing and low-level autonomy using the CASPER system, a planning and scheduling software that autonomously creates a mission plan according to the scientific observations made [6]. Since mission planning is usually not done by machine learning techniques, this part of the ASE system is not evaluated further in this thesis. However, the onboard image processor used in ASE has the same objective as an artificial neural network designed for image analysis and could therefore be replaced by the projected neuro-processor. The classifiers that form the basic framework of the ASE image processor are based on so-called Support Vector Machines (SVMs) [6]. SVMs belong to the class of supervised machine learning techniques and thus compete directly with artificial neural network architectures for supervised learning. They have, by definition, a shallow architecture, whereas artificial neural networks can have both shallow and deep architectures. In contrast to shallow architectures, a deep architecture consists of several stacked computational layers. In [7] the authors state that "Such [shallow] architectures are inefficient for representing complex, highly-varying functions", which means that SVMs are well suited for plain pattern matching but get inefficient or even fail when it comes to higher abstraction.

A mission with more sophisticated image processing techniques is the BIRD satellite developed by the German Aerospace Center DLR. Besides its primary goal to detect and monitor fires on Earth, the satellite served as technology demonstrator to test lots of different new technologies. One of those technologies was the autonomous on-board classification experiment that used the Ni1000 neuro-processor chip from Nestor Inc. in order to generate "thematic maps" [8]. The Nestor Ni1000 is a chip that was explicitly designed for classification and pattern recognition using an artificial neural network. According to Nestor Inc., it can achieve about 40,000 256-dimensional classifications (256 inputs with 5 bit each) per second [9]. This corresponds to the classification rate of a few hundred handwritten characters per second as a demonstration by Nestor Inc. showed. Internally, the chip uses a so-called radial basis function network (RBFN), a neural network type, that is nowadays not used very often anymore. In BIRD it was used to classify the captured scene into six different classes: hot spots (fire), water, bare soil, cloud, urbanization areas and "undefined" in case of an unclear decision. The chip was used to generate a "geocoded thematic map" which was then transmitted to the ground [10].
This on-board processing experiment was conducted to resolve two emerging demands: to reduce the data rate and provide immediate results without the time delay of data processing and distribution. Even though the experiment was successful and demonstrated the feasibility and usefulness of artificial neural networks onboard spacecraft, there has been no follow up project up to now. A reason for that might be, that the Ni1000 was commercially not as successful as expected and disappeared from the market without a successor. The demand for neuro-processors was just not high enough to justify the high development costs of a dedicated chip, an issue that could nowadays be solved with the increasing performance of FPGAs.

Another mission with a payload similar to ASE but at a completely different scale is currently under development at the professorship of Space Technology of University Würzburg. The SONATE satellite, a 3U cubesat that will be launched in 2019, will carry a tailored version of the ASAP system, an autonomous sensor and planning system which was also developed at the University of Würzburg [11]. Similar to ASE, ASAP consists of an optical sensor platform with an integrated image processing capability to identify predefined events like meteors, special weather phenomena or geysers [12]. The second part of ASAP, the planning system, will then adapt the mission plan according to the detected events as well as the available resources like power and memory budget and possible downlink times. Since ASAP is an experimental payload, SONATE is intended to operate in different modes: manual operation, semi-autonomous mode and full-autonomous mode. The different modes will allow careful testing without the risk of losing control of the satellite. The image processing and event detection algorithm in ASAP is based on a classification technique called cascading classifier. Even though there are rather powerful cascading classifiers nowadays, they are similar to SVMs limited when it comes to higher abstraction. Therefore, an embedded neural network chip capable of image classification and semantic scene understanding could improve the capabilities of ASAP a lot. In the Reference Scenarios section (3.1) the usage of the projected neural network system in an improved version of ASAP on-board a fictive SONATE 2 mission will be evaluated in more detail.

2.2 Neural Network Architectures and Layers

As indicated in the chapter before, there is a vast number of artificial neural network concepts and architectures. In this chapter, some relevant concepts will be introduced
2.2. Neural Network Architectures and Layers

and explained briefly to clarify terms used later in this thesis.

2.2.1 Perceptrons and Convolutional Neural Networks

In the scope of this thesis, only supervised training techniques will be regarded. Supervised training techniques are techniques including a training phase during which example input together with the desired output (also called label) is used to adapt the weights, i.e. the parameters in the mathematical relation between two nodes, of the network automatically. After the network is trained-in, it will only be used in forward-pass mode, i.e inputs are processed but there is no back propagation of errors made and thus no adaption of the trained-in weights. In contrast to that, unsupervised learning techniques work – as already mentioned – without labeled input and thus learn or adapt weights during operation.

One of the simplest forms a supervised artificial neural network can have is the (single-layer) perceptron. It consists of some input neurons and some output neurons which can have arbitrary connections. The inputs are multiplied with a weight and summed up. This sum triggers an activation function which is in classical single layer perceptrons a heavyside step function, i.e. the output can only be 0 or 1. During the training process the values of the weights are adapted (cf. fig. 2.1).

![Figure 2.1: Single-Layer Perceptron (Left: Detailed illustration showing just one output, Right: Simplified illustration with all outputs, Source: Own Illustration based on [13] p. 11/22).](image-url)
A more advanced form is the multi-layer perceptron (MLP). A multi-layer perceptron works similar to an ordinary perceptron but stacks, as the name indicates, several single-layer perceptrons, i.e. the output of the first perceptron is the input for the second perceptron, resulting in an additional hidden layer. Multi-layer perceptrons can have any number of layers. Note that the input layer is not counted when talking about N-layer networks. For example, a single-layer perceptron has only an input and an output layer, a 2-layer network has an input layer, one hidden layer and a output layer (cf. fig. 2.2). This is also where the afore mentioned differentiation between shallow and deep architectures comes into play: A single layer perceptron has a shallow architecture but is nowadays barely used whereas all MLPs have a deep architecture. A state vector machine as used in the NASA ASE is, in contrast, always shallow and can be represented with a single-layer perceptron.

Convolutional neural networks (CNNs or ConvNets) are variations of multi-layer perceptrons which recently gained lots of popularity, since they allow deeper architectures with less computational power than regular multi-layer perceptrons. Nowadays, CNNs are often used in face recognition, image recognition or natural language processing tasks.
The main differentiator between other artificial neural networks and CNNs is that CNNs have at least one or several so-called convolutional layers (cf. section 2.2.2). Neural network architectures with several convolution layers are often called deep convolutional neural networks (DCNN), the common term deep learning refers – vice versa – usually to DCNNs.

2.2.2 Convolution Layers

The input for a convolution layer usually consists of a three dimensional matrix. In the case of an image as initial input data, the three dimensions originate from the width (W) and height (H) of the image together with the color space, i.e. the third dimension (often called channel, Ch) would be 1 for a greyscale picture, 3 for a RGB picture or could have an arbitrary other value for multispectral images. The input is convolved with a \( K \times K \) kernel with a choosable size \( K \). Furthermore, the channel width \( Ch_{out} \) of the output matrix is freely choosable by the CNN architect. There is a distinct kernel for every input and output channel resulting in a 4D matrix for the kernel parameters with a size of \( Ch_{in} \times Ch_{out} \times K \times K \). This kernel parameters are the values that will be adapted during training, i.e. the layer weights. Furthermore, there is an additional bias vector which holds a bias value for each output channel \( Ch_{out} \) that is added to every value of this channel at the end. The bias value is also trained-in during the learning phase. The size of the W and H dimensions of the output matrix originate directly from the input parameters, the kernel size K and two additional parameters called stride S and padding P. The parameter P adds a specified amount of zeros around the input data in the W-H-plane, i.e. \( P = 1 \) results in an input dimension of \( (W_{in} + 2) \times (H_{in} + 2) \times Ch_{in} \). The stride specifies the step size in which the \( K \times K \) kernel is slid over the W-H input plane within the convolution. The dimension of the resulting output matrix can then be calculated to be \( W_{out} = (W_{in} - K + 2P)/S + 1 \) and \( H_{out} = (H_{in} - K + 2P)/S + 1 \) respectively. Figure 2.3 shows exemplarily the convolution for a \( 5 \times 5 \times 3 \) input with a \( 3 \times 3 \) kernel. The stride was chosen to be \( S = 2 \), the padding is \( P = 1 \) and the desired output channel width is 2 which results in an output size of \( 3 \times 3 \times 2 \). The weights for this example have a size of \( 3 \times 2 \times 3 \times 3 \) plus the two values of the bias vector. Since the output has again a dimensionality of three, this output can be directly used as input for another convolution layer or – alternatively – another arbitrary layer type.

In figure 2.3 the three blue boxes depict the three channels of the input data (also called...
Figure 2.3: Example of how a Convolution Layer works.

feature maps), the six red boxes are the kernels (three input channels times two output channels), and the green boxes represent the two channels of output data. The yellow arrows indicate the data flow in the first step, the orange arrows represent the second step (the remaining steps are not visualized for reasons of clarity). In the first step, the upper left \(3 \times 3\) \((K \times K)\) tile of the first input channel \(Ch_{in\ 1}\) is dot multiplied with the two kernels in the \(Ch_{in\ 1}\) row resulting in two interim output values, one for \(Ch_{out\ 1}\) and one for \(Ch_{out\ 2}\). The same is done with input channel \(Ch_{in\ 2}\) and \(Ch_{in\ 3}\). The resulting three interim output values of the \(Ch_{out\ 1}\) column are added together as well as the three interim values of \(Ch_{out\ 2}\). Finally, the bias value in the purple box is added to the sum calculated before which results in the upper left output value of \(Ch_{out\ 1}\) and \(Ch_{out\ 2}\) respectively. In the next step (orange), the same procedure as described before is repeated but this time a \(3 \times 3\) tile two steps further to the right is used (since \(S = 2\)). This time, the resulting output values are written two the second position (upper center) of the two output matrices. The next processing step (again shifting the \(3 \times 3\) box two steps to the right), results in the third output value of the upper column of the output matrices. In vertical direction, the step size (stride) is also two, which means that in the fourth processing step, the \(3 \times 3\) box in center left will be used. From this, one can easily
derive why the size of the two output matrices is $3 \times 3$. If the stride would have been $S = 1$ instead of $S = 2$, the size of the output matrices would be $5 \times 5$ for example.

### 2.2.3 Other Common Layers

Besides convolutional layers there may be arbitrary other layers within a CNN. Typical other layers are pooling layers, rectified linear unit (ReLU) layers, fully connected (FC) layers and concatenation layers (Concat).

Pooling layers condense a $K \times K$ grid of input values to a single output value. To determine this output value, there are two common pooling methods: maximum pooling and average pooling. In theory there are even more pooling methods but since those other pooling methods are barely used special cases, they are not described in more detail here. When choosing maximum pooling, the output value equals the highest value within its corresponding $K \times K$ input grid. In an average pooling layer, the arithmetic mean of all values in the $K \times K$ input tile is chosen as output. Pooling is used to reduce the spatial size of the data in the W-H-plane. The channel width remains the same within a pooling layer (cf. fig. [4]). Usually, there are several pooling layers within a CNN to gradually reduce the size and therefore the amount of parameters and necessary computation. In addition, a problem called overfitting can be controlled with pooling layers since they reduce the information contained in a layer. Overfitting means that a neural network adapts its parameters too strongly to certain features in the training data and therefore loses its ability to generalize (also process input that was not part of the training data) correctly.

ReLU layers are used to add nonlinearity to a neural network. They simply set every value which is smaller than zero to zero while positive values keep their value ($f(x) = \text{max}(0, x)$). Since ReLu layers are very simple, do not change the spatial size of the data and can therefore be calculated in-place, they are often not implemented as distinct layers but rather as filter at the end of other layers. This is also often reflected in the way ReLu layers are denoted or depicted (cf. CNN diagram in appendix [B]).

Concatenation layers are used to merge the data of two input layers into a single layer. In order to be able to merge two layers, two of their three spatial sizes have to be the same, for example a layer with $C h_1 \times W_1 \times H_1$ can be merged with a layer $C h_2 \times W_1 \times H_1$ but not
with a layer $Ch_2 \times W_2 \times H_1$. When merging a layer with spatial size $Ch_1 \times W_1 \times H_1$ with a layer $Ch_2 \times W_1 \times H_1$, the resulting output will have the size $(Ch_1 + Ch_2) \times W_1 \times H_1$. As one can see, the concatenation layer does not manipulate any data contained in the two input layers, it simply joins the input matrices together. As a result, concatenation layers are also often not implemented to process data like other layers but rather manipulate the address range of the data that will be accessed in the next layer.

Figure 2.5: Example of the Layers and Dimensionality within a Convolutional Neural Network.
Fully connected layers are usually applied at the very end and work like regular perceptrons: Every input neuron is connected to every output neuron with a certain relation. In the last layer, the output size is usually chosen to be $N \times 1 \times 1$ where $N$ is the amount of output classes (cf. fig. 2.5). Therefore, the output is a vector in which every entry corresponds to the probability that the given input belongs to this class. One of the reasons why FC layers are applied at the end is that the size of parameters and required computations grow drastically with the input size. Therefore, other layers like convolutions and pooling are applied as kind of pre-processing steps to reduce the size of the input data before using a FC layer.

During training an additional layer after the last ordinary layer is applied, called loss layer. The loss layer can implement different types of loss functions, e.g. softmax loss, sigmoid cross-entropy loss or euclidean loss. The calculated loss specifies how the deviation between actual and expected output of the neural network during training is treated. Since the work in this project focuses on the forward pass, i.e. the execution of neural networks, and back propagation (learning) is a rather complex topic on its own, the working mechanism of loss layers is not explained in further detail here.
In the following chapters, the system design is elaborated. For this purpose, two sample scenarios are worked out which will make use of the projected neuro-processor. Afterwards, a set of requirements is defined upon these scenarios which will eventually lead to the system concept in chapter 3.4. Furthermore, an overview of existing products, concepts and software frameworks is given in chapter 3.3 to elaborate their possible utilization within this project.

3.1 Reference Scenarios

3.1.1 Scenario 1: SONATE 2

SONATE is a triple cubesat currently under development at the professorship of space technology at the University of Würzburg. One of its main payloads is a tailored version of ASAP, an autonomous sensor and planning system. ASAP is capable of detecting different unpredictable short-term events and automatically adapt the satellite’s mission plan to track this event as long as possible. Furthermore, it can autonomously manage the satellite’s resources like memory and power budget. ASAP consists of two parts: The first part is the imaging and detection system, that takes an image, processes it and forwards, if applicable, information about the detected objects. The second system is the planning system. It takes the metadata (e.g. type and position of detected object) received from the imager as well as several other datasources about the current state of the spacecraft as input and creates a plan how to achieve its goal with given resources.
The system responsible for image processing in ASAP is, compared to artificial neural networks, relatively weak and inflexible. It consists mainly of a pattern matching algorithm based on a cascading classifier. The level of abstraction is thus very limited. Even though the used system is sufficient to achieve SONATE’s mission goal, it is expected that a new version based on the neuro-processor developed within this work would improve the performance of the ASAP imager significantly and could therefore be used in a follow-up mission of SONATE.

Within this fictive SONATE 2 scenario, the neuro-processor must be able to operate on-board a nano-satellite and accept an input image or image patch and decide, whether an object of a previously defined object set is visible within this image. The processing of the images should happen as fast as possible but real-time video processing is explicitly not required. The objects of interest, i.e. the object set, should be updatable from ground, online learning, i.e. executing the training phase on the chip itself, is not required.

3.1.2 Scenario 2: ASMET

Another example scenario which does not exactly match the intended use case (space) but could, due to similar requirements, also make use of the neuro-processor is the ASMET project. ASMET is a projected sensor platform network that will monitor the sky in order to detect meteors and other interesting objects like unknown light phenomena. From experience with a similar preceding project called SkyCam\(^1\), it is known that such a sky observing system will be triggered countless times per day by ordinary events like birds or insects passing by as well as planes, helicopters, balloons and so on. Therefore, a method to filter out the uninteresting events is required. Further requirements imposed on the ASMET system include a portable design and the capability to operate in remote areas, i.e. consume as little power as possible and work with a slow or even no internet connection. This is why the neuro-processor, which will be developed in the scope of this work, would also be a possible choice for image processing and semantic understanding on-board of ASMET. Even though it is not a spacecraft, the need to process images semantically under harsh conditions using as little power as possible turns this application into an ideal use case.

\(^1\)http://www8.informatik.uni-wuerzburg.de/mitarbeiter/kayal0/weitere_forschungsaktivitaeten/skycam/
3.2 Requirements

The requirements towards the neuro-chip for usage in ASMET are, as already indicated, very similar to the requirements in the SONATE 2 scenario: The neuro-processor must be able to decide whether an image shows a previously defined object (bird, insect, plane, etc.) or an unknown object. Furthermore, real-time processing is again not required since it is expected that there is a basic movement detection implemented before passing the images to the neural network. This movement detection will filter out most of the images that would anyway just show the static sky. Therefore, the neural network should have enough time to process the images of a detected event before the next event or movement occurs. Power consumption is expected to be not as critical as on a spacecraft, but should nevertheless be kept as low as possible to be more flexible regarding the installation site of the system.

3.2 Requirements

The requirements for the system are derived directly from the scenarios above. A full list of requirements is given in appendix A. This section will give a short overview of the key requirements as well as some explanations.

From the example scenarios, it is derived that the primary input data will consist of images or image-like data. Therefore, the main goal of the project is to develop an embedded neural network system, which is capable of semantic image classification, i.e. it has to be capable of recognizing predefined patterns or objects in a given image, under conditions as found on a small satellite platform. Furthermore, it should be possible to update the set of patterns and objects that can be recognized by the system in-orbit to add flexibility. Other requirements imposed on the system are modularity which includes scalability and the possibility to extend its capabilities to other input data like video material. In addition, setting up the system should be made as easy as possible to allow later reusage and the required training time for the neural network should be in a reasonable range, i.e. training the system should not take longer than a few days on an ordinary workstation.

Since object identification in images is a rather loose concept, it is required that the performance should be similar or better to the AlexNet neural network architecture, a famous network architecture that won the 2012 ImageNet Large Scale Visual Recognition Chal-
lenge (ILSVRC) and offers enough accuracy to solve real-world problems. The ILSVRC is a well-known annual contest with different tasks in the field of computer vision for large scale image indexing. Typical tasks within this challenge include object detection in images or videos, object localization and scene classification. Since the ILSVRC is used for comparison and as a benchmark test for the latest improvements and state-of-the-art image processing techniques, the approaches, reference data and results will be used a lot within this work. AlexNet was chosen as reference, since its performance is regarded to be more than sufficient for the intended tasks and the required computational power seems to be in a manageable range whereas newer and better implementations usually need much more resources and outperform AlexNet in precision just by a few percentage points.

Another important decision was whether online learning is required or not. Online learning means that learning is done on the embedded platform itself. This might be useful, when the use case involves adapting the learned parameters of a neural network during operation or if the purpose of the embedded system is to speed up learning on a bigger machine (i.e. developing a neural network accelerator). Offline learning in contrast means that the neural network is trained on another machine, e.g. a computer and the trained-in parameters or weights are then transferred to the embedded system for execution. This has the advantage that the execution of a neural network (i.e. the forward pass) requires much less resources than the training and is usually sufficient for most present-day applications. Especially in space related applications where one wants to test and qualify the system before uploading it to the spacecraft, offline learning is in most cases the better choice. From the reference scenarios it is derived, that for the projected neuro-processor online learning is not required.

3.3 Related Work

It is needless to say that the idea of transferring neural networks to embedded hardware is not new. There have been several papers and even products with similar objectives that will be evaluated in more detail within this chapter.

Some recent developments worth mentioning are the MIT Eyeriss chip as well as the Movidius Myriad 2 VPU. The MIT Eyeriss chip is an energy-efficient accelerator chip for
3.3. Related Work

deep convolutional neural networks. It features 168 processing elements and consumes just about 300 mW. This work is very interesting and worth monitoring but up to now, there is barely any information available about this chip. What militates against its usage is, that it seems like it will not be commercially available in near future. Additionally, it is a custom made chip which means it is either very expensive when manufactured in small quantities or, if produced in large quantities, won’t be scalable and adaptable to (space-) mission specific needs. [14]

The Movidius Myriad 2 VPU is another chip implementing a neural network and is optimized for video processing. Movidius claims it is capable of analyzing video sequences in real-time with a power consumption of less than 1 W. Again, this chip is very interesting but up to now only little information is available. Even though the manufacturer states that the Myriad 2 will soon be available, it is not obtainable at the moment. The downsides are, similar to the MIT Eyeriss, the proprietary design and thus its lack of scalability and adaptability.

A less proprietary work worth mentioning is the paper ”Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks” by Zhang et al. from 2015, which describes a FPGA based convolutional neural network [1]. The authors state that they achieved a performance about 5 times faster than a 2.2GHz Intel Xenon processor with just 18.6 W power consumption. Of course this power consumption is way too much for a nano or pico satellite but nevertheless remarkable compared to its performance. The architecture proposed by Zhang et al. may be used as a basis for further investigation within this thesis.

Since the training of the artificial neural network will presumably be done offline, i.e. on an ordinary computer or workstation, existing software and frameworks for this purpose have also been studied in more detail. The advantage of using existing software or frameworks is that they are well tested and far more optimized, especially regarding parallel execution on GPUs, than it would be possible within the timeframe of this project. A minor downside one has to take into account when using an existing tool or framework is that the extraction of necessary data for the embedded implementation, especially weights and parameters out of the trained model is more difficult than with a self written tool. This is due to the fact, that those tools and frameworks are usually meant for both training and execution of the neural network and thus store the trained model in an own
framework-specific format. Therefore, an important criterion for choosing an appropriate framework was a good documentation of the internal mechanisms and data structures and – if possible – freely accessible source code. Three of the most common frameworks and toolsets for designing artificial neural networks are Torch, TensorFlow and Caffe. TensorFlow and Caffe offer both a Python interface while Torch uses the more exotic Lua script language. Even though TensorFlow is very promising, it currently lacks a proper C/C++ interface and since it has been released just about one year ago (November 2015), there are not that many working examples. Caffe, in contrast, offers also a well documented C/C++ API as well as countless examples and an extensive documentation. Furthermore, the way Caffe describes network architectures is regarded as more structured in comparison to TensorFlow and thus is more suitable for translating the architecture to an embedded design.

3.4 System Overview

The intended overall system structure and workflow is depicted in figure 3.1.

![Overall Workflow Structure](image_url)

*Figure 3.1: Overall Workflow Structure.*

As a first step, a database containing the images and the corresponding categories (labels) has to be defined. Furthermore, the architecture of the artificial neural network (ANN) as well as a so-called solver, which handles the training process, have to be defined. Afterwards, the neural network is trained-in by a machine learning framework (Caffe). The resulting model containing the trained-in weights can then be tested by the
engineer and eventually, when the model behaves as expected, be extracted into VHDL code and synthesized for execution on the embedded platform (FPGA).

The system design as well as the workflow steps briefly described before are explained in more detail in the following subchapters.

### 3.4.1 Offline Learning Framework

As already indicated in chapter 3.3, the neural network framework Caffe, developed and maintained by the Berkeley Vision and Learning Center (BVLC), was evaluated to be best suited to train a neural network and afterwards extract the parameters and weights for further usage in the embedded neuro-processor. Since there are only command line tools available to control the Caffe framework, a dedicated tool including a graphical user interface (GUI) was developed within this thesis, based on the Caffe framework. This tool will be used to first create an appropriate training dataset, define the neural network architecture, train and test the network and then eventually export the trained-in weights and parameters to a format readable for the embedded neuro-processor. With this tool, it should be possible to train the neural network and set it up on the embedded execution platform without the need of deep knowledge in neural network design or hardware description languages. The long-term goal is that an engineer with basic but no extensive knowledge in machine learning is able to set up the neuro-chip according to his mission specific needs.

The workflow using Caffe is as follows: First, one needs to create a dataset for training, i.e. one needs input data (in this case images) together with the desired output (labels). Caffe accepts different formats for these datasets. In the simplest case, it consists of just a single text file with two entries per line: the path to the picture and the label. While this is easy to set up, it gets confusing when dealing with large datasets and is also rather slow compared to the other formats. Therefore, the recommendation of the authors of Caffe is to use this format only for testing, not for training of large datasets. Better suited formats for training are LevelDB or LMDB databases. In the offline learning toolchain developed within this project, LMDB was used (cf. chapter 4.1). In addition, one needs to create the neural network definition. In Caffe, neural networks are defined within so called protocol buffers. Protocol buffers are "a language-neutral, platform-neutral,
extensible way of serializing structured data for use in communications protocols, data storage, and more” [15]. Protocol buffers allow, similar to the better known standard XML, a structured way to define data but are – according to the developers of protocol buffers – smaller, faster and simpler. After the neural network structure has been defined, one needs to define a so-called solver which is also done via a protocol buffer. The solver specifies the training process itself. In this file, various parameters can be specified like base learning rate, the back propagation type or the amount of training iterations. With these three files, database, network definition and solver definition, the training can be started. The output of the training process is a .caffemodel file which contains the trained-in parameters and weights.

3.4.2 Neural Network Architecture

Even though the neuro-processor will be implemented with the most possible flexibility to allow the execution of different neural network architectures, it is not possible to build a neuro-processor which can handle all flavors of neural network architectures especially when it comes to optimizations regarding performance and power consumption. Therefore, a reference architecture is chosen and the implementation will be optimized towards this reference neural network. Nevertheless, the neuro-processor will also be able to execute different neural network architectures with no or little changes, depending on the neural network and how much it differs from the reference architecture. From the results of the ILSVRC one can clearly see that CNNs are currently the best possible choice when it comes to semantic image processing and interpretation. A rather new approach is to also replace the fully connected (FC) layer, which is normally used as last layer for the high level decision making, with convolutional layers. It has been shown, e.g. by [16], that even a network without fully connected layers can perform similarly well or even better than AlexNet, a network that is often used as reference. The big advantage of networks without FC layer is, that they have far less weights and are thus faster to train and require less memory. One of those convolution-only networks is SqueezeNet, which has a similar performance to AlexNet but requires 50 times fewer parameters and therefore achieves a model size of under 0.5 MB as compared to over 200 MB for AlexNet. This drastic reduction of required weights is attributable to a feature of convolutional layers called parameter sharing. Parameter sharing relies on the assumption that if it is useful to calculate a distinct feature at a certain spatial position, it should also be useful
to calculate the same feature at other spatial positions. Even though this assumption may not always be valid, the error made through this assumption seems to vanish when using several stacked convolutional layers as the example of SqueezeNet shows. Since memory capacity and uplink bandwidth is extremely limited in space application, the model size is regarded crucial for usage in space. While 200 MB require a rather large and expensive storage and are (almost) impossible to be uploaded in UHF or S-band, a 0.5 MB model can easily be stored and is also small enough to be updated during the mission lifetime. This extremely small model size together with the sufficiently good performance of SqueezeNet lead to the decision to use this neural network architecture as reference. SqueezeNet consists, besides various convolutional layers only of pooling and concat layers. Its structure can be found in appendix B.

3.4.3 Embedded System

On the embedded side, a FPGA based approach for executing the neural network was chosen. A FPGA has, in contrast to a microcontroller based approach the advantage, that it can be adapted much better to the properties of an artificial neural network. In neural networks, one can extensively make use of parallelization which can be perfectly implemented within a FPGA but leads to large and inefficient loops on a microcontroller. Because of the very promising results in [1] by Zhang et al., their approach will be used as a basis for the implementation of the neuro-processor but with some simplifications. The simplifications are needed for two reasons: First, the Virtex chip used by Zhang et al. is a very powerful FPGA but has a too large power consumption for the intended use on-board a nano-satellite. Second, their design is in some parts very complex and has been implemented by a whole team in years of work. This project in contrast is intended to be a starting point for further investigations within the department of Aerospace Computer Science at the University of Würzburg and does not have the aspiration to be extensive or the best possible implementation. Nevertheless, it is emphasized to design all parts of the project as modular and scalable as possible to allow later improvements in a possible follow-up project.
This page is intentionally left blank.
4.1 Offline Learning Toolchain

The offline learning toolchain is meant to guide engineers who are setting up the neuroprocessor for the intended task through the necessary steps:

1. Create an image/label database for learning
2. Define and train the artificial neural network
3. Test the trained neural network
4. Export the network structure and trained-in weights in an appropriate format for embedded execution

For this purpose, a C++ application with a Qt-based graphical user interface (GUI) has been developed, which allows an easy usage of the underlying Caffe framework. Since the Caffe framework is only available for Linux, the offline learning toolchain is also written for Linux. The GUI is separated into different tabs representing the steps enumerated above.

Figure 4.1 shows a screenshot of the first tab: Create Database. In the upper left corner of the GUI, the user can select and load any unlabeled pictures. In the middle of the screen, the currently selected image is shown. In the lower left corner, one can add or remove categories for the dataset. Furthermore it is possible to specify a shortcut key for each category in order to be able to browse quickly through the unlabeled images and categorize them. After an image was categorized, the corresponding entry is moved
from the left list (uncategorized) to the list on the right side where all entries are listed according to their category. After all images have been categorized, it is possible to export the dataset of labeled images either as text file (txt) or as LMDB database. Both formats are supported by Caffe but LMDB is recommended for training, since it allows faster data access while txt is easier to use and can be used for testing.

In figure 4.2 the GUI for the next step – Define and Train Network – is shown. Besides the previously created image/label database, Caffe needs a protocol buffer (prototxt) file specifying the neural network architecture as well as another prototxt file to configure the solver, i.e. define the parameters for the learning process. Protocol buffers are, as already described in chapter 3.4.1, structured text files similar to the better known XML standard. Those two prototxt files can either be loaded and edited or created directly within the GUI. In the table on the right side, it is intended to list all layers together with their dimensions/size and the memory capacity needed for their weights. This will help the engineer who is designing the neural network architecture to estimate whether the weights and parameters are small enough for the intended hardware and link budget. However, up to now this feature is not fully implemented since it is not mandatory and
4.1. Offline Learning Toolchain

thus has low priority. By clicking the "Trainiere Netz" (Train Net) button, the training process starts. The process can be monitored via the status messages displayed in the elongated text field at the bottom of the window.

In figure 4.3 it is shown how a trained-in neural network can be tested using the offline learning toolchain. In this example, the original SqueezeNet model trained on the ILSVRC 2012 dataset is tested with a picture showing a cat. The ILSVRC 2012 dataset contains 1,000 categories including several different cat types. In the status window below, it can be seen that SqueezeNet is about 26.8% sure that the image shows a kit fox. Its second guess with about 22% probability is a red fox. Even though this predictions are not quite accurate, it is very impressive if one considers that the neural network has to chose between 1,000 possible categories which are additionally often very similar. The files which have to be specified within the offline learning toolchain to be able to test a
model file are the network definition, the trained-in model file (caffemodel) as well as a mean data file and an additional text file called labels or synset. The mean data file is a file specifying the average value in each dimension of all input training data. This is used for normalization and helps the neural network to perform better but is not necessarily needed. The synset or labels file is a file providing human readable category descriptions. Within the neural network the categories are only denoted with indices since numbers are easier to handle for computers. If one wants human readable descriptions like "Kit fox, Vulpes macrotis" in the status window, the synset file is needed to look up the descriptor corresponding to a certain category index.

The last tab – Export VHDL Code – is not shown here since its functionality corresponds to the not yet implemented Control Unit of the embedded system (cf. section 4.2.3) and is therefore also not implemented up to now. As described in chapter 5 in more detail, it
was not possible to implement all features within the scope of this thesis. A breakdown of subsystems and components that are fully implemented and tested and a description of the remaining work that has to be done can also be found in chapter 5.

### 4.2 Embedded System Implementation

For the embedded implementation a FPGA-based approach was chosen. As already mentioned before, in contrast to a microcontroller, a FPGA allows the utilization of the parallel characteristics of artificial neural networks (especially CNNs) which results in a massive performance improvement. For implementation, the TerasIC DE2-115 development board was chosen which features an Altera Cyclone IV E FPGA together with lots of periphery. For this project, the most important additional component on this board is the external 64 Mbit SDRAM which will provide enough memory capacity for the large temporary data needed during execution. Furthermore, the board also offers a RS232/UART interface that will be used for user communication and testing. The board can also easily be equipped with a 5 MP camera over an extension port which could later be used as input for the neural network. However, in the beginning it is intended to upload an image via UART to the FPGA, process it and return the output again via UART. Even though this approach might be slow and therefore annul the performance increase achieved through the FPGA, it offers the easiest and best debugging possibilities and thus is preferred in this early development stage.

#### 4.2.1 Overview

A schematic system overview with all relevant datapaths is depicted in figure 4.5. The communication with the user is done over an UART interface which is connected to the Control Unit, an entity that is primary responsible for sequential control, i.e. loading the different layer definitions, adjust the datapath via the layer multiplexer (MUX) and calculate the right address range for weights and input/output data. The Layer MUX is a simple multiplexer forwarding the data from the external memory to the right processing unit depending on the current layer to execute. The External Memory Interface implements a multi-port SDRAM interface which is needed to read and store the input data (feature maps\[^1\]), intermediate (temporary) data as well as the output data. Since the intermediate data can become quite large (several MByte), an external memory is

---

\[^1\]A feature map (fm) denotes the data contained in a single channel (2D matrix).
stringently required. The weights however are small enough to be stored inside the FPGA which allows faster and more energy efficient execution of the neural network compared to using external storage, a beneficial side effect of the requirement for small model sizes which resulted from the low available bandwidth in space applications.

For the representation of decimal numbers, a fixed point approach has been chosen. Fixed point representations require far less resources for mathematical operations compared to floating point numbers which is why they are often used in hardware oriented embedded system. A fixed point decimal is basically an integer value that is scaled by a distinct factor. For example, the number 3.14 can be represented as 3140 with a scaling factor of 1/1000. The scaling factor must be chosen according to the expected values the system has to operate with in order to avoid overflows (values which are too large to be represented) or precision loss (values that are too small to be represented). Therefore, fixed point arithmetic only works well if the expected input data is within a certain and not too large range. For the CNN, an appropriate scaling factor has to be determined by analyzing the expected input and the data propagation through the neural network. This
is a task that could be done automatically by the offline learning toolchain. However, such a feature has not been implemented yet. An additional requirement when choosing a scaling factor for fixed point arithmetic on a FPGA is that the scaling factor should be a power of two. When multiplying two scaled numbers, the scaling factor gets squared which means that the result of the multiplication has to be divided by the scaling factor to maintain the initial scaling. However, divisions are usually very complex operations on FPGAs requiring lots of resources. The sole exception to this is the division by a power of two, which can be done with a simple shift operation. Therefore, the necessary resources for a multiplication can be dramatically reduced by choosing a scaling factor that is a power of two.

In previous studies, it has been shown that the achievable precision with 8 bit fixed point decimal numbers is sufficient for many neural networks. In [17], the author states that SqueezeNet can work with 8 bit fixed point numbers with a precision loss of less than 1% compared to the ordinary 32 bit floating point implementation. Even though 8 bit fixed point decimals seem to be sufficient for most neural networks, it was decided to work with 18 bit fixed point values in the beginning to eliminate overflows and precision losses as potential error sources. Besides the fact that 18 bits are sufficient to represent the necessary decimals, this value was chosen because it is the largest input width the embedded multipliers within the Altera Cyclone IV FPGA can handle. The embedded multipliers are very important components within the FPGA, since they allow a fast and efficient multiplication of two numbers, something that is also usually either very slow or very resource demanding when no dedicated multiplication units are available. Having an efficient way to do multiplications within the FPGA becomes especially important when considering that a convolutional layers consist almost entirely of multiplications (cf. 4.2.2).

4.2.2 Convolver Cluster

The Convolver Cluster can be regarded as one of the most important components of the neuro-processor. In a CNN, around 90% of the computational time is spent for convolutions (cf. [18]). Thus, optimizing this part yields the highest benefit. Furthermore, the convolution operation is well suited for parallelization and can therefore utilize the parallel processing capabilities of FPGAs in the best possible way. This is also why this
entity is called Convolver Cluster: It consists of several convolution (processing) units working in parallel. To understand how this can be achieved, it is helpful to have a look at figure 4.6 together with the pseudo-code in listing 4.1 that illustrates how the convolution within a convolutional layer works.

As one can see, the convolution is basically a multiply-accumulate operation (MAC) within multiple nested for-loops. For-loops are in general an indicator that the code may be parallelizable. Instead of executing the command(s) in the loop one after another, it is possible to execute several of those loop iterations in parallel on different computa-
Depending on which for-loop is unrolled, the entities will have different data sharing relations and can therefore differ a lot in complexity. In [1], the authors analyzed the data sharing relations of the convolution pseudo-code. They categorized the loops in three different types of data sharing relations: irrelevant, independent and dependent. According to [1], these data sharing relation categories are defined as follows:

- **Irrelevant.** If a loop iterator $i_k$ does not appear in any access functions of an array $A$, the corresponding loop dimension is irrelevant to array $A$.

- **Independent.** If the union of data space accessed on an array $A$ is totally separable along a certain loop dimension $i_k$, or for any given two distinct parameters $p_1$ and $p_2$, the data accessed by $DS(A, i_k = p_1) = \cup Image(F^A_S, (D_S \cap i_k = p_1))$ is disjoint with $DS(A, i_k = p_2) = \cup Image(F^A_S, (D_S \cap i_k = p_2))$, the loop dimension $i_k$ is independent of array $A$.

- **Dependent.** If the union of data space accessed on an array $A$ is not separable along a certain loop dimension $i_k$, the loop dimension $i_k$ is dependent of array $A$.

The results of this data sharing relation analysis by Zhang et al. can be found in table 4.1. Based on those results, the authors suggest to unroll the loops along the to/ti axes since.
those are the only axes without dependent data relations. However, since the axes to/ti are in some layers too large to be completely unrolled, they have to be sliced, i.e. only a part can be unrolled (tii/too) and the remaining part will remain a loop (cf. listing 4.2).

<table>
<thead>
<tr>
<th></th>
<th>input_fm</th>
<th>weights</th>
<th>output_fm</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td>dependent</td>
<td>irrelevant</td>
<td>independent</td>
</tr>
<tr>
<td>col</td>
<td>dependent</td>
<td>irrelevant</td>
<td>independent</td>
</tr>
<tr>
<td>to</td>
<td>irrelevant</td>
<td>independent</td>
<td>independent</td>
</tr>
<tr>
<td>ti</td>
<td>independent</td>
<td>independent</td>
<td>irrelevant</td>
</tr>
<tr>
<td>i</td>
<td>dependent</td>
<td>independent</td>
<td>irrelevant</td>
</tr>
<tr>
<td>j</td>
<td>dependent</td>
<td>independent</td>
<td>irrelevant</td>
</tr>
</tbody>
</table>

This increases the complexity of the Control Unit significantly since an output value cannot be calculated in one run before sending it to the external memory. Instead a specific number of interim values would be calculated, sent to the external memory and in a later step loaded again out of the external memory to finally sum up the final output value. Even though this approach allows a maximum utilization of all resources available on the FPGA, in this work another approach was chosen: Unrolling the i/j axes. Even though this choice brings the drawback of having to oversize the computation units for some layers, it simplifies the Control Unit and External SDRAM Interface significantly and was therefore preferred. Another factor that should be taken into account is that the approach chosen by Zhang et al. requires a rather large buffer in the internal memory to store the tile that is currently processed. With the approach chosen in this work, the only buffer required is a comparatively small FIFO to translate between the different clock domains of the external memory and the processing cores. Therefore, the remaining internal memory can be used to store the weights of the entire CNN, what could – despite the fact that the processing cores are not always working at full capacity – be more efficient since the weights do not have to be transferred from external to internal memory anymore. It is suggested that the impact of unrolling the loops along different axes on the overall system performance – especially when using convolution-only networks with very few weights – should be evaluated in more detail in a follow up project.

\[2\text{Note: Loop designators changed since the original table referred to different pseudo-code.}\]
4.2. Embedded System Implementation

Listing 4.2: Proposed Convolution Architecture by Zhang et al. (Source: [1]).

```c
for(row=0; row<H_out; row+=Tr) {
    for(col=0; col<W_out; col+=Tc) {
        for(to=0; to<Ch_out; to+=Tm) {
            for(ti=0; ti<Ch_in; ti+=Tn) {
                // load output feature maps
                // load weights
                // load input feature maps

                // on chip data computation
                for(trr=row; trr<min(row+Tr,R); trr++){
                    for(tcc=col; tcc<min(col+Tc,C); tcc++){
                        for(too=to; too<min(to+Tm,M); too++){
                            for(tii=ti; ti<min(ti+Tn,N); tii){
                                for(i=0; i<K; i++) {
                                    for(j=0; j<K; j++) {
                                        output_fm[too][trr][tcc] += weights [too][tii][i][j] * input_fm[tii][S*trr+i][S*tcc+j];
                                    }
                                }
                            }
                        }
                    }
                }

                // store output feature maps
            }
        }
    }
}
```

The Convolver units for the neuro-processor developed in the scope of this project work as follows: Every Convolver consists of $K_{max} \times K_{max}$ embedded multipliers with $K_{max}$ being the maximum kernel size of all layers in the neural network. For the reference architecture SqueezeNet, $K_{max}$ is 7. The outputs of all those multipliers are then added together in an parallel adder unit (cf. 4.7).

Since there are more embedded multipliers available than necessary for one Convolver Unit, several Convolver Units are instantiated and pooled together in an entity called Convolver Cluster. How many Convolver Units exist within a Convolver Cluster depends on the available resources. For the used Altera Cyclone IV FPGA and a maximum kernel size of 7, the Convolver Cluster can hold 5 Convolver Units. Each Convolver Unit operates on one input channel at a time, which means that in this example 5 input channels can be processed at a time. The outputs of the Convolver Units are added together. If the number of input channels is larger than the amount of Convolvers within the Convolver Cluster, the output is stored in a register and added to the sum of the next execution cycle. After all input channels have been processed, the output value is written into
Figure 4.7: Illustration of a Convolver Unit to Multiply Up to $K_{\text{max}} \times K_{\text{max}}$ Input Values and Weights in a Single Clock Cycle.

an output FIFO from where it can be transferred to the external memory. The input FIFOs, weights FIFOs and output FIFOs are necessary to separate the processing units from the memory unit in a clean way since they work at different clock speeds. After all input channels have been processed, the register holding the temporary sum is set to zero. Afterwards the convolution kernel is moved with the specified stride over the input feature maps and the next output value is calculated by repeating the steps described before. If the amount of input feature maps is not dividable by the cluster size, some of the Convolver Units are idle during the last step since the amount of necessary clock cycles needs to be rounded up. For example, if the input data has 8 channels and the Convolver Cluster holds 5 Convolver Units, two clock cycles are needed to process all 8 input channels resulting in two idle Convolver Units in the second cycle. Furthermore, if the kernel is smaller than the specified maximum kernel size, some of the multipliers within the Convolver Units are idle too. This was already indicated in the section above: With the need to oversize the computation units in this approach, not all hardware resources available on the FPGA can be used during some computation steps. However, the approach was nevertheless regarded to be best suited at the current stage of development for the reasons already mentioned. The sequential management of all those steps
and the supply of the Convolver Units with the right values and weights as well as the storage of the output value after it has completely been processed is part of the tasks of the Control Unit.

![ConvCluster Diagram](image)

**Figure 4.8: Illustration of the Convolver Cluster.**

### 4.2.3 Control Unit

The Control Unit is responsible for handling the sequential execution of the necessary steps in the different layers. This involves managing the right timing of all entities within the FPGA as well as calculating the addresses for internal and external memory to load and store the data from or to the right position. Since timing is a crucial part of FPGA development due to the parallel execution of commands, this unit can be regarded as a connection point that fuses together all other units.
As indicated before and described in more detail in chapter 5, this unit could unfortunately not be implemented yet.

4.2.4 External SDRAM Interface

The core of the External SDRAM Interface consists of a so-called IP (intellectual property) core. SDRAMs have a rather complicated interface but since dozens of FPGA developers were facing this problem before, there are good and well tested implementations of SDRAM interfaces available on the internet. For this project, a free implementation by Mike Field available on hamsterworks.co.nz was used. Even though this generic implementation does most of the hard work, the code had to be adapted to the properties of the specific memory unit available on the DE2-115 board. Furthermore, the connection between the generic SDRAM implementation and the rest of the system had to be developed by hand. Since the processing units of the neural network rely heavily on parallel processing, several values from the external memory have to be present at the same time as input to the neural network. Therefore, the external memory interface is driven with a higher clock frequency than the processing cores to avoid a bottleneck at the memory interface. This is possible since the SDRAM interface implementation is basically a large state-machine and thus, besides its apparent complexity, executable very fast. In order to separate the different clock domains of the memory interface and the slower processing cores in a clean way, some buffer FIFOs had to be implemented.

4.2.5 UART Interface

The UART Interface is held rather simple and is also based on a freely available IP core. Again, some adaptions had to be made to the generic IP core implementation to make it work with the present hardware but since UART is in general a very simple system only minor changes were necessary.

In order to store and interpret the received commands, a ringbuffer together with a parser had to be implemented. Furthermore, a connection between UART Interface and the Control Unit as well as the SDRAM Interface was needed to allow a download of test data (images) to the memory for tests and debug purposes with the neural network.

3http://hamsterworks.co.nz/mediawiki/index.php/SDRAM_Memory_Controller
4https://github.com/pabennett/uart
Since it was not possible to develop a complete, working prototype within the timeframe of a master’s thesis, only key components were implemented and tested independently from each other. Those components will be used in further development steps. Additionally, the conducted tests give a good impression of the amount of work to expect, the feasibility of the system and potential weak points in this concept. The elements implemented so far as well as the necessary parts to complete the system are depicted in figure 5.1.
5.1 Setup and Tests of the Offline Learning Toolchain

In order to test the workflow with the developed offline learning toolchain, an example dataset was trained-in. First of all, the Caffe framework had to be set up. This can be done by following the instructions on the Caffe project website\footnote{http://caffe.berkeleyvision.org/installation.html}. Setting up the CUDA drivers under Linux can be challenging since the proprietary CUDA drivers do not work well together with the default open-source drivers in Linux, but to train-in SqueezeNet in a reasonable timeframe, GPU acceleration with CUDA is highly recommended.

Afterwards a training dataset can be created using the offline learning toolchain developed in the scope of this thesis (cf. 4.1). For this test a dataset similar to the one expected in reference scenario 2 (ASMET) was created using images obtained from one of the SkyCam cameras (cf. chapter 3.1.2). The images were categorized into three different categories: Birds, Insects and Undefined. The possibility to define shortcuts for each category in the offline learning toolchain proved to be very handy: Within half an hour, more than 300 images could be categorized. Besides the training dataset itself, a second dataset for validation was created. This second dataset contains images categorized with the same categorization scheme as the training dataset but includes only images that are not part of the training dataset. The validation dataset allows to autonomously measure the learning progress during the training phase.

In addition, the neural network architecture has to be adapted to match the created dataset. By default, the SqueezeNet input layer expects images with a size of $227px \times 227px \times 3ch$. In contrast, the images in the self created SkyCam dataset have a size of $640px \times 480px \times 3ch$. Therefore, either the input data layer has to be adapted, which would also require the adaptation of at least one of the following layers to match this new size or to use the scale/crop options in Caffe to let the Caffe framework resize the images before forwarding them to the neural network. In this case, the latter option was used since it does not require redesign of the whole network architecture. On the other hand, downsampling an image always implicates a loss of information. The second adjustment that has to be made concerns the last layer. The original SqueezeNet architecture is designed for 1,000 output categories. However, the experimental SkyCam training dataset only has 3 output categories (bird, insects and undefined). This can easily be corrected by specifying the channel width of the last convolutional layer (conv10) to be
3 instead of 1,000. Yet, it should be kept in mind that such a large change in size could potentially influence the performance of the network. If the training process results in an insufficient precision, it might be necessary to also adapt some layers before the last one.

Since the 300 images included in the self created SkyCam dataset are usually far too few to train-in a complex CNN like SqueezeNet, a technique called fine-tuning is used. Fine-tuning means that the already trained-in weights of a model that were created using similar input data are re-used in the new training process and only the weights of some distinct layers are changed during the learning process with the new dataset. In this case, the model file of SqueezeNet trained-in for the ImageNet challenge was used as a base. Even though the ImageNet dataset may seem completely different at first glance, it should be possible to reuse the weights of the first layers since those layers usually react to features like edges and corners in the image and are thus almost generally applicable.

Another issue that had to be solved was that the neural network was designed to process images as input, but lots of objects in the SkyCam footage look very similar on single images. For example, it is often hard or even impossible to tell whether an image obtained from the SkyCam shows a bird, insect or plane since the main difference between those objects consists in their movement profile. This could either be resolved by using a different network architecture that is capable of processing videos (for example a so-called recurrent convolutional neural network RCNN) or by averaging several subsequent images which leads to a mapping of the movement onto a single image (cf. fig. 5.2). With the latter – rather simple – workaround, the neural network can learn to react to the distinct movement pattern created by an object.

Unfortunately, the tests conducted with the trained-in example model did not work as well as expected. First of all, a lot of different settings within the solver definition had to be tried out by trial and error, since their exact impact on the final model is nearly unpredictable. Furthermore, different network fine-tuning approaches as described before were tried out, starting from re-training only the last layer (conv10) to re-training the whole neural network. Each test took several hours or even days due to the large necessary processing power for training. The resulting network models did either not converge, which basically means that the network did not learn anything since the prediction accuracy is not better than guessing, or the network overfitted, which means that the network did learn to categorize the provided examples but lost its ability to generalize, i.e. to also
categorize images that were not part of the training data. One reason for this may be a far too small training dataset. For example, the popular ILSVRC 2012 dataset for which SqueezeNet was initially developed contains 1,300 images per category, resulting in 1.3 million images in total. Another, often used reference dataset called MNIST contains 60,000 images of handwritten digits, which equals 6,000 images per category. As indicated before, the test dataset used here contained only slightly more than 100 images per category, which could be too few, despite using the fine-tuning technique described before. Other reasons could be inappropriate solver settings or – as already suggested before – the necessity to further adapt the neural network architecture of SqueezeNet for this task. In conclusion, even though some trained-in models provided promising results, building up a large enough training database and finding an optimal network architecture together with the right solver setting to get a neural network capable of solving real world tasks reliably enough is a huge work package on its own that can not be solved satisfyingly within the scope of this thesis.

5.2 Embedded System Tests

When developing on FPGAs, usually a test-driven approach is chosen. Test driven development means that a testbench is written based on the module specifications before the real code for the module is developed. This approach ensures, that each module is tested according to its requirements and specifications instead of just testing the functionality
of the module, which does not necessarily correspond to the requirements. Furthermore, FPGA development requires a bottom-up approach which means that first all small entities are implemented and afterwards gradually put together in larger entities. To test a design on real hardware is usually not possible before all entities are implemented. Therefore, testing with testbenches is even more important when developing on FPGAs in comparison to programming microcontrollers.

This bottom-up approach has also been used in the development of the embedded system for this project. Since the timeframe of the project did not allow to implement the full system, it was not possible to run many tests on the real hardware. However, as indicated before, a test-driven approach was chosen for development which allowed to demonstrate the operability of the developed modules using testbenches. Especially the Convolver Units and the Convolver Cluster, which are some of the key elements of the embedded neuro-processor, could be tested successfully using testbenches.

In figure 5.3, an exemplary excerpt of the waveforms generated in one of the Convolver Cluster testbenches can be seen. The same figure is also depicted enlarged and without annotations in appendix C. In this test, the Convolver Cluster was configured to hold 5 Convolver Units, each Convolver Unit containing $7 \times 7$ multipliers. This is the configuration that would be required to execute SqueezeNet and still would fit on the Altera Cyclone IV FPGA. It should be emphasized that great effort has been put into making the embedded system modular and configurable. Almost all parameters like maximum
kernel size, convolver cluster size, data width of the fixed point decimal representation and the fixed point scaling factor can be adapted by changing just a single constant in a configuration file. The resulting, synthesized design can therefore be optimally adapted to different neural network architectures and the resources available on the selected FPGA. This is, in comparison to programming microcontrollers, a very hard task on FPGAs since hardware description languages are rather inflexible when it comes to abstracted design descriptions.

The input data in the testbench depicted in figure 5.3 had a channel width of 8, resulting in $\lceil \frac{Ch_{in}}{\text{CLUSTER}\_\text{SIZE}} \rceil = \lceil \frac{8}{5} \rceil = 2$ clock cycles necessary to calculate one output value. As one can see from the waveforms, there is a delay between the arrival of the first input data and the availability of the first output. However, after this initial delay, new output values are available every second clock cycle. This is due to the fact, that an implementation was chosen that focuses on high data throughput rather than low latency since data throughput is in this case more important in order to get a fast overall execution speed. The expected output matrix in this test had a size of $6 \times 6 \times 1$. Therefore, it took $6 \times 6 \times 1 \times 2 = 72$ clock cycles to calculate the complete output. In comparison, a system not capable of parallel processing (e.g. a microcontroller) would need $W_{out} \times H_{out} \times Ch_{out} \times Ch_{in} \times K \times K = 6 \times 6 \times 1 \times 8 \times 5 \times 5 = 7,200$ multiply accumulate operations which is, even if one assumes just one clock cycle per operation, far more than with the 72 cycles needed by the developed Convolver Cluster. The correctness of the output values was validated by calculating the same output values with a C++ program that implements the pseudo-code in listing 4.1. The Convolver Units were tested in a similar way and also performed as expected.

It is hard to tell from this tests how fast an artificial neural network will run on the real hardware but using the SqueezeNet analysis in table B.1 together with the network definition in listing B.1 (appendix B), a well-founded guess can be made. The required amount of clock cycles per convolution for every single convolutional layer within SqueezeNet can be calculated – analogously to the example above – with $W_{out} \times H_{out} \times Ch_{out} \times (\lceil \frac{Ch_{in}}{\text{CLUSTER}\_\text{SIZE}} \rceil)$. The results of this calculations are given in table 5.1. As one can see, around 61 million clock cycles are necessary to compute all convolutions. Beside the convolutional layers, there are ReLu, pool and concat layers in SqueezeNet. Concat layers do not really need any computation as already described in chapter 2.2.3. ReLu and pool layers only need comparison (comp) and addition (add).
operations. According to table 5.1, about 9.67 million comp and 225,000 add operations are necessary for the remaining layers. Assuming that every comparison and every addition takes one cycle (what is a very conservative guess since those operations can be parallelized as well), this would result in about 71 million cycles necessary to execute SqueezeNet. Of course, this estimate does not include potential overhead through memory read/write cycles but since the read/write operations should be performed mostly in parallel to the CNN execution, the overhead is estimated to be small. Therefore, it can be assumed that a single execution of the network takes around 0.7 seconds when running the FPGA at 100 MHz (100 million clock cycles per second), 1.4 seconds for 50 MHz or 2.8 seconds for 25 MHz. Note that this estimate was done using the original SqueezeNet network definition with 1000 output classes. When adapting the neural network to the mission specific needs, the results might look different. Furthermore, the drawback of having to oversize the Convolver Units in the chosen approach takes particularly effect in this network architecture since the kernel sizes are very heterogeneous. The first convolution (conv1) has a kernel size of $7 \times 7$ resulting in a necessary $K_{\text{max}}$ of 7, whereas all other convolutions in this network have only a kernel size of $3 \times 3$ or $1 \times 1$. If the kernel sizes would be more homogenous, a lot of resources within the Convolver Units could be used more efficiently resulting in a notable speed up. For example, there is now a second version of SqueezeNet available (SqueezeNet 1.1) with similar performance but with a maximum kernel size of 5 which would allow to have a cluster size of 10 instead of 5 on the same FPGA and therefore almost double the execution speed (0.42 seconds at 100 MHz).

In addition to testbench tests, large parts of the SDRAM interface as well as the UART interface could even be tested on the real hardware by writing a short demo program which writes the data received over UART on the SDRAM and afterwards loads this data again from the SDRAM to write it back to the host via UART. This test successfully validated the datapath that will later be needed to transfer the image data over UART into the memory and – in large parts – also the datapath between SDRAM and neural network since it relies on the same read/write mechanisms.

Parts which now have to be developed in order to be able to do a full end-to-end test are the Control Unit and other neural network layers like ReLu, Pooling and Concat. However, these other layers are far less complicated than a convolutional layer. Therefore, it is valid to say that large parts of the embedded system have been completed and tested within this thesis.
Table 5.1: Runtime Estimation Of SqueezeNet.

<table>
<thead>
<tr>
<th>Ch_{in}</th>
<th>\left\lceil \frac{Ch_{in}}{5} \right\rceil</th>
<th>Ch_{out}</th>
<th>W_{out}</th>
<th>H_{out}</th>
<th>Clk Cycles</th>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>96</td>
<td>111</td>
<td>111</td>
<td>1,182,816</td>
<td>conv1</td>
</tr>
<tr>
<td>96</td>
<td>20</td>
<td>16</td>
<td>55</td>
<td>55</td>
<td>968,000</td>
<td>fire2</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>55</td>
<td>55</td>
<td>774,400</td>
<td>fire2</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>55</td>
<td>55</td>
<td>774,400</td>
<td>fire2</td>
</tr>
<tr>
<td>128</td>
<td>26</td>
<td>16</td>
<td>55</td>
<td>55</td>
<td>1,258,400</td>
<td>fire3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>55</td>
<td>55</td>
<td>774,400</td>
<td>fire3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>55</td>
<td>55</td>
<td>774,400</td>
<td>fire3</td>
</tr>
<tr>
<td>128</td>
<td>26</td>
<td>32</td>
<td>55</td>
<td>55</td>
<td>2,516,800</td>
<td>fire4</td>
</tr>
<tr>
<td>32</td>
<td>7</td>
<td>128</td>
<td>55</td>
<td>55</td>
<td>2,710,400</td>
<td>fire4</td>
</tr>
<tr>
<td>32</td>
<td>7</td>
<td>128</td>
<td>55</td>
<td>55</td>
<td>2,710,400</td>
<td>fire4</td>
</tr>
<tr>
<td>256</td>
<td>52</td>
<td>32</td>
<td>27</td>
<td>27</td>
<td>1,213,056</td>
<td>fire5</td>
</tr>
<tr>
<td>32</td>
<td>7</td>
<td>128</td>
<td>27</td>
<td>27</td>
<td>653,184</td>
<td>fire5</td>
</tr>
<tr>
<td>32</td>
<td>7</td>
<td>128</td>
<td>27</td>
<td>27</td>
<td>653,184</td>
<td>fire5</td>
</tr>
<tr>
<td>256</td>
<td>52</td>
<td>48</td>
<td>27</td>
<td>27</td>
<td>1,819,584</td>
<td>fire6</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>192</td>
<td>27</td>
<td>27</td>
<td>1,399,680</td>
<td>fire6</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>192</td>
<td>27</td>
<td>27</td>
<td>1,399,680</td>
<td>fire6</td>
</tr>
<tr>
<td>384</td>
<td>77</td>
<td>48</td>
<td>27</td>
<td>27</td>
<td>2,694,384</td>
<td>fire7</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>192</td>
<td>27</td>
<td>27</td>
<td>1,399,680</td>
<td>fire7</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>192</td>
<td>27</td>
<td>27</td>
<td>1,399,680</td>
<td>fire7</td>
</tr>
<tr>
<td>384</td>
<td>77</td>
<td>64</td>
<td>27</td>
<td>27</td>
<td>3,592,512</td>
<td>fire8</td>
</tr>
<tr>
<td>64</td>
<td>13</td>
<td>256</td>
<td>27</td>
<td>27</td>
<td>2,426,112</td>
<td>fire8</td>
</tr>
<tr>
<td>64</td>
<td>13</td>
<td>256</td>
<td>27</td>
<td>27</td>
<td>2,426,112</td>
<td>fire8</td>
</tr>
<tr>
<td>512</td>
<td>103</td>
<td>64</td>
<td>13</td>
<td>13</td>
<td>1,114,048</td>
<td>fire9</td>
</tr>
<tr>
<td>64</td>
<td>13</td>
<td>256</td>
<td>13</td>
<td>13</td>
<td>562,432</td>
<td>fire9</td>
</tr>
<tr>
<td>64</td>
<td>13</td>
<td>256</td>
<td>13</td>
<td>13</td>
<td>562,432</td>
<td>fire9</td>
</tr>
<tr>
<td>512</td>
<td>103</td>
<td>1,000</td>
<td>15</td>
<td>15</td>
<td>23,175,000</td>
<td>conv10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60,935,176</td>
<td>SUM</td>
</tr>
</tbody>
</table>
It is obvious, that there is still a lot of work left to be done. However, as already mentioned before, it was never the goal of this project to develop a fully working system but rather to build up knowledge, conduct tests and elaborate the usefulness and feasibility of embedded neural network systems in space applications. It has been shown that the potential of this technique for on-board data processing is very large. In addition to some possible applications briefly described in the introduction of this thesis, two reference scenarios that are of special interest to the chair of Aerospace Computer Science at the University of Würzburg were elaborated in more detail. Furthermore, existing products and approaches have been analyzed in detail and a complete system design has been developed upon the gained knowledge. Even though the feasibility of developing a neuro-processor with the specified requirements cannot be finally proved, the test conducted with the sample implementation suggest that building such a system is not only possible but also opens completely new capabilities for autonomous spacecraft.

As a next step, the remaining parts of the system developed within this thesis should be implemented and tested. On the embedded side, this is, as already mentioned, the Control Unit as well as some additional layer types like ReLu, pooling or concat. Furthermore, different network architectures should be evaluated to achieve a system that converges better during training. In addition, it should be considered to switch the hardware platform for execution. Even though the convolution operation could be accelerated a lot using the parallel processing capabilities on the FPGA, lots of other subsystems that are necessary to build up a complete system are unnecessarily hard to implement in hardware. Therefore a system combining a microcontroller for high level interaction and a FPGA for accelerating the neural network would be ideal to implement such an embedded
neuroprocessor. Even though it is also possible to instantiate a microprocessor within the Cyclone IV FPGA used in this project, such so-called soft-core processors always consume much more resources compared to a microprocessor implemented in hardware, leaving less resources for the neural network accelerator.

Fortunately, nowadays so-called system-on-chips (SoCs) are available combining the best of both worlds: A hardware microprocessor paired with FPGA fabric. One of those systems is the Microsemi SmartFusion 2 SoC which has already been used here at University Würzburg in a star sensor developed within project AROS (cf. [19]). Even though SmartFusion 2 SoCs target mainly low-power applications and are therefore comparatively weak, it is worth evaluating if these chips are powerful enough to execute CNNs in reasonable time. An additional reason for the use of SmartFusion 2 SoCs is their eligibility for pico and nano satellites due to their low power consumption and promise for high reliability under harsh conditions, one of the strong points of Microsemi chips.

Another option that is worth some consideration, especially regarding a potential use in ASMET, is a combined system of a dedicated microcontroller and FPGA. For example, there are various FPGA extension boards available for the popular Raspberry Pi single-board computer. Since there have already been other subsystems developed on a Raspberry Pi in the scope of bachelor and master projects that might be used in ASMET, it would be a logical next step to further evaluate whether a Raspberry Pi FPGA extension board can be used as neural network accelerator to allow a fast and energy efficient execution of neural networks and still have the flexibility and ease of use of the Raspberry Pi.

In conclusion, this thesis revealed some exciting new possibilities embedded neural networks can bring to the space sector. It can be understood as a starting point which will hopefully encourage further research in this interesting topic of artificial neural networks to increase autonomy in spaceflight.


This page is intentionally left blank.
The main goal of the project is to develop an embedded neural network system, which is capable of semantic image classification, i.e. it has to be capable of recognizing predefined patterns or objects in a given image, under conditions as found on a small satellite platform. Detailed user requirements can be found in table A.1, system requirements in tables A.2 and A.3, other requirements in table A.4.
Table A.1: User Requirements.

<table>
<thead>
<tr>
<th>No.</th>
<th>Requirement</th>
<th>Value/Remark</th>
<th>Priority</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>UR-0100</td>
<td>The system must be able to recognize predefined patterns and objects in a given image.</td>
<td>-</td>
<td></td>
<td>Project Statement</td>
</tr>
<tr>
<td>UR-0200</td>
<td>The system must be able to work under conditions found on a small satellite, especially regarding power and space.</td>
<td>-</td>
<td></td>
<td>Project Statement</td>
</tr>
<tr>
<td>UR-0300</td>
<td>It should be possible to update the set of patterns and objects that can be recognized by the system.</td>
<td>High</td>
<td></td>
<td>Usability</td>
</tr>
<tr>
<td>UR-0400</td>
<td>The system should be modular, i.e. it should be possible to scale it (on ground) to different requirements like low-power/low-performance or high-power/high-performance.</td>
<td>Medium</td>
<td></td>
<td>(Re-) Usability</td>
</tr>
<tr>
<td>UR-0500</td>
<td>The system should be usable (i.e. configurable and trainable) for technicians and scientists who are not experts in NN.</td>
<td>High</td>
<td></td>
<td>(Re-) Usability</td>
</tr>
<tr>
<td>UR-0600</td>
<td>The system must be trainable in reasonable time, i.e. training should not take longer than a few days.</td>
<td>-</td>
<td></td>
<td>Usability</td>
</tr>
</tbody>
</table>
Table A.2: System Requirements - Requirements on the Embedded System.

<table>
<thead>
<tr>
<th>No.</th>
<th>Requirement</th>
<th>Value/Remark</th>
<th>Priority</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR-ES-0100</td>
<td>The system must consume less than [] power.</td>
<td>2 W</td>
<td>-</td>
<td>UR-0200</td>
</tr>
<tr>
<td>SR-ES-0150</td>
<td>The system should consume less than [] power.</td>
<td>500 mW</td>
<td>High</td>
<td>UR-0200</td>
</tr>
<tr>
<td>SR-ES-0200</td>
<td>The system should be implemented with as few electrical components as possible.</td>
<td></td>
<td>High</td>
<td>UR-0200, SR-ES-0100/0150</td>
</tr>
</tbody>
</table>

Table A.3: System Requirements - Requirements on the Neural Network Architecture.

<table>
<thead>
<tr>
<th>No.</th>
<th>Requirement</th>
<th>Value/Remark</th>
<th>Priority</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR-NN-0100</td>
<td>The NN should have no fully-interconnected layers.</td>
<td>Reduces memory consumption significantly, In-orbit update possible and no/less ext. memory</td>
<td>High</td>
<td>UR-0300, SR-ES-0100/0150</td>
</tr>
<tr>
<td>SR-NN-0200</td>
<td>The NNs accuracy should be similar to AlexNet or better.</td>
<td>ImageNet contest winner 2012, acceptable accuracy for real-world-problems</td>
<td>High</td>
<td>UR-0100</td>
</tr>
<tr>
<td>SR-NN-0300</td>
<td>The system does not need to support back-propagation, i.e. no online-training capability.</td>
<td>Training will be performed on ground, reduces complexity significantly</td>
<td>-</td>
<td>UR-0200</td>
</tr>
</tbody>
</table>
### Table A.4: Other Requirements.

<table>
<thead>
<tr>
<th>No.</th>
<th>Requirement</th>
<th>Value/Remark</th>
<th>Priority</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR-O-0100</td>
<td>An existing framework should be used for NN training.</td>
<td>Fast training utilizing CPU+GPU, tested, less dev. effort</td>
<td>High</td>
<td>UR-0600, UR-0300</td>
</tr>
<tr>
<td>SR-O-0200</td>
<td>A rudimentary tool with GUI should be developed to guide the user through the necessary steps for training and setup.</td>
<td>If no convenient ready-to-use tool can be found, most NN frameworks just code/libraries or cmd-line</td>
<td>Medium</td>
<td>UR-0500</td>
</tr>
<tr>
<td>SR-O-0300</td>
<td>Developed code should be documented or commented.</td>
<td></td>
<td>High</td>
<td>UR-0500, Reusability</td>
</tr>
</tbody>
</table>
SqueezeNet consists, besides of ordinary pool and ReLu layers, of several so-called fire layers, an invention by the creators of SqueezeNet. Fire layers are basically an arrangement of 3 convolution layers together with a concat layer. The first convolution layer (squeeze1x1) reduces the channel width, afterwards the output of the squeeze layer is fed into two further convolution layers (expand1x1 and expand3x3), which again broaden the channel width. In the concat layer, the outputs of those two expand layers are merged together. The difference between the two expand layers is that they have different kernel sizes (1x1 and 3x3). The two convolutional layers together with the concat layer could also be replaced by a single convolutional layer with varying kernel size but since this is not supported by Caffe, the workaround described before is necessary. Up to now, this workaround is also used in the embedded implementation. In a later development stage, a dedicated fire layer processing unit may be developed to speed up execution of SqueezeNet. A table listing the different layers and their resource utilization (necessary operations and memory for feature maps (fm) and weights) as well as a graph visualizing the structure of SqueezeNet can be found below. In this table and graph, SqueezeNet was configured to solve the ImageNet challenge, a dataset with input images of size 227 x 227 x 3 (RGB) and 1,000 different output classes. If one wants to configure SqueezeNet to solve different tasks, the first and last layers have to be adapted accordingly.
Table B.1: SqueezeNet Layers and Resource Utilization (Source: Netscope CNN Analyzer).  

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>Dim In</th>
<th>Dim Out</th>
<th>Ops</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data</td>
<td>Data</td>
<td>3 x 227 x 227</td>
<td>3 x 227 x 227</td>
<td>fm 154.59k</td>
<td>1.18M</td>
</tr>
<tr>
<td>2</td>
<td>conv1</td>
<td>Conv</td>
<td>3 x 227 x 227</td>
<td>96 x 111 x 111</td>
<td>MAC 173.87M</td>
<td>1.18M</td>
</tr>
<tr>
<td>3</td>
<td>relu_conv1</td>
<td>ReLu</td>
<td>96 x 111 x 111</td>
<td>96 x 111 x 111</td>
<td>COMP 1.18M</td>
<td>1.18M</td>
</tr>
<tr>
<td>4</td>
<td>pool1</td>
<td>Pool</td>
<td>96 x 111 x 111</td>
<td>96 x 55 x 55</td>
<td>COMP 2.61M</td>
<td>290.4k</td>
</tr>
<tr>
<td>5</td>
<td>fire2</td>
<td>Fire</td>
<td>96 x 55 x 55</td>
<td>128 x 55 x 55</td>
<td>MAC 35.62M</td>
<td>1.26M</td>
</tr>
<tr>
<td>12</td>
<td>fire3</td>
<td>Fire</td>
<td>128 x 55 x 55</td>
<td>128 x 55 x 55</td>
<td>COMP 1.18M</td>
<td>1.26M</td>
</tr>
<tr>
<td>19</td>
<td>fire4</td>
<td>Fire</td>
<td>128 x 55 x 55</td>
<td>256 x 55 x 55</td>
<td>MAC 136.29M</td>
<td>1.26M</td>
</tr>
<tr>
<td>26</td>
<td>pool4</td>
<td>Pool</td>
<td>256 x 55 x 55</td>
<td>256 x 27 x 27</td>
<td>COMP 1.18M</td>
<td>1.26M</td>
</tr>
<tr>
<td>27</td>
<td>fire5</td>
<td>Fire</td>
<td>256 x 27 x 27</td>
<td>256 x 27 x 27</td>
<td>MAC 136.29M</td>
<td>1.26M</td>
</tr>
<tr>
<td>34</td>
<td>fire6</td>
<td>Fire</td>
<td>256 x 27 x 27</td>
<td>384 x 27 x 27</td>
<td>MAC 309.56M</td>
<td>1.26M</td>
</tr>
<tr>
<td>41</td>
<td>fire7</td>
<td>Fire</td>
<td>384 x 27 x 27</td>
<td>384 x 27 x 27</td>
<td>MAC 76.14M</td>
<td>1.26M</td>
</tr>
<tr>
<td>48</td>
<td>fire8</td>
<td>Fire</td>
<td>384 x 27 x 27</td>
<td>512 x 27 x 27</td>
<td>MAC 136.29M</td>
<td>1.26M</td>
</tr>
<tr>
<td>55</td>
<td>pool8</td>
<td>Pool</td>
<td>512 x 27 x 27</td>
<td>512 x 13 x 13</td>
<td>COMP 778.75k</td>
<td>1.26M</td>
</tr>
<tr>
<td>56</td>
<td>fire9</td>
<td>Fire</td>
<td>512 x 13 x 13</td>
<td>512 x 13 x 13</td>
<td>MAC 33.23M</td>
<td>1.26M</td>
</tr>
<tr>
<td>63</td>
<td>drop9</td>
<td>Dropout</td>
<td>512 x 13 x 13</td>
<td>512 x 13 x 13</td>
<td>COMP 86.53k</td>
<td>1.26M</td>
</tr>
<tr>
<td>64</td>
<td>conv10</td>
<td>Conv</td>
<td>512 x 13 x 13</td>
<td>1000 x 15 x 15</td>
<td>MAC 115.2M</td>
<td>1.26M</td>
</tr>
<tr>
<td>65</td>
<td>relu_conv10</td>
<td>ReLu</td>
<td>1.000 x 15 x 15</td>
<td>1.000 x 15 x 15</td>
<td>COMP 225k</td>
<td>1.26M</td>
</tr>
<tr>
<td>66</td>
<td>pool10</td>
<td>Pool</td>
<td>1.000 x 15 x 15</td>
<td>1.000 x 1 x 1</td>
<td>ADD 225k</td>
<td>1.26M</td>
</tr>
</tbody>
</table>

**TOTAL**  
MAC 861.34M  
COMP 9.67M  
ADD 225k  
fm 12.73M  
weights 1.24M

---

1https://dgschwend.github.io/netscope/#/preset/squeezenet
Listing B.1: SqueezeNet Network Definition.

input: "data"
input_shape {
  dim: 10
  dim: 3
  dim: 227
  dim: 227
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 96
    kernel_size : 7
    stride : 2
  }
}
layer {
  name: "relu_conv1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size : 3
    stride : 2
  }
}
layer {
  name: "fire2/squeeze1x1"
  type: "Convolution"
  bottom: "pool1"
  top: "fire2/squeeze1x1"
  convolution_param {
    num_output: 16
    kernel_size : 1
  }
}
layer {
  name: "fire2/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire2/squeeze1x1"
  top: "fire2/squeeze1x1"
}
layer {
  name: "fire2/expand1x1"
  type: "Convolution"
  bottom: "fire2/squeeze1x1"
  top: "fire2/expand1x1"
  convolution_param {
    num_output: 64
    kernel_size : 1
  }
}
layer {
  name: "fire2/relu_expand1x1"
  type: "ReLU"
  bottom: "fire2/expand1x1"
  top: "fire2/expand1x1"
}
layer {
  name: "fire2/expand3x3"
  type: "Convolution"
  bottom: "fire2/squeeze1x1"
  top: "fire2/expand3x3"
  convolution_param {
}
layer {
  name: "fire4/squeeze1x1"
type: "Convolution"
bottom: "fire3/concat"
top: "fire4/squeeze1x1"
convolution_param {
  num_output: 32
  kernel_size : 1
}
}
layer {
  name: "fire4/relu_squeeze1x1"
type: "ReLU"
bottom: "fire4/squeeze1x1"
top: "fire4/squeeze1x1"
}
layer {
  name: "fire4/expand1x1"
type: "Convolution"
bottom: "fire4/squeeze1x1"
top: "fire4/expand1x1"
convolution_param {
  num_output: 128
  kernel_size : 1
}
}
layer {
  name: "fire4/relu_expand1x1"
type: "ReLU"
bottom: "fire4/expand1x1"
top: "fire4/expand1x1"
}
layer {
  name: "fire4/expand3x3"
type: "Convolution"
bottom: "fire4/squeeze1x1"
top: "fire4/expand3x3"
convolution_param {
  num_output: 128
  pad: 1
  kernel_size : 3
}
}
layer {
  name: "fire4/relu_expand3x3"
type: "ReLU"
bottom: "fire4/expand3x3"
top: "fire4/expand3x3"
}
layer {
  name: "fire4/concat"
type: "Concat"
bottom: "fire4/expand1x1"
bottom: "fire4/expand3x3"
top: "fire4/concat"
}
layer {
  name: "pool4"
type: "Pooling"
bottom: "fire4/concat"
top: "pool4"
pooling_param {
  pool: MAX
  kernel_size : 3
  stride : 2
}
}
layer {
  name: "fire5/squeeze1x1"
type: "Convolution"
bottom: "pool4"
top: "fire5/squeeze1x1"
convolution_param {
  num_output: 32
}
kernel_size : 1
}
}
layer {
  name: "fire5/relu.squeeze1x1"
type: "ReLU"
bottom: "fire5/squeeze1x1"
top: "fire5/squeeze1x1"
}
layer {
  name: "fire5/expand1x1"
type: "Convolution"
bottom: "fire5/squeeze1x1"
top: "fire5/expand1x1"
convolution_param {
  num_output: 128
  kernel_size : 1
}
}
layer {
  name: "fire5/relu.expand1x1"
type: "ReLU"
bottom: "fire5/expand1x1"
top: "fire5/expand1x1"
}
layer {
  name: "fire5/expand3x3"
type: "Convolution"
bottom: "fire5/squeeze1x1"
top: "fire5/expand3x3"
convolution_param {
  num_output: 128
  pad: 1
  kernel_size : 3
}
}
layer {
  name: "fire5/relu.expand3x3"
type: "ReLU"
bottom: "fire5/expand3x3"
top: "fire5/expand3x3"
}
layer {
  name: "fire5/concat"
type: "Concat"
bottom: "fire5/expand1x1"
bottom: "fire5/expand3x3"
top: "fire5/concat"
}
layer {
  name: "fire6/squeeze1x1"
type: "Convolution"
bottom: "fire5/concat"
top: "fire6/squeeze1x1"
convolution_param {
  num_output: 48
  kernel_size : 1
}
}
layer {
  name: "fire6/relu.squeeze1x1"
type: "ReLU"
bottom: "fire6/squeeze1x1"
top: "fire6/squeeze1x1"
}
layer {
  name: "fire6/expand1x1"
type: "Convolution"
bottom: "fire6/squeeze1x1"
top: "fire6/expand1x1"
convolution_param {
  num_output: 192
  kernel_size : 1
}
layer {
  name: "fire6/relu_expand1x1"
  type: "ReLU"
  bottom: "fire6/expand1x1"
  top: "fire6/expand1x1"
}
layer {
  name: "fire6/expand3x3"
  type: "Convolution"
  bottom: "fire6/squeeze1x1"
  top: "fire6/expand3x3"
  convolution_param {
    num_output: 192
    pad: 1
    kernel_size: 3
  }
}
layer {
  name: "fire6/relu_expand3x3"
  type: "ReLU"
  bottom: "fire6/expand3x3"
  top: "fire6/expand3x3"
}
layer {
  name: "fire6/squeeze1x1"
  type: "Convolution"
  bottom: "fire6/concat"
  top: "fire6/squeeze1x1"
  convolution_param {
    num_output: 48
    kernel_size: 1
  }
}
layer {
  name: "fire7/relu_squeeze1x1"
  type: "ReLU"
  bottom: "fire7/squeeze1x1"
  top: "fire7/squeeze1x1"
}
layer {
  name: "fire7/expand1x1"
  type: "Convolution"
  bottom: "fire7/squeeze1x1"
  top: "fire7/expand1x1"
  convolution_param {
    num_output: 192
    kernel_size: 1
  }
}
layer {
  name: "fire7/relu_expand1x1"
  type: "ReLU"
  bottom: "fire7/expand1x1"
  top: "fire7/expand1x1"
}
layer {
  name: "fire7/expand3x3"
  type: "Convolution"
  bottom: "fire7/squeeze1x1"
  top: "fire7/expand3x3"
  convolution_param {
    num_output: 192
    pad: 1
    kernel_size: 3
  }
}
layer {
  name: "fire7/relu_expand3x3"
  type: "ReLU"
  bottom: "fire7/expand3x3"
  top: "fire7/expand3x3"
}
layer {
  name: "fire7/concat"
  type: "Concat"
  bottom: "fire6/expand1x1"
  bottom: "fire6/expand3x3"
  top: "fire6/concat"
}
layer {
  name: "fire7/squeeze1x1"
  type: "Convolution"
  bottom: "fire6/concat"
  top: "fire7/squeeze1x1"
  convolution_param {
    num_output: 192
    pad: 1
    kernel_size: 3
  }
}
layer {  
  name: "fire7/relu_expand3x3"  
  type: "ReLU"  
  bottom: "fire7/expand3x3"  
  top: "fire7/expand3x3"  
}  
layer {  
  name: "fire7(concat)"  
  type: "Concat"  
  bottom: "fire7/expand1x1"  
  bottom: "fire7/expand3x3"  
  top: "fire7/concat"  
}  
layer {  
  name: "fire8/squeeze1x1"  
  type: "Convolution"  
  bottom: "fire7/concat"  
  top: "fire8/squeeze1x1"  
  convolution_param {  
    num_output: 64  
    kernel_size : 1  
  }  
}  
layer {  
  name: "fire8/relu_squeeze1x1"  
  type: "ReLU"  
  bottom: "fire8/squeeze1x1"  
  top: "fire8/squeeze1x1"  
}  
layer {  
  name: "fire8/expand1x1"  
  type: "Convolution"  
  bottom: "fire8/squeeze1x1"  
  top: "fire8/expand1x1"  
  convolution_param {  
    num_output: 256  
    pad: 1  
    kernel_size : 3  
  }  
}  
layer {  
  name: "fire8/relu_expand1x1"  
  type: "ReLU"  
  bottom: "fire8/expand1x1"  
  top: "fire8/expand1x1"  
}  
layer {  
  name: "fire8/expand3x3"  
  type: "Convolution"  
  bottom: "fire8/expand1x1"  
  top: "fire8/expand3x3"  
}  
layer {  
  name: "pool8"  
  type: "Pooling"  
  bottom: "fire8/concat"  
  top: "pool8"  
}
bottom: "fire9/concat"

convolution_param {
  num_output: 1000
  pad: 1
  kernel_size : 1
}

layer {
  name: "relu_conv10"
  type: "ReLU"
  bottom: "conv10"
  top: "conv10"
}

layer {
  name: "pool10"
  type: "Pooling"
  bottom: "conv10"
  top: "pool10"
  pooling_param {
    pool: AVE
    global_pooling: true
  }
}

layer {
  name: "prob"
  type: "Softmax"
  bottom: "pool10"
  top: "prob"
}
APPENDIX C

Enlarged ModelSim Waveform

The enlarged screenshot of the ModelSim waveform can be found on the next page.
Figure C.1: Enlarged Screenshot of the Waveforms of One of the Convolver Cluster Testbenches in ModelSim.
I hereby affirm that the thesis at hand is independent work of myself and that work of others used herein is attributed as such. Furthermore, this thesis was neither submitted as whole, nor in part, to another board of examination in pursuit of attainment of an academic degree.

Würzburg, 15.10.2016

Place, Date

Andreas Schartel