Inference Acceleration For Neural Networks Using External Hardware

Lukas Gutbrunner

Leopold Franzens Universität Innsbruck

Besi Austria GmbH

supervised by
Karin Schnass & Michael Sandbichler

October 7, 2019

Contents

1 Introduction

This bachelor thesis is about neural networks for computer vision tasks and the time needed to run them. Larger models need more computations and therefore have longer runtimes. This is a problem for time-critical tasks like security cameras or self-driving cars. A solution for this problem could be external hardware, optimized for these computations. Here, external hardware is benchmarked and compared to ordinary CPUs for different tasks in computer vision.
In the course of the thesis, we describe and benchmark different models and analyze the resulting times.

2 Setting

We first provide a short introduction to neural networks. Additionally, the problem with bigger networks on small computers is described and solution strategies are discussed.

2.1 Neural Networks

Neural networks are a sub-field of AI[1]-research. They consist of many parameters and operations. These operations are rather elementary and therefore can be computed fast individually.
They are motivated by neuroscience, where Donald Hebb postulated that neurons in the brain grow stronger connections if they work together. If one neuron is activated when another one gives off an impulse, their connection strengthens. These neurons and connections are modelled mathematically by taking linear combinations of some input vector. The weights of these linear combinations first were calculated by hand, and later, with more sophisticated models, could be implemented such that the model can learn the right parameters from data. One reason for a mathematical modelling of this biological postulate was the working example of the brain. Further, if the models work like a brain, it could give insight into its biological counterpart, as mathematical models can be examined more easily than the human brain [11].

One simple example of a neural network is the **multilayer perceptron**, which consists of matrix-vector multiplications, vector additions and element-wise use of functions σ_ℓ after the ℓ-th multiplication. In practice, the functions σ_ℓ are non-linear functions to break the linearity that would result from a series of matrix multiplications. This leads to better approximations [11].

[1] "The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages" [15]

Definition 2.1: Multilayer Perceptron (MLP)
Let $m_i \in \mathbb{N}$ for $i = 1, \dots, \ell, x \in \mathbb{R}^{m_0}, \sigma_i \in C(\mathbb{R}), b_i \in \mathbb{R}^{m_i}$ and $A_i : \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}$ be a linear mapping.
For this, σ_i will be applied to $x \in \mathbb{R}^{m_i}$ via

$$\begin{aligned}\tilde{\sigma_i} :&\mathbb{R}^{m_i} \to \mathbb{R}^{m_i},\\ &x \mapsto (\sigma_i(x_1), \sigma_i(x_2), \dots, \sigma_i(x_{m_i}))\end{aligned}$$

From now on, this function $\tilde{\sigma_i}$ will also be denoted with σ_i. We call a function of the form

$$\begin{aligned}f :&\mathbb{R}^{m_0} \to \mathbb{R}^{m_\ell},\\ &x \mapsto \sigma_\ell(A_\ell(\sigma_{\ell-1}(A_{\ell-1}(\dots(\sigma_1(A_1(x) + b_1))\dots) + b_{\ell-1})) + b_\ell)\end{aligned}$$

a multilayer perceptron(MLP).
The function $\sigma_i \circ A_i$ is called the i-th layer of the MLP and m_i the number of its neurons.
The functions σ_i are called activation functions.
The total number of layers ℓ is called the depth.

One of the most used activation functions σ is the so called *ReLU*, that simply is the identity for $x \geq 0$ and 0 everywhere else.
It is easy to see, that when using functions σ_i that are fast to compute, the MLP f can also be computed efficiently.
It is also evident that the amount of neurons in one layer represents the number of linear combinations produced with the input of this layer. Because in one layer, every input neuron is connected to every output neuron, these layers are called fully connected layers.

Another popular activation function is the softmax [11, pp. 77] function.

Definition 2.2: softmax function
Let $x \in \mathbb{R}^n$. The softmax function softmax : $\mathbb{R}^n \to \mathbb{R}^n$ *is defined by*

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

It is easy to see that the output vector has entries between zero and one, with all entries adding up to one. It is mostly used as the activation function in the last layer of image classification models. It is used here due to its property as a continuous approximation of the argmax function: The softmax function approximates a multinoulli distribution, giving a percentage at each entry. These percentages are more intuitive to use than normal linear combinations, as they can be interpreted as the confidence a neural network has into a prediction. It is different to the ReLU function, in that it is not applied element-wise, but rather takes a vector, and outputs another vector containing the probabilities. Although the model seems rather simple, a multilayer perceptron with enough neurons and suitable non-linear functions σ_i can represent a given function arbitrarily well. We have the following theorem due to G. Cybenko [6].

Theorem 2.3: Cybenko
Let I_n denote the unit cube, $\alpha_j, \theta_j \in \mathbb{R}$ and $y_j \in \mathbb{R}^n$. Let σ be any continuous sigmoidal function. Here, σ is called sigmoidal, when $\lim_{t\to-\infty} \sigma(t) = 0$ and $\lim_{t\to\infty} \sigma(t) = 1$. Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \epsilon$$

for all $x \in I_n$.

Models of the above form and variants of it can be used for a broad field of applications [9] [10].
However, there is a major drawback: This result is purely theoretical and gives no information about the nature of the parameters α_i, y_i and θ_i. In practice, a model consists of a definition of the layers used, called the *architecture* and its parameters, called the *weights*. While the architecture of a model is fixed from the beginning, the best weights have to be learned to get a good approximation.
To find the best parameters, statistical estimates for the error made with the current parameters are made. To do so, a metric has to be specified, by which the performance of the model is measured. A popular choice is the Euclidean distance between predicted output and expected output. Then an optimizer is used to compute better parameters based on the expected error of the model. One inconvenience here is that knowledge of the expected error requires one to know the true distribution of the input. As this is practically impossible, empirical estimates are used. This leads to the transition from optimization problem (1) to (2):

$$\min_{f\in\Omega} \mathbb{E}(f(x) - y) \tag{1}$$

$$\min_{f\in\Omega} \frac{1}{n} \sum_{i=1}^{n} f(x_i) - y_i \tag{2}$$

Here, Ω denotes the set of functions defined by the network architecture, x and x_i denote the inputs and y and y_i the corresponding desired outputs.
As the errors now are estimates, more samples to train the model (i.e. optimizing the weights based on the training samples) lead to better models, as the error-approximations

get better [11].

For more specific tasks like image recognition, a more distinct type of linear function, compared to arbitrary matrix vector multiplications is used, namely convolutions. These are more effective in finding patterns and additionally can be defined with less parameters: The main idea is that similar features can occur at different regions in one image. For example, looking at the picture of a seven, one can see multiple horizontal lines at different positions. If a MLP would want to learn that there is more than one horizontal edge, it would need to learn the right parameters for every location. This is because the parameters in one layer do not share information with each other. With a convolution, there are fewer parameters, but they are shared across the layer. This parameter sharing leads to a translation invariance positively influencing the image recognition. Mathematically speaking, a convolution of a matrix with another matrix describes the following operation:

Definition 2.4: Convolution
Given a matrix $I \in \mathbb{R}^{n \times m}$ *and a matrix* $K \in \mathbb{R}^{(2a+1)\times(2b+1)}$ *with* $2a+1 \leq n$ *and* $2b+1 \leq m$. *Then the convolution of* I *with* K *or in symbols:* $I * K \in \mathbb{R}^{n-2a \times m-2b}$ *is given by*

$$(I * K)(i,j) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} K(s,t) I(i-s, j-t)$$

with $I(i,j) = 0$ *for* $i,j < 0$ *or* $i > n, j > m$. *The matrix* K *is called a kernel or a filter. Note that for this definition, the entry with indices* $(0,0)$ *indicates the center of the matrix.*

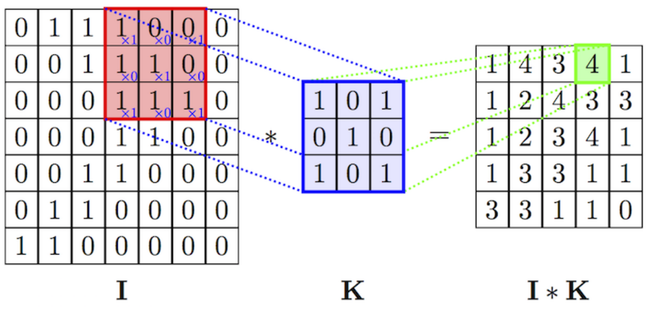

Figure 1: Visualization of a convolution [2]

Now, when one kernel for feature detection is learned, a convolution with this kernel returns all occurrences of this feature. It is not difficult to see, that this operation

[2]Picture from: https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution; 2019.06.26

can profit immensely from proper implementation on processors: Every entry of the output is the same linear combination of the entries of the input overlapping with the kernel.

Additionally, fast implementations make use of the Fourier convolution theorem [13, p. 256-258]:

Theorem 2.5: Fourier Convolution Theorem

Let f, g be two functions, $\mathcal{F}(f), \mathcal{F}(g)$ their Fourier transforms and $f * g$ their convolution. Then

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

with $\cdot$ denoting pointwise multiplication of two functions.

With this, $f * g$ can be computed as follows:

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))$$

Note that $\mathcal{F}^{-1}$ is the inverse Fourier transform. Also note that multiplications in the frequency domain correspond to convolutions with periodic input. To avoid the problems occuring with non periodic inputs, one can simply extend the input with a sufficient amount of zeros in each direction. Now, the matrix $I \in \mathbb{R}^{m \times n}$ can be reshaped into a vector $v \in \mathbb{R}^{mn}$ and the kernel $K \in \mathbb{R}^{p \times q}$ rewritten into a circulant matrix $M \in \mathbb{R}^{mn \times mn}$. This matrix can be obtained by simply observing which linear combinations of the vector v are computed at certain positions in a normal convolution. In this form, it follows that the complexity of $M * K$ is $\mathcal{O}(mnpq)$. When using the Fourier convolution theorem, one gets a complexity of $\mathcal{O}(nm \log(mn))$. Especially with large kernels, this can greatly improve computation time.
For better properties when optimizing models using convolutions, more kernels are used on the same input. This also leads to different features being extracted at once by the different kernels, which are then saved in different matrices. Additionally, kernels of different sizes allow to extract features of different complexity, so it is desirable to use kernels with different sizes. Also note that in a standard convolution as described above, the entries of the matrix I positioned in the middle of the matrix contribute to more outputs than the entries in the first and last column and row. To counter this, the matrix can be extended (for example with zeros or periodically). In doing so, a matrix is created whose convolution uses every entry equally often.

Notation 2.6: Convolutional Layer

A convolutional layer describes the use of one or more convolutions on a given input. A convolutional layer is written in the form:

Conv(kernels, kernel-shape, padding)

with the arguments meaning:

kernels: number of different kernels to use in this layer
kernel-shape: tuple containing height and width of the kernels
padding: this can be "valid" or "same". Valid describes a standard convolution, whereas same fills the input matrix with zeros at the edges. This leads to an output size identical to the input size.

A second type of layer used in convolutional networks are pooling-layers. They have a structure similar to the convolutional layers but now, the kernels that get shifted over the input do not need to be some simple matrix, but instead can also - for example - return the maximum of a region.

Notation 2.7: Pooling Layer
There are different types of pooling. The two mostly used are MaxPooling, which returns the maximum of a region and AveragePooling, which averages over the region and returns this value. A pooling layer has the from:

Pooling(kernel-shape) with stride k

with kernel-shape meaning the height and width of the kernel. The stride indicates how far the kernel gets shifted over the matrix for the next computation.
In contrast to other layers, pooling layers do not use activation functions.

Note that even though the pooling kernels do not need to be linear, they can still be. One example of this is the AveragePooling, with the kernel being a matrix of the form: $K \in \mathbb{R}^{n \times n}, K_{i,j} = \frac{1}{n^2}$.

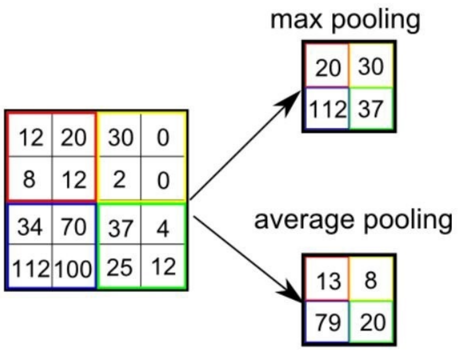

Figure 2: Visualization of different poolings [3]

The strides are useful when doing MaxPooling over a large area (e.g. a 6×6 area). The maximum probably does not change much, if the observed area is moved by one to the right. The leftmost column gets lost and one column is added from the right side. When using a stride of three, the observed area is moved by three columns, and by doing this, not so much data needs to be computed and the approximate location of maxima is still conserved.

The curious reader could ask what makes pooling layers useful. By averaging or computing the maximum over a region, the network does not predict from fixed positions of features in an image, but from approximate positions. This means, moving a picture one pixel to the right, does not heavily affect our network, as after a maxpooling layer, it

[3] Picture from: https://qph.fs.quoracdn.net/main-qimg-cf2833a40f946faf04163bc28517959; 30.06.2019

will look mostly the same. Furthermore, big kernels and large strides reduce the output size compared to the input size. While extracting the important features, unimportant details are omitted, resulting in less computations and faster runtime.
In practice, as convolutional and pooling layers rely on the structure of the input, these layers are mostly used at the beginning of a network to extract features. Then, fully connected layers make predictions from these features. However, fully connected layers need a vector as an input. Because of this, before using a fully connected layer, the different matrices obtained to this point need to be reshaped into a vector. A layer doing exactly this is called *Flatten*.

2.2 Inference Problem

There are many scientific papers [12] [18] and competitions [16] that demonstrate the power and accuracy of neural networks. From simple spam filters for emails, to picture enhancement, they provide a powerful way to solve these tasks as long as one runs them on a powerful enough computer or does not need them to be fast: Even though they consist of elementary operations, with deeper architectures, the time to calculate a prediction increases as well. The process of running models on some data to compute the prediction is called *inference*. This is no problem, if the purpose of a model is to determine if an email is spam. But with real time tasks like security cameras detecting suspicious behaviour or cameras used in quality assurance for fast production lines, an inference time of 3 seconds per frame is not desirable. However, faster and more efficient computations require very good hardware. Powerful processors for example, need good cooling and compatible mainboards. Therefore, a computer powerful enough for fast image processing takes up some space. This can be problematic, thinking of a hidden, small security camera with a big, chunky computer at its side. Additionally, such a hardware can be quite expensive.
This leads to the central questions of this thesis: Can external hardware with no need for a good computer accelerate inference time? How large is the difference in inference time on different processors? Are some models better for one type of hardware, but other models for other types of hardware?
To analyze this, two computers with different hardware specifications were benchmarked on different models and compared to external hardware: The Intel Neural Compute Stick 2 (NCS). Differences and similarities in runtime are observed and it is explored which performance of the NCS can be expected. The two computers are a desktop computer with an AMD Ryzen 7 1700 CPU and a notebook with an Intel core i5 6300U CPU. The Ryzen 7 CPU serves as a reference point for how a good processor scales with model-growth, and what times could be expected from a powerful CPU. For an average user, the Intel core i5 is a good comparison, as it is built into many mid-range computers.

The testing of models on hardware always follows the same procedure: Models are loaded onto the processing units, and preprocessed test-inputs are loaded. Now, a timer is started, and all the examples are run through the model. After the last input, the time is stopped and logged. As a result, all the given times are absolute times over all the test-examples.

2.3 Neural Compute Stick 2

To address the problem of overly long inference times, Intel released the Neural Compute Stick 2 [5]. It is a USB-stick that features Intel's vision processing unit, the Intel Movidius Myriad X VPU. As a dedicated vision processing unit, it should perform well on computer vision tasks, making it an option for faster inference on smaller computers. What needs to be considered though, is that all the data needs to be copied onto the NCS from the computer. Even though it features a USB 3.0 port, this means the Neural Compute Stick takes longer accessing the data, compared to on-board processors like a CPU or a GPU.
Additionally, the NCS comes with the software toolkit *Openvino* [4]. This can be used, among other things, to convert a neural network to an intermediate representation, that consists of two files containing the model architecture and its weights. These files can then be loaded onto a CPU, GPU or the NCS, and predictions can be computed. This has the advantage that only the *Openvino* toolkit is needed on the computer used for inference, instead of the rather big TensorFlow [1] library. This is especially useful on very small computers like a Raspberry Pi, that only has 1 GB of RAM.

The NCS cannot be used for training of networks, but only for inference. As a consequence, the models have to be designed and trained on a high performance computer, until they are ready for use. Further, not all neural networks can be converted to an intermediate representation: There is a list of layers that are supported, but there are still unsupported ones. For example, convolutional layers are supported, but self-defined layers that perform exotic operations on an input are not. Additionally, all the algorithms that should be converted need to be neural networks. As a consequence, for a conversion, an algorithm needs to be rewritten in the form of layers with special weights. This is impractical, as the advantage of ordinary algorithms mostly is that good parameters are already known. Another peculiarity is the precision required by different processors in Openvino: CPUs only work with intermediate representations using 32 bit precision, whereas the NCS can only work with intermediate representations of 16 bit precision. The precision can be specified when converting. Because of this, the CPUs and the NCS do not operate with the same models, but with different intermediate representations created from the same model.

3 First Steps

In this section, a first model is described that was used to analyze the performance of the NCS. This model shows, that there are significant differences between inference time on a CPU and on the NCS.
With the model description, the runtimes are shown. In Section 3.2 some problems that occurred are discussed.

3.1 Architecture

The first model used was a simple MLP with nine layers. It was used as a starting point for further exploration, as it uses simple operations and is rather small. The model gets an input vector $x \in \mathbb{R}^7$ and outputs a two dimensional vector. This makes it different from later models, in that it does not infer from images, but small vectors.

Experiment 3.1: MODEL1

MODEL1 is defined as a MLP with the following layers:

Layer 1: $m_1 = 7, \sigma_1 = Id$

Layer 2: $m_2 = 24, \sigma_2 = ReLU$

Layer 3: $m_3 = 48, \sigma_3 = ReLU$

Layer 4: $m_4 = 96, \sigma_4 = ReLU$

Layer 5: $m_5 = 192, \sigma_5 = ReLU$

Layer 6: $m_6 = 96, \sigma_6 = ReLU$

Layer 7: $m_7 = 48, \sigma_7 = ReLU$

Layer 8: $m_8 = 24, \sigma_8 = ReLU$

Layer 9: $m_9 = 2, \sigma_9 = ReLU$

The total number of parameters is 49.186*, so this model is small, compared to later architectures.*

Testing this model on 1000 test examples resulted in the times seen in Table 1.

While the Ryzen 7 and Intel i5 perform nearly equally well with about 0.12 seconds, the

Hardware	NCS	Ryzen7	Intel i5
Times	0.389	0.118	0.128

Table 1: Times for MODEL1

NCS takes nearly 0.4 seconds. This means that the NCS takes 3.03 times as long as the Intel i5, which is in no way an improved inference time. This displays a need for further testing with more models, especially larger models with more need for computational efficiency. The results can be explained with the rather small model: The slowness of the NCS presumably results from the fact, that the data needs to be copied onto it for inference over a USB-port, whereas the CPUs can load the data from the much faster RAM. The performance on the other hand cannot be improved much, as only few operations need to be performed.

3.2 Problems

There are many problems when creating a model for the NCS. These problems are discussed in this section, as they affected all further models and the approach that was

taken from here on.
First of all, as it is designed for image processing, the NCS needs the input to be a rank four tensor with the shape [B,W,H,C] - B being the batch size (number of inputs processed at once) and W,H,C being the width, height and the number of channels/colors respectively. This means that the used models need to have exactly this format, as otherwise the input gets reshaped in a wrong way and no correct predictions can be made.
Secondly, some further testing revealed even more problems. Normally, a pooling layer using a kernel going beyond the matrix - and therefore using entries not specified in the matrix - is no problem. Normally, all poolings should be computed until one kernel goes beyond the matrix. All further poolings, that need values outside of the matrix get omitted. This way, some data at the edge of the input gets lost, but the model still works. The NCS, however, cannot compute such a layer and returns an error. Therefore, pooling layers need to be defined in a way that all edges are perfectly aligned.

The biggest let-down is the immense time difference between the CPUs and the NCS. But one must keep in mind, that all the data so far comes from one model that is rather small, simple in structure and probably cannot be parallelized. This means the advantages one hopefully gets from the NCS could be overshadowed by the data transfer. For better comparisons, bigger models need to be analyzed that also use other types of layers apart from fully connected layers. Also, the input size may influence the overall runtime of the NCS, so different input-sizes should be observed.

4 More Models

In this section, models I have written to be supported on the NCS are described and benchmarked on the hardware. The models described are of different size and structure so the effect of different parameters can be observed.
In Section 4.1, the used architectures are described and the times observed from the benchmarking are given. The analysis of these times and a discussion of the results takes place in Section 4.2.

4.1 Architectures

First off, a model for classification of the MNIST-dataset [14] was created. The goal of the model is to classify handwritten digits given as a 28×28 pixel image. To solve this task reasonably well (the model used here achieved about 98% accuracy), no big model is necessary. It suffices to use one convolutional and one fully connected layer.
Experiment 4.1: MODEL-MNIST
To solve the MNIST task, MODEL-MNIST was created, which has the following architecture:

Layer 1: $Conv(28, (2, 2), valid), \sigma_1 = Id$

Layer 2: $MaxPooling(3, 3)$ *with stride* 1

Layer 3: Flatten

Layer 4: $m_4 = 10, \sigma_4 =$ *softmax*

This model has 204.242 *parameters and* 28 *convolution-kernels and uses one convolutional layer.*

Running the model on the CPUs and the NCS and logging the inference time used on all the test examples (10.000 images), the following times are obtained:

Hardware	NCS	Ryzen	Intel i5
Times	19.386	1.044	1.057

Table 2: Times of MODEL-MNIST

While Ryzen 7 and Intel i5 perform reasonably well taking about one second, the NCS shows an exorbitant time of nearly 20 seconds, rendering it useless for real time application. This is especially depressing as the NCS is supposed to work well on image-processing tasks like MNIST. The NCS takes nearly 20 times as long as the Ryzen 7 and Intel i5! As already observed in the last section, one problem could be the bigger input size, that needs to be transferred onto the NCS.

To get a feeling whether the time difference between on-board CPUs and the NCS is caused by the simple structure and small size of the network, a model for the EMNIST [3] task was written. The EMNIST task consists of classifying hand-written digits and letters given as 28×28 pixel images. Together with MODEL-MNIST it provides a way to compare different models working on similar data. A model operating on this dataset probably needs a more complex structure and can take even more advantage of convolutional layers. So, MODEL-EMNIST was created to see if a similar, but overall larger model results in inference times that show the same trend as the times in Table 2.

Experiment 4.2: MODEL-EMNIST
MODEL-EMNIST is given by the architecture

Layer 1: $Conv(8, (10, 10), same), \sigma_1 = Id$

Layer 2: $MaxPooling(4, 4)$ *with stride* 2

Layer 3: $Conv(30, (4, 4), same), \sigma_3 = Id$

Layer 4: $MaxPooling(5, 5)$ *with stride* 4

Layer 5: $Conv(50, (2, 2), valid), \sigma_5 = Id$

Layer 6: $MaxPooling(2, 2)$ *with stride* 2

Layer 7: Flatten

Layer 8: $m_8 = 500, \sigma_8 = ReLU$

Layer 9: $m_9 = 100, \sigma_9 = ReLU$

Layer 10: $m_{10} = 47, \sigma_{10}$ = *softmax*

with 2.566.075 *parameters and* 88 *convolution-kernels.*

Running 3000 preprocessed images through MODEL-EMNIST on CPUs and the NCS for benchmarking, a nice improvement can be observed (see Table 3):
Even though 784 numbers per image need to be copied onto the NCS as before, it takes

Hardware	NCS	Ryzen	Intel i5
Times	43.651	4.234	4.495

Table 3: Times of MODEL-EMNIST

about 43.6 seconds compared to approximately 4.5 seconds on the Intel i5. This does not seem like much. However, comparing the ratios of NCS to Intel i5, a decrease from 18.3 to 9.7 can be observed. This means an increase in efficiency of a factor of about 2! This suggests, that bigger models can be computed more efficiently on the NCS compared to smaller models.
The next model was one used to operate on the Caltech 101 dataset [8]. The network should classify images from 102 different classes. Each class contains images depicting one object, for example pianos or anchors. MODEL-CALTECH has fewer parameters than MODEL-EMNIST, but more convolution kernels and therefore could yield even better times on the NCS.

Experiment 4.3: MODEL-CALTECH
The MODEL-CALTECH is a network with the following layers:

Layer 1: $Conv(100, (3, 3), same), \sigma_1$ = *Id*

Layer 2: $MaxPooling(3, 3)$

Layer 3: $Conv(100, (3, 3), same), \sigma_3$ = *Id*

Layer 4: $MaxPooling(3, 3)$

Layer 5: $Conv(80, (2, 2), valid), \sigma_5$ = *Id*

Layer 6: $MaxPooling(3, 3)$

Layer 7: Flatten

Layer 8: $m_8 = 204, \sigma_8$ = *ReLU*

Layer 9: $m_9 = 102, \sigma_9$ = *softmax*

This model contains 1.479.050 *parameters and* 280 *convolution-kernels. It is evident that the model needs more computations to infer from a given input.*

MODEL-CALTECH was benchmarked by loading 300 preprocessed images and running them through the model. It is worth noting that even though the model on the CPU works with a 32 bit precision and the NCS with a 16 bit precision, inference accuracy is the same on all processing units.

Hardware	NCS	Ryzen	Intel i5
Times	1.004	1.046	1.356

Table 4: Times of MODEL-CALTECH

The times from this benchmark can be seen in Table 4 and show a positive result: Even the Ryzen 7 takes longer than the NCS. This shows, that for certain architectures, an increase in efficiency of the NCS can be obtained.
The question is, which architectures work well, and which do not.

4.2 Enlightenment

In this section, we combine observations gathered up to now. Also, connections between different model parameters and the inference times are explored. In particular, we are looking for correlations between the architecture of models and the effectiveness of the NCS. Here, the effectiveness of the NCS means the ratio of the times needed by a CPU divided by the times needed by the NCS. It is evident, that a smaller ratio means a relatively longer time needed by the NCS. The parameters and times observed on the models are:

A Time of Intel i5 divided by the time of NCS

B Time of Ryzen 7 divided by the time of NCS

C Input size multiplied by the number of convolution-kernels (this approximates the number of convolutions performed)

D Number of parameters

E Number of layers

F Input size

G $C/(D \cdot F)$ (this captures the proportion of convolutions used, while taking into account that the time needed for loading the data onto the NCS scales with size).

H Number of operations performed in fully connected layers

I Input size multiplied by number of convolutional layers divided by number of layers

Note, that G is the number of convolution-kernels divided by the number of parameters, but for a better understanding of its meaning, it was written-out in its full form.
When computing a correlation matrix of these quantities, one can see, that some correlations are stronger than others (see Table 6). However, the data collected up to know comes from four samples, meaning that while giving a first idea of some correlations, the results are not powerful. The correlations indicate that the number of layers is unlikely to

Model Name	Model Parameters					Inference Time		
	Parameters	Layers	Conv-Kernels	FC-Operations	Input Size	Ryzen 7	Intel i5	NCS
MODEL1	49186	9	0	49186	7	0.118	0.128	0.389
MNIST	22830	4	28	22690	784	1.044	1.057	15.797
EMNIST	91075	11	88	530347	784	4.234	4.495	34.876
CALTECH	9546414	9	280	9400524	202500	1.046	1.360	1.004

Table 5: Architecture specific parameters of different models

affect the effectiveness of the NCS. Further, it can be seen that the number of convolution kernels divided by the number of parameters is negatively correlated to the effectiveness. This indicates that more convolution kernels could make the NCS less effective, when the total number of parameters stays the same or gets smaller. This, however, could just be a quirk of the small dataset. All the other correlations are very strong. This would mean that with a greater amount of learned parameters, layers and input size, the effectiveness should rise, which is somewhat contradictory: When comparing the effectiveness of the NCS on MODEL1 and on MODEL-MNIST, the effectiveness drops, but the model gets bigger. Interestingly, there is also a strong correlation between input size multiplied by percentage of convolutional layers and the effectiveness.
In order to get better insights, we need to observe more models, so hopefully some correlations calculated on the small dataset get diminished.

	A	B	C	D	E	F	G	H	I
A	1.00	1.00	0.98	0.98	0.25	0.98	-0.70	0.98	0.98
B	1.00	1.00	0.97	0.97	0.26	0.97	-0.73	0.97	0.97
C	0.98	0.97	1.00	1.00	0.17	1.00	-0.55	1.00	1.00
D	0.98	0.97	1.00	1.00	0.17	1.00	-0.55	1.00	1.00
E	0.25	0.26	0.17	0.17	1.00	0.17	-0.45	0.20	0.17
F	0.98	0.97	1.00	1.00	0.17	1.00	-0.55	1.00	1.00
G	-0.70	-0.73	-0.55	-0.55	-0.45	-0.55	1.00	-0.54	-0.55
H	0.98	0.97	1.00	1.00	0.20	1.00	-0.54	1.00	1.00
I	0.98	0.97	1.00	1.00	0.17	1.00	-0.55	1.00	1.00

Table 6: Correlation matrix with features computed from Table 5

5 Data Gathering

The goal of this section is to create even more usable models to benchmark and to collect enough data to analyze the NCS reasonably well. To get data from bigger nets that have real world applications, Section 5.1 deals with big models that are available online. Section 5.2 serves to analyze models whose main purpose is to take a better look at the correlation between runtime and the number of convolutions in a model.

5.1 Predefined Models

In this section we investigate predesigned architectures available at [7]. The big advantage is that they are supported by the NCS and can be easily imported over the `tensorflow.keras` [2] module in Python. This allows us to skip the tedious process of making a useful model and getting it to work on the NCS.
The following architectures are all designed for image classification. Because of this, they all are benchmarked by giving them ten preprocessed images and letting them infer what is depicted in them. It has to be noted however, that not all models process the same input sizes. This needs to be taken into account when comparing inference times for the NCS, as different amounts of data need to be transferred.

As a first model, called MODEL-RESNET50, the model ResNet50 [12] was used. It has 25.636.712 parameters and 34 layers and an input size of 224×224, making it nearly four times as deep as the biggest model used up to now. It uses residual blocks, which means combining outputs of some layers with outputs of previous layers. This adds to the complexity of the network, as some information needs to be stored separately, indicating that the NCS could perform badly with this model.

Testing MODEL-RESNET50 on the different processors, the times in Table 7 can be obtained. One can see, that on this model, the Intel i5 and the NCS perform equally.

Hardware	NCS	Ryzen 7	Intel i5
Times	0.558	0.331	0.550

Table 7: Times of MODEL-RESNET50

This is a further indicator, that the NCS works well on bigger models, even if the input-size is rather large and more complex structures are used in the architecture of the neural network. Unsurprisingly, the Ryzen 7 clearly outperforms both processors, requiring only about 2/3 of the time.

The next model is the inceptionV3 [19], a 159 layer network with 23.851.784 parameters designed by Google. As before, MODEL-INCEPTIONV3 refers to the inceptionV3 model, that can be downloaded via `keras`. Predictions can be made by feeding it a 299×299 pixel image.

Note that this means that bigger inputs are loaded onto the NCS compared to the other models in this section. When benchmarking it by running the model on the ten images, the times in Table 8 were obtained.

Like before, the Intel i5 and the NCS perform nearly the same, but this time with a slightly bigger advantage of the Intel i5 and the Ryzen 7 being noticeably faster.

Hardware	NCS	Ryzen 7	Intel i5
Times	0.828	0.452	0.761

Table 8: Times of MODEL-INCEPTIONV3

Lastly, the network VGG19 [17] was loaded onto the different processors and benchmarked.
With its 26 layers and 143.667.240 parameters, it has fewer layers than the previous two models, but more parameters and needs more computations.

This bigger number of operations gives hope for a good performance of the NCS, compared to the CPUs.
As can be seen from Table 9, this hope is fullfilled: The NCS clearly outperforms the

Hardware	NCS	Ryzen 7	Intel i5
Times	2.166	1.999	2.741

Table 9: Times of MODEL-VGG19

Intel i5 and even though it is still slower than the Ryzen 7, the ratio between the inference times is much closer to one than for the last two models: On MODEL-VGG19, the ratio is about 1.08, on MODEL-INCEPTIONV3 approximately 1.83 and on MODEL-RESNET50 approximately 1.69. This further indicates, that there may be a correlation between model size and effectiveness of the NCS.

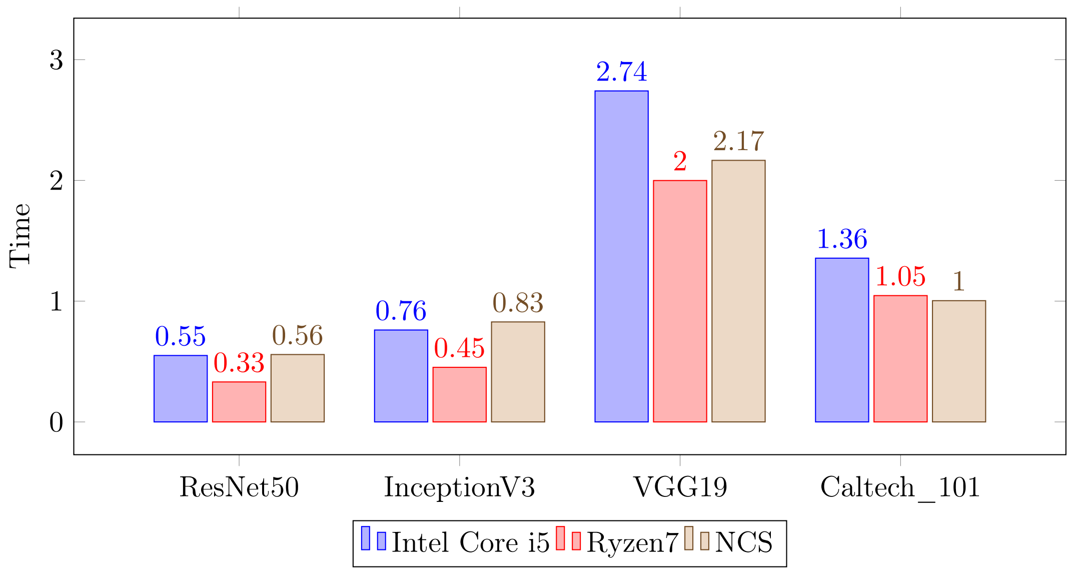

Figure 3: Times of different models on different hardware

As can be seen in Figure 3, the NCS performs badly on MODEL-RESNET and MODEL-INCEPTIONV3, beats the Intel i5 on MODEL-VGG19, and outperforms both CPUs on MODEL-CALTECH. An interesting observation from the above figure is, that the only model on which the NCS performed better than the CPUs was the model that had the biggest input size. More accurately, as MODEL-CALTECH needs 450×450 pixel images as input, it needs about 4 times as much information to be transferred compared to MODEL-RESNET and MODEL-VGG19. This confirms the insight, that on some models, the transferring time can be compensated with the more efficient and faster computation of the NCS.
Now with more data to consider, one could ask if these numbers support the hypothesis about the correlations from Section 4.2, or if they negate it. Therefore, again, some parameters of the different models are gathered and a correlation matrix is calculated (see Table 11).
Because these models are larger, fewer parameters were noted and used for the calculations, as the main point of interest is the correlation between the effectiveness and the percentage of layers that use convolutions multiplied by the input size.

Like before, one can see that the effectiveness of the NCS correlates in a very similar manner with regard to either CPU. The correlation between effectiveness and the input size also stays high, although now smaller. This can be explained with the fact, that the input size is part of the quantity H from the last section. Here, the positive correlation makes more sense. One reason to believe that the correlation between effectiveness and input size is a quirk of this dataset can be seen when looking at MODEL1, MODEL-MNIST and MODEL-EMNIST. Larger inputs yielded slower times for the NCS. Fortunately, the correlation between the number of parameters and the effectiveness is rather low, compared to the correlation matrix from Section 4.2. This comes in combination with the high correlation of effectiveness and the percentage of convolutional layers multiplied by the input size. This term can be thought of as an approximation of the number of convolutions computed in the network relative to other operations, as bigger inputs mean more convolutions computed in each layer.
This indicates that the main reason, why the NCS is more effective on some models than on others, could be the good implementation for parallelizable operations. As the

Model Name	Model Parameters				Inference Time		
	Parameters	Layers	Input Size	ConvLayers	Ryzen 7	Intel i5	NCS
MODEL1	49186	9	7	0	0.118	0.128	0.389
MNIST	22830	4	784	1	1.044	1.057	15.797
EMSNIST	91075	11	784	3	4.234	4.495	34.876
CALTECH	9546414	9	202500	3	1.046	1.360	1.004
RESNET50	25636712	117	50176	53	0.331	0.550	0.658
INCEPTIONV3	23817352	313	89401	94	0.452	0.761	0.828
VGG19	143667240	26	50176	16	1.999	2.741	2.166

Table 10: Architecture specific parameters of more models

dataset above still only consists of eight samples and therefore is of dubious power, this correlation is fully explored in the next section.

For the full correlation matrix, see Table 11, with the letters indicating the following features:

A	Time of Intel i5 divided by time of NCS
B	Time of Ryzen 7 divided by time of NCS
C	Number of parameters
D	Number of layers
E	Input size
F	Percentage of layers that are convolutional layers times input size

	A	B	C	D	E	F
A	1.00	0.98	0.58	0.26	0.81	0.90
B	0.98	1.00	0.57	0.09	0.82	0.92
C	0.58	0.57	1.00	0.00	0.05	0.26
D	0.26	0.09	0.00	1.00	0.19	0.13
E	0.81	0.82	0.05	0.19	1.00	0.97
F	0.90	0.92	0.26	0.13	0.97	1.00

Table 11: Correlation matrix for more models

5.2 Proportion of Convolution

Up to now, the models have shown a tendency to work better on the NCS, when there are proportionally many convolutions and few fully connected layers. In combination with the bigger inputs, this also corresponds to the idea one gets from the correlation matrix in the last section. Therefore, we construct models aiding to analyze this behaviour: In this section, models that all share the same structure but a different number of convolutional layers are described and benchmarked. This will show, that the structure of the model clearly affects the efficiency of the NCS.
To get comparable data, the architectures used start with some convolutional layers and then use some fully connected layers.

Experiment 5.1: MODEL-CONV P
A model with P percent of the layers being convolutional layers is called MODEL-CONV P. Overall 22 layers are used and P is a multiple of 5. *This provides the architecture*

Layer 1 to $P/5$*: Conv(50,(4,4),valid),* $\sigma_1 = Id$

Layer $P/5+1$*: Flatten*

Layer $P/5+2$*:* $m = 1, \sigma =$ *ReLU*

Layer $P/5+3$ *to* 22*:* $m = 1000, \sigma =$ *ReLU*

Here, every fully connected layer adds 100100 parameters, every convolutional layer 800 parameters and 50 convolution-kernels.

These models were then benchmarked on the same ten preprocessed images the models in Section 5.1 were evaluated on. To get a comparison between the proportion of convolutions and the sole number of operations in a model, benchmarks with different input sizes were executed. The first test is conducted with an input of 512×512 pixel images, the second one with 256×256 pixel images. Therefore, in the first test, four times as much data needs to be copied onto the NCS, and roughly four times as many convolutions need to be computed compared to the second test.
The times gathered for the 512×512 pixel input image are shown in Figure 4. (For a detailed table of the times, see the appendix.)

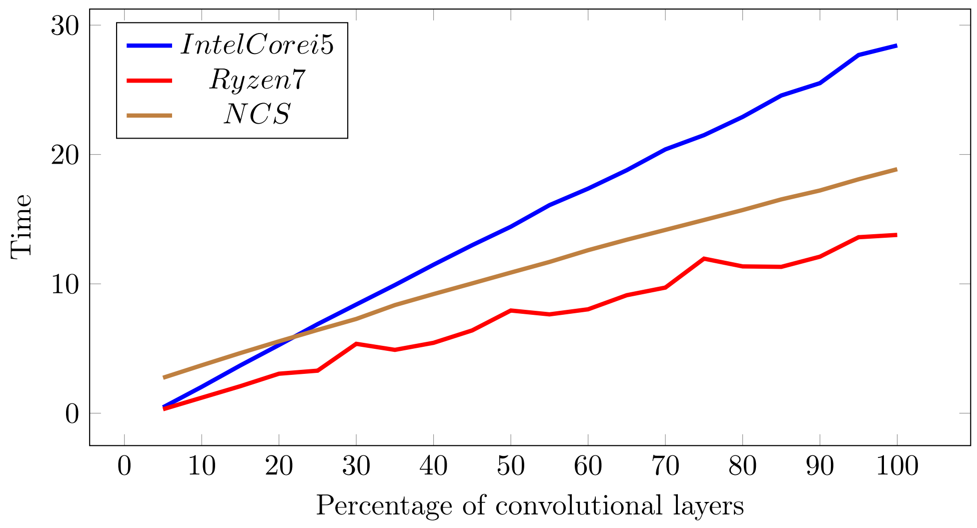

Figure 4: Times of MODEL-CONV series with input size 512×512

It can be seen, that in all cases, the runtime scales with the proportion of convolutional layers. This is no surprise, as convolutions need more computations than a simple matrix vector multiplication. However, the graph indicates that the different processors scale differently with more operations to be executed: While the Ryzen 7 scales roughly like the NCS, the Intel i5 loses much of its speed with the increase of convolutions.
Doing some linear regression, one can calculate that the slope of the graph regarding the NCS equals approximately 0.17 while the slope of the Intel i5 is approximately 0.29. This

means with each fully connected layer replaced by a convolution layer, the Intel i5 needs twice as much additional inference time as the NCS. Asymptotically, the NCS should perform in half of the time needed by the Intel i5. The Ryzen 7 on the other hand only has a slope of about 0.14.

The Ryzen 7 is always faster than the NCS, but when comparing the stick to the Intel i5, the times are in favour of the NCS for all models containing more than 20% convolutional layers. These numbers lead to a confirmation of the previous hypothesis: The NCS takes advantage of the more efficient way of computing convolutions. Higher proportions of convolutions in a model lead to better performance, compared to mid-range CPUs. The ratios of the inference times between the NCS and the CPU times are also interesting: These are better indicators how much faster the NCS really is, and if a reasonable acceleration is achieved. These ratios are shown in Figure 5.
With decreasing proportion of convolutions the ratios get bigger. Additionally, the dif-

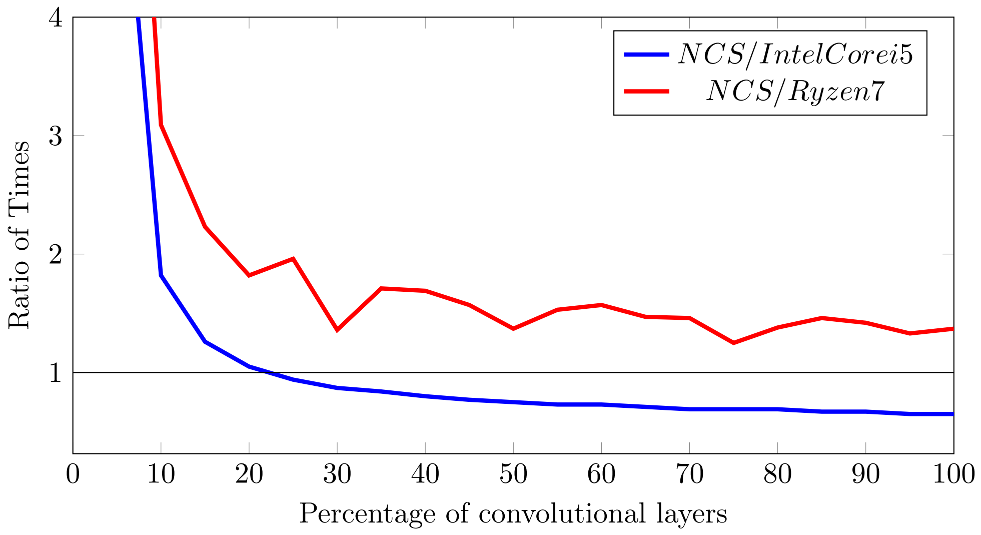

Figure 5: Ratios of hardware, MODEL-CONV series with input size 512×512,

ferent graphs show a similar pattern: Altough the NCS/Intel i5 ratio is always lower than the NCS/Ryzen7 one, indicating that there is a better time ratio of the processing units, they both show a rather gentle slope up from 100% to about 20%. Then, the slope gets very steep. This indicates a bad performance of the NCS on small models with big inputs.

What remains to do, is to analyze if this behaviour can be expected with similar models, which process smaller inputs and therefore need to compute less convolutions. This additionally provides the opportunity to analyze how the inference times scale with smaller inputs, if the remaining architecture stays the same. To achieve this, the same benchmark as before was performed, but now with models of the shape MODEL-CONV P, trained for input images with a size of 256×256 pixel, so a quarter of the number of

pixels from before. This benchmark supplied the times one can see in Figure 6. (For a detailed table of the times, see the appendix.)
Like in Figure 4, the NCS and the Ryzen 7 scale approximately the same with increasing

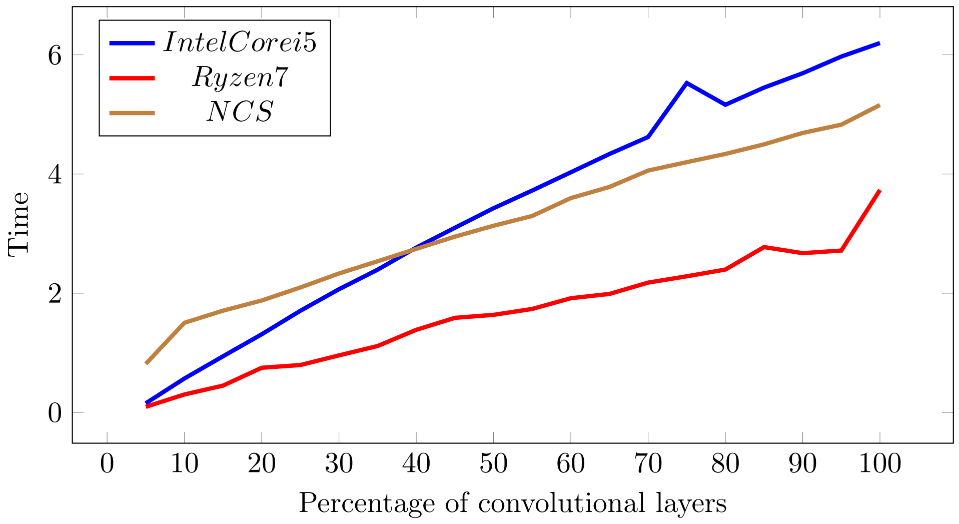

Figure 6: Times of hardware, MODEL-CONV series with input size 256 × 256,

number of convolutional layers, with the biggest difference being that the NCS is always about 1.5 seconds slower than the Ryzen 7 CPU. This is no surprise, as one needs to take the data-transferring time into account.
Again, with the help of linear regression, the slopes of the graphs can be approximated, leading to the slopes in Table 12. One can see that again, the Ryzen 7 performs best, with the NCS being second.
Calculating the ratio of the slope of the NCS over the one of the Intel i5, one gets about

Hardware	NCS	Ryzen	Intel i5
slopes	0.042	0.032	0.064

Table 12: Slopes of linear regression models

2/3, indicating that now the NCS, although still performing considerably better than the i5, does not give the same speed boost as with the models inferring from 512 × 512 pixel images.

The biggest difference can be observed when comparing the Intel i5 and the NCS: Even though less information needs to be copied, the NCS is slower than the Intel i5 on more models. From 5% to 40%, the Intel i5 is faster. The fact that smaller inputs result in less efficiency fits right in with the current hypothesis: A smaller input means that fewer convolutions need to be computed per layer. This again means that less computations can be parallelized, which explains the difference in performance. This also is supported by the fact, that the relative difference in the slopes is smaller with the smaller image size, as pointed out above. Additionally, it confirms the correlations explored above.

Another interesting observation can be made, when comparing the slopes of the hardware when performing on the bigger input image, with the respective slopes for the smaller input image: Independently of the processor, with an input that is four times as large, the slopes get approximately doubled. This is a bit unintuitive, as one would expect the slopes going up with the same factor as the input size, as also the number of convolutions, which are the computationally costly part, grows in the same way as input size.
Comparing the times needed by the NCS and the Intel i5 for MODEL1, MODEL-CONV 5 with input size 256×256 and MODEL-CONV 5 with input size 512×512, one can see that independently of the input size, the NCS performs considerably worse than the Intel i5 on architectures that use many fully connected layers and very few to no convolutions.

6 Conclusion

In this section we summarize the results of the analyses and point out, what the advantages and disadvantages of the NCS are. In Section 6.2, some other noteworthy results are mentioned, even though they are not crucial to the NCS. Finally, the last section features a discussion of potential further analysis that could be done.

6.1 NCS: What Is It Good For?

As seen in Section 4.2, the NCS performs well on some models and badly on others: On MODEL1 it takes three times as long as the on-board CPUs, whereas on MODEL-CALTECH, it outperforms even the Ryzen 7. Furthermore, our statistical analysis suggests that there could be a strong correlation between the effectiveness of the NCS and the proportion of convolutions used in a model. This is supported by the models in Chapter 5.2: Logging the time for similar models that use different proportions of convolutions showed the increasing efficiency with more convolutions and less fully connected layers.

As a consequence, when using large models with many convolutions, one can expect a good performance boost using the NCS. On the other hand, models with few convolutions, but large inputs, are better performed on a conventional CPU.
One big advantage, however, is that models can now also be used efficiently on small computers like a Raspberry Pi, which can be expected to perform considerably worse than the Intel Core i5 CPU without the NCS. It is noteworthy that the benchmarks originally should have been performed on a Raspberry Pi Model 3B+. This proved impossible, because it was not possible to use the intermediate representation used by the *Openvino* tool kit on a Raspberry Pi CPU. Here, only inference on the NCS is supported. Running the models with TensorFlow would have yielded inference times not comparable to the

other CPU's times that were tested with intermediate representation models.

As already mentioned in Section 3.2, the NCS exhibits problems when creating models that use unsupported or not optimally defined layers like pooling layers that need to cut off parts of the input. Although not posing a problem with professionally designed models, this means that small models that still work fine with TensorFlow have to be rewritten and tweaked to work on the NCS. A bigger annoyance is the lack of user defined layers being supported. Some tasks can profit much by non-standard layers, that are not commonly used and therefore not supported by the NCS.

In conclusion, it can be said that the Neural Compute Stick 2 proves effective for models that use many convolutions and few non convolutional operations. However, more complex networks can be problematic to implement, as the toolkit used does not work for all layers.

6.2 Serendipitous Discoveries

Besides the sole benchmarking of the NCS, other interesting results were made. These range from the runtime of neural networks on traditional CPUs to how much architecture specific parameters of a neural network influence the performance of a model.

Firstly, the different graphs show how much of a difference a good processor can make on the inference time. This can best be seen when observing the inference times of the MODEL-CONV series. Here, the Intel i5 shows a much steeper increase of time with a bigger proportion of convolutions than the Ryzen 7. Further it can be observed that with four times the input size, and therefore about four times the computations in convolutional layers, the additional time with each convolutional layers only gets doubled. Overall, this shows how crucial good processors are when using inference in real world applications that need to be computed fast.

The models in Section 4 show how efficient convolutional neural networks can be for computer vision tasks: Only a small model with four layers can perform reasonably well on the MNIST task. Writing an algorithm that performs equally well but which does not use machine learning would be much more complex.

6.3 What Remains To Do

While this thesis gives some information about the inference times needed for various models on different processing units and explores some possible correlations, these correlations are not supported by deeper analyses of the processor architectures. For a better and more theoretical description of the behaviour of the processing units, the specific pipelines should be analyzed. Understanding how the CPUs handle the convolutions and how well they parallelize them compared to the Intel Movidius Myriad X VPU in the NCS would certainly give a better understanding of the correlations between network

architecture and runtime. Further, it remains to analyze how the conversion of a model to an intermediate representation by the *Openvino* SDK affects the way certain layers in a network are computed. This means exploring whether layers are merged together if some operations are not necessary to compute. An example of this would be a ReLU function that always returns the input, because it always gets non-negative input.
As the models used here were, with the exception of MODEL1, used for image classification, the conversion of the model to a 16 bit precision did not affect the output drastically. With MODEL1, however, a big difference was observed between the outputs of NCS and CPUs. It is still an open question, if these wrong outputs were caused by the model-specific architecture and a bad combination of weights or if such an error propagation must be expected in all models used for regression tasks.

The models used in Section 5.2 and MODEL-CALTECH showed the best performance of the NCS. Nevertheless, they are still rather small, having no more than 25 layers. Certainly, with even more convolutions, a bigger time difference between the Intel core i5 and the NCS could be observed. This suggests that in order to harness the true power of NCS one needs to consider even deeper models with even more convolutions.

Another interesting question is whether networks can be tweaked in such a way, that the NCS gets more efficient, without destroying the existing architecture in a way that would harm the accuracy of the predictions. To do this would again require knowledge how the different CPUs compute different operations and in which way a model is influenced by the conversion to an intermediate representation. If such knowledge is acquired, this would greatly help to increase the performance of the NCS. Additionally, analyzing how much of an improvement these tweaks really provide, would be interesting.

7 Bibliography

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] François Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7(8):T1, 2015.

[3] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: an extension of mnist to handwritten letters. *arXiv preprint arXiv:1702.05373*, 2017.

[4] Intel Corporation. Intel distribution of openvino toolkit. https://software.intel.com/en-us/openvino-toolkit, 2019. accessed: 2019-06-26.

[5] Intel Corporation. Intel neural compute stick. https://software.intel.com/en-us/movidius-ncs, 2019. accessed: 2019-06-30.

[6] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[7] Keras Documentation. Keras applications. https://keras.io/applications/, 2019. accessed: 2019-08-1.

[8] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer vision and Image understanding*, 106(1):59–70, 2007.

[9] Matt W Gardner and Stephen R Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.

[10] Gavin J Gibson, Sammy Siu, and Colin F N Cowan. Application of multilayer perceptrons as adaptive channel equalisers. In *Adaptive Systems in Control and Signal Processing 1989*, pages 573–578. Elsevier, 1990.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[13] Thomas William Körner. *Fourier analysis*. Cambridge university press, 1989.

[14] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[15] Lexico. Defintion of artificial intelligence. https://www.lexico.com/en/definition/artificial_intelligence, 2019. accessed: 2019-07-29.

[16] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[18] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[19] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

8 Appendix: Tables

	256 × 256			512 × 512		
% Conv	Intel i5	NCS	Ryzen7	Intel i5	NCS	Ryzen 7
100	6.198	5.158	3.732	28.430	18.863	13.783
95	5.971	4.828	2.716	27.690	18.083	13.604
90	5.691	4.689	2.672	25.515	17.218	12.100
85	5.450	4.498	2.774	24.561	16.530	11.314
80	5.162	4.337	2.397	22.906	15.705	11.348
75	5.527	4.199	2.284	21.496	14.936	11.948
70	4.621	4.058	2.178	20.394	14.170	9.712
65	4.336	3.783	1.988	18.785	13.410	9.121
60	4.029	3.596	1.916	17.361	12.602	8.033
55	3.723	3.296	1.736	16.084	11.693	7.641
50	3.426	3.132	1.637	14.417	10.872	7.939
45	3.098	2.949	1.587	12.990	10.036	6.401
40	2.766	2.744	1.386	11.480	9.210	5.442
35	2.395	2.536	1.114	9.909	8.359	4.897
30	2.067	2.331	0.957	8.403	7.286	5.364
25	1.705	2.096	0.795	6.885	6.438	3.289
20	1.311	1.878	0.749	5.272	5.557	3.053
15	0.942	1.707	0.450	3.693	4.649	2.089
10	0.569	1.504	0.301	2.038	3.705	1.200
5	0.153	0.814	0.094	0.456	2.741	0.309

Table 13: Times of MODEL-CONV series with different input sizes