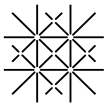


University of Basel

Performance Analysis of Neural Network Models on Vision Processing Units

Master Project

Faculty of Science
Department of Mathematics and Computer Science
High Performance Computing Group
https://hpc.dmi.unibas.ch/en/

Examiner: Prof. Dr. Florina Ciorba
Supervisor: Ahmed Hamdy Mohamed Eleliemy, MSc.

Gowthaman Gobalasingam
gowthaman.gobalasingam@stud.unibas.ch
16-050-619

January 15, 2021

Abstract

The promising growth of Deep Learning in the last few years allows to perform highly complex applications on IoT devices. In particular, computer vision applications like object detection, object recognition, image segmentation, and object tracking can be realized with the help of Convolutional Neural Networks (CNN). However, deploying high parametrized and complex deep learning models, the hardware needs to face challenges in terms of computational workload in computer vision and AI on the edge. Intel provides the Movidius Vision Processing Unit (VPU) as such that exactly incurs these challenges.
In this project, we studied the feasibility of deploying different CNN models for image classification on Intel's VPU and tested their performance with regard to accuracy and classification rate.

Table of Contents

1 Introduction

The interest in Artificial Intelligence (AI) arose with the question of whether we can make a machine to automatically solve problems intelligently and detect patterns in data. In order to build an AI model, we need a theoretical understanding of building intelligent systems and an artifact, meaning a machine that can deploy the systems. The latter was problematic at the beginning of AI Research since the hardware was slow and costly at that time. Later on, further generations and optimizations in hardware improved the performance at a lower cost. Applications of AI models on these hardware became realistic.

In today's era of Big Data, AI models became useful to identify patterns in huge data collection. Internet of Things devices (IoT devices) use Machine Learning (ML) models that are typically hosted on a cloud based infrastructure connected to a remote server. So, if a device wants to use a model, it sends a request for inference to the associated server (Cloud Computing) as for example robots and sensors in industrial automation do. To get rid of bandwidth restrictions in Cloud Computing, we would place a local server on the top of the edge devices which can also host ML models (Edge Computing).

On the contrary, devices that work offline are required to run AI models directly on their local hardware. Common devices that use visual intelligence like security cameras, drones and service robots apply typical visual applications like object detection with motion tracking, facial detection, image segmentation and image classification. Deep Learning models, especially Convolutional Neural Networks (CNN) play an essential role in computer vision applications, and have presented in the past many CNN models with promising accuracy.

Executing Deep Learning models on devices with restricted resources is not convenient. The model can have a complex architecture with a high number of parameters that require massive compute power to justify an inference [7]. A different processor can assimilate these models. One artifact is the Movidius Neural Compute Stick (NCS) provided by Intel that contains a Vision Processing Unit (VPU). It is a chip-level architecture that can balance the computational workload and efficient power consumption. Overall, the NCS can achieve the following goals:

- **Accelerated Performance**: Quick inference, e.g. rapid real-time detection
- **Privacy**: Data are not shared
- **Efficiency**: Computations at low power consumption
- **Adaptivity**: NCS can be plugged to various devices
- **Scalability**: Building a system with multiple NCS to distribute inference requests

The aim of this project was to study the feasibility of deploying different CNN models for image classification on the NCS, and to test their performance in terms of accuracy and classification rate.

The remaining of this report is structured as follows. In Chapter 2, we go through the theoretical aspects of CNN, and describe the models we chose for our analysis. In Chapter 3, we briefly describe the technical details of the Movidius NCS. In Chapter 4, we delineate the OpenVino environment which we mainly used to work with the NCS. In Chapter 5, we present our approach for the performance evaluation which will be assessed in Chapter 6. In Chapter 7, we amplify the challenges and issues we had while working with the NCS. Finally, we conclude the work.

2

Artificial Neural Networks

Deep Learning is a sub-field of Machine Learning, and studies the way of extracting patterns of input data with Artificial Neural Networks (ANNs), and assigning them to some class or label. The idea of building a "Neural Network" [9][14] was inspired by the composition of biological neurons in the central nervous system. This observation was introduced in 1943 by Warren McCulloch. He presented a computational model based on propositional logic. This invention created the belief in intelligent machines, but quickly became impractical due to the limited hardware of that time. In the beginning of the 1990's, with the tremendous increase of computing power, huge availability of data and improved training algorithms, another wave of interest in ANNs was observable. ANNs have been studied very well since then, and many ANN models were published.

2.1 Fundamentals of Artificial Neural Network

An ANN [8][16] is built of multiple computing units, called neurons, that are distributed among multiple layers. The network is composed of an input layer, where the input data are fed into, and the results are issued by the output layer. The intermediate layers are called the hidden layers. Each neuron takes a weighted sum of the output from the previous layer, and applies some activation function. In order to solve hard, complex problems we need to take nonlinearity into account, and therefore use nonlinear activation functions like the Sigmoid function. The results of the output layer are evaluated by a loss function whereby we can learn the network by updating the model parameters (weights and biases) to improve the overall accuracy. This procedure is called the Backpropagation where we apply the Gradient Descent technique to update these parameters. This method yields the best approximation for the target values of the network.

Stacking multiple hidden layers became fundamental to better capture the underlying data generation process of complex problems. However, they were unstable in training. The famous issues that appear are Vanishing and Exploding Gradients. Deep Learning introduced the Backpropagation friendly ReLU function for activation, extensions and variations in the architecture design and regularization techniques for preventing overfitting that have been essential for better performance in learning. The two main diverged state-of-the-art archi-

tectures in ANN are Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). In Section 2.2, we focus on CNN architectures from which we chose four models, see Section 2.3, for the performance analysis.
Modern libraries like TensorFlow[1], PyTorch[2] and Caffe[3] support Backpropagation automatically. With the help of Graphical Processing Units (GPUs), the training can be even more simplified with parallelisms.
In the next section, we take a tour to CNN that is the core of this project work.

2.2 Convolutional Neural Networks

The importance of CNN arose from different types of multi-dimensional input data, e.g. images. Images are composed of pixels, and each pixel expresses some color intensity. For example an image of pixels 300×300 and assuming an RGB color system, we would feed an image of dimension $300 \times 300 \times 3$ to a fully connected ANN that would enormously increase the number of parameters. Since we are limited by computational resources, we need to constrain the network by sharing the weights, and reduce the dimension of hidden layers [13]. The first intention of building a CNN came up by interpreting the human's perception and information processing by the visual cortex of the cat's brain [8]. David H. Hubel and Torsten Wiesel discovered in 1958/1959 that subsets of neurons only react to so-called local receptive fields of the entire visual field. This inspiration led to the idea of CNN where neurons of a convolutional layer evaluate only pixels in some receptive fields of the image. By stacking multiple convolutional layers, each capturing only subsets of previous layers, finally extract high-level features from low-level features of an input image. This architecture became successful, and led to interesting applications in computer vision like image classification, object detection and semantic segmentation, see Fig. 2.1. Apart from visual applications, CNN found also interest in other areas like voice recognition, natural language processing and bioinformatics.

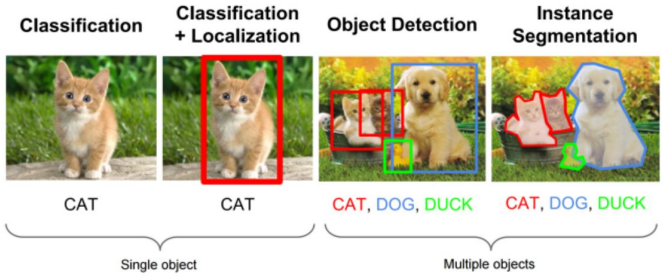


Figure 2.1: Examples of visual applications by means of CNNs [13]

In the next two sections, we describe the two crucial building blocks of CNNs.

[1] https://www.tensorflow.org/
[2] https://pytorch.org/
[3] https://caffe.berkeleyvision.org/

2.2.1 Convolutional Layers

A convolutional layer is nothing else than a hidden layer. The neurons only take a weighted sum of a subset from outputs from the previous layer, the so-called convolved feature. The neurons in the first convolutional layer have to be fed by multiple convolved features of the image which can be gathered by a so-called convolutional filter. A filter is a matrix of small dimension, e.g. 3×3 or 5×5 containing some weights in each cell, and strides over the whole image. This allows the network to capture features independent of feature variations and distortions. This is essential since the same objects can be represented differently in different images. A simple filter would return a weighted sum of receptive fields. The resulting feature map of a layer gives an abstraction of the pixel intensities.

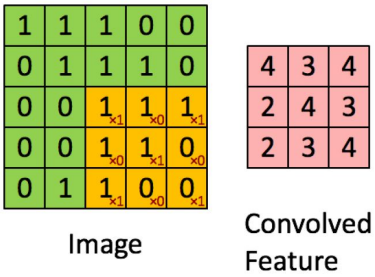


Figure 2.2: Convolved feature [16]

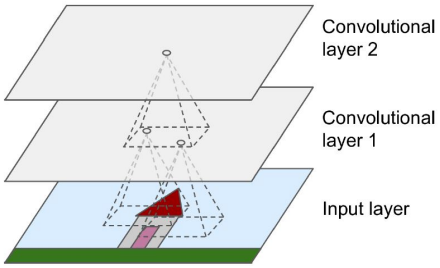


Figure 2.3: Convolutional layers [8]

Formally, a convolution is an operation of two functions, one the pixel values, and the other the weights stored in a kernel [9][16]. In practice, different filters are used to extract different patterns, and the parameters are learned by Backpropagation. Typically, each convolutional layer will produce multiple feature maps, see Fig. 2.4, using different filters that allow to extract complex structures.

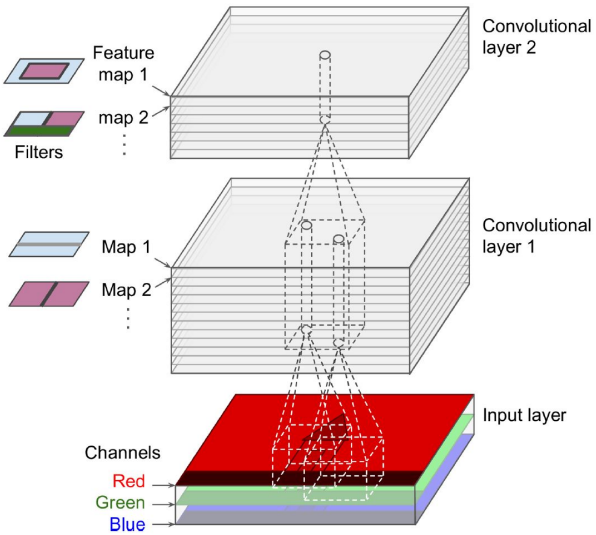


Figure 2.4: Convolutional layers with resulting multiple feature maps [8]

2.2.2 Pooling Layers

The large number of model parameters can be reduced by downsizing the feature maps with pooling layers. Similar to convolutional layer, we use again a filter, but without any kernel. The result is an aggregation of the captured inputs by the filter, and reduces the number of parameters. This allows us to mitigate the computational overload, getting rid of overfitting, and avoid some small invariances that can still take place. Commonly used pooling filters are mean and max-pooling filters where the latter works better, see Fig. 2.5. The resulting subsampled feature map will be fed to the next layer.

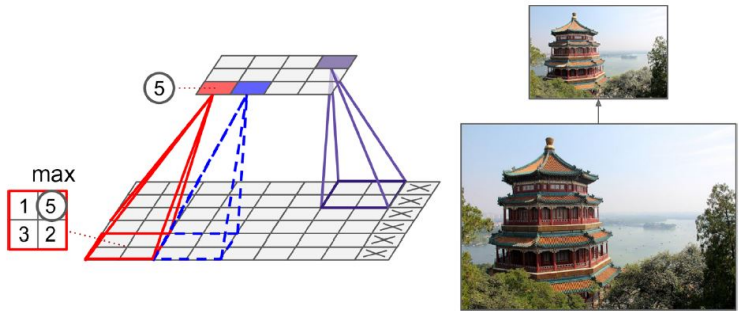


Figure 2.5: Max pooling [8]

Finally, a CNN can be built with convolutional layers and pooling layers, and is able to classify any image. Fig. 2.6 shows a basic CNN architecture template:

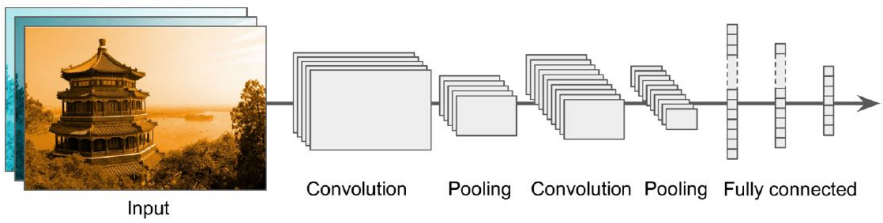


Figure 2.6: Basic CNN architecture [8]

2.3 CNNs Models

In this project work, we considered four CNN architectures: AlexNet, GoogleNet, ResNet and SqueezeNet. They all are trained with the ImageNet dataset[4]. We briefly describe each CNN model in the next sections.

2.3.1 AlexNet

AlexNet was introduced by Krizhevsky et al. [12] in 2012. The network is built with five convolutional layers and three fully connected layers. The neurons apply the non-linear

[4] http://www.image-net.org/

Rectified Linear Unit (ReLU) function for gaining a better performance in the training. For downsizing the inputs, max-pooling filters are used for the first, second and fifth convolutional layer. The input image will be first downsized to $224 \times 224 \times 3$, and filtered by a convolutional filter of size $11 \times 11 \times 3$ with 96 kernels, and strides the image by four pixels. The last fully connected layer feeds the output to a softmax function producing a probability distribution over 1000 classes. Due to the large size of the network, two GPUs were used for training. Containing around 60 million parameters, the network was initially prone to overfitting. Hence, they enlarged the Imagenet dataset by transforming images through translations, horizontal reflections and augmenting the color channel intensities. Additionally, they used the dropout technique that disables neurons with a probability of 0.5. This is a common method for tuning the network. In every training period, different neurons were dropped out, but the parameters are still shared by convolutional layers. Fig. 2.7 illustrates the architecture.

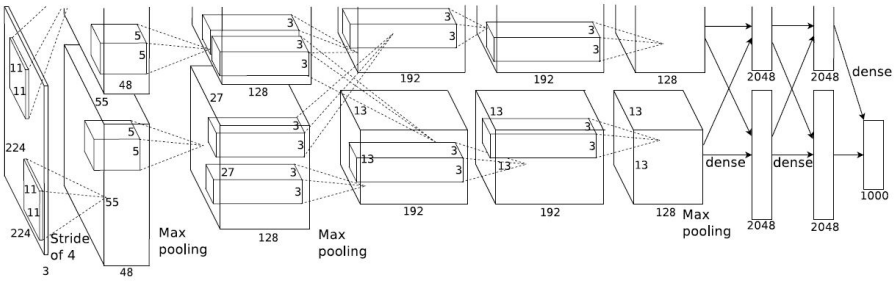


Figure 2.7: AlexNet Architecture [12]

2.3.2 Inception v4

Szegedy et al. [17] introduced the Inception v4, also called GoogleNet v4. The network is composed of so-called Inception blocks. It is a mix of convolutional and pooling layers with different filters that can be stacked in parallel, and executed independently on an input. The resulting feature maps of the layers in that block are then finally concatenated, see Fig. 2.8.

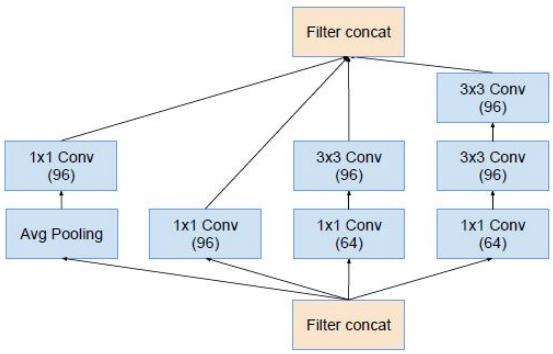


Figure 2.8: Schema of the first Inception block in Inception v4 [17]

Inception modules allow to build deeper networks. Initially, the first Inception model was partitioned among these inception blocks, and are trained individually which allowed them to make decisions in terms of tuning the network. However, bringing all sub-networks together made the resulting network complicated. Inception v4 overcomes this issue by restricting to uniform decisions, and introduces more inception models than the former model.
The network can be further extended with residual frameworks, see Section 2.3.3, and makes the Inception blocks cheaper. The resulting hybrid network is the Inception-ResNet. It has roughly the same cost as the pure Inception Network, and has also around the same error rate on the ImageNet classification task. We will not cover this network here. Details can be found in the publication [17].

2.3.3 ResNet-50

ResNet stands for Resolutional Network, and was first introduced by He et al. [10] in 2016 and won first place in the ILSVRC classification task 2015[5]. The network is based on so-called residual frameworks, a way of reducing training overhead. The problem arose with the large depth of the network. Deep networks are crucial for extracting patterns from the input data, however, it can lead to higher training errors when it starts converging. They introduced a deep residual learning framework with shortcut connections, meaning performing a so-called residual mapping by skipping convolutional layers, see Fig. 2.9.

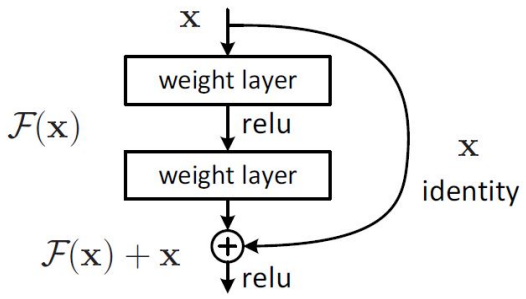


Figure 2.9: Residual Learning: Building block [10]

This framework asymptotically approximates the desired identity mapping that we would originally do with stacked layers. We will not go further in details here, but consolidate that the resulting network is easier to optimize, and gains high accuracy with no extra parameters.
We considered for this project work the version ResNet-50 with 50 layers.

2.3.4 SqueezeNet

SqueezeNet was introduced by Iandola et al. [11] in 2016. They intended to build a smaller CNN with competing accuracy based on AlexNet. The core building block is the so-called Fire module, see Fig. 2.10. It squeezes a convolutional layer with 1×1 filters, and feeds

[5] http://image-net.org/challenges/LSVRC/2015/

into an expanded layer that is composed of 1×1 and 3×3 filters. This module helps to reduce the number of parameters by nine times.

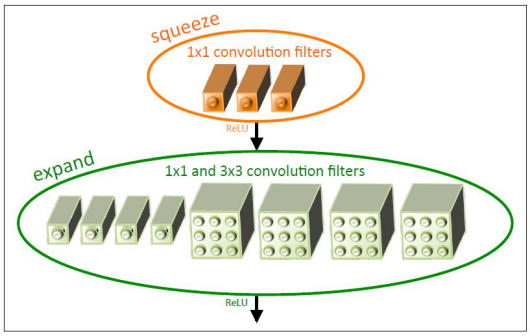


Figure 2.10: Abstract illustration of the Fire module [11]

Another strategy along with these Fire modules is the late placement of pooling. Feature maps are preserved longer, and yield higher accuracy. With these building blocks, the SqueezeNet is roughly constructed as follows: a convolutional layer with 3×3 filter followed by eight Fire modules and at the end again a convolutional layer. The max-pooling takes place after conv1, fire4, fire8 and conv1. Compared to AlexNet, this architecture has 50 times smaller parameters with nearly the same accuracy. By additionally compressing the model by Deep Compression[6], it becomes 510 times smaller than AlexNet, and fits a memory size less than 0.5 MB.
The advantages are the following: a better scaling among a distributed system with less communication overhead, quicker model update (on-the-air update) on clients, e.g. autonomous driving cars like Tesla, and storing the model for inference on the chip memory of an FPGA.

[6] Han et al., 2015

3

Intel's Movidius Neural Compute Stick

In this chapter, we look at the Movidius Neural Compute Stick device containing the VPU. In Chapter 1, we motivated us with AI applications on the edge, and in Chapter 2, we saw Deep Learning as a complex Machine Learning concept for which the appropriate hardware and acceleration techniques are required. An alternative to CPU and GPU is the VPU for running ANN models whereby we conduct the goal of enabling accelerated AI solutions under low power consumption.
The SDK for Movidius NCS is the Intel Movidius Neural Compute SDK. Unfortunately, we had troubles working with this SDK which we cover in Chapter 7. Nevertheless, we will still take a brief look at in Section 3.2. The environment we used for our evaluation is the OpenVino Toolkit which we consider in the next chapter.

3.1 Movidius Neural Compute Stick

The Movidus NCS [2] comes with USB connection, a built-in architecture for loading and prototyping Deep Learning models for AI application inference. The core component is the Movidius Myriad 2 Vision Processing Unit[7]. It contains a Leon Microprocessor, a 4GB LPDDR3 Memory and 12 so-called Streaming Hybrid Architecture Vector Engines (SHAVE) processors that incorporate parallelism. Fig. 3.1 shows an abstract illustration of the NCS's architecture, and the following table consists of some technical specifications copied from [3].

[7] https://ark.intel.com/content/www/us/en/ark/products/122461/intel-movidius-myriad-2-vision-processing-unit-4gb.html

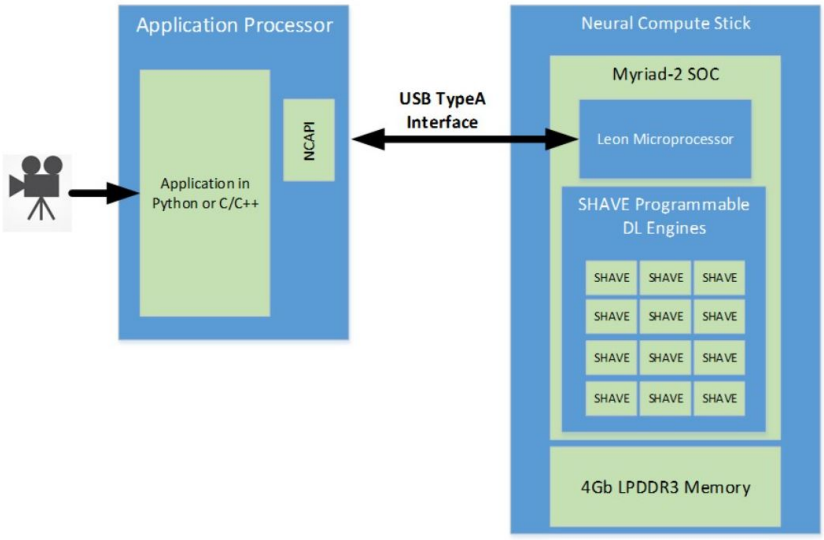


Figure 3.1: Architecture Movidius NCS [2]

Processor Base Frequency	933 MHz
Memory Types	4GB LP-DDR3 with 32-bit interface at 733 MHz
Maximum Memory Speed	733 MHz
Operating Temperature Range	−40° C to 105° C

Table 3.1: Intel Movidius Myriad 2 Specifications [3]

The second generation is the Intel's Neural Compute Stick 2 (NCS 2)[8] with Intel's Myriad X processor that contains 16 SHAVE processors and an inference engine that allows to speed up the performance up to 10 times. The NCS 2 is best compatible with the latest version of OpenVino.

3.2 Intel Movidius NCSDK

The NCSDK [5] contains various software tools, the crucial one for compiling deep neural network models with the associated parameters into a Graph file that can be loaded only to Movidius NCS. Additionally, tools exist for checking the network and profiling, meaning providing a statistical, and performance overview of the model in HTML. Along with the software tools, it also comes with an API (NCAPI) for developing AI applications in Python and C/C++. The progress of deploying models onto the NCS is the same as in OpenVino. Therefore, any application written with NCAPI can be transitioned into an OpenVino based application and vice versa[9]. For our project, we had problems working with NCSDK, and therefore turned to OpenVino.

[8] https://ark.intel.com/content/www/us/en/ark/products/140109/intel-neuralcomputestick-2.html?_ga=2.228662617.419505977.1609766294164914 5100.1605605305

[9] https://software.intel.com/content/www/us/en/develop/articles/transitioning-from-intel-movidius-neural-compute-sdk-to-openvino-toolkit.html

4

OpenVino

For the performance analysis, we worked with OpenVino [6]. It is a software toolkit provided by Intel. OpenVino stands for Open Visual Inference Neural Network Optimization, and was introduced for developing AI applications with high-performance, especially CNN models, on Intel's architectures that is the Intel's CPU, GPU, VPU and FPGA. Applications are mainly deployed on the edge [4], e.g. surveillance cameras, healthcare products and driving cars.

The OpenVino Toolkit is available for Windows, macOS and Linux, and works on the command line. The current version is 2021.2, but we used version 2020.3 that is known to work well for sending models to Intel's Movidius NCS.

OpenVino is meant for inference, meaning directly deploying the model on a device, and for creating a benchmark for inference. The full inference flow of OpenVino is illustrated in Fig. 4.1[10]:

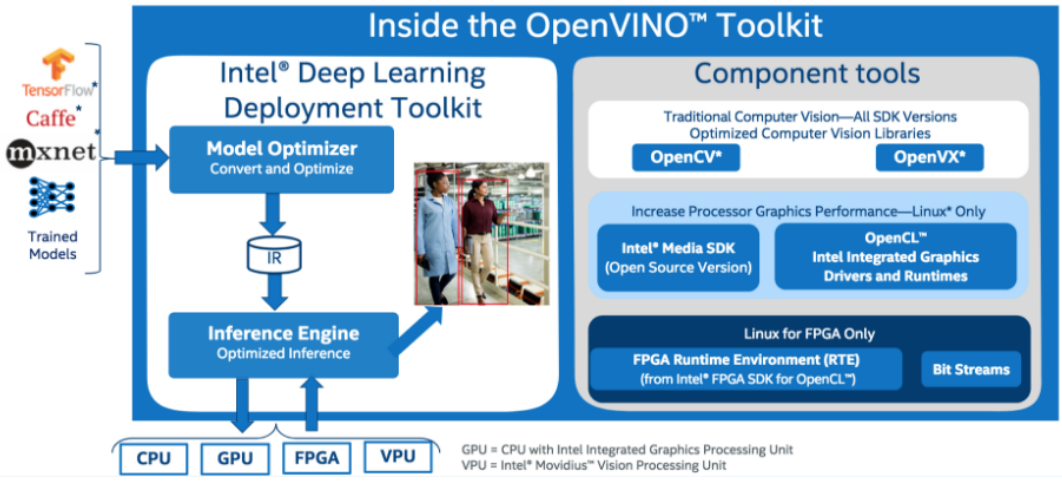


Figure 4.1: OpenVino Workflow

[10] Image source: https://hypraptive.github.io/2018/08/13/optimized-inferenceedge.html

In the next two sections, we take a glance into the two main components, the Model Optimizer and Inference Engine.

4.1 Model Optimizer

The Model Optimizer is responsible to transform the given CNN model into an Intermediate Representation (IR) that is composed of XML and Binary file. The former defines the network topology in a descriptive manner, and the latter contains the associated parameters in binary format. The model must be trained before and frozen afterward, meaning saving the model parameters as constants. This can be done with the Deep Learning library that is used to develop the model. The necessary scripts for producing the IR files are available for TensorFlow, Caffe, MXNet, ONNX and Kaldi models.

4.2 Inference Engine

The Inference Engine provides a facile API for C++ and Python for developing Deep Learning based applications. It reads the input network from the given IR files, and loads it to the desired device for deployment. After that, for given input data, it infers the result.

4.3 OpenVino Model Zoo

The OpenVino Zoo contains a broad collection of pre-trained models, and most of them are already frozen. Many of them are well-known CNN models, the models we described in Section 2.3 are also included. Other models are provided by Intel itself. With OpenVino, these models can be downloaded directly from the command line tool, and can be transformed into IR representation with the Model Optimizer. The corresponding GitHub repository[11] contains all details of all models and all necessary scripts.

[11] https://github.com/openvinotoolkit/open_model_zoo

5

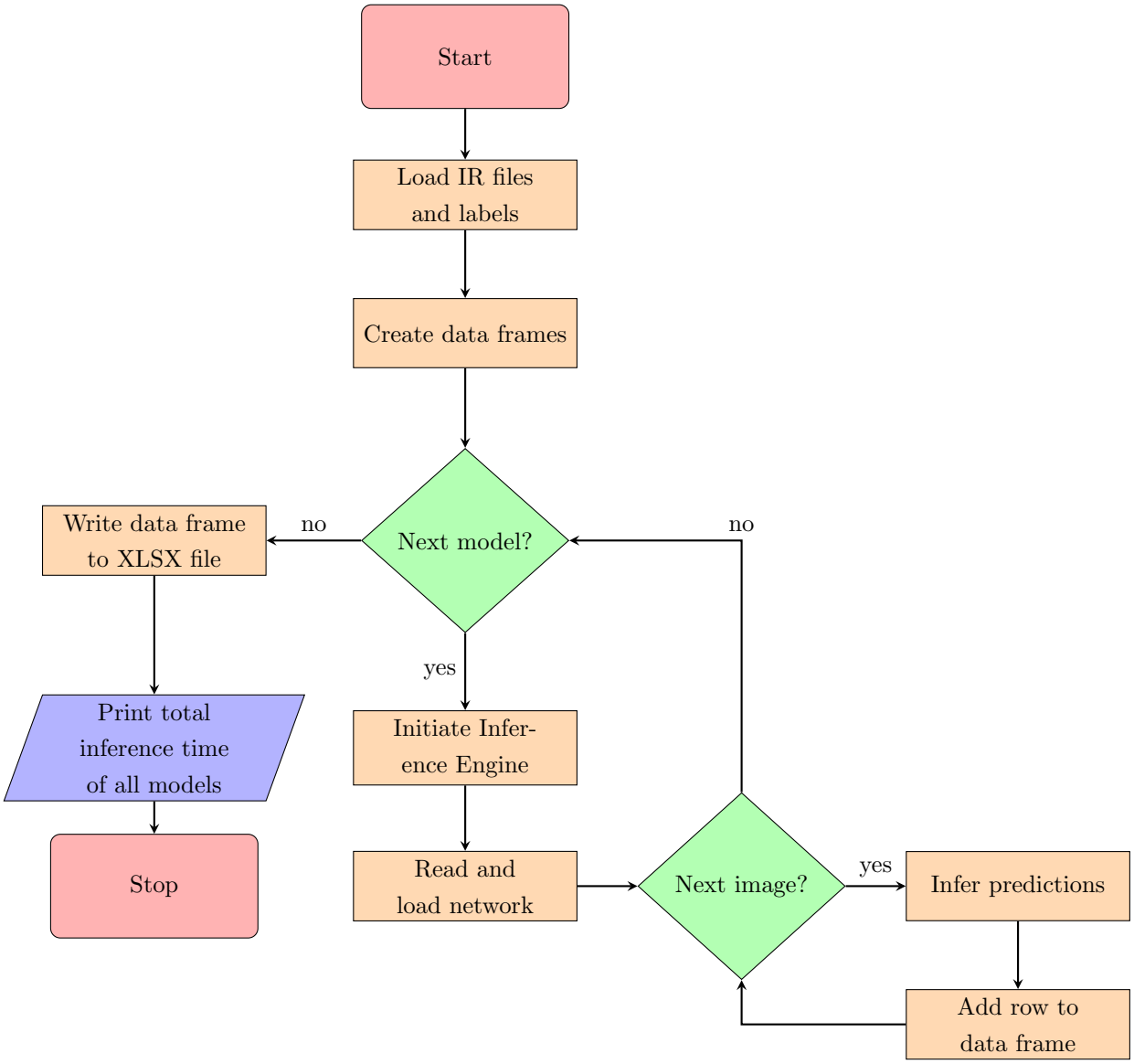
Proposed Evaluation Approach

In this chapter, we describe our approach for the model evaluation. We used OpenVino to deploy all models onto the NCS. We first installed the OpenVino distribution and all required dependencies[12]. We wrote a Python script by applying the Inference Engine's API. All commands we describe here were executed on Windows. The OpenVino environment must be initialized every time. For that, we have run the `setupvars.bat` script that is located in OpenVino's `bin/` directory.

To get the models, we used the `downloader.py` script located in `../open_model_zoo/tools/downloader/`. By typing the `--print_all` argument, it will list all available ANN models. We then chose our models by typing their name as an argument and our desired directory for storing the files. We then used the `mo_tf.py` script for TensorFlow models and the `mo_caffe.py` script for Caffe models, both located in `openvino/deployment_tools/model_optimizer/`. Several times, we had to add the corresponding input shape, mean or scale values of the model. These are explicitly mentioned in the model's Readme documentation in the OpenVino Zoo GitHub repository.
For the images, we considered the ImageNet dataset. We randomly selected 50 images from the test set, and used them for the evaluation of all four models.

Our script `classification_evaluation.py` is based on OpenVino's sample script `classification_sync.py`, and we made our changes for the purpose of this project. The following illustration shows roughly the process of the script:

[12] https://docs.openvinotoolkit.org/2020.3/_docs_install_guides_installing_openvino_windows.html



In the beginning, the script loads the IR files of all models and the ImageNet label file. It additionally initiate four data frames for each model using Pandas[13]. We wanted our results to be exported as an Excel file. After that, two nested loops follows. The outer loop induces one model, and the inner loop sends one image from the given path to the device in each iteration. In the beginning of the outer loop, the Inference Engine is initiated, and the IR files of the model are loaded into the Movidius NCS. In the second loop, the image is first resized, and then sent to the model for inference. The model gives as output the labels and the corresponding predictions. We decided to consider only the top-1 label for the accuracy measure. The intermediate results are written in the corresponding data frame as a new row. At the end, all data frames are written in a XLSX file, and the total inference time in milliseconds are printed out.

[13] https://pandas.pydata.org/

6 Evaluation Results

Our objective of this project work was to test and compare several CNN models on the VPU. We first present our results for accuracy. Consequently, the latency of inferring images are shown. After that, we show the classification rate. Along with the NCS, we also tested the models on two CPUs. The following lists all three processors: Intel Movidius NCS with Myriad 2 with 933 MHz, Intel processor i5 Dual-Core with 2.9 GHz and AMD processor Ryzen 7 1700-X 8-Core with 3.8 GHz.

6.1 Results

For accuracy, we stored, as explained in Chapter 5, the predictions of all 50 images from all four models in an Excel table. The ground truth for the test images from ImageNet was not given. Therefore, we evaluated each prediction manually whether they are classified correctly. The following table and chart show the classification results:

NN Models	**Correct Classified**	**Misclassified**
AlexNet	36	14
Inception v4	48	2
Resnet-50	39	11
SqueezeNet	41	9

Table 6.1: Classification results

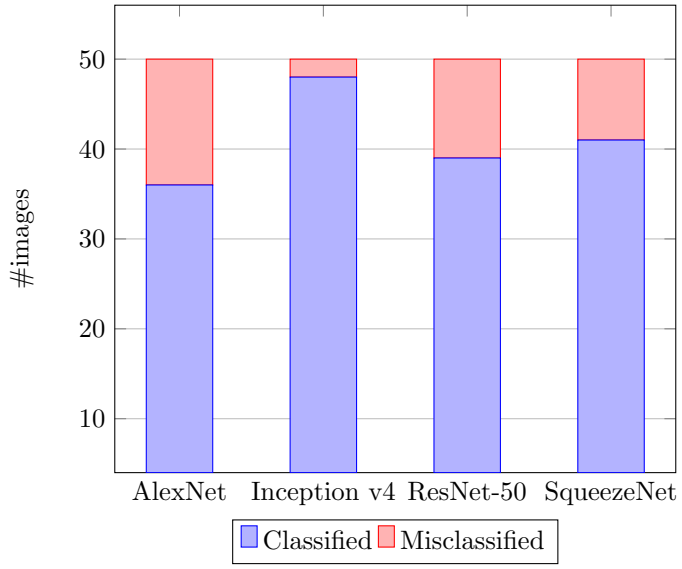


Figure 6.1: Classification results as bar chart

Subsequently to accuracy, we also measured the latency, meaning the time for inferring the prediction of each image in milliseconds, and stored also in the Excel file. In the following a table and a box chart are shown containing measurements for all four models:

Intel Movidius NCS:

NN Models	Min	1. Quartile	Median	3. Quartile	Max
AlexNet	91.029ms	91.947ms	92.021ms	92.234ms	95.099ms
Inception v4	686.003ms	686.955ms	688.720ms	690.133ms	692.721ms
Resnet-50	224.548ms	225.263ms	226.054ms	227.018ms	228.766ms
SqueezeNet	43.006ms	43.009ms	43.010ms	43.011ms	44.011ms

Table 6.2: Latency measurements on NCS

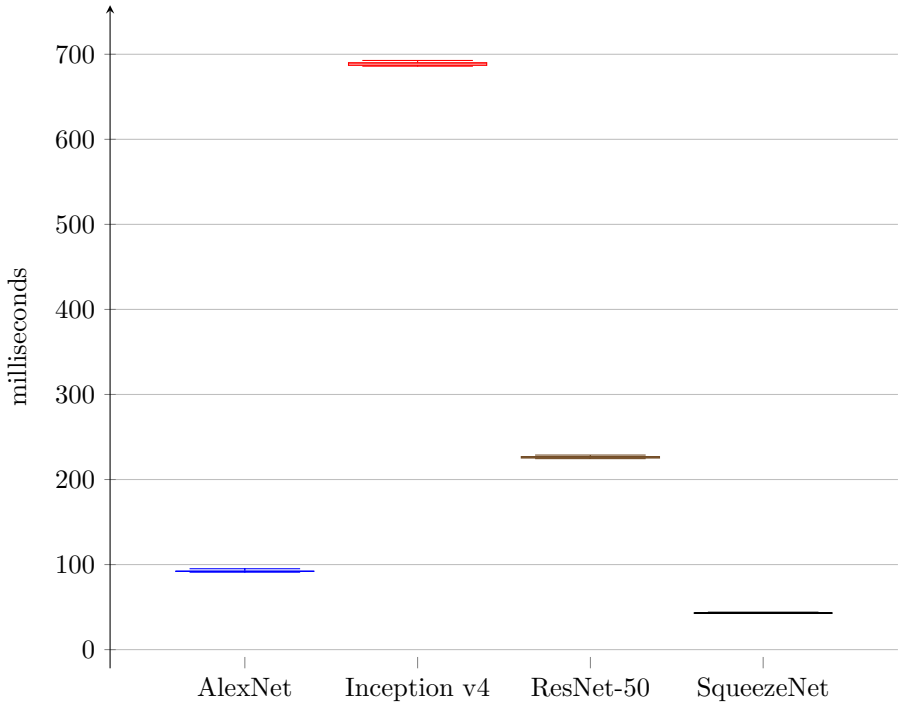


Figure 6.2: Latency measurements on NCS as box chart

Intel processor:

NN Models	Min	1. Quartile	Median	3. Quartile	Max
AlexNet	31.074ms	33.161ms	35.112ms	37.018ms	40.154ms
Inception v4	277.354ms	280.131ms	282.223ms	285.566ms	288.543ms
Resnet-50	119.675ms	121.984ms	123.561ms	125.332ms	127.452ms
SqueezeNet	11.034ms	11.201ms	11.514ms	12.102ms	12.945ms

Table 6.3: Latency measurements on Intel's processor

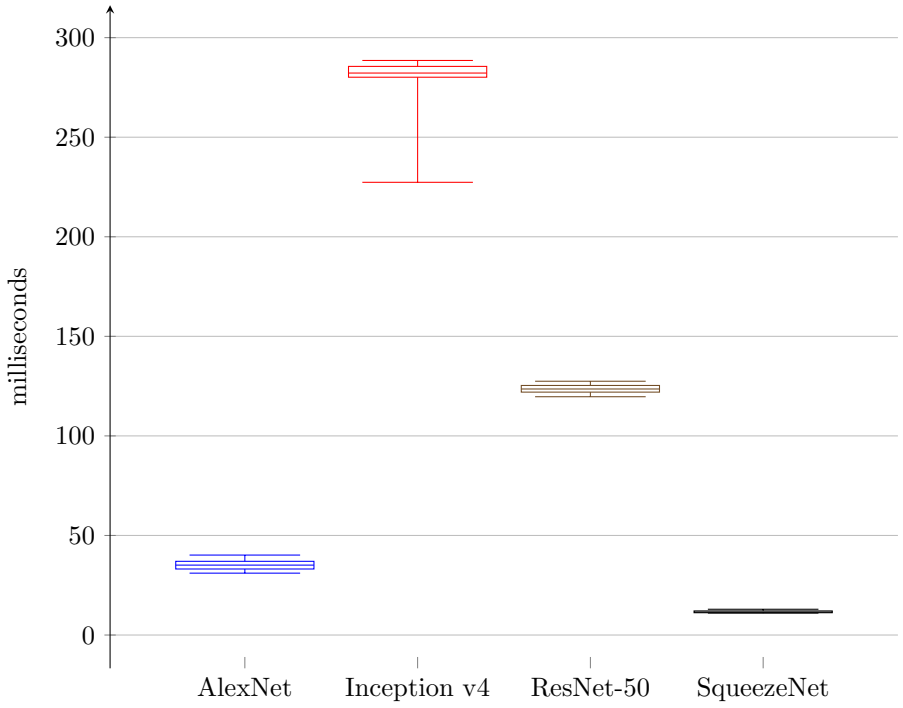


Figure 6.3: Latency measurements on Intel processor as box chart

AMD processor:

NN Models	Min	1. Quartile	Median	3. Quartile	Max
AlexNet	12.013ms	14.006ms	15.005ms	16.008ms	28.491ms
Inception v4	77.017ms	82.023ms	85.020ms	90.031ms	115.026ms
Resnet-50	26.000ms	29.008ms	30.009ms	31.016ms	45.013ms
SqueezeNet	3.000ms	4.001ms	5.001ms	5.002ms	7.003ms

Table 6.4: Latency measurements on AMD's processor

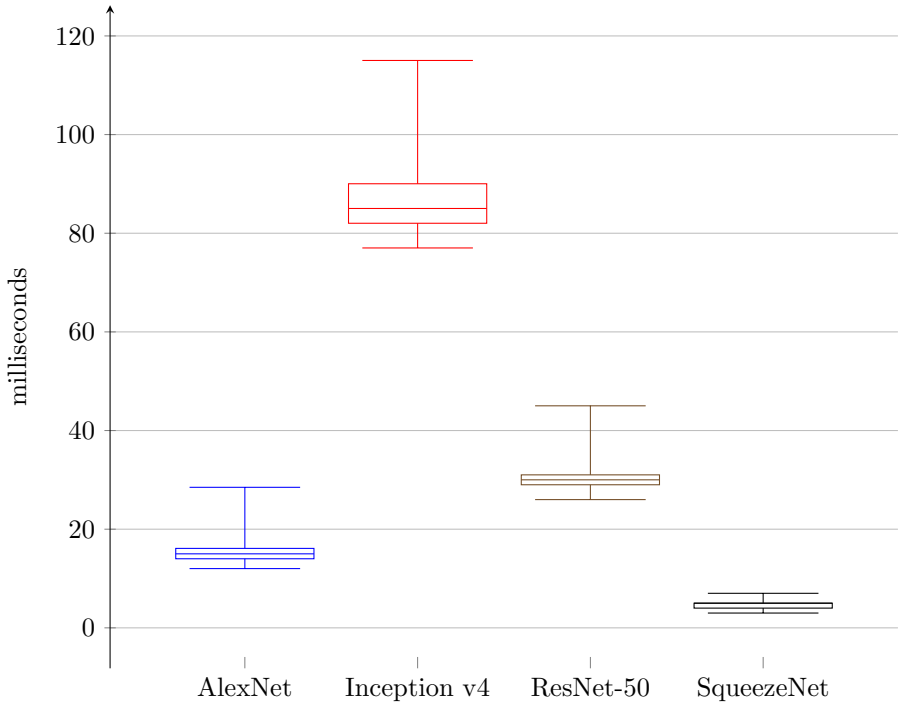


Figure 6.4: Latency measurements on AMD's processor as box chart

Finally, we measured the classification rate of each model, meaning how many images it classified within a second. For each model, we divided the number of images by the overall inference time in seconds. In the following table and chart, the classification rates for all four models are depicted, tested on all three processors.

NN Models	**Movidius NCS**	**Intel Processor**	**AMD Processor**
AlexNet	10.8	30.16	67.93
Inception v4	1.45	3.57	12.74
Resnet-50	4.42	8.21	33.69
SqueezeNet	23.18	89.29	249.25

Table 6.5: Classification rates

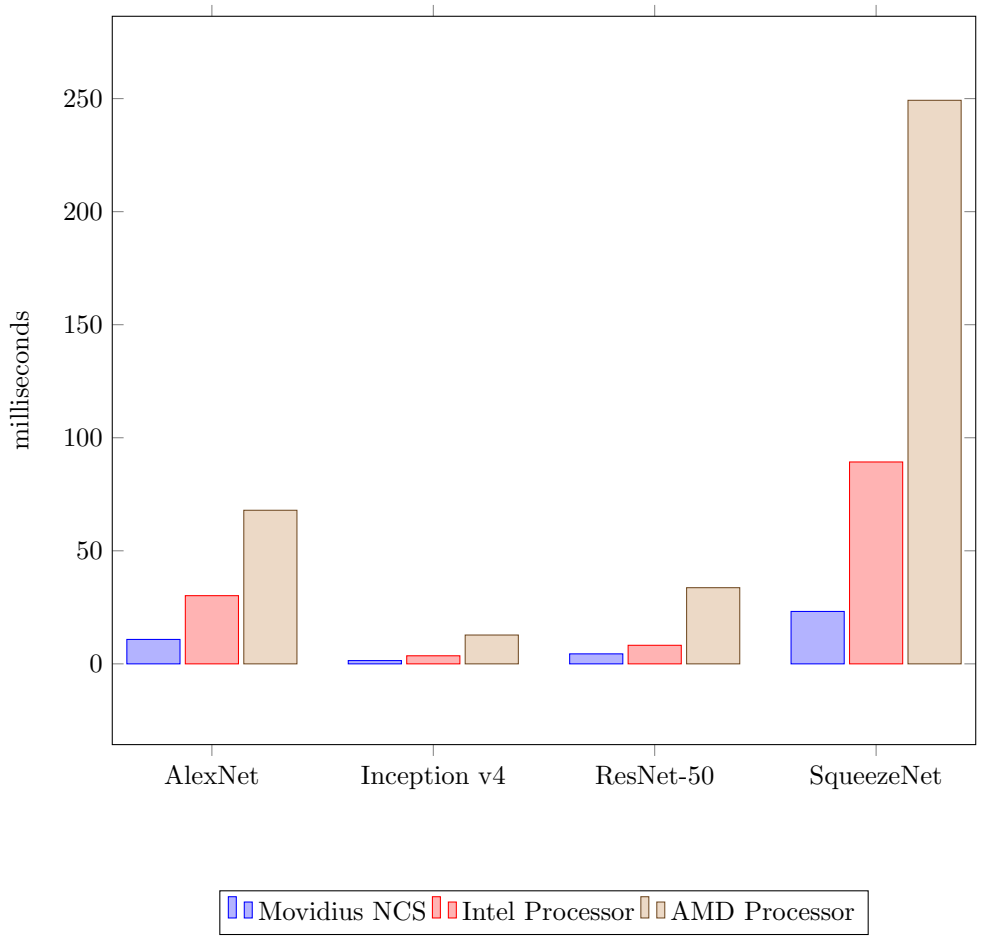


Figure 6.5: Classification rates as bar chart

6.2 Discussion

Overall, all models performed well. Especially, the Inception v4 model results in the best accuracy. For given 50 images, it only misclassified two images that are ambiguous, see Fig. 6.6 and 6.7:

Figure 6.6: Misclassified as “plastic bag” by Inception v4

Figure 6.7: Misclassified as “hook, claw” by Inception v4

Another two ambiguous examples are shown in Fig. 6.8 and 6.9. The first image is classified as “wig” by AlexNet with probability 0.24 and as “hair spray” by Inception v4, ResNet-50 and SqueezeNet with probabilities of 0.99, 1.0 and 0.36. The second image is predicted by AlexNet as “eskimo dog, husky” with probability 0.44, by Inception v4 and Resnet-50 as “malamute” with probabilities of 0.94 and 0.82, and as “siberian husky” by SqueezeNet with probability 0.36.

Figure 6.8: Classified as “wig” and as “hair spray”

Figure 6.9: Classified as “husky”, “siberian husky” and as “malamute”

These sort of images hindered us to decide the correctness of the classification. Since we did not possess the ground truth of our test images, we eased the assessment and considered multiple proposed labels as correct if they have made sense.
In the first round of the evaluation, we asserted that the models can still predict the images correctly even if the probabilities are poor. This instance appeared seldom. Thus, we decided to not set any confidence level for the inference, otherwise we would have discarded many true positives. Fig. 6.10 shows an example that is classified correctly by AlexNet, but with a probability of 0.29.

Figure 6.10: Classified correctly as “crash helmet” by AlexNet

The accuracy measure indicates that AlexNet has the most misclassifications. However, this fact does not allow us to determine AlexNet as a non-prominent model. It won the ILSVRC 2012, and all further models are at least based on this model. To be more certain, a lot more images are needed for the assessment.
The gain of accuracy with Inception modules is replicated in this result. However, the time that took for inferring the results for 50 images is according to our perception too long. This can be improved by applying the asynchronous inference request which we cover in Chapter 7. Another interesting observation that we already predicted is the performance of SqueezeNet with similar accuracy as AlexNet. It has fewer misclassifications than AlexNet, and is around two times faster than AlexNet and 16 times faster than Inception v4. This observation is crucial for running applications on the edge with low memory capacity.
Regarding the classification rate, we can clearly see that the NCS took much longer time for inference than the CPUs. On average, the AMD’s processor can classify around 6-10x and the Intel’s processor around 2-4x times more images per second than the NCS. For example, the AMD’s processor is able to recognize roughly 250 images with SqueezeNet, but the Movidius NCS only 23 images. The lowest classification rate is 1-2 by GoogleNet v4 which is poor. For this reason, we rather use SqueezeNet for the NCS whose accuracy is acceptable. With asynchronous inference method, however, the performance would be invincible with Inception v4.

6.3 Future Work

As future work, we can scale this evaluation to an HPC cluster. An example would be the $\mu-$cluster[14] from the High-Performance Computing Research Group of the University of Basel. It is composed of 64 Odroid computers, each equipped with a Movidius NCS. An extension of this work would be to develop a complete benchmark to measure the performance over 64 NCSs with several measurements. The Benchmarking tool from OpenVino[15] can be taken as reference. Similar to this work, we could again test the performance of CNN models and observe whether the NCSs behave differently.
Along with that, multiple images or video frames can be distributed among the NCSs. The distribution can be realized with Message Parsing Interface (MPI) for Python[16] by selecting one Odroid node as master and all others as slaves. Apart from image classification models, other models for object detection and image segmentation can be considered.

Another idea for the future work, a sophisticated approach would be the distribution of CNN models onto the HPC $\mu-$cluster. Rivas-Gomez et al. [15] presented an exploration of a multi-VPU configuration. They tested the performance by utilizing eight NCS devices simultaneously, and determined that they can keep up with the CPU's and GPU's performance. This could motivate us to develop a system for offloading scientific tasks on the HPC $\mu-$cluster with the help of OpenMP[17].

[14] https://hpc.dmi.unibas.ch/en/research/micro-cluster/
[15] https://docs.openvinotoolkit.org/2020.3/_inference_engine_tools_benchmark_tool_README.html
[16] https://bitbucket.org/mpi4py/mpi4py/src/master/
[17] https://www.openmp.org/

7
Lessons Learned

In this chapter, we describe the sort of troubles we had to face, mainly while with NCSDK.

The first restriction was that NCSDK only works on Ubuntu 16.04. We, therefore, prepared a virtual machine and installed NCSDK by downloading the corresponding GitHub repository[18]. There exist two versions of NCSDK, but we considered v1.
Similar to OpenVino, a GitHub repository[19] is available containing shared apps along with pre-trained TensorFlow and Caffe models. The Make file for each model contains all commands for downloading, compiling and checking the model (NCSDK software tools). But the links to the model were deprecated. This was the case for many Caffe models. Alternatively, we tried to use the files from OpenVino Model Zoo and to compile the Graph file from them, but it did not work.

The second try was with TensorFlow models. We tried Mobilenet-SSD, TinyYolo and Inception. Only the Mobilenet-SSD model was working. TinyYolo has, same for Caffe models, deprecated links in the Make file. For TensorFlow models, the Make file downloads the corresponding models as TAR-file directly from the TensorFlow page and extracts it. The issues arose for exporting model weights using the TensorFlow Library. The error was: `AttributeError: module 'tensorflow_core.compat.v1' has no attribute 'contrib'`. This error appeared for TensorFlow versions >1.14 which is not supported anymore. When trying version <1.14 the following error arose: `ImportError: No module named 'tensorflow.compat.v1'`. This problem turned up almost for all models. Due to the lack of experience, we lost a lot of time in the beginning of this project. For the future, we propose to download the models directly from the TensorFlow and Caffe pages, and try to freeze them manually. We tried this way, but failed in exporting the Graph file.

One challenge we had working with OpenVino is to employ synchronous and asynchronous

[18] https://github.com/movidius/ncsdk
[19] https://github.com/movidius/ncappzoo

inference requests [1] in the performance evaluation script. These paradigms are utilized to obtain results for multiply data, e.g. multiple images or batches from images. The difference is in terms of processing images through the model. The synchronous method conducts the images sequentially, and the asynchronous method prepares the next image in parallel while the former image is still be processed. These requests allow a higher throughput, especially helpful for working with large number of images. Unfortunately, neither of them did work for the Movidius NCS. We could not find any explanation, but we assume that these inference requests are only supported by Intel Neural Compute Stick 2.

8 Conclusions

The initiative of this project work was to test and compare CNN models on Intel's Movidius NCS. Efficient high-performance computing is essential for AI on the edge. We contemplated the VPU as a way for deploying and accelerating Deep Learning models. We saw the general concept of ANN, particularly CNN, and chose four CNN models for our evaluation. Furthermore, we delineated the basic network architecture of each model. After that, we briefly registered some technical details of the Movidius NCS, and saw its possibility of parallelizing the Deep Learning model with the embedded SHAVE processors. We had our troubles with NCSDK that led us to work with OpenVino. It is able to create and deploy AI based applications on various Intel platforms including the Movidius NCS. It comprises two components: the Model Optimizer for creating IR files from CNN models and the Inference Engine that facilitates developing AI applications.

For the evaluation part, we wrote a Python script based on OpenVino Inference Engine's API , where we loaded our CNN models onto the NCS and measured the performance. As test data, we randomly picked 50 images from the ImageNet test set and inferred our results. We measured the models in terms of accuracy, latency and classification rate for VPU, and additionally on two further CPUs. In the results, we saw the Inception v4 model as the best model for predicting the images with high accuracy, but is slow in inference. The SqueezeNet, however, has a lower accuracy than Inception v4 but higher than AlexNet and ResNet-50, and has the best classification rate with 250 images per second. In the end, we proposed a potential improvement to include asynchronous request method from the Inference Engine's API, that can help to improve the accuracy of the models and to infer batches of a larger data set.

In the future work, we suggested extending our evaluation approach to a complete Benchmark to enable further measurements of the models and to distribute the data among multiple NCSs. Another further work would be to scale tasks among multiple distributed NCSs on the HPC μ-cluster.

Bibliography

[1] Openvino Documentation: Overview of Inference Engine Plugin Library. https://docs.openvinotoolkit.org/latest/ie_plugin_api/. Accessed: 2020-12-30.

[2] Intel Movidius^{TM} Neural Compute Stick. https://movidius.github.io/ncsdk/ncs.html, . Accessed: 2020-12-30.

[3] Intel Movidius^{TM} Myriad^{TM} 2 Vision Processing Unit 4GB. https://ark.intel.com/content/www/us/en/ark/products/122461/intel-movidius-myriad-2-vision-processing-unit-4gb.html?_ga=2.178120876.888339587.1609612383-1649145100.1605605305, . Accessed: 2020-12-30.

[4] Intel Distribution of Openvino^{TM} Toolkit. https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html, . Accessed: 2020-12-30.

[5] Intel Movidius^{TM} Neural Compute SDK Documentation. https://movidius.github.io/ncsdk/index.html. Accessed: 2020-12-30.

[6] Openvino Toolkit Documentation. https://docs.openvinotoolkit.org/2020.3/index.html. Accessed: 2020-12-30.

[7] Sophie Lebrecht Director of Operations, xnor.ai at 2018 GeekWire Cloud Tech Summit. Ai at the edge: Ultra efficient AI on low power compute platforms. URL https://www.youtube.com/watch?v=XfNhN1UPESo. Accessed: 2021-01-06.

[8] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2nd edition, 2019. ISBN 9781492032649.

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

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 06 2016. doi: 10.1109/CVPR.2016.90.

[11] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. URL http://arxiv.org/abs/1602.07360.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. URL https://doi.org/10.1145/3065386.

[13] Dennis Madsen. Pattern Recognition Course University of Basel, Deep Learning for Computer vision: Convolutional Neural Networks (CNNs), 2019.

[14] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN 0262018020.

[15] Sergio Rivas-Gomez, Antonio J. Peña, David Moloney, Erwin Laure, and Stefano Markidis. Exploring the vision processing unit as co-processor for inference. *CoRR*, abs/1810.04150, 2018. URL http://arxiv.org/abs/1810.04150.

[16] Volker Roth. Machine Learning Course University of Basel, Section 5: Neural Networks, 2020.

[17] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 4278–4284. AAAI Press, 2017.