technische universität dortmund

Master Thesis

RESOURCE-AWARE ACCELERATION FOR NANO PARTICLE CLASSIFICATION

Mikail Yayla

June 2019

LS 12 (Embedded Systems)
Department of Computer Science
TU Dortmund University

Supervisors:
Prof. Dr. Jian-Jia Chen
Dr.-Ing Kuan-Hsun Chen

ABSTRACT

Due to the evolution and emergence of viruses, and the increased global travel and transport, there is a need for a mobile system that performs virus detection in real-time. The Plasmon Assisted Microscopy Of Nano-sized Objects (PAMONO) biosensor is a viable candidate to fulfil the requirements of a mobile virus detection device. The sensor uses Surface Plasmon Resonance (SPR) images to detect virus particle bindings. For virus detection techniques, state-of-the-art methods which use multiple convolutional neural networks (CNNs) in series can provide high accuracy. However, due to the resource constraints, such techniques are not suitable for small mobile and embedded devices due to high computational requirements. In this thesis, two main methods are explored, that aim to make the virus detection more suitable for resource-constrained systems by optimizing the PAMONO image analysis pipeline.

In the first part of this thesis, a frequency domain analysis based nano-particle classification is proposed, which is faster and less resource-demanding than the CNN method. The approach is implemented by modifying the classification stage of the PAMONO nano-particle image analysis pipeline. Extensive evaluations are run on different platforms, and the results show that the modified classification is significantly faster than the state-of-the-art methods based on CNNs by sacrificing only 1 to 2.5% points of accuracy. The frequency analysis method is 2.6 times faster than the CNN, when using a mobile GPU as an accelerator. The method is further sped up by performing the classification as a CPU-only version, i.e., from 1280 to 29 microseconds for the Fourier features, and from 1500 to 17 microseconds for the Haar wavelet features. Nanoparticle classification systems using frequency analysis methods are a more resource-efficient alternative to CNNs, thus the methods presented in this part of the thesis can serve as a blueprint for the design of resource-efficient virus classification methods for SPR images.

In the second part of the thesis, the usefulness of the Single Shot Object Detector (SSOD) yoloV2-CNN architectures are explored to optimize several parts of the image analysis pipeline, by combining the detection and classification stage into one step. The results show that a simple yolo architecture with a shallow structure and fewer layers and kernels than the tiny-yolo architecture can reliably detect and classify particle bindings in 200 nm images, i.e., with around 95% precision and 80% recall. How to improve the classification performance for 100 nm, i.e., around 90% precision and 55% recall, remains an open problem, and a few ideas are discussed.

PUBLICATIONS AND CONTRIBUTIONS

The materials from the following works are included in an extended version:

- Mikail Yayla, Anas Toma, Jan Eric Lenssen, Victoria Shpacovitch, Kuan-Hsun Chen, Frank Weichert, and Jian-Jia Chen. Resource-Efficient Nanoparticle Classification Using Frequency Domain Analysis. In: Bildverarbeitung für die Medizin (BVM), Lübeck, Germany. 2019

- Mikail Yayla, Anas Toma, Kuan-Hsun Chen, Jan Eric Lenssen, Victoria Shpacovitch, Roland Hergenröder, Frank Weichert, Jian-Jia Chen. Nanoparticle Classification using Frequency Domain Analysis on Resource-limited Platforms. In: Sensors, Special Issue on SPR-Based Sensors and Their Biological Applications

Following parts of the thesis are based on the works of Dr. Anas Toma, who produced the results for our collaboration in the unpublished journal version. The materials have been added to the thesis for completeness.

- The work on the feature analysis, i.e., Section 3.1.2

- The production of Figure 14 in Section 3.2.6

ACKNOWLEDGEMENTS

I would like to thank my supervisors Prof. Dr. Jian-Jia Chen and Dr.-Ing. Kuan Hsun Chen for their excellent support and for providing me with many valuable opportunities to learn, explore, and grow.

I would also like to thank Dr.-Ing. Anas Toma, Jan Eric Lenssen, and Dr. Frank Weichert for their help in improving the work for this thesis.

This work was supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), subproject B2 and A1.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

PAMONO	Plasmon Assisted Microscopy Of Nano-sized Object
SPR	Surface Plasmon Resonance
CPU	Central Processing Unit
GPU	Graphics Processing Unit
FFT	Fast Fourier Transform
FWT	Fast Wavelet Transform
CNN	Convolutional Neural Net
DT	Decision Tree
RF	Random Forest
SSOD	Single Shot Object Detection
MPSoC	Multi-Processor System-on-Chip
FPGA	Field Programmable Gate Array
PL	Programmable Logic
IP	Intellectual Property
TP	True Positive
FP	False Positive
FN	False Negative

1 INTRODUCTION

This thesis explores modifications on the existing image analysis pipeline of the PAMONO biosensor. The work in this thesis is split in two seperate parts, in which two methods are explored: Using frequency analysis techniques for the classficiation stage, and using a SSOD for detecting and classifying the virus excitations.

1.1 MOTIVATION

The evolution and emergence of viruses coupled with global travel and transport entail the risk of spreading epidemic diseases. Therefore, there is a need for accessible mobile real-time virus detection, preferably in the size of a small mobile device. The PAMONO biosensor is a viable candidate to perform such a task [29]. It captures a sequence of images which need to be analyzed by an image analysis pipeline. The state-of-the-art solutions based on multiple Convolutional Neural Networks (CNNs) in series, e.g., proposed by Lenssen et al. [14], provide good classification and execution time performance on general purpose hardware. However, CNN models have complex architectures which require a considerable amount of resources, such as high-performance GPUs, to be evaluated [4, 17]. Although the research of developing dedicated hardware for the evaluation of the CNNs is in progress [4, 21], there is still no general purpose solution with low power consumption available for mobile and embedded systems according to the best of the author's knowledge. Therefore, modifications of the existing image analysis pipeline are evaluated in this thesis, which aim to reduce the demand for resources, while keeping the execution time and the detection performance high.

1.2 CONTRIBUTIONS

The contributions of this thesis can be summed up in two parts.

Firstly, the classification stage of the PAMONO image analysis pipeline is sped up significantly compared to the CNN solution, e.g., from 3370 µs to 29µs for the Haar wavelet features and to 17µs for the Fourier features, when measured on an Intel Core i7 4600U.

Secondly, the yoloV2-CNN architecture is explored to simplify and combine several steps of the PAMONO image analysis pipeline. The results show precision values of around 95% and recall values of 80% for 200 nm particles, and for the 100 nm case around 90% precision and around 55% recall. A more detailed overview follows below.

In the first part of this thesis, modifications on the classification stage of the PAMONO image analysis pipeline, which was originally proposed by Siedhoff et al. [29], are explored. The modifications comprise using frequency domain analysis features for the feature extraction stage and small decision trees for the classification stage. The modifications make the image analysis pipeline more suitable for small mobile and embedded systems. The idea to use frequency domain analysis for the classification is originally given by Lenssen et al. in [14]. The results show high classification performance, but are one percent point less in accuracy than the CNN. However, the resource-efficiency of the approaches compared to the CNN is not explored. As examples of frequency domain analysis techniques, Fourier and Haar wavelet features which are acquired computing the Fast Fourier transform (FFT) and the Fast wavelet transform (FWT), are explored. These feature extraction and classification methods are evaluated on different platforms with different implementations. First, a mobile GPU-accelerated feature extraction is evaluated, considering execution time and classification accuracy. It is then compared to the CNN in terms of classification accuracy and execution time. Since the results show that the synchronization overhead between the CPU and the GPU makes up a big part of the execution time, a CPU-only version of the frequency analysis methods is evaluated, and is compared to the accelerated version. To evaluate the execution time of the feature extraction and classification on a small embedded system, the methods are also evaluated on the ARM chip of the Zedboard as a CPU-only version, and as a version which offloads the FFT computation to the Progammable Logic (PL) region using the xfft IP from Xilinx.

In the second part of the thesis, the use of the Single Shot Object Detector (SSOD) yoloV2-CNN for virus detection is explored. The goal is to detect the virus excitations by combining the detection and classification stages of the original pipeline into one step. In the literature, SSOD techniques were not tried before on this problem since they were too resource demanding a few years ago. Only recently a resource-efficient version called tiny-yolo was proposed by [23]. Xilinx recently showed that their MPSoC FPGAs can run small versions of the yoloV2-CNNs with quantized and binarized parameters [20] for real-time image processing with around 16 FPS. They execute the darknet framework [22] on the CPU and redirect the layer operations of the CNN to software-defined hardware functions in the PL. This enables the use of the MPSoC platform, such as the Zedboard, as a potential platform candidate to execute the virus detection with yolo. For improving the existing image analysis pipeline, the benefit of yolo is that it combines several parts of the image analysis pipeline into one step, while being one of the most resource efficient SSOD methods up to date. Therefore, in this work, the usefulness of several yolo architectures is explored, with the goal to simplify and improve the current virus detection pipeline.

1.3 THESIS STRUCTURE

The thesis is structured as follows:

In Chapter 2, the PAMONO sensor's principles of operation, its image analysis pipeline, and the data it provides are presented.

In Chapter 3, the modifications of the classification stage of the image analysis pipeline with frequency analysis methods are presented, and the performance of the modifications are evaluated.

In Chapter 4, the modifications of the detection stage of the image analysis pipeline with the yolo-CNN are explored, evaluated, and the usefulness for virus detection is discussed.

In the Appendix A, the implementations of the OpenCL kernels and the yolo configuration files are presented and documented.

Chapter 5 concludes the main ideas of the thesis and discusses the future work.

2 PAMONO SENSOR

The Plasmon Assisted Microscopy Of Nano-sized Objects (PAMONO) sensor has been a topic of research for many years in a collaboration between TU Dortmund and the Leibniz-Institute for Analytical Science. It was originally invented by Zybin et al. [32]. The sensor can be used to detect virus particle bindings in a liquid sample in real-time on a mobile system. It has a wide range of applications in medicine, e.g., for the automatic diagnosis of virus infections [32].

This chapter describes the sensor's principles of operation, presents its image analysis pipeline, and introduces the data provided by the sensor for the automatic detection of the virus bindings. The contents for this chapter are based on the work in Chapter 2 of Libuschewski's PhD thesis in [15], and on the work by Lenssen et al. [14] and Zybin et al. [32].

2.1 PAMONO SENSOR OVERVIEW

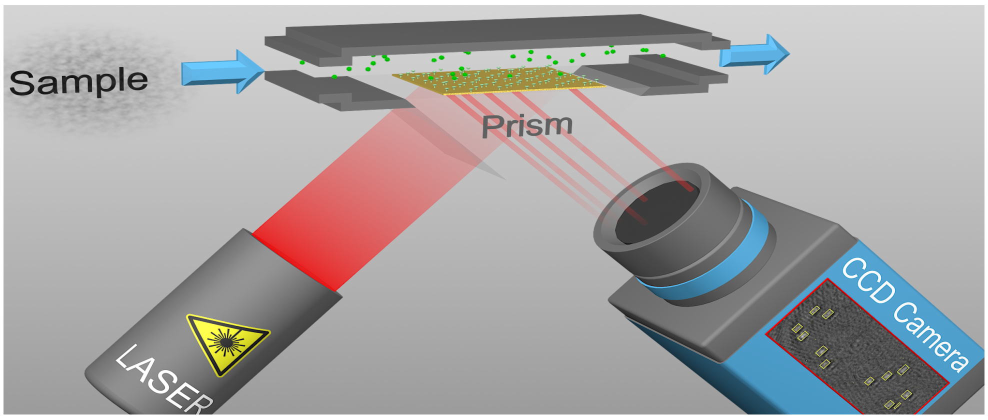


Figure 1: A schematic of the PAMONO biosensor, adapted from [16].

The PAMONO sensor is an optical biosensor which is used to detect nano-particles in liquid or gas samples [27, 32]. In general, viruses are too small to be detected with light. The sensor however utilizes Kretschmann's scheme [13] of plasmon excitation to visualize the individual bindings of particles to antibodies that are applied to a gold surface. For the devices, only common optics and cameras are used. A sketch of the setup and operation of the PAMONO sensor is shown in Figure 1. First, the sample is pumped with a rotatory pump or a syringe into the flow cell with the gold plate, onto which the viruses can attach due to the antibodies on it. A laser is pointed to the gold surface such that a CCD camera can record the surface over time. The camera produces image sequences which can then be analyzed. When a nano-particle binding occurs on the gold surface, intensity steps can be observed in the image sequences. These steps appear on images as blob-like structures with wavelike excitations around them.

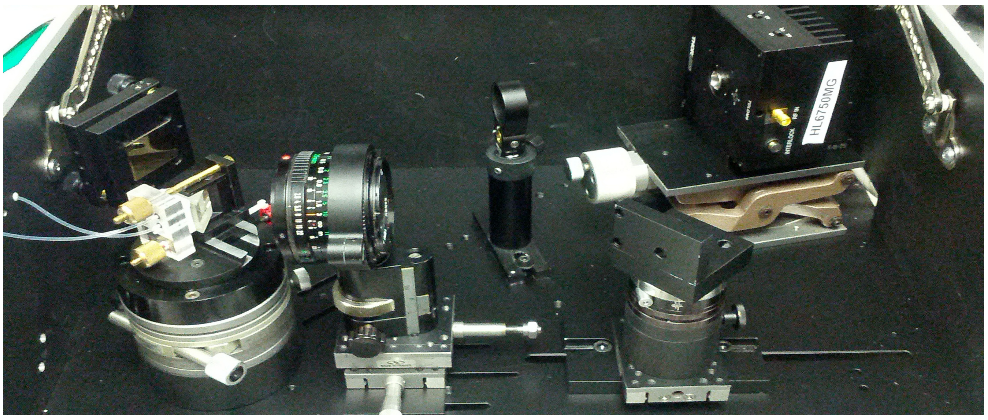


Figure 2: The PAMONO biosensor, adapted from [15].

In Figure 2, a mobile version of the actual sensor setup is shown. The mobile version needs to be operated on top of a stabilizing platform, to protect it from vibrations. In the top right corner is the laser, which is directed towards the sample in the very left. Between the laser and the sample is a lens, which focuses the laser beam. The laser beam is reflected from the gold plate to the camera, which is located in the bottom right of the sensor, but is unmounted in this figure for visibility. Note that the camera should have a bit depth of at least 12 bit, since e.g., a brightness increase of only 6% is caused by the binding of 200 nm polystyrene particles. The quantization error would be greater and would lead to a more difficult detection, if, e.g., an 8 bit depth was used [15]. The sensor is currently the size of an average Desktop PC, around $450\text{mm} \times 300\text{mm} \times 200\text{mm}$, but, as there is still a lot of empty space in the housing, it can be miniaturized further to the size of smaller device, which can possibly be hand held. More details about the PAMONO sensor and the devices used to assemble it can be found in [14, 15, 27].

2.2 PAMONO IMAGE ANALYSIS PIPELINE

Figure 3 shows a general view of the image analysis pipeline of the PAMONO sensor. The size of the raw input images can vary, e.g., 1080×145, 706×167, or 450×170 pixels, depending on the given dataset. In the preprocessing stage, the constant background noise signal is removed from the raw sensor image using the model proposed in [29]. In the detection stage, the resulting image is segmented into smaller patches, e.g., of size 48×48. The smaller patches are then classified as virus or no virus in a subsequent classification step.

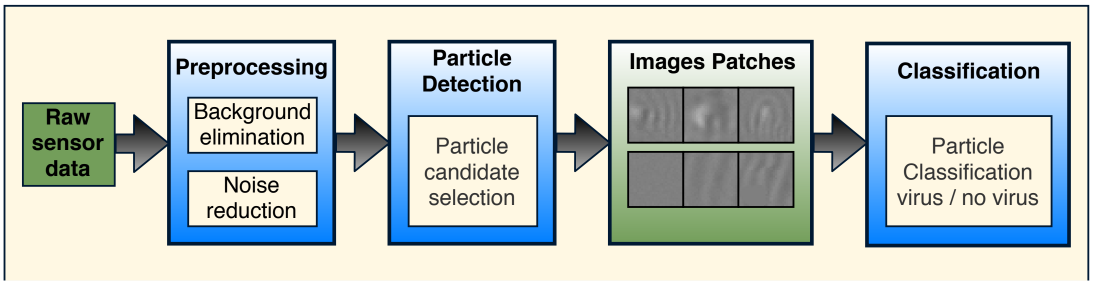


Figure 3: The PAMONO image analysis pipeline.

Several techniques for the detection and classification stage and their performance are detailed by Lenssen et al. in [14]. To summarize the state-of-art approaches briefly, in [14], the original PAMONO pipeline by Libuschewski [15] was modified with light weight CNNs, which are executed in series. The current state-of-the-art method for the detection stage is a fully convolutional neural network which uses a stack of 8 consecutive images from the data stream as input. It detects the virus binding excitations by performing binary segmentation by outputting a 2-class confidence map. It uses Fire Modules, which have three convolutional layers, one 1×1 convolution for feature reduction, followed by two convolutions, 3×3 and another 1×1. The results of these last two convolutions are concatenated along the feature dimension. Two maxpool operations are used, and one $4\times$ upscaling to produce a segmentation with the size of the input. Note that the number of filters for any convolution is no greater than 32. The second CNN classifies the proposed image patches. It uses three Fire Modules, two convolutional layers, five pooling layers, and one fully connected layer. The number of filters for any convolution is no greater than 64. Between detection and the classification, a region proposal algorithm by Libuschewski [15] is executed. It is based on marching squares on the segmentation map to produce contours.

2.3 PAMONO SENSOR DATA

For the PAMONO sensor data, different types of datasets are available. There are datasets from signals of Virus Like Particles (VPLs) and polystyrene particles, for particle sizes of 200 nm and 100 nm, the latter of which are harder to detect since they are smaller in size and are more faint.

Virus Like Particles (VPLs) have a similar structure to viruses and a similar binding behaviour to antibodies. The benefit of VPLs is that they can be used without risk of contamination. They have a similar structure and dimensions to real viruses, but are harmless. On the other hand, for binding, the polystyrene particles do not need antibodies, only a small electrical charge is sufficient to bind them to the gold surface. The bindings recorded by polystyrene particles are almost indistinguishable from the bindings produced by real viruses or by VLPs. In this thesis, only datasets containing bindings from polystyrene particles are used for training and testing.

After the datasets are recorded and stored, they need to annotated by experts. Depending on the particle concentration in the liquid, hundreds of bindings can be observed. Using experts to annotate such an amount of data anew for every sample is time-consuming, therefore synthetically generated datasets are created using the Synthesis-Optimization-AnalySis (SynOpSis) approach proposed by Siedhoff in [28]. Data from already annotated virus bindings are used to produce plausible synthetic datasets. For the work in this thesis, only synthetic data is used for training and testing.

Polygon annotations which surround the binding are created by experts for every binding occurring in the sample. The annotation file contains the image name and frame number, (x, y)-position and height and width of the bounding box surrounding the excitation, all coordinates of the vertices, and other annotations, which are not relevant for this thesis. The differences in intensity which the binding excitations induce are however not visible for the human eye, therefore the constant background noise and the camera noise have to removed. Contrast and brightness changes are performed to make the binding excitations more visible.

The datasets from which the Figures 4-8 originate from can be found on the webpage of the SFB876-B2 project [26]. In Figure 4, the raw signal image from a synthetic dataset with polystyrene particles is shown. In contrast, in Figure 5, the constant background noise is shown. For the human eye, these two images are not distinguishable.

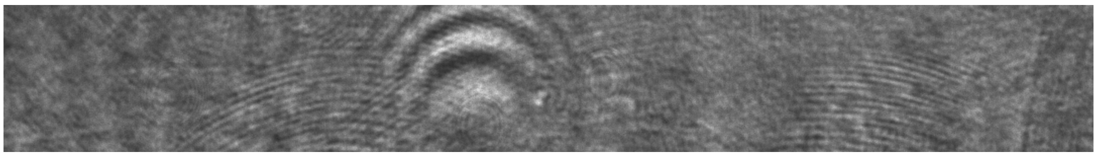


Figure 4: The raw PAMONO sensor data. The blob in the middle belongs to constant background noise. The curved lines as well.

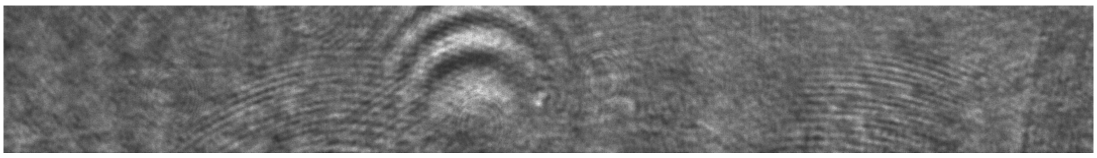


Figure 5: The PAMONO sensor background signal. It is not distinguishable from the raw sensor data for the human eye.

In Figure 6, the portion of the signal with removed background and camera noise is shown. In the middle of the figure the excitation can be seen as blobs. Figures 7 and 8 show images with more excitations. When more time passes, more excitations are recorded, and more excitations can be found on the images.

As described in Section 2.2, in the detection stage of the PAMONO image analysis pipeline, regions are proposed, to classify them as virus or no virus. In Figure 9

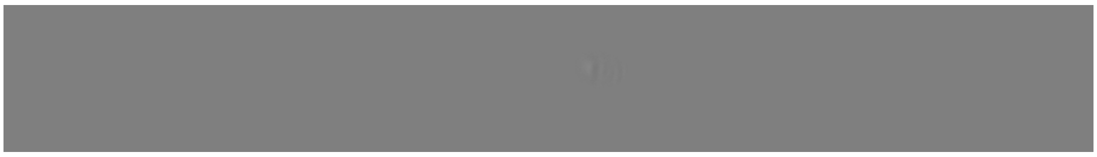


Figure 6: The particle signal extracted from the raw PAMONO sensor data. In this figure, only one excitation is present, as not much time has passed since inserting the sample.

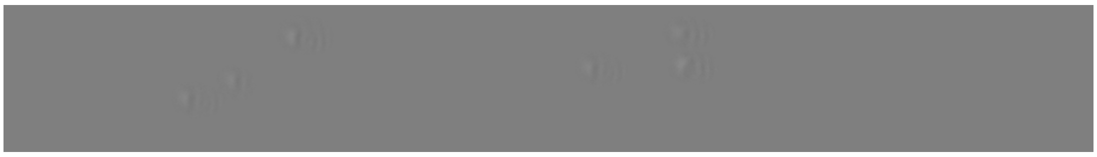


Figure 7: The particle signal extracted from the raw PAMONO sensor data. A few excitation are present, and a bit more time passed since the recording in Figure 6.

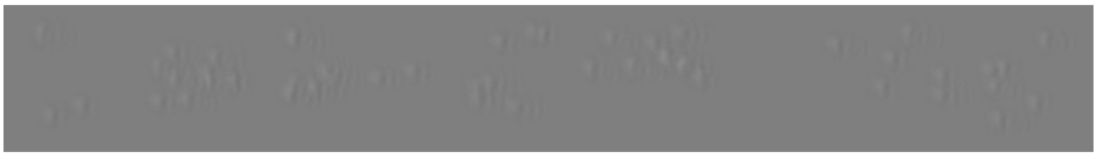
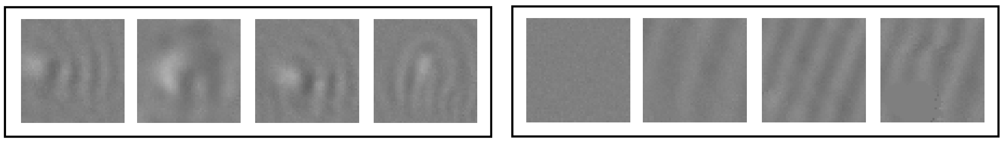


Figure 8: The particle signal extracted from the raw PAMONO sensor data. Many excitation are present, because a lot more time has passed since the insertion of the sample, thus many binding have occurred.

positive and negative region proposals are shown. Positive samples have a blob-like excitations with surrounding periodic patterns around them. Negative samples on the other hand do not have blob-like excitations, they may only contain periodic patters or other artifacts.



(a) Positive samples

(b) Negative samples

Figure 9: Proposal regions in (a), negatives samples in (b). These proposal regions can be generated with different particle binding detection methods.

2.4 SUMMARY

In this chapter, the PAMONO biosensor and the PAMONO datasets have been presented. The sensor can process images in real-time and it is possible to build a portable, hand held device for mobile use. The benefit of the PAMONO sensor over other SPR sensors is that it can observe and analyze individual virus bindings, which enables a wide range of applications in medicine. The training and testing images used for the modified image analysis pipeline in the following chapters originate from the datasets described in this chapter.

3 PAMONO CLASSIFICATION STAGE MODIFICATION

In this chapter, alternative classification methods for the PAMONO image analysis pipeline are presented, which have less resource demands than CNNs, and sacrifice only a small loss in accuracy. CNN solutions for the classification stage have been proposed by Lenssen et al. in [14] and show high accuracy. However, CNNs are not suited well for low power mobile and embedded devices due to high computational requirements. Therefore, this chapter introduces a method based on frequency domain analysis, which is faster and less resource demanding than the CNN, with only a small, tolerable loss in accuracy. The idea to use frequency domain analysis for the classification is originally given by Lenssen et al. in [14], but the resource efficiency of the methods is not explored. To evaluate the frequency domain based method, the classification stage of the PAMONO sensor image analysis pipeline is modified.

This chapter is organized as follows:

In Section 3.1, the modifications of the PAMONO image analysis pipeline are described. In the first subsection, the considered frequency domain analysis methods, the Fourier and the Haar wavelet transforms are introduced, and the corresponding feature extraction processes are illustrated. The second subsection is about the feature selection, in which the selection of subsets from the whole sets of features is considered.

In Section 3.2, the classification based on frequency domain analysis is executed on different platforms. A comparison to the CNN method in terms of classification performance and execution time is given as well.

Some materials of this chapter have already been published in the author's publications in [31].

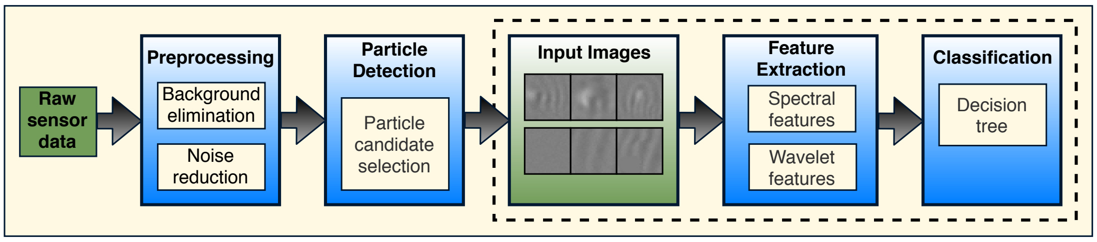


Figure 10: The modified image Analysis pipeline for the classification with frequency analysis features.

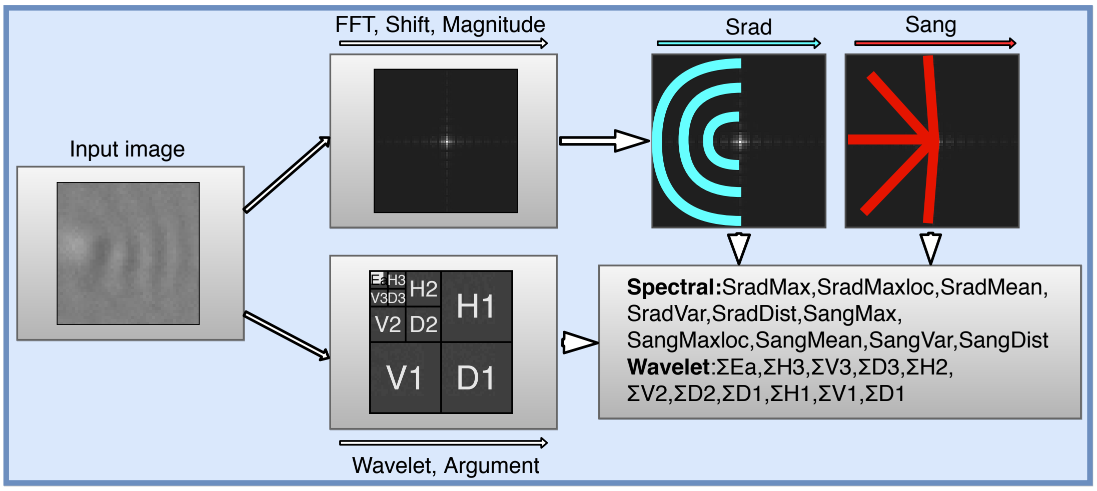

FFT, Shift, Magnitude
Srad
Sang
Input image
H3
V3D3
H2
H1
V2
D2
V1
D1
Spectral:SradMax,SradMaxloc,SradMean, SradVar,SradDist,SangMax, SangMaxloc,SangMean,SangVar,SangDist
Wavelet:ΣEa,ΣH3,ΣV3,ΣD3,ΣH2, ΣV2,ΣD2,ΣD1,ΣH1,ΣV1,ΣD1
Wavelet, Argument

Figure 11: The main steps of feature extraction process. In the top branch, the extraction of the Fourier features is illustrated. In the bottom branch, the extraction of the Haar wavelet features is illustrated.

3.1 MODIFICATION OF THE IMAGE ANALYSIS PIPELINE

Figure 10 shows the modified image analysis pipeline, which can be divided into four main stages: preprocessing, particle detection, feature extraction and classification. Several techniques for the detection and classification stage and their performance are detailed by Lenssen et al. in [14]. The state-of-the-art methods for the detection and classification stage are summarized briefly in Section 2.2. In this chapter the focus is on the feature extraction and the classification stages which are highlighted by the dashed rectangle.

3.1.1 *Extraction of Fourier and Wavelet Features*

The image patches which are provided by the detection stage of size 48×48 are subsampled from the center as 32×32 patches and are considered as the input for the classification system. The image patches may contain particles in addition to the existing artefacts and non-constant noise. In the feature extraction stage, the images are loaded in the classification pipeline, which extracts features in the frequency domain. For the classification of the features, decision trees and random forests are evaluated to decide whether excitations due to a particle binding were observed.

Frequency domain analysis is a classical method for signal processing, and has many application cases in the field of medicine [3]. The provided data from the PAMONO sensor are actually signals which show light intensity steps over time. In the previous work and in this thesis, the data is analyzed with methods that are often applied in image processing. The goal of this analysis is to detect blob-like and periodic patterns in the images from the PAMONO sensor. In Figure 10 positive and negative examples of input images can be seen in the first and second row, respectively. Images with excitations due to particle bindings show blob-like excitations with surrounding periodic patterns resembling sinusoidal waves. Therefore, frequency analysis methods are proposed to classify particle bindings. As examples of frequency analysis techniques, the Fourier and Haar wavelet transform are used to extract features.

For the feature extraction, at first, the input image is transformed from the spatial domain into the target domain. Then, several features are extracted from the transformed and further processed image to extract texture information, using the information of which the type of pattern or excitation can be identified. A general overview of the feature extraction processes is presented in Figure 11, with the extraction of the Fourier features in the upper branch and the extraction of the wavelet features in the lower branch.

3.1.1.1 *Fourier or Spectral Features*

In this work, the Fourier features are used to differentiate between periodic and non-periodic patterns in the image patches. With these features it is possible to quantify the differences between the patterns [11].

Theoretically, every signal can be represented as a superposition of sinusoidal signals with different frequencies. The Fourier transform divides a signal into its frequency components, from which the original representation can be derived again. The two dimensional Fourier transform, when integrated over $\mathbb{R}^2$, produces the frequency representation of $f(\mathbf{p})$:

$$F(\mathbf{w}) = \int f(\mathbf{p})\ e^{-2\pi i \mathbf{p}\mathbf{w}}\, d\mathbf{p} \tag{1}$$

In this work, the 2D discrete Fourier transform (DFT) is considered, since the inputs are grayscale images. Every pixel of the 2D transform represents a sinusoidal function with information about the phase, magnitude, and frequency. With an image $I(p_x, p_y)$ of dimension $N \times N$ and its Fourier transform $F(p_x, p_y)$ at $\mathbf{p} = (p_x, p_y)^T$, then the DFT can be obtained by:

$$F(p_x, p_y) = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} I(p_x, p_y) e^{-2i\pi(\frac{kp_x}{N} + \frac{lp_y}{N})} \tag{2}$$

Instead of computing Equation 2 directly, the Fourier transform can first be applied on all the rows of the image, and then on all the columns of the image, because of the algorithm's separability property. For the single rows and columns, a divide and conquer method, the Fast-Fourier Transform (FFT) algorithm can be used for efficient computation. In the FFT algorithm, the signal samples are divided into two sets, one set with the values of even and one set with the values of odd indices.

This division of sets is applied to the resulting two sets as well, and this process is repeated recursively $O(\log(N))$ times, when there are N signal samples, till only two elements are left in every set. DFTs of size two are computed at the very bottom of the recursion tree, which are also referred to as butterfly operations. More DFTs of size two are computed when combining of results of the lower recursion levels. FFTs with butterfly operations with two inputs are also referred to as radix-2 FFT. The butterfly calculates a DFT of size two, which corresponds to two multiplications and two additions. Combining the results of the subproblems in one recursion level needs linear time.

For one row of an image, $O(N\log(N))$ is the complexity. To transform images, first the rows are all transformed by considering them as 1D FFTs. Then, in a second step, the columns are considered as 1D FFTs. This approach still computes the FFT of a square image in $O(N\log(N))$. Since the signal samples are halved at every step, an input of a power of two is necessary for a radix-2 FFT, otherwise zeropadding may be performed before transformation.

For the feature extraction, the FFT algorithm is performed on the input image first. To compute spectral features from the Fourier transform of the input image, the zero-frequency component is shifted to the center, and magnitude of the spectrum is computed. Then two sets of magnitude values are extracted, which are referred to as S_{rad} and S_{ang}. A visualization of the extraction of the Fourier features is shown in the upper branch in Figure 11. S_{rad} is an array of size r and stores the sums of magnitudes over half circles for different radiuses r. This means that S_{rad} contains one entry for each halfcircle with radius r that can be drawn into the image. S_{ang} consists of sums over magnitudes lying on straight lines between the image center and the outer half circle. S_{ang} contains one entry for each straight line between the center and the outer half circle This way of summing is illustrated in Figure 11 by the blue and red lines, for S_{rad} and S_{ang} respectively. Positive and negative images can be distinguished by observing high magnitude bursts in half circles and lines in the frequency spectrum. Therefore, the information about dominant frequencies from S_{rad} and dominant periodic patterns from S_{ang} is used to generate features. From both sets S_{rad} and S_{ang} the mean, the maximum, the location of the maximum, the variance, and the difference between minimum and maximum values are extracted, leading to a feature vector containing ten spectral features in total.

As an attempt to use the available resources efficiently for computing the extraction of the spectral features, two computation phases can be distinguished. In the the first phase, the computations are performed on the GPU, since these can be parallelized on the GPU for efficiency. The second phase, in which the values are summed, is computed on the CPU. Summing the values on the GPU with halfcircles and straight lines would require transferring many-dimensional bitmaps to the GPU, since the memory regions which need to be summed overlap. Then a parallel sum reduction would need to be performed. As it is more resource-efficient to transfer the spectum to the CPU once instead, the summation is performed on the CPU.

For this application, the 2D Fourier transform was implemented by the author in OpenCL from scratch as a radix-2 out of place GPU accelerated FFT algorithm, for which the Bealto website [5] served as a guideline. For the details of the GPU computations, in the first phase, the FFT and the the magnitude of the transform is computed. Then the zero-frequency component is shifted to the center of spectrum with an own implementation by the author, which was inspired by the efficient shift algorithm by Abdellah in [1]. The resulting spectrum is then transferred from the GPU memory to the CPU memory, to sum magnitudes of the spectrum in the second phase. The implementation of the OpenCL algorithms for the 2D FFT and the pseudocode for the host code are documented in Appendix A. A CPU-only implementation of the Fourier analysis can also be viewed at the authors Github repository in [33].

3.1.1.2 *Wavelet Features*

The two-dimensional Wavelet transformation is also explored in this work to perform frequency domain analysis. It is derived from the correlation between the image and the wavelet ψ. Integrating over $\mathbb{R}^2$, the continuous two-dimensional wavelet transformation is defined as:

$$W_f(\mathbf{t}, s, \theta) = \int f(\mathbf{p})\psi^*_{\mathbf{t},s,\theta}(\mathbf{p})d\mathbf{p} \qquad (3)$$

where $\mathbf{t}$ represents the translation vector, s the scaling parameter, and θ the rotation angle. The wavelet transformation is related to the Fourier transformation, but also provides positional information by analyzing the frequency content within different resolutions, which is also called multiresolution analysis. It offers high positional accuracy for high frequencies, and offers high frequency resolution for low frequencies. The computational complexity of the wavelet transform is the same as FFT, $O(N\log(N))$, as the wavelet transform can be computed as a product between the Fourier transform of both the input and the wavelet function in the frequency spectrum according to the convolution theorem. As an example of a wavelet function, the simplest form is used, the Haar wavelet. 3 scales are used to produce 10 different channels, so that excitations can be detected by analyzing the image in different scales.

In the case of the Haar wavelet transformation, the computational complexity is $O(lN)$, since the discrete Haar wavelet transformation can be computed efficiently as a Fast Wavelet Transformation (FWT) by performing N summations per row on the first level, and N/l summations in level l per row. As in the 2D FFT, for the 2D FWT, the transformation needs to be executed on all rows first, then on all columns. The processing on the rows and columns have to be performed subsequently before changing the level, otherwise a different type of a wavelet transform will result.

When the 2D Haar wavelet transformation is applied, the image is decimated into four channels. The decimation of the image is illustrated in the bottom branch in Figure 11. In the output of a level one Haar wavelet transform, the top left channel

(Ea) is a smaller version of the original image, containing the low frequency information, which is obtained by low-passing the original image, which is realized by a scaling function averaging neighboring values. The three remaining channels contain the high frequency portions (or the wavelet portions) of the signal, which are the horizontal (Hx), vertical (Vx), and diagonal (Dx) edge information. For transforms with levels higher than one, the top left image is decimated recursively as described above.

To extract the wavelet features, the the energy of each channel is computed as follows:

$$E_{f_d} = \frac{1}{NM} \sum_{p_x=0}^{N-1} \sum_{p_y=0}^{M-1} |W_{f_d}(\mathbf{p})|, \tag{4}$$

with $N-1$ and $M-1$ as the image dimensions and $W_{f_d}(\mathbf{p})$ as the discrete Haar wavelet transform in position $\mathbf{p} = (p_x, p_y)^T$ in the image. This results in a vector consisting of ten extracted wavelet features. Images with blob-like excitations have large energies in both low and middle frequencies, while the low frequency channels are dominant in smooth regions.

There are also two phases for generating the wavelet features. In the first phase, in which the level three Haar wavelet transform is computed, is performed on GPU. The energies are computed on the GPU as well, as the operations can be parallelized well. The energy terms are then transferred to CPU, to sum all values. The 2D Haar wavelet transform was implemented by the author from scratch in OpenCL as a GPU accelerated fast Haar wavelet transform (FWT), according to the guidelines in [6], in an attempt to use available resources efficiently. A CPU-only version was implemented by the author of this thesis as well. Both implementations are documented in Appendix A, and the CPU-only implementation can be viewed in the author's Github repository in [33].

3.1.2 *Feature Selection*

To optimize the feature extraction, the number of extracted features could be reduced. In this subsection, feature analysis using principal component analysis (PCA) is performed. The goal is to find out whether selecting only a subset of the generated features would reduce the classification performance significantly. With less features to extract, the complexity of the approach and the execution time could potentially be reduced significantly.

It is computationally expensive to evaluate all combinations of features. With n features, there would be $2^n - 1$ combinations. An alternative is to select a subset of the features from the original set sequentially. The selection is performed by calculating the variance each feature accounts for in a principal component analysis. In practice, this means that the feature which accounts for the most variance is selected first, and then the other features are added incrementally. At each step, the accuracy is evaluated.

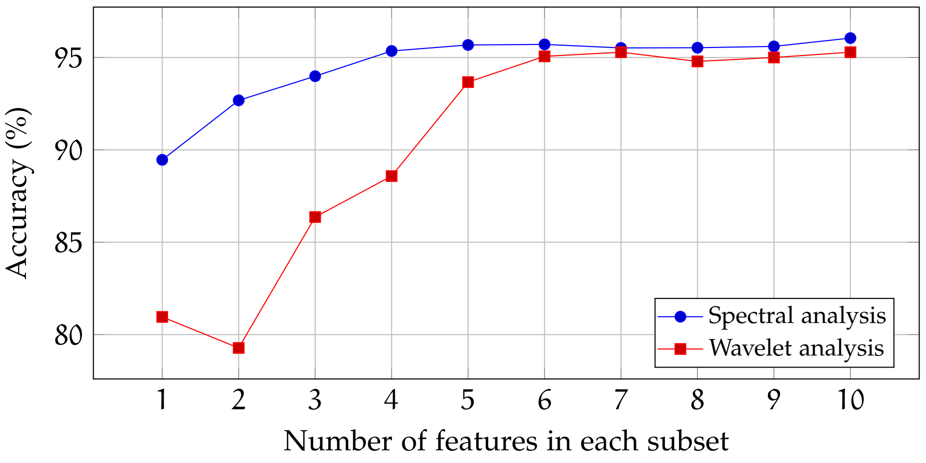


Figure 12: Accuracy of the classification for different subsets of features.

The results of this methods are illustrated in Figure 12. It shows that the accuracy of the spectral features is higher than the wavelet features. For the spectral features, the accuracy does not increase a lot after selecting four features. For the wavelet features, only after selecting six features, a saturation of the classification performance can be observed. This result can be explored further when optimizing the system performance. E.g., it could be the case that extracting fewer features reduces the execution time by a big factor.

3.2 EVALUATIONS OF FREQUENCY DOMAIN ANALYSIS METHODS

For evaluations, the classification performance and the execution time of the frequency analysis approach is measured. For evaluating classification performance, precision, recall, and accuracy are used. The accuracy results are then compared with the CNNs by Lenssen et al. [14].

To evaluate the execution time, two experimental settings are considered. In the first setting, for a fair comparison between the frequency analysis method and the CNN, the execution time for computations of the deepRacin graphs are considered. The execution time for this setting is measured on a mobile CPU, the Intel Core i7-4600U with the integrated Intel HD Graphics 4400 GPU, to which the computation of the 2D FFT and 2D FWT are offloaded with OpenCL kernels. For the 2D FFT, shift and argument steps are accelerated, and the for 2D FWT the energy computation is accelerated. For the execution of the CNN, the convolutional, maxpool, and fully connected layers are offloaded as OpenCL kernel to the GPU of the aforementioned Intel CPU.

In the experiments, offloading the computations to GPUs via OpenCL kernels requires synchronization with significant overhead compared to the time needed for computation of the algorithms. For this reason, in the second experiment setting, the computations are not accelerated; CPU-only implementations of the feature extraction methods are evaluated. For the CPU-only experiments, the Intel Core i7-4600U *without* GPU acceleration is used as a general purpose evaluation platform. For testing on an embedded platform, the feature extraction is run on the ARM Cortex-A9 chip on the Zedboard, again *without* any acceleration.

In a separate subsection, the FFT computation is accelerated using the official xfft IP (Intellectual Property) accelerator from Xilinx. It is compared against the FFTW [9] library in terms of computation time on the Zedboard.

3.2.1 *Training and Classification*

The dataset used for training and testing the classifier is provided by Lenssen et al. [14]. It consists of 38871 images and contains particles of sizes 100 nm and 200 nm. The size of all the images is 48×48 pixels. The provided train/test split with 19526 images for training and 19345 for testing is used, which is approximately symmetrical. All images are annotated with binary labels, indicating the existence or non-existence of a particle in the image patch.

For training the classifier using the FFT and FWT, the decision tree (DT) and random forest (RF) library in sk-learn [19] is used. DTs with small depths are very fast classifiers and it is convenient to obtain their C++ code for the use in embedded systems. After training the classifier model on features extracted from the training set, the tree structure and split values of the tree from the trained model are extracted, then converted to deployable C++ code, as proposed by Buschjäger and Chen et al. [7] as a standard If-else tree. The tree then is appended to the image analysis pipeline. The tree is executed after feature extraction with the spectral, the wavelet, or with both feature sets as input. The image analysis pipeline with the decision tree at the end returns a binary decision result for each extracted features. Note that the depths of all DTs are limited, and have a maximum depth no greater than 12. Since the execution time of the DT with depths less or equal to 12 is smaller than 1 μs, it is negligible for for execution time considerations. Increasing the depth of the DTs further does not increase classification performance by a large margin in this case.

Please note that the DTs and RFs need to be trained for every platform anew to achieve the best accuracy possible. For example, to train a DT to classify on a certain model of an ARM CPU, the features for the training of the DT have to be extracted on this exact same ARM CPU. If a certain model of an Intel CPU is used, the ARM DT may not be compatible to the features extracted on the Intel CPU. For this exact same Intel CPU, the DT has to be trained anew. Details for the reasons are discussed in Section 3.2.8.

3.2.2 *Platforms and Development Environments*

The image analysis pipeline is evaluated on three different platform settings. For the first setting, the Intel Core i7 4600U is used, and OpenCL kernels are offloaded to its integrated Intel HD Graphics 4400 using the framework deepRacin [10]. The deepRacin framework was used in [14] to accelerate CNNs computations to GPUs. To compare the performance of the feature based method to CNNs, the classification pipeline is implemented as deepRacin computation graphs as well.

The classification pipeline is also implemented as a CPU-only version on the Intel Core i7 4600U, the second platform, and on the ARM Cortex-A9 on the Zedboard, the third platform, to compare the execution time to the deepRacin implementation.The Zedboard is used as an embedded evaluation platform since it also has flexible acceleration capabilities.

3.2.2.1 *deepRacin*

The framework deepRacin is used for the first part of our experiments. In deepRacin, OpenCL kernels are used to offload computations to accelerators. Computation graphs with tensor operations, such as trained deep neural networks or image processing pipelines, can be deployed in deepRacin. In [14], the CNN computations were accelerated with deepRacin. The FFT and FWT used in the feature extraction are implemented as OpenCL kernels, and their computations are offloaded as well to the Intel HD Graphics 4400 GPU using deepRacin computation graphs.

3.2.2.2 *Zedboard with Xilinx SDSoC*

Since the frequency domain analysis algorithms are tailored for small embedded systems, therefore the classification is also executed on an ARM Cortex-A9, to evaluate whether the classification can be run efficiently on small low power embedded chips. The Zedboard was chosen for additional evaluations. It is a heterogeneous Zynq SoC and MPSoC platform, with the SDSoC development environment [12] for implementing the proposed frequency analysis classification methods. The Zedboard has an ARM Cortex-A9 processor and PL (Programmable Logic). PL is a component which can be used to create reconfigurable digital circuits. The PL can be used for the acceleration of parts of the image analysis pipeline.

SDSoC offers the xfft IP (Intellectual Property) core which can be instantiated in the PL. The xfft IP can be executed with a C++ function call to accelerate the FFT. The use of the xfft IP core is evaluated against a statically linked FFTW library [9] in the image analysis pipeline on the Zedboard. Guidelines to configure, compile, and run the FFTW library and the xfft IP core for the Zedboard can be found in [30], and how to use the xfft IP core is shown in the official documentation from Xilinx.

SDSoC also provides automated software acceleration in PL, which means that functions in C++ can be accelerated by creating reconfigurable digital circuits for them, but this is not explored further in this work.

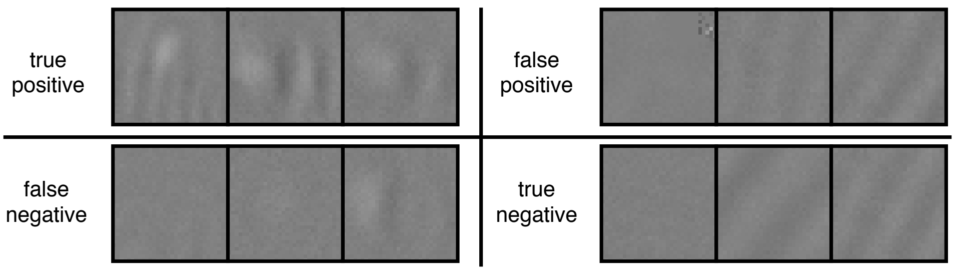


Figure 13: TP, FP, FN, and TN for the approach with spectral and wavelet features.

3.2.3 *Evaluation Metrics*

TP stands for true positive, FP for false positive, FN for false negative, etc. Precision is calculated as $\frac{TP}{TP+FP}$, and measures the ratio of correctly as positive detected elements, considering only positive predictions. FP in the denominator holds the information about the precision, if it is high, the precision will be low. However, most elements could be predicted to be false, and only a few correctly as positive, which would give a high precision value despite having many FNs. For this reason, a measure for the number of FNs is needed, which is recall. Recall is calculated as $\frac{TP}{TP+FN}$, and measures the ratio of correctly as positive detected elements, considering only elements that have positive ground truth annotations. If the number of FNs is high, the recall will be low. Accuracy is calculated as $\frac{TP+TN}{TP+FP+FN+TN}$, and measures the ratio between the correct predictions divided by all predictions. Since the dataset for this chapter is approximately symmetric, the accuracy can be used as a reliable summary metric for the classification performance.

3.2.4 *Classification performance*

The classification performance results for the frequency domain analysis methods are obtained by running the experiments on the Intel Core i7 4600U with the integrated Intel HD Graphics 4400 as the accelerator for the OpenCL kernels. The classifications are run as deepRacin computation graphs.

As shown in Table 1, the classification has the best performance when using both FFT and FWT features together. When using only one set of features, the FFT features outperform the FWT features by a small margin. Increasing the number of trees in a random forest improves the performance only slightly. When imposing no restrictions on the tree depths (can be up to 39), the classification performance increases around 0.15 percent points for all cases, however, the tree depths used by the training algorithm in that case can amount to over 39.

Overall, for the approaches shown the precision and recall are above 96% in all cases, and the accuracy is above 97% except for FWT features with one tree. Similar classification performance results are obtained in the experiments in [14], in which the classification performance of a classifier using FFT and FWT features was used with a Matlab implementation. The CNN in [14] has a higher accuracy score of 99.5% than both feature approaches in Table 2.

In Figure 13 a few classifications are shown. When excitations are not strong enough, the classification might miss them (FN). Regular high frequency patterns can occur due to other nearby excitations, or due to noise or vibrations. These can be mistaken as a particle binding (FP).

Table 1: Comparisons between different classifiers using different feature compositions. DT stands for decision tree, RFn for random forest with n DTs.

Features	Measure	DT (%)	RF10 (%)	RF100 (%)
FFT and FWT	Precision	98.49	99.33	99.33
	Recall	97.04	97.19	97.76
Only FFT	Precision	97.78	98.50	98.66
	Recall	96.33	96.02	96.48
Only FWT	Precision	96.77	97.59	97.66
	Recall	96.35	96.63	97.49

3.2.5 *Execution Time using a mobile GPU for Acceleration*

Table 2: Average execution time (Intel Core i7 4600U with integrated Intel HD Graphics 4400), and accuracy (with one DT for the feature based approaches).

Method	Execution time (ms)	Accuracy (%)
FFT features	1.28	97.07
FWT features	1.50	96.57
FFT and FWT	2.78	97.78
CNN [14]	3.37	99.50

The execution time for the frequency domain analysis methods are obtained by running the experiments on the Intel Core i7 4600U with the integrated Intel HD Graphics 4400 as the accelerator for the OpenCL kernels. The classifications are run as deepRacin computation graphs.

In Table 2 the FFT approach is 2.63 times faster than the CNN solution. The wavelet approach is 1.83 times faster than the CNN. However, when measuring the total run time of the deepRacin computation graph, and subtracting the sum of execution times of all kernel calls, it can be noticed that the synchronization overhead between CPU and GPU is dominating the execution time. For the experiment setting in Table 2, the Fourier transform and summing values take less than 0.2 ms in total, but the synchronization of GPU and CPU due to the kernel calls increase the total execution time of the computation graph. This overhead can potentially be eliminated when implementing the proposed approach as a CPU-only version.

To explore the synchronization issue further, parts of the deepRacin graphs are evaluated for the feature generation on an Intel Core i5 4460, and an AMD Radeon R7 370 as the accelerator. For the FFT of a 32×32 image on this platform only 5% of the time is spent on computation, the rest is spent on synchronization. Similar ratios are obtained for the FWT. It can be noticed that the ratio between synchronization overhead and actual calculation for FFT gets smaller the bigger the image size. From sizes above 512×512, for actual calculations the time taken is 80%, and only 20% for synchronization in the whole FFT computation. For image sizes below 512×512, the synchronization time takes above 70% of the whole FFT computation.

3.2.6 *Execution Time Results for CPU-only*

The implementation of the classification methods in deepRacin allows to compare the CNN and the feature based methods in a fair setting. However, as pointed out in the previous subsection, the synchronization overhead may be the main bottleneck in the feature based method. Hence, the execution time should be evaluated in a CPU-only version on the Intel Core i7 4600U as well (the same CPU was used for evaluating the GPU-accelerated algorithms). For the CPU-only version, the FFTW library for FFT, and a CPU version of the 2D Haar FWT is used, the latter of which was implemented by the author himself. The results show that the CPU-only executions give *significant* speedup.

In Table 3, the decomposition of the runtime of the classifications are presented for different platforms. Only the time for executing the classification is considered, not the execution time needed generate and load the test images. The FFT and FWT algorithms (Alg.) are executed on the Intel and ARM platforms (Platf.). After the transforms (Transf.), for the FFT, the post-transformation (PT) step is calculating the arguments and computing the shift. The summing (Sum) part consists of S_{rad} in the first value, S_{ang} in the second value, and final calculations to obtain the 10 features in the third value. The table shows that the computation of S_{ang} accounts for around 70 % of the execution time; the issue can be observed in the measurements for the ARM processor as well. For FWT, the main bottleneck is the 2D FWT. After the Wavelet transform, the energies and their sums are computed; the results show that the influence on the total execution time of the PT, Sum, and DT stages of the FWT classification is weak. Overall, the FWT classification outperforms the FFT classification in execution time in this CPU-only setting on this experiment setting.

According to the feature analysis in Section 3.1.2, a trade-off between the maximal accuracy and the computation time with the aforementioned procedure is possible. On the x-axis is the computation time for a subset of f features required to extract those f features. As shown in Figure 14, around 83.5% of the computation time can be saved and more than 93% accuracy can be achieved by extracting only two FFT

Table 3: Compositions of the execution time for different platforms. All values are average values obtained from 1000 runs, and are in ns.

Alg.	Platf.	Transf.	PT	Sum	DT	Total
FFT	Intel	3563	3415	2367 + 19438 + 19	43	28848
	ARM	61801	43302	21902 + 246765 + 975	1310	376058
FWT	Intel	16594	160	21	45	16821
	ARM	79612	6271	934	1123	87942

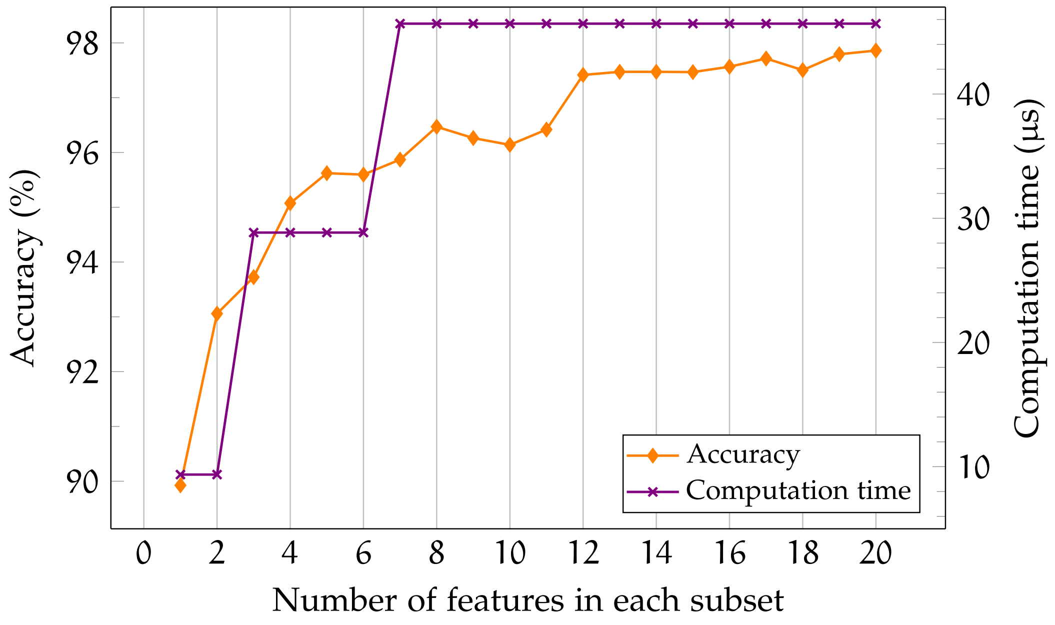


Figure 14: Trade-off between accuracy and computation time for different subsets of features.

features, where the features are ordered according to the aforementioned correlation in Section 3.1.2. If four more FFT features are extracted, about 40% of the computation time can be saved and only about 2% accuracy needs to be traded off. The maximum classification accuracy using all the 20 features is 97.78%, but the computation time is significantly increased from 60% to 100% in this case, because the FWT features have to be extracted.

3.2.7 *FFTW against xfft IP*

In the Xilinx SDSoC environment, the xfft IP core can be used, which executes a 1D FFT of any point size up to a certain limit. The use of the 1D xfft IP against the FFTW library is compared on the Zedboard.

The 32 point xfft IP is called 64 times to perform a 2D FFT on the image patches. Calling only the xfft IP in this way needs 28848 ns averaged over 1000 runs, while the FFTW, as a CPU-only version, needs 61801 ns. The FFTW is thus around four times faster than using the xfft for acceleration of the FFT on the Zedboard.

The xfft IP execution time is limited by the data transfer overhead between the ARM chip and the PL, as well as the PL clock on the Zedboard which is limited to 100 Mhz. Therefore, the xfft IP for acceleration is not used in our application. However, if the image size and thus the FFT point size is larger, or another, faster FPGA platform is used, there could be more benefit using the xfft IP, and the two methods should be compared anew.

To speed up data movement, it is possible to extend the xfft IP core with a memory controller, which stores the entire patch and intermediate FFT results, with a design in a hardware description language (HDL), to perform a 2D FFT. However, how to optimize the execution in HDL is out of the scope of this work. In this work, only readily available components are used.

3.2.8 *The Settings of the CPU-only Execution Time Measurements*

The accuracy of the frequency analysis methods were shown twofold. First, by Lenssen et al. in [14], the results are obtained in a Matlab evaluation. Second, in this work similar accuracy values are obtained by implementing the feature extraction in deepRacin with offloading to a GPU.

The decision trees obtained from the first setting are not compatible to the features extracted on the second setting. This means that the decision tree from the first setting cannot classify features which are extracted in the second setting. The reason is that different platforms, e.g., CPU and GPU, or ARM and Intel, produce different numerical errors in floating point arithmetic. Different platforms therefore may produce differing results due to floating point approximation errors. The FFT algorithm predominantly uses floating point operations, and in the FFT feature extraction thousands of floats are summed. Over 4500 floating point summing operations are performed to obtain arrays S_{rad} and S_{ang}. Additional sums are performed in subsequent calculations of the mean and variance.

In these summations the errors add up differently for different platforms, and the DTs have to be adapted to each platform. A direct consequence is that the DTs trained on the features extracted in MATLAB cannot be used to classify the features extracted in deepRacin. A new DT has to be trained by extracting features in deepRacin.

For the case of Intel and ARM (both CPU-only), as well as ARM with the xfft IP, to get the best accuracy results, it would be mandatory to train a DT by generating the features for training the DTs on every platform separately. Since the accuracy of the classification methods is confirmed in two different works, only the execution time of the classification is considered for the CPU-only evaluation. The floating point errors of the FFTW and the xfft IP are also explored in [30]. Furthermore, to preserve the portability of the evaluation framework for different platforms, 1000 random grayscale images are generated during the execution time test. During testing, a batch of 25 random grayscale images is generated at once, and the features are extracted. This procedure is repeated 40 times to reach 1000 runs.

Although there is no difference in the number of executed instructions when using random images, the extracted features are not representative for the features which are extracted from the original patches. Therefore, 1000 extracted feature vectors are loaded into memory for FFT and FWT features respectively, which were generated from the deepRacin implementation. Then, the trained DTs for FFT and FWT from the deepRacin implementation are used, to get accurate and representative features and execution times for the DT. This test flow allows to evaluate the classification even on platforms with memory limitations, without having to extract the features of the training examples for every platform.For more details on the implementation of the testing framework for the frequency domain based classification, the source code of the testing framework can be viewed at the author's Github repository in [33].

On a related note, zeropadding the input image causes interpolation values in the FFT, and introduces more errors in the summation part of the FFT classification. This problem was encountered when zeropadding the 48×48 image patches to 64×64, to benefit from powers of two in the FFT and FWT algorithms. With such interpolation values, the errors add up more. The errors due to interpolation were directly noticeable in the accuracy of the FFT classification. The FFT classification trained on a dataset zeropadded to 64×64 showed around 10 percent points less accuracy than the FFT classification trained on a dataset without zeropadding (32×32). On the other hand, the FWT classification showed no significant changes in accuracy when zeropadding to 64×64 (there is no interpolation for the FWT in this case, zero values stay zero values for FWT), or cropping to 32×32 from the original 48×48 images.

3.3 SUMMARY

In this chapter, frequency domain analysis based features with a decision tree as a classifier are proposed as a resource-efficient alternative to CNNs for mobile and low power virus classification methods. It is shown that the virus classification pipeline can be built using readily available, low cost components. The execution time improved from 3370 μs, and is in the scale of 17 to 29 μs per classification on a Intel Core i7 4600U, without the need for acceleration. The classification accuracy of the feature based approaches is sufficiently high for virus detection, which indicates the usefulness of frequency domain analysis for this application. While the CNN approach performs better by approximately 2.5 percent points in accuracy, the frequency analysis based approaches outperform it in execution time by a large factor. In conclusion, ways to utilize the trade-off between classification quality and execution time were found. The methods presented in this thesis can serve as a blueprint for the design of resource-efficient virus classification methods which use SPR images.

4 PAMONO DETECTION STAGE MODIFICATION

In this chapter, the usefulness of several yolo architectures for the virus detection on PAMONO sensor images is explored. The goal is to explore new techniques and platforms for the virus detection, and to simplify and improve the current virus detection pipeline. The yolo-CNNs could also potentially be used as a light-weight region proposal CNN; this possibility is discussed at the end of the chapter.

Single Shot Object Detectors (SSOD) are CNNs which predict size, position, and the class of certain objects in images using bounding boxes. Usually, the CNN models use a confidence score per bounding box, which describe the confidence that there is an object inside the bounding box. The models then filter boxes based on this score with a threshold, while also considering the predicted class probabilities. For real-time object detection, the processing speed per frame should be high enough, e.g., 30 FPS for video processing. SSOD methods were not tried before on the images of the PAMONO sensor for virus detection, because they were thought to be too resource-demanding for the use in resource-constrained systems.

Currently, the yolo-CNNs are one of the fastest available SSODs that can also be run on resource-limited platforms. An efficient way of executing SSODs on resource-limited platforms is proposed by Preußen et al. at Xilinx in a recent work [20]. They demonstrate that an architecture based on the tiny-yoloV2-CNN [23] can be executed efficiently on their MPSoC (multi-processor system-on-chip) boards. For inference (applying the CNN on unseen data), the darknet framework (with which yolo-CNNs can be trained and tested) which has C++ functions for layer operations, is used. When executing the compiled darknet implementations of the layer operations on the ARM chip of the MPSoC platforms, the framerate is too low for real-time object detection. Therefore, several optimization techniques are implemented, inter alia, using the hardware compiler which synthesizes C/C++ functions into op-

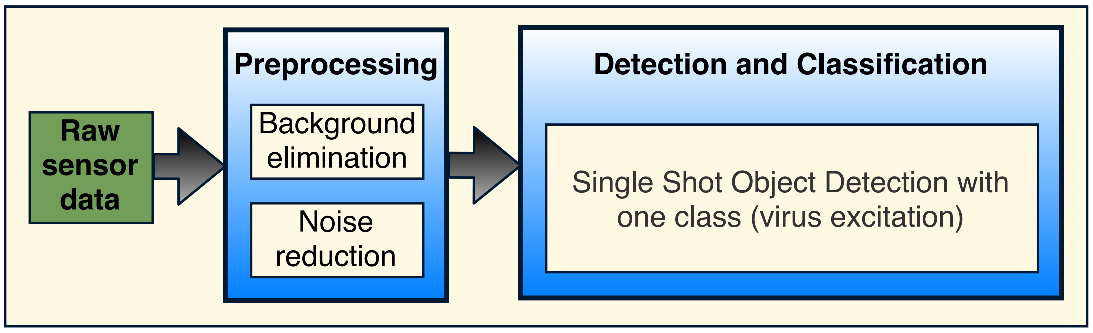

Raw sensor data
Preprocessing
Background elimination
Noise reduction
Detection and Classification
Single Shot Object Detection with one class (virus excitation)

Figure 15: The modified image analysis pipeline for the detection with yolo.

timized hardware functions which are executed in the PL [20], and quantization or binarization of the CNN parameters and feature maps. The function calls of the layer operations of the CNN, the computations of which are simplified as a consequence, are redirected to software-defined hardware functions in the PL. If the virus detection performance of the yolo-CNN were high enough, it would mean that MPSoC FPGA platforms, e.g., the Zedboard, would be a viable platform candidate to perform the virus detection efficiently. Therefore, the goal of the work in this chapter is to explore whether yolo-CNNs can be used to detect virus excitations reliably.

For improving the existing image analysis pipeline, the benefit of yolo is that it combines several parts of the image analysis pipeline into one step, i.e., it performs detection and classification in one single step. The simplified PAMONO image analysis pipeline is shown in Figure 15.

In this chapter, an overview over the yolo-CNN architecture and its principles of operations is given in Section 4.1. In Section 4.2, the tools, methods, and evaluation metrics for the training and evaluation results of the yolo-CNNs are presented. At the end of the section, the detection performance is presented and the usefulness of yolo-CNNs for virus detection is discussed.

4.1 YOLO OVERVIEW

The yolo-CNN architecture belongs to the Single Shot Object Detection (SSOD) methods. It detects objects in a complete image and annotates them with bounding boxes. Yolo stand for You Only Look Once, which implies that the CNN only processes the whole image once, to produce bounding box predictions and class probabilities. Other SSOD methods use several processing stages in series, i.e., one region proposal CNN followed by a classifier to produce class probabilities. They are difficult to optimize since the parts of the methods have to be considered separately. Yolo processes the image in one single CNN to output bounding boxes. Redmond et al. claim that yolo can generalize well, and is that it also works well when applied to new domains [25]. There are also some limitations of yolo, e.g., it generally has a lower recall than other SSOD methods. It also struggles with many objects that are close to each other. However, detecting many objects that are close to each other are a problem for SSOD methods in general.

In Figure 16, the general structure of a yolo-CNN is shown. The CNN takes an image as input, and then processes it with several layers. The layers consist of convolutions with multiple kernels followed by maxpool operations. These layers are executed in series, till the tensor is flat enough, and then fully connected layers are executed to produce bounding box annotations.

In general, all yolo-CNNs produce a last layer of size $S \times S$, this means that the input image is separated into regions of equal size with a grid. For each cell in the $S \times S$ grid, B bounding boxes are predicted. The bounding box annotations consist of five values: The center of the box with respect to the grid cell for the first two values, the width and height of the bounding box in the following two values, and

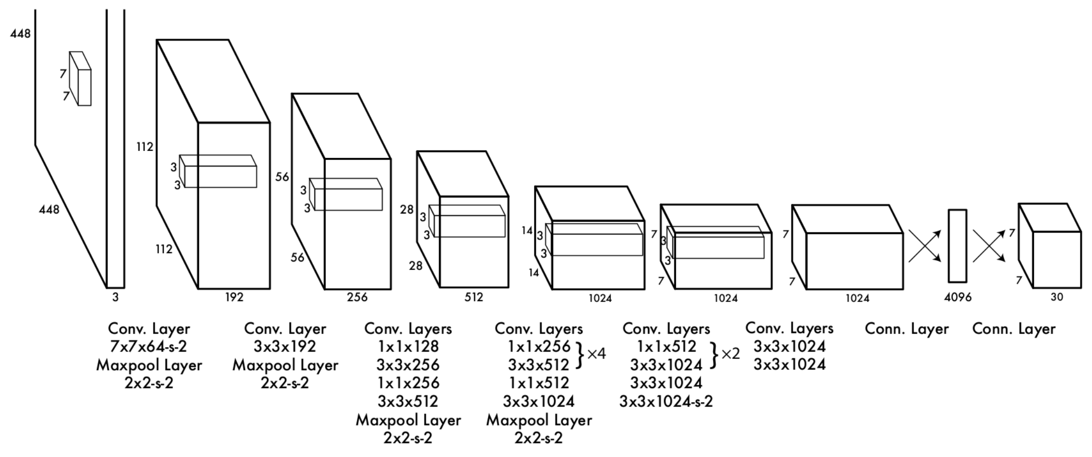

448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 }×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 }×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 16: The general structure of a yolo-CNN, adapted from [25].

a confidence value per class $C_i = Pr(Class_i)IOU^{thruth}_{pred}$, describing how confident the prediction is. IOU^{thruth}_{pred} means intersection over union, which is a metric that describes how big the area of the intersection between the true bounding box label and the predicted label is, divided by the area of the union of the bounding box labels. There is also the class probability which is computed in the output, denoted as C. In the end, the yoloV1-CNN has an output tensor of $S \times S \times (B \times 5 + C)$, where C stands for the probability for the classes defined in the problem. Here, the bounding boxes for one region in the $S \times S$ grid can only detect one class.

Redmon et al. also published a second version of yolo, referred to as yoloV2 [23], with several small improvements for detection performance and faster execution. In yoloV2, the limitation for per grid area classes is abolished, then every cell in the output grid contains class probabilities for every class. This means that the output tensor is of size $S \times S \times (B \times 5 + C_{all})$, where C_{all} are the class probabilities for every class. Also, the bounding box widths and heights may not need to be predicted directly. Instead, bounding box dimensions can be defined as priors, so that only the scale and the offset need to be predicted. The CNN then does not need to predict the bounding box dimensions on its own, when the rough object dimensions are known by the expert. The CNN can chose between the boxes with different height-to width ratios and takes the one that has the highest IOU. To generate the priors for the bounding boxes based on the dataset, a k-means clustering algorithm can be run on the training data to find good prior anchors (bounding box priors). This makes it easier for the CNN to learn and increases the mean average precision (mAP) score, which is a summary statistic for the area under the precision-recall curves, the values of which are averaged over all classes. An implementation of the k-means clustering algorithm is provided in AlexeyAB's darknet fork [2].

Another few small updates in yoloV3 introduce the use of pass-through layers, in which not only the last layer produces bounding boxes, but it is allowed to let previous layers produce bounding box predictions as well. This increases the number of available bounding boxes and improves the object detection for objects which significantly differ in size [24].

In this thesis, the yoloV2 architecture is used for training and testing. It has a simple architecture with only convolution and maxpool operations. This simplifies finding a suitable architecture for virus detection. Redmond et al. report that yoloV2 is one of the fastest state-of-the-art SSOD methods [23]. Xilinx also reported that the execution time for an optimized tiny-yolo architecture [20] on their MPSoC is 16 FPS. In the authors graduate project group PG611, a simple yoloV2 architecture (shallow architecture with less than ten convolutional layers and less than 100 filters in a convolutional layer) was executed in around 20 ms on an Intel Atom Z530 with 1.6 GHz. For these reasons, the yoloV2 architecture was chosen for exploration.

4.2 TRAINING AND EVALUATION OF THE CNNS

In this section, the datasets which are used for training and testing are presented. Then the evaluation metrics for the yolo-CNNs are discussed. The training environment used for training and testing is presented next. Subsequently, a proof of concept example will be shown to motivate further study. Finally the training of a CNN with all the available data and the evaluations of it are discussed.

4.2.1 *Organization of the Datasets*

The dataset used for training and testing is from the SFB 876-B2 website [26]. The terms used in this subsection are introduced in Section 2.3. These are all the available datasets for 100 nm and 200 nm images. For training for the 200 nm case, following sets are used, with the total number of available annotations and the image dimensions in round brackets:

```
PAMONO Sensor Data 200nm_10Apr13 [v2.0] (470 annotations, 1080x145)
PAMONO Sensor Data 200nm_11Apr13_1 [v2.0] (387 annotations, 706x167)
PAMONO Sensor Data 200nm_11Apr13_2 [v2.0] (250 annotations, 742x127)
```

For training for the 100 nm case, following sets are used:

```
PAMONO Sensor Data 100nm_27Sep13_exp2 [v2.0] (382 annotations, 450x170)
PAMONO Sensor Data 100nm_27Sep13_exp3 [v2.0] (1046 annotations, 750x230)
```

The number of annotations is also equal to the number of distinct images. For the organization of the datasets, there is a directory called *synthetic* inside each directory inside every dataset. Within that directory, there are the folders called *1*, *2*, and *3* (referred to as dataset index in this thesis), which can be used as training, validation, and test sets. Permutations of these sets for training, validation and test is also possible. Each folder can have hundreds of images, and a csv-file with polygon annotations. Folders *1*, *2*, and *3* in one datatet have roughly the same amount of annotations or images.

In the folder *particle_component* are the images with the particle signal only, with noise and background signal removed. Within it are the synthetically generated images, with the annotations in a separate file; these are the images that are used as an input to the yolo-CNN. Each image can contain up to hundreds of binding excitations. Since the data stream of the PAMONO sensor provides the images over time, the first image in the folder has fewer excitations than the images which were recorded later in time. The recorded images that belong to the same experiment run always contain the binding excitation from the previously recorded images in the same experiment run. One folder, e.g., folder *1/particle_component* from one of the datasets, contains the images of one experiment run.

In this thesis, the folder *1* is used to choose the model, e.g., number of layers, the kernel configuration, and other parameters and hyper-parameters. Folder *2* is used for estimating the model performance, and perform changes on the model. Folder *3* is used last, when a model is chosen and trained fully, to finally asses the detection performance. The CNNs are then only evaluated once with the data in folder *3*.

In general, the binding excitations of the 100 nm images are more difficult to detect, as the excitations are smaller, more faint, and more numerous compared to the 200 nm case. 80 nm images are also available, however, the quality of the annotations are not as high as in the 100 nm and 200 nm case, as the excitations are too difficult for the experts to be distinguished.

4.2.2 *Training and Testing Environment*

As the training and testing environment, the darknet fork from AlexeyAB [2] is used. It offers better documentation and calculation of detection performance measures for the CNN compared to pjreddie's darknet [35]. Both environments train the parameters of the CNNs using a gradient descent algorithm. The configuration files of the CNNs which contain the information about the CNN model, hyperparameters, and explanations for all other settings which are needed for the training and testing can be found in Appendix A.2. The guideline and details on how to set up the darknet environment for training and testing the CNNs can be found in the author's yolo-Pamono Github repository in [34]. For the hardware used for training and testing, a local server station with a 1080 GTX GPU was used, which supports CUDA and cuDNN.

4.2.3 *Converting the PAMONO Annotations to Yolo Annotations*

The author of this thesis wrote a Python script to convert the .csv annotations in the PAMONO datasets to yolo annotations. The corresponding script can be viewed at the author's Github repository in [34]. The PAMONO annotations include the coordinates of all vertices of the polygon surrounding the excitation. Conveniently, the first tuple in the PAMONO annotations are the (x, y)-coordinates of the top left of the rectangle that encloses the excitation completely. The second tuple contains the object width and height of this rectangle. Leaving out annotations which are not relevant for the yolo-CNN, the PAMONO annotations have following structure:

```
[top left X] [top left Y] [width in X] [height in Y]
```

Whereas the yoloV2 format uses a slightly different setting, so the values have to be converted accordingly to:

```
[category number] [center in X] [center in Y] [width in X] [width in Y]
```

Where the category number is always "0" since the CNN is configured to only detect one class, which is a binding excitation. The annotations were converted with the aforementioned script to yoloV2 annotations. More details about the conversion process can be found in the comments of the script file.

4.2.4 *Evaluation Metrics*

IOU means intersection over union, which is a metric that describes how big the area of the intersection between the true bounding box label and the predicted label is, divided by the area of the union of the ground truth bounding box and the detected bounding box. For the evaluation of object detection, the TP (true positive), FP (false positive), and FN (false negative) definitions are dependent on the IOU. A TP is a correct detection with $IOU \geqslant IOU_{threshold}$, where $IOU_{threshold}$ is always set to 0.5 in this thesis. A FP is a wrong detection with $IOU < IOU_{threshold}$ A FN is ground truth annotation which was not detected. A metric which evaluates classification and detection by summarizing the area under the precision-recall curve is the average precision (AP). It calculates the precisions for the recall values for different classifier thresholds by increasing the recall in discrete steps of equal size, and averages among them. In this thesis, mean average precision (mAP) is the same as average precision, since there is only one class to be detected. More details about the AP evaluation metric can be found in the work of Everingham et al. in [8].

4.2.5 *Proof of concept for 200 nm only*

To explore whether the yolo-CNNs can work at all for detecting particle bindings, a proof of concept case for the 200 nm particles is performed, using a simple yoloV2 architecture. The CNN is trained with 30 images from "PAMONO Sensor Data 200nm_10Apr13 [v2.0]" with the dataset index *1*, and 30 other images from the same dataset, but dataset index *2*, which the CNN never saw during training, were used to asses the detection performance. Images with more than 20 excitations on them were chosen in the training and test sets. The training process was stopped manually after around 15 minutes, which correspond to around 500 iterations with the gradient descent algorithm. The CNN architecture used in this experiment was inspired by the robot detection CNN in the Graduate Project Group 611, in which the author also took part in:

```
yolo-nd-v2
layer     filters       size              input                output
    0 conv     16  3 x 3 / 1   704 x 160 x   1   ->   704 x 160 x  16
    1 max          2 x 2 / 2   704 x 160 x  16   ->   352 x  80 x  16
    2 conv     16  3 x 3 / 1   352 x  80 x  16   ->   352 x  80 x  16
    3 max          2 x 2 / 2   352 x  80 x  16   ->   176 x  40 x  16
    4 conv     24  3 x 3 / 1   176 x  40 x  16   ->   176 x  40 x  24
    5 max          2 x 2 / 2   176 x  40 x  24   ->    88 x  20 x  24
    6 conv     24  3 x 3 / 1    88 x  20 x  24   ->    88 x  20 x  24
    7 max          2 x 2 / 2    88 x  20 x  24   ->    44 x  10 x  24
    8 conv     32  3 x 3 / 1    44 x  10 x  24   ->    44 x  10 x  32
    9 conv     32  3 x 3 / 1    44 x  10 x  32   ->    44 x  10 x  32
   10 conv     30  1 x 1 / 1    44 x  10 x  32   ->    44 x  10 x  30
```

Figure 17: The CNN architecture of the proof of concept architecture to detect 200 nm particles in a simple setting. The architecture is inspired by the architecture which was used in the PG611 graduate project group (to detect robots) in which the author also took part in.

Qualitatively, the results look convincing, as illustrated in Figure 18. In image A in Figure 18, a particle signal with more than fifty bindings is shown. In image B, the predictions by the yolo-CNN are shown with a detection threshold of 0.3. In image C, the detection threshold is set to 0.5, and in image D to 0.8. Judging by the eye, the bounding boxes in all images seem to catch many blob-like excitation. Quantitatively however, the detection performance is very low, see Table 4 where all the considered measures are below 35%. This first result motivated for a further study for training with more data and a CNN with higher model capacity.

Table 4: The detection performance of the yolov2-tiny-nd-v2-overfit CNN architecture, for an IoU threshold of 50 %. All values are in percent.

CNN	Dataset	mAP	recall	precision	IoU
yolo-nd-v2	200 nm	20.48%	32%	35%	20.22%

4.2.6 *Finding a Suitable Yolo Architecture*

Currently, for finding a useful CNN architecture to a given problem, no optimal algorithms available. One can only try to follow guidelines or take architectures as inspiration that worked well on similar problems before. Detecting nano-particle bindings is a new application case and has not been tried before in the literature according to the best of the author's knowledge. Furthermore, most of yolo architec-

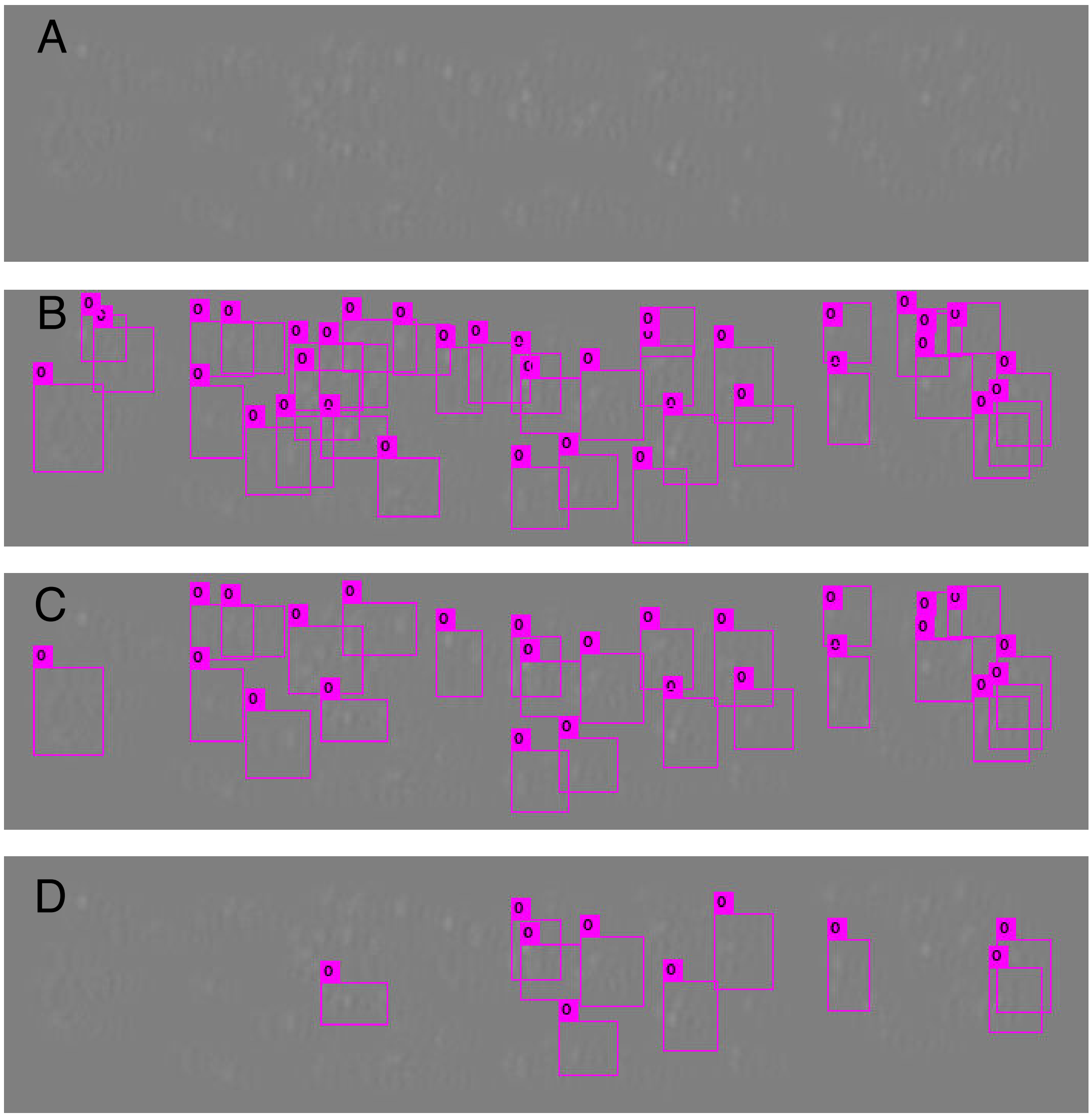


Figure 18: The predictions of the simple 200 nm yolo-CNN. In A is the image of the binding excitations. In B the detection threshold is 0.3, in C 0.5, and in D 0.8.

tures are used to detect more than ten or hundred classes at one, while in this thesis only one class, the binding excitation, needs to be detected.

One subgroup in the author's Graduate Project Group PG611 applied yoloV2 for detecting one class as well; just the robot class. Surprisingly, a shallow CNN architecture with only a few layers and all kernel sizes below 100 were used. The detection performance of this CNN was around 90% precision and 60% recall. It ran on an Intel Atom Z530 1.6 GHz CPU with 1 GB RAM in around 20 ms. However, an architectures similar to the PG611 architecture produced useless results for the PAMONO sensor data, similar to the results of the proof of concept architecture.

The original tiny-yoloV2 architecture [23] doubles its number of kernels in every layer, beginning with 16 in the first layer and increasing till 1024 kernels. This causes the weight file of the CNN to grow past 20 MB. The model would be too big for the efficient use in mobile systems.

For this reason, a smaller CNN architecture needs to be found, and in the end a well-performing architecture with a weight file which is only around 2 MB in size is found.

In a recent work, Pedoeem et al. proposed to **cut** the layers of the tiny-yoloV2 architecture and leave the last few layers out, i.e., they leave the kernels with a size of 512 and 1024 out. This architecture is called yolo-lite [18], and it has a kernel configuration of 16-32-64-128-256-1024-1024-125. According to the authors of the work, it can be run as a CPU-only version with 21 FPS on a Dell XPS 13 laptop [18]. Leaving these kernels out cuts the size of the weight file of the architecture.

Using the work in [18] as an inspiration, in this thesis the CNN is cut by two more layers, one at the top and one at the bottom, leaving the CNN with a kernel configuration of 32-64-128-256->128->54. The cutting also enables the last layer to produce many bounding box candidates. The less maxpool layers in the CNN, the more bounding box candidates the CNN can produce, which is crucial for good detection performance in this application case, since many excitation are close to each other in the input.

As in other works [18] which try to find CNN architectures, the CNN architecture for the detection of binding excitations are found empirically, training many CNN architectures by changing the yolo-lite architecture till the results are useful. In total, around 15 different architectures are evaluated, and it does not make sense to present all the variations in detail. Three tables are provided that show the trial and error steps for many CNN architectures. Only the successful *v14* version is presented in detail, which is the 14th CNN the author of this thesis tried. It outperformed all other CNNs in terms of classification performance.

In Table 5, all the architectures of all trial CNNs are collected. Architectures v1-v4 are based on the tiny-yolo architecture. Only the number of filters in the last layer is changed, since the number of classes in this application case is one. Then, the yolo-lite architecture with a modified last layer is tried in v5. In v6, the layer with 1024 filters is added, and the first layer is removed. In v7, the v6 architecture is cut by two layers at the end, so that there are only 3 maxpool operations, which allow the CNN to have more bounding box candidates than the previous layers. In v8, the first layer with 16 filters is added again. In v9, the last convolution block is changed to mostly increasing filter sizes. In v10, the use of filter configurations of size "a->b->a" and "b->a->b" is tried. The principle is inspired by the fire module from Lenssen et al. [14] which classifies the image patches of size 32×32 with high accuracy, where the feature maps are reduced first, and then expanded again. In v11, the filter with size 256 is removed. In v12, a simplified version of v1-v4 is tried. In v13, a very shallow architecture is tried, with only two maxpool operations. In v14, the yolo-lite architecture is cut one layer from below and one from above, so that the output of the CNN can produce many bounding boxes, while the CNN is not too shallow; the last layer is simplified as well. In v14-vf, where vf stands for Version Fire, the v14 version is tried in combination with a fire module inspired filter configuration. In v14-rf, where rf stands for Reverse Fire, the v14 architecture is mixed with a filter configuration which is inspired by the fire modules but the configuration is inverted. This filter configuration is also used in the last layer of the original tiny-yolo architecture. The architecture of the v14 CNN is shown in detail in Figure 19.

Table 5: Description of trail CNNs. "-" means "convolution followed by maxpool", and "->" means "convolution only".

Trial CNN	Architecture
v1-v4	16 - 32 - 64 - 128 - 256 - 512 - 1024->256->512->30
v5	16 - 32 - 64 - 128 - 256->512->30
v6	32 - 64 - 128 - 256 - 512->1024->30
v7	32 - 64 - 128 - 256->128->30
v8	16 - 32 - 64 - 128->256->128->30
v9	16 - 32 - 64 - 128->256->512 - 256->30
v10	16->32->16 - 32->16->32 - 64->32->64 - 128->256->128->30
v11	16->32->16 - 32->16->32 - 64->32->64 - 128->64->128 - 128->128->30
v12	16 - 32 - 64 - 128 - 256 - 512 - 1024->512->54
v13	128 - 256 - 128->30
v14	32 - 64 - 128 - 256->128->54
v14-vf	64->32->64 - 128->64->128 - 256->128->256 - 512->256->512->128->54
v14-rf	32->64->32 - 64->128->64 - 128->256->128 - 256->512->256->128->54

4.2.7 *Training of the CNNs*

From the datasets which are presented in Subsection , folders *1* and *2* from the sets are taken as training, and folders *3* from the sets as testing images. The images in folder 1 are used to find a suitable model for this problem. The images in folder 2 are used to evaluate the changes that are made on the model. Finally, the CNN is trained on datasets *1* and *2*. The CNN was then evaluated once on the images in folder *3*, and then the architecture was not changed anymore, so that it cannot overfit for the images in folder 3. The images in the different datasets have different sizes, but this is not a problem since yolo resizes the input images all to one size, and the detection performance produces reasonable results in many cases. It may also help the CNN generalize well.

```
v14
layer      filters       size             input                   output
    0 conv      32   3 x 3 / 1   448 x 160 x   1    ->    448 x 160 x  32
    1 max            2 x 2 / 2   448 x 160 x  32    ->    224 x  80 x  32
    2 conv      64   3 x 3 / 1   224 x  80 x  32    ->    224 x  80 x  64
    3 max            2 x 2 / 2   224 x  80 x  64    ->    112 x  40 x  64
    4 conv     128   3 x 3 / 1   112 x  40 x  64    ->    112 x  40 x 128
    5 conv     256   3 x 3 / 1   112 x  40 x 128    ->    112 x  40 x 256
    6 max            2 x 2 / 2   112 x  40 x 256    ->     56 x  20 x 256
    7 conv     128   3 x 3 / 1    56 x  20 x 256    ->     56 x  20 x 128
    8 conv      54   1 x 1 / 1    56 x  20 x 128    ->     56 x  20 x  54
```

Figure 19: The v14 CNN architecture. Max means maxpool operation, and conv stands for convolution.

4.2.8 *Detection Performance of the CNNs*

Table 6: Performance of trial CNNs that had at least either precision or recall above 50%, for an IoU threshold of 50 %.

Trial CNN	Dataset	mAP	recall	precision	IoU
v5	100 nm	42.91%	43%	84%	57.39%
v7	100 nm	52.33%	52%	90%	62.05%
v8	100 nm	41.88%	46%	82%	56.23%
v10	100 nm	58.94%	56%	86%	60.12%

Table 7: The detection performance of the v14 CNN architecture with different training sets, for an IoU threshold of 50 %.

CNN	Evaluation dataset	mAP	recall	precision	IoU
v14	200 nm	79.68 %	75%	90%	64.34 %
v14	200 nm (+ 100 nm in train.)	81.17%	80%	95%	70.70%
v14	100 nm	59.21%	54%	85%	59.72%
v14	100 nm (+ 200 nm in train.)	60.21%	54%	89%	61.94%

Only the trial CNNs that have a precision or recall value above 50% are shown, the rest of the results are not shown for brevity. The results of the training and testing are collected in Tables 6 and 7. In Table 6, the detection performance of architectures v5, v7, v8, and v10 are shown. Note that these well-performing architectures all have in common that there are only three to four maxpool operation stages, and the filter design is rather simplistic, except for v10. Only the 100 nm dataset is considered in this case for training and testing, as it is the more difficult problem, and well-performing architectures can be filtered more easily.

In all the results, the precision is below or equal to 90 %, and the recall is below 60% in all cases. This means that many excitations are missed. But of the excitations that were found, most are annotated correctly. The results of the v14 architecture are shown in Table 7 separately. Note that the v14 architecture has significantly higher mAP and IOU values then the other architectures, which shows on one hand that the precision-recall curve has more area under it and on the other hand indicates that the predicted bounding boxes are much closer to the ground truth labels.

The yolo-v14-CNN architecture is first trained with only the 200 nm data, and then evaluated on 200 nm data only. In the second training setting, the same architecture is trained on 100 nm data only, and then evaluated on 100 nm data only. In a third step, the architecture is first trained with the 100 nm datasets, and then in next step on the 200 nm as well, but only evaluated on the 100 nm data, theorizing that the CNN will improve by transfer learning and that more data will yield better detection performance.

The results show that for 200 nm, the recall is more than 20% higher than in the 100 nm case. This is reasonable, as 100 nm is more difficult to detect, since the excitations are smaller and more faint. When adding the 100 nm data to the training set, and only evaluating the detection performance for 200 nm, the results improve around 5% points in recall and precision. For the 100 nm case, the results show that the recall is at 54%, while precision is at 89%. Unfortunately, the recall does not increase much when adding the 200 nm images to the 100 nm training set.

4.2.9 *Analysis of the Detection Results*

For 200 nm images the detection performance is high, around 90% precision and 80% recall. For 100nm, there is around 90% precision, but around 55% recall. This means that the evaluation, more than 45% of all virus particles are missed, which means that there are a lot of false negatives.

It should be analyzed why the recall is so low. For this, in Figure 20, 21 and 22, sensor images with a different number of excitations are shown. The boxes are predicted with the v14 architecture. Consider the drawn bounding boxes for different numbers of excitations. In Figure 20, there are only six excitations in total. The CNN can detect all excitations in this image without a problem. In Figure 21, a sensor image with more excitations is shown. It looks like the CNN draws bounding boxes in the regions, where there are excitations. Flat regions have no bounding boxes at all. However, binding excitations can also occur in the same place as a previous excitation. This means that the objects to be detected are very close together. In [23], Redmon et al. claim that yolo has problems when there are many objects very close to each other, and this is the case here. In Figure 22, an version of the sensor image with hundreds of excitations is shown. Here, there are many excitations very close to each other and possibly on top of each other. Evaluating the performance of the v14 architecture on single images, the results are that for images with few excitations, e.g., below 20, the precision and recall are near one, like in Figure 20 and 21. For images like in Figure 22, the precision values of around 80% and recall values of around 50% are common. This supports the argument that too many excitations that are too close together are too difficult for the CNN.

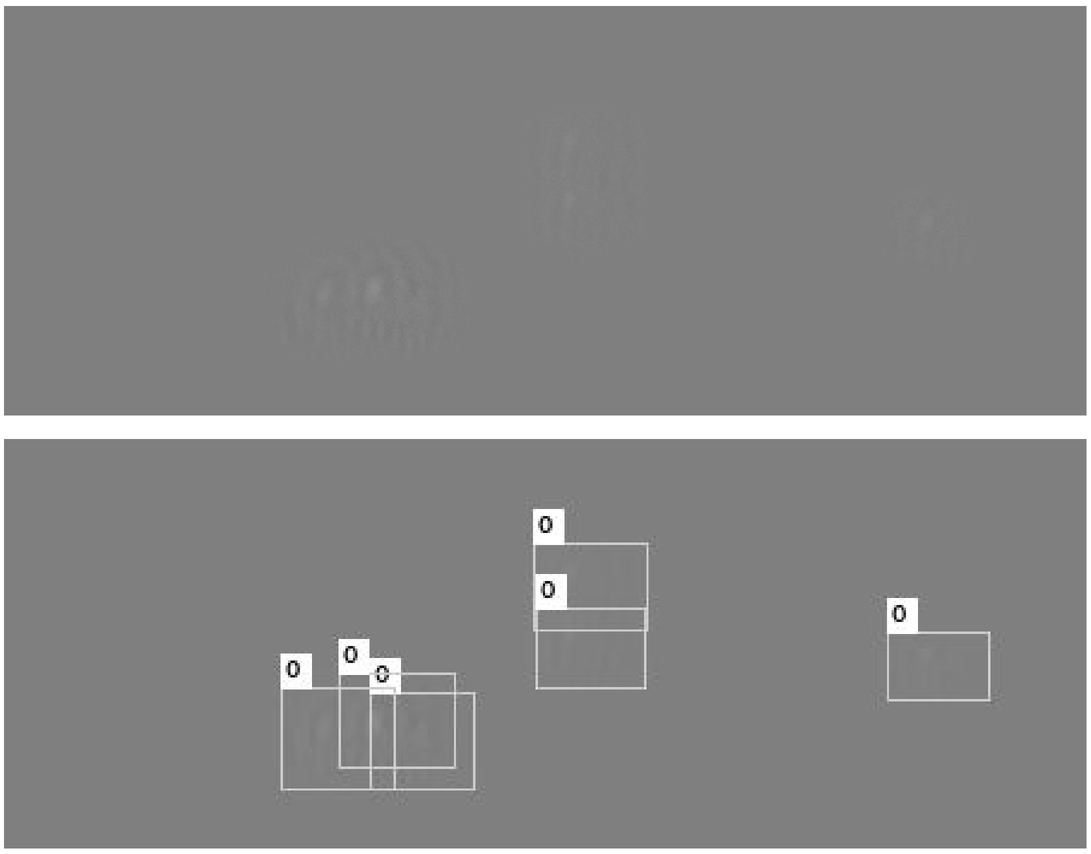


Figure 20: A sensor image for 100 nm with six excitations.

Furthermore, it could be possible that yolo has a technical limit and cannot generate the needed number of bounding boxes to detect all bindings in such extreme cases. Due to the three maxpool layers, the input image is halved three times. When there are excitations too close to each other in the input, the information could be lost due to the maxpool operation. In the experiments for v14, an output layer of $56 \times 20 \times 54$ is used, which are 1120 bounding boxes, each having nine candidate anchors and values for (x, y)-position, width, height, confidence, and class (six by nine values is 54 values). Increasing this size by a factor of two, or decreasing the size of a factor of two decreases the detection performance to a point where the detection is useless. Redmon et al. claim in [24], that the yoloV3 architecture can deal better with objects close to each other. YoloV3 also has more bounding box candidates, as the CNN has multiple output branches, meaning an output layer can be placed anywhere in the CNN architecture, e.g., one at the last layer, and one at the last but

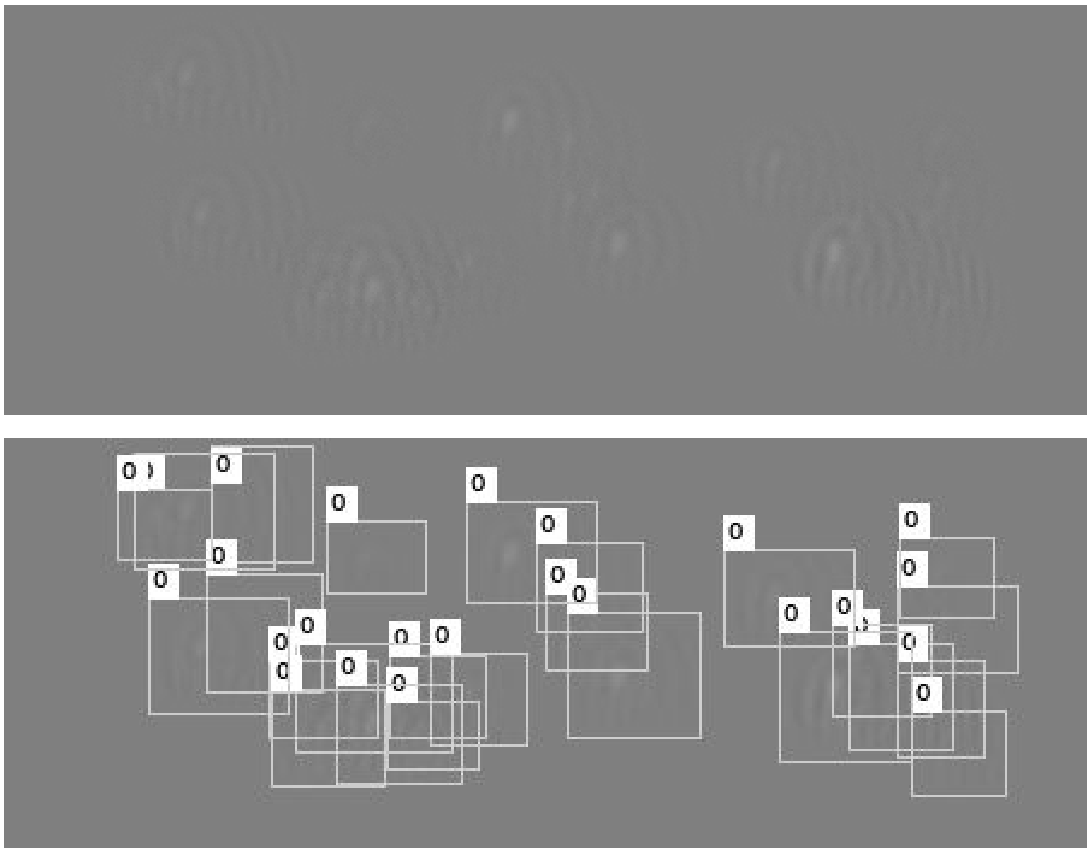


Figure 21: A sensor image for 100 nm with many excitations. The CNN only draws bounding boxes where bindings occur. However, there are some regions where too many bindings occur, and the bounding boxes overlap significantly.

one layer. A yoloV3 architecture is tried by the author, with one output tensor of $56 \times 20 \times 54$ and one output tensor half this size in the (x, y)-dimension. Otherwise, the architecture is equal to v14. However, the results were not significantly different than for yoloV2 v14 architecture, while more convolution operations are needed, which makes the yoloV3 architecture more complex. Therefore, more experiments with yoloV3 were not performed.

4.2.10 *Ideas for Improvement*

Measuring the detection performance of the CNN could be done in the following way, which could be fairer to the yolo method:

- When the center of a binding excitation is within any box, it is not counted as a FN. It is counted as correctly detected.
- Determine the center of excitations by calculating the center of the polygon of the annotation

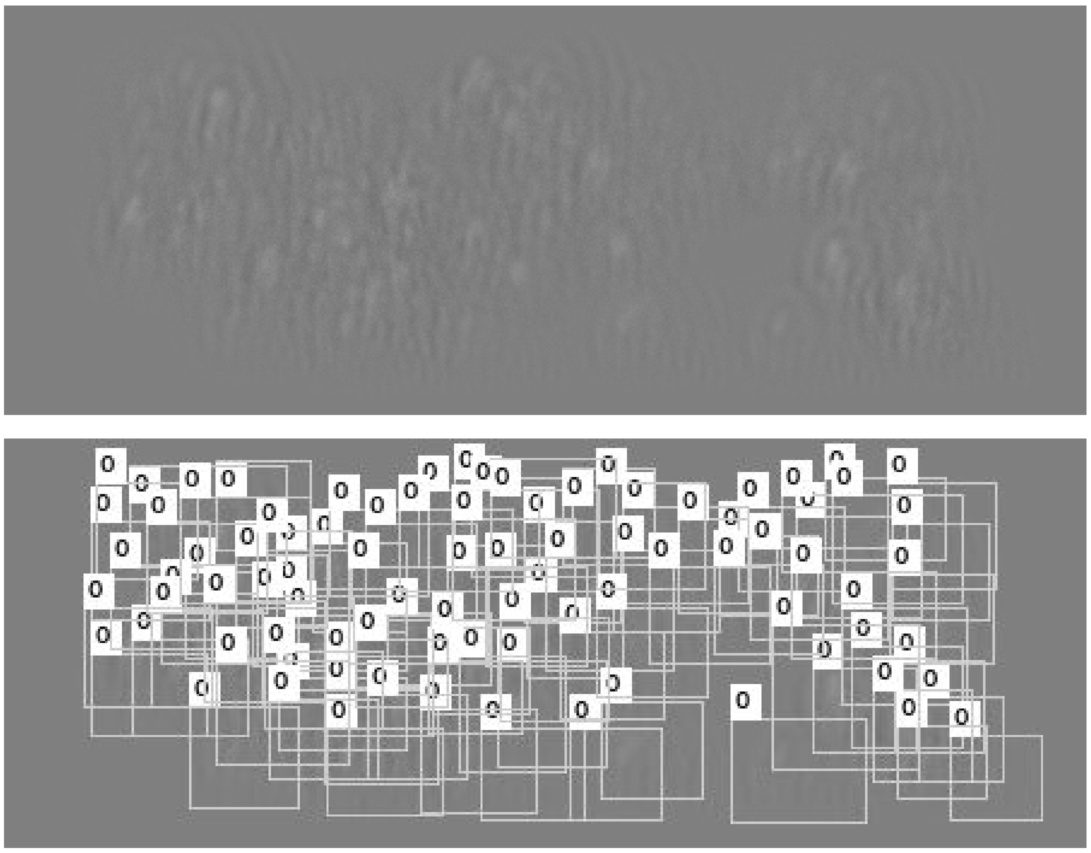


Figure 22: A sensor image for 100 nm with hundreds of excitations. The CNN only draws bounding boxes where bindings occur. However, almost everywhere in the image the bounding boxes overlap significantly.

The reason is that within the yolo bounding boxes, there can be multiple binding excitations. For example, there can be 4 binding excitations, which are very close to each other, within one single bounding box. But only one single box can be generated by yolo for this region, due to the technical limit. In this case there are 1 correct and 3 FN detections. With the new way of measuring the detection performance, all the detection of these cases would be correct detections. However, the biological plausibility of this way of measuring cannot be assessed by the author. Another issue that arises with this way of measuring is that when the center of a binding excitation is very close to the edges of the bounding box (just slightly outside or inside the bounding box), whether these cases have to be incorporated into the evaluation, or whether these "border line" excitations can be left out in the evaluation.

Another idea is to use the yolo-CNN to suggests patches. It can be noticed in the evaluations that the CNN does not draw any bounding boxes in flat areas of the image. If the middle of every excitation is within a bounding box that the yolo-CNN produced, then a patch can be cropped from the middle of the bounding box, and the patch can be classified again with the FFT/Wavelet classifier proposed in this thesis. However, for this approach, the generated patches of the yolo CNN would need to be annotated with "virus" or "no-virus", so that the performance of the yolo CNN with the FFT/Wavelet classifier can be evaluated. How to decide whether a generated patch contains a binding excitation for bounding boxes with an error cannot be assessed by the author. An expert would be needed to annotate the patch dataset produced by yolo.

One could also try other single shot object detection CNNs that may solve the problem better. But there is the problem again, that a useful CNN architecture has to be found by trials. Trying models and training costs a lot of time. Other SSOD also have problems with small objects that are close to each other, like yoloV3. Furthermore other SSOD methods may not be well suited for resource-constrained systems.

4.3 SUMMARY

In this chapter, a new yoloV2 CNN architecture was found, that could be used in the PAMONO biosensor image analysis pipeline to simplify the nano-particle detection. It as a precision of around 95%, and a recall of around 80% for 200 nm particles. For 100 nm particles it has a precision of around 90% and a recall of around 55%. The CNN has under 10 convolutional layers, and has a weight file of 2.8 MB. It can be executed efficiently on an MPSoC FPGA platform with 16 FPS, using quantization and binarization of the network parameters.

The results cannot be compared directly to the results of the detection or classification methods described by Lenssen et al. in [14], as in their case the detection and classification results are evaluated independently. To have a rough idea for a comparison, only the detection (excluding the classification) performance, a precision of 93.3% and a recall of 90% is achieved for the 200 nm case, and a precision of 93.8% and recall of 80% for the 100 nm case. These results however would need an inclusion of the classification performance for a fair comparison between the yolo-CNN and the results in [14]. The necessary information to calculate the precision and recall of the detection and classification however is not given in [14].

The question remains whether "missing" half the excitations when there are hundreds of excitations anyway, is a problem from the perspective of biology and medicine. One could argue that if only half of the virus excitations are detected, it can still be useful for the application. Then there is a trade off in missing around 45% of all excitations, but having high accuracy and speed is the benefit. Another point which remains unaddressed is whether the CNN should be evaluated by using the total number centers of excitations within any bounding box.

5 CONCLUSION

In this thesis, several new approaches for the PAMONO nano-particle image analysis pipeline were explored. In the first part of the thesis, the classification stage of the pipeline was optimized for speed with minimal loss in accuracy. In the second part of the thesis, the usefulness of SSOD methods for virus detection were explored.

Modifications on the classification stage of the image analysis pipeline were made by replacing the classification CNN with a frequency domain analysis method, which beats the CNN in execution time by a big factor, and reduces the resource-demand, with only a small loss in accuracy. The frequency domain analysis based classification was executed on several different platforms: 1) on a mobile CPU using the integrated GPU for acceleration, 2) on a mobile CPU-only version, and 3) on the ARM chip of the Zedboard with and without acceleration of the frequency domain analysis in the PL region. The frequency analysis method is 2.6 times faster than the CNN, when using a mobile GPU as an accelerator. The method was further sped up by performing the classification as a CPU-only version, i.e., from 1280 to 27 microseconds for the Fourier features, and from 1500 to 17 microseconds for the Haar wavelet features when evaluated on a Intel Core i7 4600U. For comparison, the classification CNN needs 3370 microseconds on the same platform. Nanoparticle classification systems using frequency analysis methods are a more resource-efficient alternative to CNNs, thus the methods explored in this part of the thesis can serve as a blueprint for the design of resource-efficient virus classification methods for SPR images.

Modifications on the detection and classification stages of the image analysis pipeline were made to assess the usefulness of the Single Shot Object Detector yoloV2-CNN architectures, by combining the detection and classification stage into one step. The results show that a simple yolo architecture with a shallow structure and fewer layers and kernels than the tiny-yolo architecture can reliably detect and classify particle bindings in 200 nm images, i.e., with around 95% precision and around 80% recall. How to improve the classification performance for 100 nm, i.e., around 90% precision and 55% recall, remains an open problem, and a few ideas were discussed.

In conclusion, a resource-efficient classical method of using frequency domain analysis was added to the design space of the classification methods, and the usefulness of previously untried SSOD methods were explored for the detection and classification in one single step.

Since the frequency domain analysis method as a CPU-only version is extremely fast compared to the accelerated versions and the CNN, the author believes that in practice, the nano-particle analysis pipeline should be run using a simplified version of the detection stage and the frequency domain based classification. With the contributions of this thesis in the first part, around 100 times more patches can be checked in the same time compared to the CNN.

Another idea for optimizing the work from Lenssen et al. in [14] is to quantize and binarize their CNN parameters and feature maps to find out whether there is a significant speedup in execution time with a tolerable loss in accuracy.

Exploring other, more lightweight methods for the detection and classification stage, which are also more suitable for the use in small resource-constrained systems, and the optimization of the CNNs for the detection and classification are left as future work.

A APPENDIX: CODE AND CONFIG FILES

A.1 THE IMPLEMENTATION OF THE FEATURE EXTRACTION ALGORITHMS

In this section, the implementation of the feature extraction algorithms are documented. In Listing 1, the preparations and function definitions are shown.

```
/* The implementation is inspired by http://www.bealto.com/gpu-fft_opencl
    -1.html */
#ifndef M_PI
#define M_PI 3.1416 // TODO: more exact PI
#endif

/* Return real or imaginary component of complex number */
inline float real(float2 a){
     return a.x;
}
inline float imag(float2 a){
     return a.y;
}

/* Complex multiplication */
#define MUL(a, b, tmp) { tmp = a; a.x = tmp.x*b.x - tmp.y*b.y; a.y = tmp.x*
    b.y + tmp.y*b.x; }

/* Butterfly operation */
#define DFT2(a, b, tmp) { tmp = a - b; a += b; b = tmp; }

/* Return e^(i*alpha) = cos(alpha) + I*sin(alpha) */
float2 exp_alpha(float alpha)
{
  float cs,sn;
  sn = sincos(alpha,&cs);
  return (float2)(cs,sn);
}
```

Listing 1: The data structures and functions for the 2D FFT OpenCL kernel implementations.

In Listing 3, the implementation of the 2D FFT OpenCL kernel is shown. The FFTs are parallelized across rows, i.e., for a 32×32 image, all the rows FFTs are computed in parallel. To call this kernel and transform an image, it has to be called in a loop. Two buffers are needed, as it is an out-of-place algorithm. Below the pseudo-code as a guideline for calling the OpenCL kernels to acquire the FFT features, in Listing 2. The OpenCL code of the transpose kernel is shown in Listing 4 and the implementation of the magnitude kernel is shown in 5.

```
for(int p = 1; p <= width; p *= 2)
    configure kernel arguments
        input memory and output memory
        p (level of the FFT algorithm)
        height and weight of the image

    execute FFT on rows
    transpose the result with the transpose-kernel
    execute FFT on rows (equivalent to executing FFT on columns)
    transpose the result

call the shift kernel
call the magnitude kernel
compute sums on the CPU
```

Listing 2: Pseudocode for using the OpenCL 2D FFT kernel.

```
/**
* \brief 2D grayscale out-of-place FFT algorithm
* \param[in] in Input image, grayscale or spectral. In the first half of
    the array are the real parts, and in the second half the imaginary parts
    . E.g., for 4x4 images, in the first 16 elements are the real parts, and
     in the seconds 16 elements are the imaginary parts. In case of a
    grayscale image, only one array with 16 elements is allowed as input
    aswell.
* \param[out] out Output of the transformation, in the first half of the
    array are the real parts, and in the second half the imaginary parts. E.
    g., for 4x4 images, in the first 16 elements are the real parts, and in
    the seconds 16 elements are the imaginary parts
* \param[in] p A power of two, denotes the level of the fft algorithm
* \param[in] r_c Value to determine whether input real or complex, r_c == 1
     means real, 0 means complex
* \author mikail
*/

__kernel void fft(
  __global float * in,
  __global float * out,
  int p,
  int r_c
)
{
  int gx = get_global_id(0);
  int gy = get_global_id(1);
  int w = (int) get_global_size(0)*2;
  int h = (int) get_global_size(1);
  int offset = gy*w;
  int gid = w*gy + gx;
  int imag_offset = w*h;

  int k = (gx) & (p-1);

  float2 u0;
  float2 u1;

  if (p == 1 && r_c == 1) // first fft iteration, input is real
  {
    u0.x = in[gid];
    u0.y = 0;

    u1.x = in[gid + (w/2)];
    u1.y = 0;
  }
  else // get real and imaginary part
  {
    u0.x = in[gid]; //real part
    u0.y = in[gid + imag_offset]; //imag part

    u1.x = in[gid + (w/2)];
    u1.y = in[gid + (w/2) + imag_offset];
  }

  float2 twiddle;
  float2 tmp;

  twiddle = exp_alpha( (float)(k)*(-1)*M_PI / (float)(p) );

  MUL(u1,twiddle,tmp);

  DFT2(u0,u1,tmp);

  int j = ((gx) << 1) - k;
  j += offset;

  out[j] = real(u0);
  out[j + p] = real(u1);

  out[j + imag_offset] = imag(u0);
  out[j + p + imag_offset] = imag(u1);
}
```

Listing 3: The 2D FFT OpenCL kernel. Operations are parallized across rows.

```
/**
* \brief Transposes the input into the output
* \param[in] in Input array
* \param[out] out Output array
* \param[in] real_width The real width of the array, used for zeropadding
* \param[in] real_height The real height of the array, used for zeropadding
* \author mikail
*/
__kernel void transpose(
  __global float * in,
  __global float * out
)
{
  int gx = get_global_id(0);
  int gy = get_global_id(1);
  int gz = get_global_id(2);
  int w = (int) get_global_size(0);
  int h = (int) get_global_size(1);
  int gid  = w*h*gz + w*gy + gx;
  int t_gid  = w*h*gz + w*gx + gy;

  out[t_gid] = in[gid];
}
```

Listing 4: The OpenCL kernel for transposing an image.

```
/*
1 2  becomes  4 3
3 4           2 1
*/
/**
* \brief Shifts the magnitudes of the fft to the middle
* \param[in] in Input array
* \param[out] out Output array
* \author mikail
*/
__kernel void shiftFFT(
  __global float *in,
  __global float *out
  )
  {
    int width = (int) get_global_size(0);
    int height = (int) get_global_size(1);
    int gx = get_global_id(0);
    int gy = get_global_id(1);
    int gid = gy*width + gx;
    int imag_offset = width*height;
    int eq1 = ((width*height) + width)/2;
    int eq2 = ((width*height) - height)/2;

    if (gx < width/2)
    {
      if (gy < height/2)
      {
        // swap first quadrant with fourth
        out[gid] = in[gid+eq1];
        out[gid+eq1] = in[gid];

        out[gid + imag_offset] = in[gid + eq1 + imag_offset];
        out[gid + eq1 + imag_offset] = in[gid + imag_offset];
      }
    }
    else
    {
      if (gy < height/2)
      {
        // swap second quadrant with third
        out[gid] = in[gid + eq2];
        out[gid+eq2] = in[gid];

        out[gid + imag_offset] = in[gid + eq2 + imag_offset];
        out[gid + eq2 + imag_offset] = in[gid + imag_offset];
      }
    }
}
```

Listing 5: The OpenCL shift kernel for the 2D FFT according to the CUDA implementation in [1].

```
/**
* \brief Calculates the magnitudes of a fft
* \param[in] in Input array
* \param[out] out Output array
* \author mikail
*/
__kernel void absFFT(
  __global float *in,
  __global float *out
  )
  {
    int width = (int) get_global_size(0);
    int height = (int) get_global_size(1);
    int gx = get_global_id(0);
    int gy = get_global_id(1);
    int gid = gy*width + gx;
    int imag_offset = width*height;

    float real = in[gid];
    float imag = in[gid + imag_offset];
    real *= real;
    imag *= imag;
    out[gid] = sqrt(real + imag);
  }
```

Listing 6: Magnitude calculation of a 2D FFT OpenCL kernel.

In Listing 8, the implementation of the 2D FWT OpenCL kernel is shown. The FWTs are parallelized across rows, i.e., for a 32×32 image, all the rows FWTs are computed in parallel. To call this kernel and transform an image, it has to be called in a loop. Below the pseudo-code as a guideline for calling the OpenCL kernels to acquire the FWT features, in Listing 7. The implementation of the transpose for the FWT is shown in Listing 9 and the implementation of the energy computation is shown in Listing 10.

```
for(int i = 2; i <= level; i*=2)
    configure kernel arguments
        input memory and output memory
        i (level of the FWT algorithm)
        height and weight of the image

    call FWT kernel on all rows
    transpose
    call FWT kernel on all rows
    transpose

calculate energy terms
sum on the CPU
```

Listing 7: Pseudocode for using the OpenCL 2D FWT kernel.

```
/**
* \brief Computes the 2D Haar Wavelet transformation
* \param[in] in Input array
* \param[out] out Output array
* \param[in] img_width The width of the image
* \author mikail
*/
__kernel void haarwt(
  __global float *in,
  __global float *out,
  int img_width
  )
  {
    int width = (int) get_global_size(0);
    int gx = get_global_id(0);
    int gy = get_global_id(1);

    out[img_width*gy + gx] = in[img_width*gy + 2*gx] + in[img_width*gy + 2*
    gx + 1];
    out[img_width*gy + gx] /= SQRT2;
    out[img_width*gy + width + gx] = in[img_width*gy + 2*gx] - in[img_width
    *gy + 2*gx + 1];
    out[img_width*gy + width + gx] /= SQRT2;
  }
```

Listing 8: The 2D FWT OpenCL kernel, according to [6].

```
/**
* \brief Simple transpose for the haarwt transformation
* \param[in] in Input array
* \param[out] out Output array
* \author mikail
*/
__kernel void hwttranspose(
  __global float * in,
  __global float * out
)
{
  int gx = get_global_id(0);
  int gy = get_global_id(1);
  int width = (int) get_global_size(0);
  int gid =  width*gy + gx;
  int t_gid = width*gx + gy;

  out[t_gid] = in[gid];
}
```

Listing 9: The OpenCL 2D FWT transpose kernel.

```
/**
* \brief Calculate all energy summands in-place
* \param[in] in Input array, is also the output array
* \author mikail
*/
__kernel void wenergy2All(
  __global float * in
)
{
  int gx = get_global_id(0);
  int gy = get_global_id(1);
  int width = (int) get_global_size(0);
  int height = (int) get_global_size(1);
  int gid =  width*gy + gx;

  in[gid] = in[gid]*in[gid];
}
```

Listing 10: The OpenCL 2D FWT energy computation kernel.

The 2D FWT was also implemented as a CPU-only version in Listing 11.

```
#include <Fwt.h>
#include <cmath>
#include <cstring>
#include <stdlib.h>

using namespace std;
void Fwt::haarWT(float* features, unsigned int patchWidth, unsigned int
    patchHeight, unsigned int levels)
{
  int w = patchWidth;
  int h = patchHeight;
  for (unsigned int level = 0; level < levels; level++)
  {
    // rows
    for (int row = 0; row < h; row++)
    {
      int offset = patchWidth * row;
      dwt1level_rows(features, offset, w);
    }

    // columns
    float* temp = (float*) malloc(sizeof(float) * h);
    for (int col = 0; col < w; col++)
    {
      for (int row = 0; row < h; row++)
      {
        temp[row] = features[col + patchWidth * row];
      }
      dwt1level(temp, w);
      for (int row = 0; row < h; row++)
      {
        features[col + patchWidth * row] = temp[row];
      }
    }
    w /= 2;
    h /= 2;
    free(temp);
  }
}
```

Listing 11: The 2D FWT computation without OpenCL acceleration in C++.

```cpp
void Fwt::dwt1level(float* input, int width)
{
  std::vector<float> temp(width);
  int n = width / 2;

  for (int i = 0; i < n; i++)
  {
    temp[i] = input[i*2] + input[i*2 + 1];
    temp[i] /= 2;

    temp[i + n] = input[i*2] - input[i*2 + 1];
    temp[i + n] /= 2;
  }
  memcpy(&input[0], &temp[0], width * sizeof(float));
}

void Fwt::dwt1level_rows(float* input, int offset, int width)
{
  std::vector<float> temp(width);
  int n = width / 2;

  for (int i = 0; i < n; i++)
  {
    int i2 = offset + 2 * i; // access correct row in features array with
    offset

    temp[i] = input[i2] + input[i2 + 1];
    temp[i] /= 2;

    temp[i + n] = input[i2] - input[i2 + 1];
    temp[i + n] /= 2;
  }
  memcpy(&input[offset], &temp[0], width * sizeof(float));
}
```

Listing 12: The 2D FWT functions for the CPU-only version in C++.

A.2 THE YOLO CONFIG FILE FOR VIRUS DETECTION

In Listing 13 the yoloV2 configuration file for the v14 architecture is shown. Many of the configurations have to be chosen by trial and error. The explanations of the configuration parameters are in the configuration file after the "#". Some configuration options are not documented well in the darknet documentations, that is why some explanations may be sparse.

Listing 13: The yolo configuration file for the v14 architecture.

```
[net]
batch=64 # Number of images considered for one gradient update step
subdivisions=8 # Number of images that are loaded in parallel to GPU memory
width=448 # Image width, to which the input width is resized
height=160 # Image height, to which the input height is resized
channels=1 # 1 For grayscale, 3 for color channels
momentum=0.9 # 0.9 makes the gradient descent more stable for this application
decay=0.0005 # Factor to diminish the weight updates
angle=0 # Data augmentation by rotating the image
saturation=0 # Data augmentation by saturation, not useful for grayscale
exposure=0 # Data augmentation by exposure, not useful for grayscale
hue=0 # Data augmentation by hue, not useful for grayscale

learning_rate=0.001 # The factor with which the gradient update is multiplied
burn_in=0 # Number of iterations in which a fitting learning rate can be found
    by successively decreasing it
max_batches=500200 # Upper limit for the maximum number of iterations
policy=steps
steps=500,1000 # Iterations which the learning rate should be multiplied by the
    values specified in "scale"
scales=.1,.2,.2,0.3 # The factors with which the learning rate should be
    multiplied at steps iterations

# A definition of a convolutional layer
[convolutional]
batch_normalize=1 # activates batch normalization during traininf
filters=32 # Number of filters used
size=3 # Size of the filter (3x3 in this case)
stride=1 # Size of the strides
pad=1 # Standard padding
activation=leaky # relu is also possible

# A definition of a maxpool layer
[maxpool]
size=2
stride=2

[convolutional]
```

```
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

###########

[convolutional]
batch_normalize=1
size=3
stride=1
pad=1
filters=128
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=54
activation=linear

[region]
```

```
# Anchors contain the informations about the bounding box candidate dimensions
anchors = 0.8330,0.8870, 0.9817,0.7778, 1.1471,0.8899, 1.0153,1.0617,
    1.1333,1.2006, 1.2797,1.1555, 1.4413,1.0621, 1.4215,1.2711, 1.7420,1.2199
bias_match=1 # With "1", anchor dimensions are used, with "0", they will be
    refined
classes=1 # Number of predicted classes
coords=4
num=9 # Number of anchors used
softmax=1
jitter=0.2 # The extent to which random cropping should be used for data
    augmentation
rescore=0 # Determines which loss function should be used

# The following four parameters are for adapting the yolo cost function
object_scale=5
noobject_scale=1
class_scale=1
coord_scale=1

absolute=1
thresh = .6 # Minimum IOU during training
random=0 # If 1, random resizing will be activated for data augmentation
```

BIBLIOGRAPHY

[1] Marwan Abdellah. "CufftShift: High performance CUDA-accelerated FFT-shift library." In: *High Performance Computing Symposium*. 2014.

[2] *AlexeyAB's Darknet Fork*. `https://github.com/AlexeyAB/darknet`. Accessed: 2019-06-01.

[3] Inad Aljarrah, Anas Toma, and Mohammad Al-Rousan. "An Automatic Intelligent System for Diagnosis and Confirmation of Johne's Disease." In: *Int. J. Intell. Syst. Technol. Appl.* 14.2 (2015), pp. 128–144.

[4] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights." In: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2016, pp. 236–241.

[5] Eric Bainville. *Bealto Website*. `http://www.bealto.com/gpu-fft2.html`. Accessed: 2019-06-01.

[6] Hovhannes Bantikyan. "CUDA Based Implementation of 2-D Discrete Haar Wavelet Transformation." In: *Parallel and Distributed Computing Systems (PDCS)*. 2014, pp. 48–57.

[7] S. Buschjager, K. Chen, J. Chen, and K. Morik. "Realization of Random Forest for Real-Time Evaluation through Tree Framing." In: *IEEE International Conference on Data Mining (ICDM)*. 2018, pp. 19–28.

[8] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. "The Pascal Visual Object Classes (VOC) Challenge." In: *Int. J. Comput. Vision* 88.2 (2010), pp. 303–338.

[9] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3." In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[10] *Github: deepRacin*. "`https://github.com/mrjel/deepracin`". Accessed 2019-06-01.

[11] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (4rd Edition)*. Pearson, 2018.

[12] Vinod Kathail, James Hwang, Welson Sun, Yogesh Chobe, Tom Shui, and Jorge Carrillo. "SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC." In: *Xilinx Research*. 2016.

[13] Erich Kretschmann. "The determination of the optical constants of metals by excitation of surface plasmons." In: *Eur Phys J A* 241 (1971), pp. 313–324.

[14] Jan Lenssen, Anas Toma, Albert Seebold, Victoria Shpacovitch, Pascal Libuschewski, Frank Weichert, Jian-Jia Chen, and Roland Hergenröder. "Real-time Low SNR Signal Processing for Nanoparticle Analysis with Deep Neural Networks." In: *BIOSIGNALS*. 2018, pp. 36–47.

[15] Pascal Libuschewski. "Exploration of cyber-physical systems for GPGPU computer vision-based detection of biological viruses." PhD thesis. Technical University of Dortmund, Germany, 2017.

[16] Olaf Neugebauer, Pascal Libuschewski, Michael Engel, Heinrich Müller, and Peter Marwedel. "Plasmon-based Virus Detection on Heterogeneous Embedded Systems." In: *International Workshop on Software and Compilers for Embedded Systems*. 2015, pp. 48–57.

[17] Kalin Ovtcharov, Olatunj i Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware." In: *Microsoft Research* (2015).

[18] Jonathan Pedoeem and Rachel Huang. "YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers." In: *CoRR* abs/1811.05588 (2018).

[19] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *J Mach Learn Res* 12 (2011), pp. 2825–2830.

[20] T. B. Preußer, G. Gambardella, N. Fraser, and M. Blott. "Inference of quantized neural networks on heterogeneous all-programmable devices." In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 833–838.

[21] Yuran Qiao, Junzhong Shen, Tao Xiao, Qianming Yang, Mei Wen, and Chunyuan Zhang. "FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency." In: *Concurrency and Computation: Practice and Experience* 29.20 (2016), e3850.

[22] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. `http://pjreddie.com/darknet/`. Accessed 2019-06-01.

[23] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In: *arXiv preprint arXiv:1612.08242* (2016).

[24] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement." In: *CoRR* abs/1804.02767 (2018).

[25] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection." In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 779–788.

[26] *SFB 876-B2 Website*. `https://sfb876.tu-dortmund.de/SPP/sfb876-b2.html`. Accessed: 2019-06-01.

[27] Victoria Shpacovitch et al. "Application of surface plasmon resonance imaging technique for the detection of single spherical biological submicrometer particles." In: *Analytical Biochemistry* 486 (2015), pp. 62 –69.

[28] Dominic Siedhoff. “A parameter-optimizing model-based approach to the analysis of low-SNR image sequences for biological virus detection.” PhD thesis. Technical University Dortmund, Germany, 2016.

[29] Dominic Siedhoff, Pascal Libuschewski, Frank Weichert, Alexander Zybin, Peter Marwedel, and Heinrich Müller. “Modellierung und Optimierung eines Biosensors zur Detektion viraler Strukturen.” In: *Bildverarbeitung für die Medizin*. 2014, pp. 108–113.

[30] Avnet Whitepaper. *Implementing FFT Accelerators with SDSoC TM Using Open-Source Software and C-Callable IP*. 2018.

[31] Mikail Yayla, Anas Toma, Jan Eric Lenssen, Victoria Shpacovitch, Kuan-Hsun Chen, Frank Weichert, and Jian-Jia Chen. “Resource-Efficient Nanoparticle Classification Using Frequency Domain Analysis.” In: *Bildverarbeitung für die Medizin*. 2019, pp. 339–344.

[32] Alexander Zybin, Yurii Kuritsyn, Evgeny Gurevich, Vladimir Temchura, Klaus Überla, and Kay Niemax. “Real-time Detection of Single Immobilized Nanoparticles by Surface Plasmon Resonance Imaging.” In: *Plasmonics* 5 (2010), pp. 31–35.

[33] *myay's Github with the Testing Framework for the Frequency Domain Classification*. `https://github.com/myay/PamonoClassificationFFTFWT`. Accessed: 2019-06-01.

[34] *myay's yolo-Pamono Github*. `https://github.com/myay/yolo-Pamono`. Accessed: 2019-06-01.

[35] *pjreddie's Darknet*. `https://github.com/pjreddie/darknet`. Accessed: 2019-06-01.

DECLARATION

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 01.06.2019
Mikail Yayla

Eidesstattliche Versicherung

Name, Vorname   Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum   Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum   Unterschrift

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of October 11, 2019 (`classicthesis` version 4.2).