Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Aksel Lindbæk Gundersen

Hardware-Software partitioned implementation of an autoencoder-based hyperspectral anomaly detector

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Milica Orlandic
June 2021

NTNU
Norwegian University of Science and Technology

Aksel Lindbæk Gundersen

Hardware-Software partitioned implementation of an autoencoder-based hyperspectral anomaly detector

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Milica Orlandic
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

The Hyper-Spectral Small-Satellite for Oceanographic Observations (HYPSO) is an ongoing mission at the small-satellite laboratory at the Norwegian University of Science and Technology (NTNU). The objective is to capture and process hyperspectral images of the ocean with a satellite. The obtained information will be used to monitor algal blooms along the Norwegian coastline. Hyperspectral Anomaly Detection (HAD) is a feature that can increase the performance of the mission by autonomously detecting regions of interest in the captured hyperspectral images. HAD is the process of locating rare pixels in hyperspectral images that notably differ from their surrounding pixels, without a priori knowledge concerning the pixels.

An autoencoder-based method for HAD is adapted from [1] and modified to increase the accuracy in this thesis. The HAD algorithm uses an autoencoder to learn the high-level features of the hyperspectral image before using that information to perform a novel method for computing the anomaly scores for all pixels in the hyperspectral image. The anomaly score is the probability that a pixel is an anomaly determined by the HAD algorithm. An extensive review of known baseline and state-of-the-art HAD algorithms is presented in this thesis. The HAD algorithm here is evaluated in Matlab and compared to known baseline and state-of-the-art HAD algorithms. The results show that it achieves the third-highest average accuracy on a set of hyperspectral images widely used to evaluate HAD algorithms.

A simplified version of the HAD algorithm proposed in this thesis is implemented using the C-programming language. The C-programming language is chosen as the programming language used in the processing pipeline onboard the satellite for the HYPSO mission. A computationally expensive part of the C implementation is accelerated in customized hardware using high-level synthesis. A range of optimization techniques, including increased concurrency and fixed-point arithmetic in high-level synthesis, are explored and compared.

Finally, a hardware-software partitioned implementation of the HAD algorithm is tested using a Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit. The C implementation runs on the application processing unit on the ZCU104. The accelerated part of the algorithm is implemented as a hardware kernel in the programmable logic on the ZCU104. The optimized hardware kernel achieves a maximum of about ten times faster computation of the anomaly scores than the C version of the hardware kernel on a 2.9GHz CPU and up to 100 times faster than the C version on the application processing unit on the ZCU104.

Sammendrag

Hyper-Spectral Small-Satellite for Oceanographic Observations (HYPSO) er et pågående oppdrag på laboratoriet for små satellitter på Norges Tekniske og Naturvitenskapelige Universitet (NTNU). Målet med oppdraget er å ta og prosessere hyperspektrale bilder av havet med en satellitt. Informasjonen fra de hyperspektrale bildene skal bli brukt til å overvåke alge-oppblomstringer langs Norskekysten. Deteksjon av avvik i hyperspektrale bilder (Hyperspectral Anomaly Detection (HAD)) er et hjelpemiddel som kan bidra til oppdraget ved å detektere områder av interesse i de hyperspektrale bildene som blir tatt autonomt. HAD bilder omhandler det å finne sjeldne piksler i bildene, som skiller seg klart ut fra sine omgivelser, uten a priori kunnskap om pikslene.

I denne masteroppgaven er en autoenkoder basert metode for HAD adaptert fra [1] og modifisert for å øke deteksjonspresisjonen. HAD algoritmen bruker en autoenkoder til å lære høy-nivå karaktertrekk i et hyperspektralt bilde, før denne informasjonen er brukt til å utføre en ny metode for å regne ut sannsynligheten for at pikslene avviker fra sine omgivelser. En omfattende oversikt over de siste fremskrittene og tradisjonelle metoder innen deteksjon av avvik i hyperspektrale bilder er presentert i denne masteroppgaven. Den foreslåtte HAD algoritmen er evaluert i Matlab og sammenlignet med kjente HAD algoritmer. Resultatene viser at den foreslåtte algoritmen oppnår de tredje høyeste gjennomsnittlige presisjons resultatene på et sett med hyperspektrale bilder som er mye brukt til evaluering av HAD algoritmer.

En forenklet versjon av den foreslåtte algoritmen er implementert i programmeringsspråket C. C er valgt ettersom dette er programmeringsspråket som er brukt i prosesseringen ombord HYPSO satellitten. En del av den foreslåtte algoritmen er akselerert i spesialtilpasset maskinvare ved hjelp av høynivå syntese. En rekke optimaliserings teknikker, inkludert å øke antall operasjoner som skjer samtidig, er utforsket og sammenlignet.

Til slutt er en maskinvare-programvare partisjonert implementasjon av den foreslåtte algoritmen testet på et Zynq Ultrascale+ MPSoC ZCU104 evaluerings brett. C implementasjonen kjører på applikasjons prossesserings enheten på ZCU104, mens den akselererte delen av algoritmen er implementert som en maskinvare kjerne i den programmerbare logikken på ZCU104. Den optimaliserte maskinvare kjernen oppnår maksimalt ca. 10 ganger raskere utregning av resultatene enn C versjonen kjørt på en 2.9GHz CPU, og ca. 100 ganger raskere enn C versjonen kjørt på prosesserings systemet på ZCU104.

Preface

This master's thesis is the last part of my five years at the master's degree program in Electronics Systems Design and Innovation at NTNU. Though it has been hard at times, I have enjoyed my years at NTNU greatly.

Multiple persons have aided me in the process of finishing this master's thesis. My supervisor Milica Orlandić has been very supportive and has provided significant assistance on the theoretical and practical aspects of the thesis. My family and girlfriend have given their full support. I want to thank my mother, Elise, for all the good discussions regarding the thesis work. My co-students at the HYPSO project Aksel Danielsen and Sondre Tagestad, have been very important, both socially and by answering my many questions during the work with this thesis. Lastly, I would like to thank the project leader of HYPSO, Evelyn Honoré-Livermore, for taking the time to provide feedback on my work.

Table of Contents

List of Figures

List of Tables

Abbreviations

ABU	Airport–Beach–Urban
AE	Autoencoder
AED	Attribute and Edge-preserving filters Detector
AI	Artificial Intelligence
AMBA	Arm Advanced Microprocessor Bus Architecture
AN	Artificial Neuron
ANN	Artificial Neural Network
APU	Application Processing Unit
ASIC	Application-Specific Integrated Circuit
AVIRIS	Airborne Visible/Infrared Imaging Spectrometer
AWDBN	Adaptive Weights Deep Belief Network
AUC	Area Under Curve
AXI	Advanced eXtensible Interface
BIP	Band Interleaved by Pixel
BP	Backpropagation
BRAM	Block Random Access Memory
CD	Contrastive Divergence
CLB	Configurable Logic Block
CRX	Causal RX
CRD	Collaborative Representation Detector
CPU	Central Processing Unit
DAE	Denoising Autoencoder
DBN	Deep Belief Network
DL	Deep Learning
DSP	Digital Signal Processor
DSW	Double Sliding Window
DWRX	Dual Window RX
FeFR-RX	Fractional Fourier Entropy RX
FF	Flip-Flop
F-MGD	Fast Morphological and guided filters Detector
FN	False Negatives
FPGA	Field-Programmable Gate Array
FPR	False Positive Rate
GD	Gradient Descent
GRX	Global RX
HAD	Hyperspectral Anomaly Detection
HDL	Hardware Description Language
HLL	High-Level Language
HLS	High-Level Synthesis
HSI	Hyperspectral Image
HW	Hardware
HYPSO	Hyper-Spectral Small-Satellite for Oceanographic Observations
I/O	Input/Output
IP	Intellectual Property
KRX	Kernel RX
LRX	Local RX
LUT	Look-Up Table
ML	Machine Learning
MPAF	Morphological Profile and Attribute Filters
NTNU	Norwegian University of Science and Technology
PCA	Principal Component Analysis
PDF	Probability Density Function
PUT	Pixel Under Test
RBM	Restricted Boltzmann Machine
RMSE	Root Mean Square Error
RX	Reed-Xiaoli Detector

ROC	Receiver Operator Characteristics
RTL	Register-Transfer Level
SW	Software
SAE	Sparse Autoencoder
SDBP	Spatial Density Background Detector
SVDD	Support Vector Data Description
SW	Software
TP	True Positive
TPR	True Positive Rate

1 Introduction

1.1 Motivation

The surface of planet Earth consists of roughly 70 percent water, with the clear majority being held by the ocean. Information about the ocean is essential to understand the impact of human activity and the challenges of climate change we are facing in the near and distant future. Ocean monitoring has traditionally been performed by ships, airplanes, and satellites carrying a wide range of sensors. The utilization of ships and airplanes for ocean monitoring has severe limitations and obstacles due to the size and harsh conditions of the oceans. Satellites can cover large geographical areas in a short space of time compared to ships and airplanes. The advances in technology in recent years have enabled the gathering of detailed information from space by using methods such as hyperspectral imaging [3].

1.1.1 Hyperspectral imaging

Hyperspectral imaging is a method that combines the aspects of traditional imaging and spectrometry to obtain both spectral and spatial information about an area or object. Common images are acquired by measuring reflected visible light and obtaining the intensity of the colors red, green, and blue. The intensity of these three colors determines how each pixel of the image should present a graphical representation of the captured object or area. Spectrometry is the measurement of the intensity of electromagnetic radiation at different wavelengths. A hyperspectral camera combines these two methods by measuring the intensity of hundreds of narrow frequency bands in the electromagnetic spectrum for each pixel. A Hyperspectral Image (HSI) consists of 2 spatial and one spectral dimension and is commonly represented as a cube [14]. Figure 1 illustrates the construction of a HSI with the two spatial dimensions w and h and the spectral dimension λ representing different wavelengths. The Figure further shows how the HSI cube can be viewed both as a stack of two-dimensional greyscale images captured at different wavelengths or as an image where each pixel contains a spectrum.

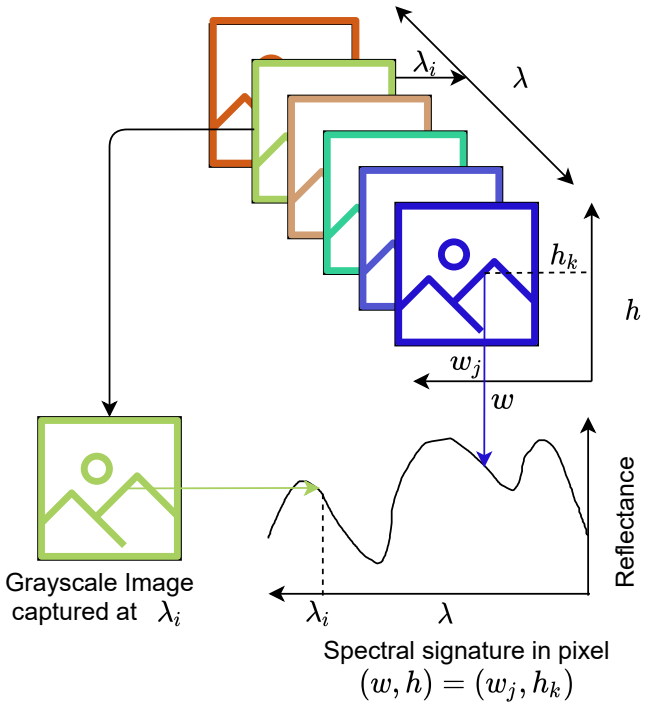


Figure 1: Conceptual illustration of a HSI adapted from [2]

The spectrum that each pixel contains in a HSI is called a spectral signature and can be used to detect similarities and differences between pixels that are not visible to the human eye. The high spectral resolution gives a highly detailed representation compared to common imaging and is therefore rapidly emerging as the technique utilized in remote sensing [14].

1.1.2 Remote sensing

Remote sensing concerns techniques for obtaining information about the Earth from a distance. The information is commonly acquired by measuring the reflected radiation of the electromagnetic spectrum from Earth with sensors onboard satellites or airplanes. Technological advances have led to an increased interest in utilizing hyperspectral cameras for remote sensing. The information is more detailed than common digital images and can be used to perform more accurate detection, and monitoring of objects and phenomena on earth [15]. The spectral signature that each pixel of a HSI contains promotes the classifying of different materials as shown in Figure 2.

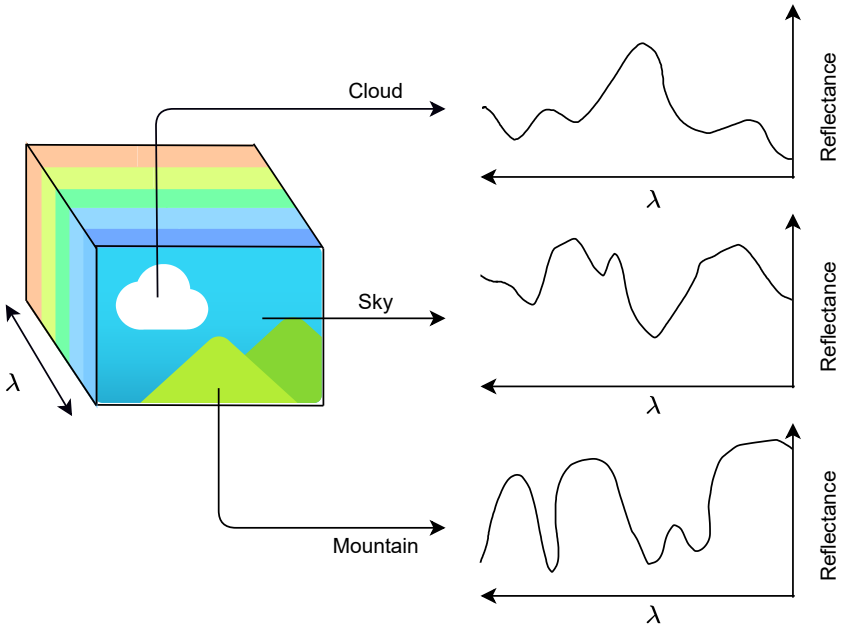


Figure 2: Illustration of spectral differences between different materials in HSI.

Target detection is a widely used method in remote sensing that seeks to detect specific observations such as hurricanes, oil spills, wildfires et cetera. There are two categories of target detection, namely supervised and unsupervised. Supervised target detection requires a priori information about the target, which most commonly is the target's spectral signature. Unsupervised target detection is named Anomaly Detection and does not require any a priori information regarding the target. The definition of an anomaly is an unfamiliar and infrequent instance that notably stands out from most other instances in the same data set. HAD aims at detecting rare spectral signatures that clearly stand out from their neighboring pixels [16, 17].

1.1.3 The HYPSO mission

NTNU Small Satellite Lab is an initiative aimed at growing the space-technology environment at the NTNU. More specifically, the initiative aims at increasing the focus on the design and development of small satellites. A common type of small satellite is the cube satellite [18] constructed by a number of 10 cm x 10 cm x 10 cm cubes. The Cube satellite is a popular choice in universities seeing as they are significantly less expensive and faster to design than other alternatives.

The HYPSO mission is currently in operation at the NTNU small satellite Lab. The participants are a combination of professors, PhD-students and master's students all collaborating to achieve the following objectives as stated in [3]:

- To provide and support ocean color mapping through a Hyperspectral Imager payload, autonomously processed data, and on-demand autonomous communications in a concert of robotic agents at the Norwegian coast.
- To collect ocean color data and to detect and characterize spatial extent of algal blooms, measure primary productivity using emittance from fluorescence generating micro-organisms, and other substances resulting from aquatic habitats and pollution to support environmental monitoring, climate research and marine resource management.
- Build strong competence and strengthen the prospect of nano- and micro-satellite systems as supporting intelligent agents in integrated autonomous robotic systems dedicated to marine and maritime applications in Norway and internationally, these being applicable to communications and remote sensing (altimetry, SAR, radiometry etc).

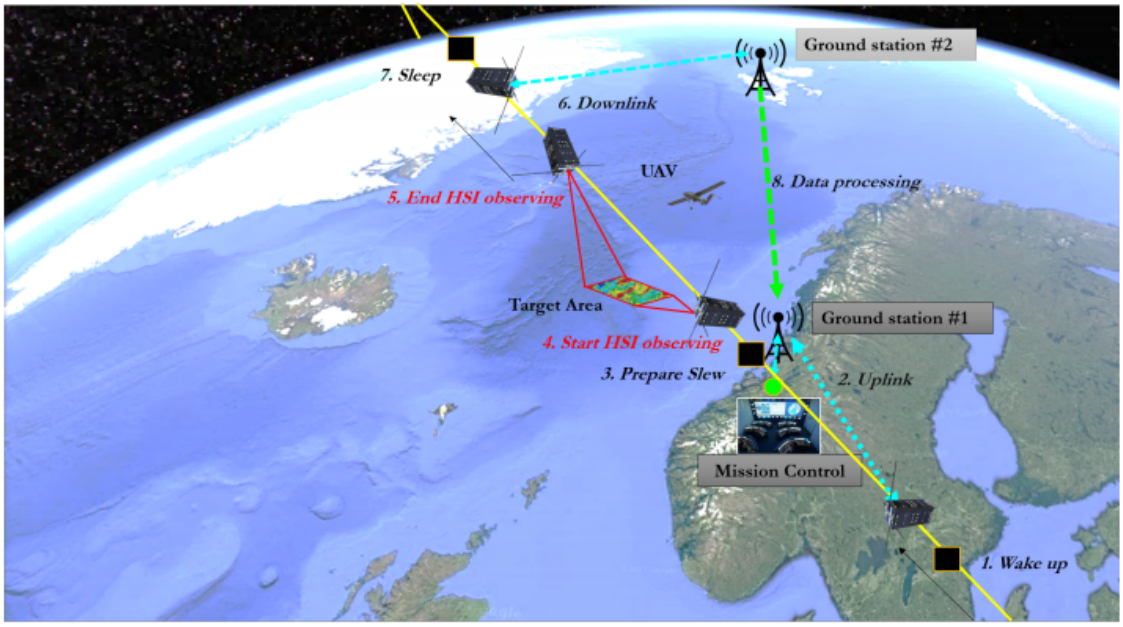


Figure 3: Conceptual illustration of the HYPSO mission operation [3].

Anomaly detection results are an essential part of the autonomously processed data that the mission aims to provide. The downlink step of the operation provides a limited data rate, and it is an advantage to autonomously find regions of interest within the HSI onboard the satellite. The regions of interest in the captured HSI can be sent to the ground station instead of the entire HSI, saving both time and power. Another possible use is to perform supervised target detection on the areas of interest, which is less time-consuming than performing it on the entire HSI.

1.2 Main contributions

The main objective of this thesis is to provide the HYPSO mission with an implementation of a state-of-the-art HAD algorithm. The implementation should fulfill the requirements necessary to be added to the HYPSO onboard processing pipeline. The algorithm must be evaluated and compared with baseline and state-of-the-art HAD algorithms to verify that the chosen algorithm achieves state-of-the-art results. Research questions related to this objective are as follows:

1. What are the most recent advances in hyperspectral anomaly detection?
2. How can a HAD algorithm be evaluated?
3. What are the strengths and weaknesses of the current state-of-the-art algorithms for HAD?
4. How much can the execution time of the proposed HAD algorithm be improved by hardware acceleration?

The main contributions of the thesis are listed below:

- An extensive review of known baseline and state-of-the-art HAD algorithms.
- Evaluation of a HAD algorithm adapted and improved from an existing state-of-the-art algorithm in Matlab. The algorithm is compared with 11 known baseline, and state-of-the-art HAD algorithms using 13 real HSIs.
- Software implementation in C and C++ programming languages of a simplified version of the proposed HAD algorithm.
- Acceleration in hardware of a part of the software implementation using High-Level Synthesis.
- Comparison between different optimization techniques in the High-Level Synthesis implementation.
- Verification of the implemented hardware-software partitioned HAD system on a Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit.

1.3 Outline of thesis

The thesis is organized as follows: **Chapter 2** contains background theory on HAD algorithms and technologies utilized in the implementation of the HAD algorithm proposed in this thesis. **Chapter 3** describes the methodology of the proposed HAD algorithm and how it is evaluated and compared to other HAD algorithms in Matlab. **Chapter 4** describes the implementation of the proposed HAD algorithm in C-programming language and how a part of the algorithm is accelerated in customized hardware. **Chapter 5** shows the results. **Chapter 6** gives a discussion on all aspects covered in the thesis. **Chapter 7** gives the conclusion.

2 Background

This chapter describes the theoretical aspects used in implementing and evaluating the HAD algorithm proposed in this thesis. The chapter starts by describing deep learning and autoencoders because autoencoders are the basis of the algorithm. The second part of this chapter describes what HAD is and several known HAD algorithms that are used in the evaluation of the proposed HAD. The third part of this chapter describes the concept of hardware acceleration and tools that can be utilized for hardware acceleration. The last part describes the hyperspectral datasets that are used for testing in this thesis. The background on deep learning, HAD and the datasets are adapted from the specialization thesis of the author [13] and elaborated upon.

2.1 Deep learning

Artificial Intelligence (AI) is a widely researched field in computer science that originates from the desire to mimic human intelligence using machines. The prime objective in the field of AI is to understand intelligence and how it can be employed in machines to perform valuable tasks [19].

Machine Learning (ML) is a collective expression for a range of methods related to AI that aims to enable computer applications to independently and automatically adapt to a specified task based on experience. ML methods are commonly designed to learn how to identify patterns in data and use the discovered patterns to predict subsequent data or make decisions. ML methods are categorized as either supervised or unsupervised based on the use of labeled training data. Labeled training data is input representing the input that the application will be provided with after training. The labeled training data also contains the correct outputs that the program should provide from the corresponding inputs. By using the labeled training set, the supervised methods train the application to produce the desired outputs when exposed to the different inputs. Unsupervised ML methods do not rely on a priori knowledge about the correct output of the program. The aim is instead to identify patterns that stand out as interesting based on the input data. Unsupervised ML is commonly considered a more ambiguous field than supervised ML since the correct behavior is not accurately defined [20].

Deep Learning (DL) is a subset of ML where Artificial Neural Networks (ANNs) are utilized to learn high-level features from large data sets. ANNs are logic structures that are designed roughly inspired by neuroscience [4]. The relation between AI, ML, and DL is illustrated in Figure 4.

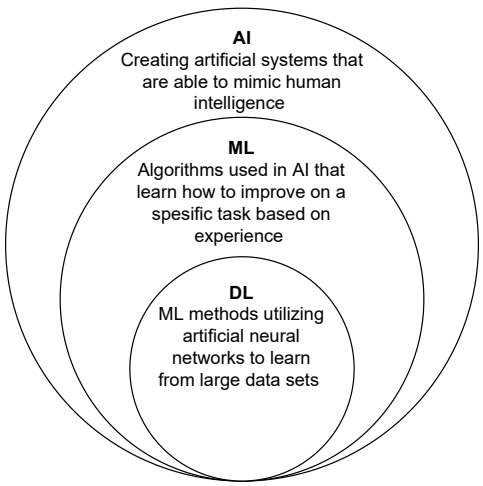


Figure 4: Diagram showing the relation between AI, ML and DL, adapted from [4].

ANNs are built up by Artificial Neurons (ANs) as shown in Figure 5.

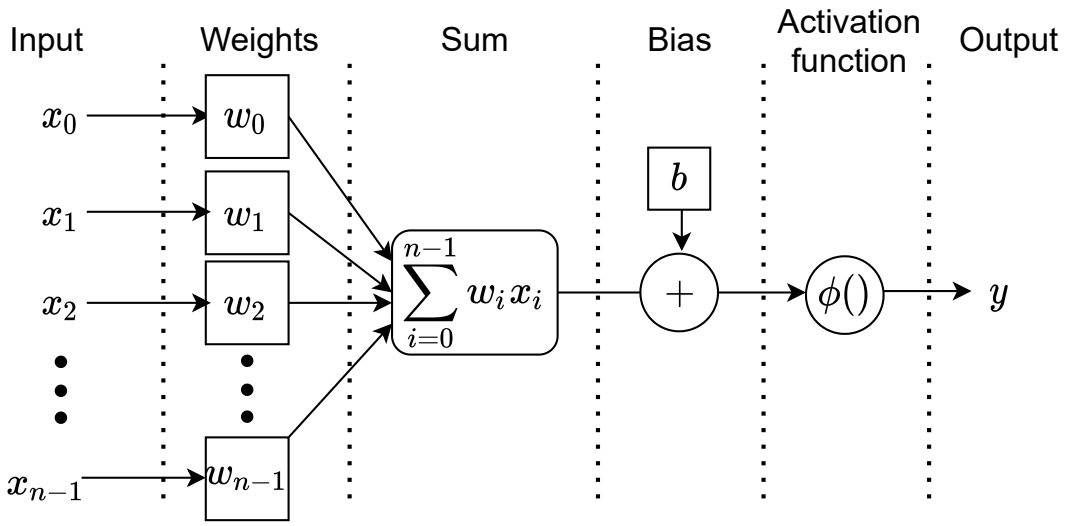


Figure 5: Illustration of an artificial neuron.

The mathematical expression for y from a single AN as illustrated in Figure 5 is given as

$$y = \phi((\sum_{i=0}^{n-1} w_i x_i) + b), \tag{1}$$

Where parameters x_i and w_i are elements i of the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}$ both of length n respectively. Parameter b is the bias value belonging to the AN and the function $\phi()$ is a non-linear activation function. The activation function plays a vital part in learning non-linear patterns of high complexity [21]. Examples of activation functions are the Sigmoid function

$$\phi(x) = \frac{1}{1 - e^{-x}}, \tag{2}$$

the Tanh function

$$\phi(x) = tanh(x), \tag{3}$$

the ReLu

$$\phi(x) = max(0, x), \tag{4}$$

and the Leaky-ReLu

$$\phi(x) = max(\alpha x, x). \tag{5}$$

Figure 6 shows how several ANs build up an ANN. In the Figure, each circle in the hidden layers represents an AN. The lines between the layers represent the weights. The word deep in DL originates from the multiple layers that the ANN is constructed by. This Figure shows a feed-forward ANN where the information moves in a single direction.

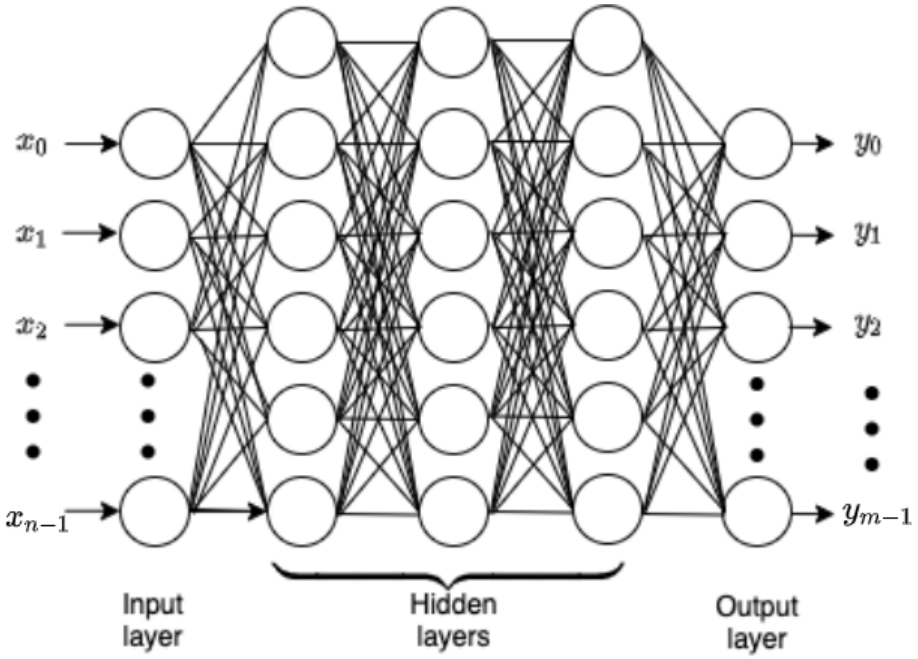


Figure 6: Illustration of an artificial neural network.

The mathematical expression for the entire ANN is derived by using matrices to represent the weights, biases, inputs, and outputs. The input and output can be of different lengths and are represented as $(1 \times n)$ and $(1 \times m)$ arrays where n is the number of inputs, and m is the number of outputs. The hidden layers are the layers between the input and output layers. n_l is the number of ANs that layer l consists of, and K is the number of hidden layers. $\mathbf{W}^{(l)}$ is a $(n_l \times n_{l+1})$ matrix containing all weights between layer l and the following layer. The equation used to obtain the first hidden layer $\mathbf{h}^{(1)}$ is

$$\mathbf{h}^{(1)} = \phi(\mathbf{x}\mathbf{W}^{(\mathbf{0})} + \mathbf{b}^{(\mathbf{0})}). \tag{6}$$

The remaining (K-1) hidden layers are obtained by

$$\mathbf{h}^{(l+1)} = \phi(\mathbf{h}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}). \tag{7}$$

The output is lastly obtained by using

$$\mathbf{y} = \phi(\mathbf{h}^{(K)}\mathbf{W}^{(K)} + \mathbf{b}^{(K)}). \tag{8}$$

2.1.1 Training

An ANN learns how to identify patterns in the input by training. The training is a process that makes minor alterations in the weights and biases in numerous iterations. The direction of change is calculated by taking the derivative of a loss function with respect to the adjustable parameters of the ANN, namely the weights and biases. The loss function is a function that aims to map the weights and biases of the ANN to a single number that represents the performance of the ANN. This method is called Gradient Descent (GD) since it utilizes the gradient of the loss function to repeatedly move in the opposite direction of the gradient to decent in the loss function [22].

GD training relies on the ability to find the gradient of the loss function. Backpropagation (BP) is

a technique that efficiently computes the gradient of a given loss function in a feed-forward ANN by utilizing the chain rule of derivatives [23]

$$\frac{\partial z(y(x))}{\partial x} = \frac{\partial z(y(x))}{\partial y} \cdot \frac{\partial y(x)}{\partial x}. \tag{9}$$

BP propagates backward one layer at a time while computing the derivatives of each layer. The following mathematical derivation of the GD and BP techniques follows the description in [6]. A commonly used loss function L that is a function of $\mathbf{W^l}$ and $\mathbf{b^l}$ for all layers l is defined as

$$L = \frac{1}{2} \sum_{i=0}^{m-1} (y_i - t_i)^2, \tag{10}$$

where parameter y_i is element i in the output vector of length m. Parameter t_i is the corresponding desired result from the labelled training set. The notations $z_j^{(l+1)} = \sum_{i=1}^{N_l} W_{i,j}^{(l)} h_i^{(l)} + b_j^{(l)}$ and $\partial_i^{(l)} = \frac{\partial L}{\partial z_i^{(l)}}$ are used in the following derivation to increase readability. Using this notation and equations 6, 7 and 8 for the propagation in ANNs we can express element j in hidden layer l as

$$h_j^{(l)} = \phi(z_j^{(l)}). \tag{11}$$

To calculate the gradient of L for layer l we need the two derivatives

$$\frac{\partial z_j^{(l+1)}}{\partial W_{i,j}^{(l)}} = h_i^{(l)} \tag{12}$$

and

$$\frac{\partial z_j^{(l+1)}}{\partial b_j^{(l)}} = 1. \tag{13}$$

If the activation function is chosen to be the Sigmoid (equation 2), the derivative of $\phi()$ is

$$\frac{\partial \phi(z_j^{(l)})}{\partial z_j^{(l)}} = (1 - \phi(z_j^{(l)}))\phi(z_j^{(l)}). \tag{14}$$

The BP algorithm starts at the last layer K by computing

$$\partial_i^{(K)} = (y_i - t_i) \frac{\partial \phi(z_i^{(K)})}{\partial z_i^{(K)}}, \tag{15}$$

before propagating in a backward fashion while computing

$$\partial_i^{(l)} = \frac{\partial \phi(z_i^{(l)})}{\partial z_i^{(l)}} \cdot (\sum_{j=1}^{N_{l+1}} W_{i,j}^{(l)} \partial_j^{(l+1)})) \tag{16}$$

for each layer l. When the algorithm has propagated through all the layers of the ANN, the gradients for each neuron in each layer can be calculated by

$$\frac{\partial L}{\partial W_{i,j}^{(l)}} = h_j^{(l)} \cdot \partial_i^{(l+1)} \tag{17}$$

and

$$\frac{\partial L}{\partial b_i^{(l)}} = \partial_i^{(l+1)}. \tag{18}$$

BP obtains these gradients for each iteration of the GD algorithm. For each iteration up to the total number of iterations T, the gradients are used to update the weights and biases with

$$W_{i,j}^{(l)} = W_{i,j}^{(l)} - \epsilon \frac{\partial L}{\partial W_{i,j}^{(l)}} \tag{19}$$

and

$$b_i^{(l)} = b_i^{(l)} - \epsilon \frac{\partial L}{\partial b_i^{(l)}}. \tag{20}$$

The parameter ϵ is called the step ratio and influences the amount of change per iteration.

2.1.2 Autoencoders

An autoencoder (AE) is a type of ANN where the objective is to compress and decompress the input in an efficient manner with minimal loss of information. The overall behavior of an AE is to encode the input to a code representation and then decode the code representation to an output as close to the original input as possible. The most widely used application is dimensionality reduction, where the code representation is in a lower dimension than the in- and output. An AE can be used as an unsupervised ANN where it uses the inputs as desired outputs and thus does not need labeled training data to learn and improve on the task it is performing [4]. Figure 7 shows how an AE is composed of two parts, namely the encoder and the decoder. The layer located in between the two parts is called the code layer and contains the code representation of the input.

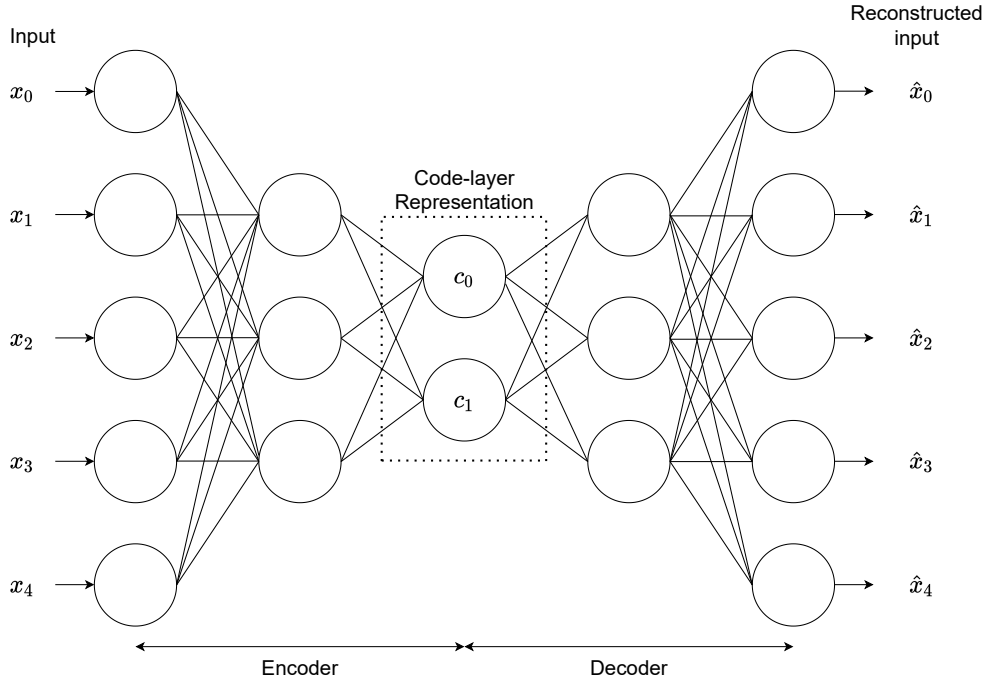


Figure 7: Illustration of an autoencoder, adapted from [5]

The loss function for a regular AE is

$$L = \frac{1}{2}\sum_{i=1}^{n}(\hat{x}_i - x_i)^2, \tag{21}$$

where parameters $\hat{x}_i$ and x_i are the i-th elements in the output and input, respectively. Multiple modifications of the AE emphasize different aspects during training. Sparse AEs (SAEs) emphasize sparsity in training, meaning that most neurons in a layer or network are inactive (output sat to zero). Makhzani and Frey state that emphasizing sparsity can lead to better performance for AEs on classification-related problems in [24]. SAEs include a training criterion $\mathbf{\Omega}(\mathbf{c})$ for the code layer $\mathbf{c}$ in the loss function. Multiple implementations of $\mathbf{\Omega}(\mathbf{c})$ have been proposed in the past. A commonly used implementation can be found in [25]. The loss function for a SAE is then

$$L_{sparse} = \frac{1}{2}\sum_{i=1}^{n}(\hat{x}_i - x_i)^2 + \mathbf{\Omega}(\mathbf{c}). \tag{22}$$

Denoising AEs (DAEs) use the original loss function instead of adding a criterion. The difference is that DAEs add random noise to the input and penalize the difference between the original input and the output computed with the noisy input. The random noise forces the AE to become more robust during training and find the most critical structures in the input [4, 26].

2.2 Hyperspectral anomaly detection algorithms

The desire to detect small, peculiar objects or phenomena in large geographical scenes has led to many HAD algorithms being proposed over the recent decades. The HAD algorithms vary significantly in methodology, but all share the following objectives [16]:

- Increasing the percentage of rightly detected anomalies while decreasing the percentage of falsely detected anomalies.
- Low computational cost since real missions often demand proximity to real-time processing of the pixels as they are captured.
- High robustness to changing conditions in the HSI and the varying nature of anomalies.

The percentage of falsely detected anomalies, commonly called false alarm rate, is sensitive to noise in the HSI, complex geographical scenery, borders between materials, et cetera. The desire for near real-time processing originates from time-sensitive applications such as wildfire detection, war-zone detection, and other cases where detection requires an immediate response. An anomaly is unknown, which also applies to the size and shape of the anomaly. The size of a true anomaly can range from sub-pixel to multiple pixels resulting in the need for algorithms that are robust in regards to detecting anomalies of varying sizes and shapes [16].

The most frequently utilized HAD algorithms are based on statistical or geometrical modeling [27]. Statistical modeling methods model the background of the HSI with known Probability Density Functions (PDFs). The parameters of the chosen PDF are estimated using the data from the HSI before the algorithm searches for spectral signatures in the HSI that significantly deviate from the estimated model. Most statistical methods utilize a model PDF related to the Gaussian distribution. Real HSIs tend to fit the model PDFs poorly, leading to high false alarm rates [17]. Contamination from anomalous pixels in the estimated model is a challenge for statistical methods. When no a priori knowledge of the target spectral signature exists, the whole HSI, including the anomalies, contributes to estimating the background statistics. The most widely used statistical HAD algorithm is the Reed-Xiaoli Detector (RX) proposed in 1990 [28]. RX models the background with a multivariate Gaussian distribution before searching for spectral signatures that deviate from the model. RX is considered the benchmark HAD algorithm and is often used to compare and

evaluate novel approaches. After 1990, many variations of the RX optimized for one or more of the objectives listed above has been proposed. Geometrical methods model the background by extracting or computing rudimentary spectral signatures or bases from the HSI. The assumption is that all background pixels can be represented by these spectral components, while anomalous pixels cannot. The Signal Sub-Space Processing Detector [29] is a widely used geometrical HAD method. It utilizes linear transformations called Singular Value Decomposition [30] to obtain a set of bases that represent the background. The spectral signatures of the HSI are projected down on the subspace created from the obtained bases, and the assumption is that this will separate the background pixels from the anomalous pixels. HAD algorithms can be split into local and global methods. Global HAD algorithms use all pixels in the HSI to compute the anomaly score for the Pixel Under Test (PUT), while local HAD algorithms only utilize surrounding pixels. A widely utilized tool that enables HAD algorithms to locate local fractions of HSIs is the Double Sliding Window (DSW).

2.2.1 Double sliding window

Figure 8 illustrates the composition of a DSW, where w_{inner} and w_{outer} are the widths of the two squares.

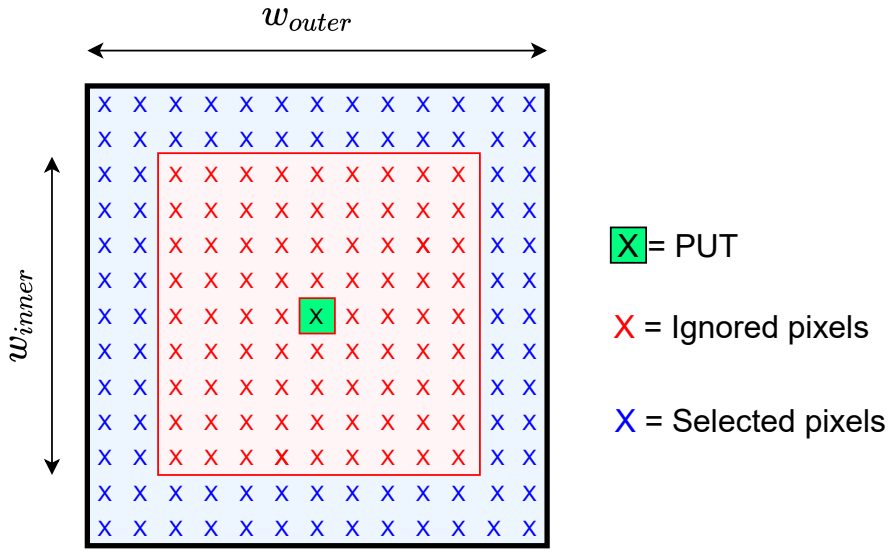


Figure 8: Double sliding window, adapted from [6]

The PUT is the center of two squares of different sizes. The pixels inside the smallest square are not used in computations because of an assumption that anomalies can be of approximately the same size as the inner square. The PUT is rather compared with the pixels that are contained between the two square windows. The window slides over all pixels in the HSI such that each PUT is compared with its locally surrounding pixels and given an anomaly score representing the calculated probability that the pixel is anomalous. By changing the size of both windows, it is possible to control the number of pixels included in computing the anomaly score for the PUT.

2.2.2 Dimensionality reduction

Dimensionality reduction is a method that is widely used in the field of hyperspectral image processing. The objective of dimensionality reduction in hyperspectral image processing is to reduce spectral bands in the HSI cube while preserving as much of the information as possible. Dimensionality reduction reduces the size of the HSI cube, which reduces memory utilization. Dimensionality reduction also reduces the computational complexity of processing the HSI cube by reducing the number of bands. HSIs can consist of hundreds of spectral bands per pixel,

where a significant part of them does not carry relevant information for anomaly detection. The two dimensionality reduction methods, Principal Component Analysis (PCA) [31] and Maximum Noise Factoring [32] are among the most utilized dimensionality reduction methods for HAD.

2.2.3 Baseline and state-of-the-art hyperspectral anomaly detection algorithms

HAD is an emerging field of study, and there are many publications in recent years proposing new methods. Current state-of-the-art approaches to HAD include machine learning algorithms, variants of the RX algorithm, morphological attribute filters, and collaborative representation. The algorithms described in this section are widely cited as either baseline or state-of-the-art HAD algorithms and used for comparison in very recent literature.

The original Global RX Detector (GRX) was proposed by Reed and Yu in 1990 [28]. GRX is a binary hypothesis test where each pixel in the HSI is either a background pixel or an anomalous pixel. The pixels are assumed to be distributed in a multivariate fashion with

$$PDF(\mathbf{x}) = \frac{e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)}}{\sqrt{(2\pi)^n |\mathbf{\Sigma}|}}, \tag{23}$$

where n is the number of bands in pixel $\mathbf{x}$. μ is the mean of the pixel vectors estimated by all K pixels in the HSI with

$$\mu = \frac{\sum_{i=0}^{K-1} \mathbf{x_i}}{K}. \tag{24}$$

The parameter $\mathbf{\Sigma}$ is the covariance matrix estimated from all K pixels in the HSI. The GRX uses Mahalanobis distance [33] together with the estimated parameters of the multivariate Gaussian distribution [20] to obtain the detector

$$D_{GRX}(\mathbf{x}) = (\mathbf{x} - \mu)^T \mathbf{\Sigma^{-1}} (\mathbf{x} - \mu). \tag{25}$$

The Local RX (LRX) utilizes a DSW as described in section 2.2.1 to estimate the covariance and mean locally around the PUT, instead of for the entire HSI. This increases the latency significantly since the parameters must be estimated again for each pixel. The detector is given as

$$D_{LRX}(\mathbf{x}) = (\mathbf{x} - \mu_{\mathbf{local}})^T \mathbf{\Sigma^{-1}_{local}} (\mathbf{x} - \mu_{\mathbf{local}}). \tag{26}$$

The Dual Window RX (DWRX) [34] uses the DSW differently than the LRX. The pixels inside the inner window are used to calculate a mean μ_{inner} that replaces the PUT in the detector instead of disregarding it. This leads to the detector

$$D_{DWRX}(\mathbf{x}) = (\mu_{\mathbf{inner}} - \mu_{\mathbf{local}})^T \mathbf{\Sigma^{-1}_{local}} (\mu_{\mathbf{inner}} - \mu_{\mathbf{local}}). \tag{27}$$

The Causal RX (CRX) [35] is a HAD algorithm motivated by the desire to achieve real-time processing of the pixels as they are captured. The other RXs all rely on computing the μ and $\mathbf{\Sigma}$ making real-time implementation impossible. To compute the anomaly score for pixel k, the CRX utilizes the sample correlation matrix defined as

$$\mathbf{R}(\mathbf{x_k}) = \frac{\sum_{i=1}^{k} \mathbf{x_i} \mathbf{x_i^T}}{k}, \tag{28}$$

to create the detector

$$D_{CRX}(\mathbf{x_k}) = \mathbf{x_k^T R(x_k)_{local}^{-1} x_k}. \tag{29}$$

The Kernel RX (KRX) [36] is a non-linear implementation of the RX. KRX maps the pixel vectors to a feature-space of higher dimension by utilizing a non-linear function $\theta()$. Kernel functions $k()$ allow for efficient calculation of the dot-product without the need to identify the mapping function in the following manner:

$$k(\mathbf{x_i}, \mathbf{x_j}) = \langle \theta(\mathbf{x_i})\theta(\mathbf{x_j}) \rangle = \theta(\mathbf{x_i}) \cdot \theta(\mathbf{x_j}) \tag{30}$$

A widely used $k()$ for HAD is the radial basis function kernel [34]

$$k(\mathbf{x_i}, \mathbf{x_j}) = e^{\frac{-||\mathbf{x_i}-\mathbf{x_j}||^2}{2\sigma^2}}. \tag{31}$$

The Fractional Fourier Entropy RX (FrFE-RX) aims at reducing noise and increasing the difference between anomalous, and background pixels [37]. The algorithm starts by extracting features with fractional Fourier transform before applying the RX on the extracted features. This approach leads to increased performance but also significantly higher latency due to the required pre-processing.

Collaborative Representation Detector

Li and Du proposed the Collaborative Representation Detector (CRD) in [38]. The method relies on the assumption that it is possible for pixels belonging to the background class to be roughly represented by pixels surrounding it spatially. At the same time, it is impossible for anomalous pixels. A significant drawback for this method is its low flexibility seeing as the number of surrounding pixels to include in the computation must be pre-defined. Another drawback is that, like the RXs, the anomalous pixels contaminate the calculations when they are included in the surrounding area leading to higher false alarm rates.

For a pixel, $\mathbf{x}$ the surrounding pixels are found with a DSW as described in section 2.2.1. $\mathbf{X_s}$ is a matrix containing all the surrounding pixels, and α is an array of weights. The Lagrange multiplier is denoted as λ. The CRD algorithms objective is to compute α in a way that minimizes

$$\underset{\alpha}{argmin} ||\mathbf{x} - \mathbf{X_s}\alpha||_2^2 + \lambda||\alpha||_2^2. \tag{32}$$

The solution to the equation above, as described in [38], is

$$\hat{\alpha} = (\mathbf{X_s^T X_s} + \lambda \mathbf{I})^{-1} \mathbf{X_s^T x}, \tag{33}$$

and the detector is then

$$D_{CRD} = ||\mathbf{x} - \mathbf{X_s}\hat{\alpha}||_2. \tag{34}$$

Li and Du also elaborate on the above detector by further optimizing the expression for obtaining $\hat{\alpha}$. Please refer to [38] for more details on the optimized CRD algorithm.

The Spatial Density Background Purification (SDBP) Detector described in [39] aims at reducing anomaly contamination. The algorithm is pre-processing the HSI with a density peak clustering algorithm before employing the CRD algorithm. This pre-processing step is reported to increase both the accuracy and latency of the CRD.

Attribute filters

Attribute filters have in recent years emerged as a technique for HAD. Attribute filters refer to adaptive morphological filters and were first introduced in [40]. They can be utilized in HAD to exploit the spatial relationship between pixels belonging to the same class. Before the filtering, connected components of the HSI built up by pixels connected by a given connectivity rule must be found as described in [41]. The attribute filter then decides to either keep or remove each of the connected components based on a size predicate $T^{Size}_{Threshold}$. If the size of the connected component is larger than the threshold, the predicate is true, and if the size is smaller than the threshold, the predicate is false. Based on the assumption that anomalies are small, the anomalies are extracted from the HSI by the size of the connected components. The size attribute is the one most used in HAD algorithms, but other attributes are available.

The Attribute and Edge-preserving filters Detector (AED) proposed in [27] starts by pre-processing the HSI cube to find the principal component in the spectral dimension and uses this to represent the HSI as a grey-scale image. The anomalies are separated from the background using an attribute filter with a pre-defined size threshold. After the separation of the two classes, an edge-preserving filter described in [42] is used for post-processing to increase the detection accuracy by removing incorrectly classified anomalous pixels. This HAD provides high accuracy and low execution time, but there are several drawbacks. The AED is highly sensitive to wrong thresholds for the size predicate since too high thresholds lead to background pixels being wrongly classified as parts of the anomalies. Low thresholds lead to larger anomalies being wrongly classified as background.

The Morphological Profile and Attribute Filters Detector (MPAF) proposed by Andika et al. and described in [43] reduce the dimension in the spectral domain by using a novel approach to improve the execution time. After selecting one spectral band, morphological profiles as described in [41] are used to separate the anomalies from the background before the anomalies are filtered with an attribute filter to remove pixels wrongly classified as anomalies. Andika et al. also describes an algorithm to calculate a suiting threshold for the attribute filters to improve the drawback of the AED.

Autoencoder-based HAD algorithms

AE-based HAD algorithms exploit how an AE learns to compress and decompress input when trained, combined with the assumption that anomalies are rare occurrences. When an AE is configured to compress and decompress a HSI pixel-wise after being trained on that or a similar set of HSIs, the concept is that it has been thoroughly exposed to background pixels but only negligibly exposed to anomalous pixels during training. This leads to significantly higher reconstruction errors for the Anomalous pixels than the background pixels. Figure 9 illustrates the use of an AE in the context of hyperspectral anomaly detection.

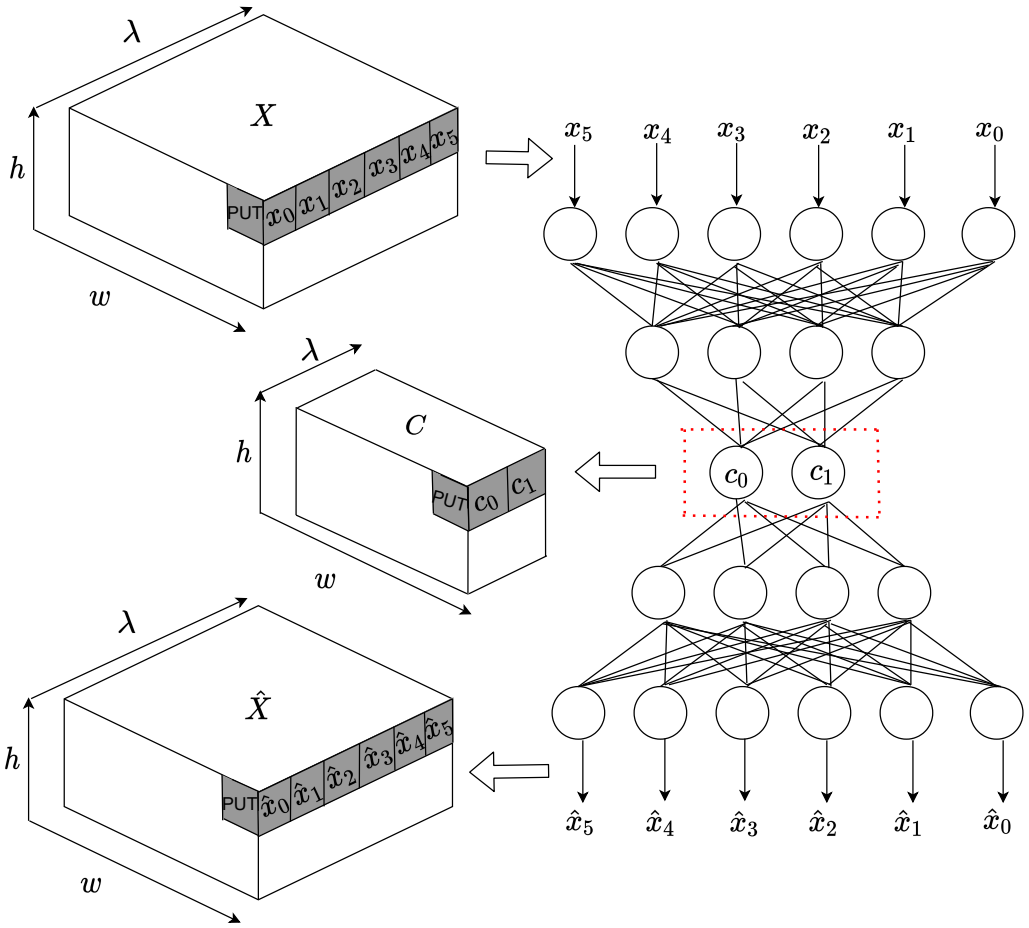


Figure 9: Illustration of hyperspectral anomaly detection with an autoencoder for a HSI with 5 spectral bands, adapted from [1].

The difference r between the original pixel $\mathbf{x}$ and the reconstructed representation of the original pixel $\hat{\mathbf{x}}$ is calculated as the root mean square error (RMSE) and used as the anomaly score. The expression to compute r for the PUT is

$$r = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(\hat{x}_i - x_i)^2}. \tag{35}$$

An advantage of using an AE is that while statistical models strive to estimate the parameters of a PDF that most likely is a sub-optimal fit, an autoencoder learns to represent the background pixels through training. The autoencoder-based HAD algorithms provide highly competitive results in both accuracy and execution time, thus leading to a range of different methods being proposed since it was first introduced in the field of hyperspectral anomaly detection. The drawbacks of the described method are that it does not utilize spatial information and that many parameters can be hard to choose optimally. Bati et al. published a paper describing a HAD algorithm utilizing a sparse AE in 2015 [44]. Chang et al. proposed a HAD algorithm utilizing a sparse AE combined with a DSW to perform the anomaly detection in [45]. This approach used the DSW to train the AE locally and thus incorporating spatial information in the HAD algorithm. Denoising AEs are used combined with PCA in the HAD algorithm described in [26]. The HSI is initially altered by reducing the dimension of the spectral domain with PCA before it is exposed to whitening as a part of the pre-processing. The resulting pixels are fed to an AE to learn the complex features of the HSI, which are then used as input to a GRX that computes the anomaly score.

2.2.4 Deep belief network autoencoder algorithms

Ma et al. describe a HAD algorithm that uses a Deep Belief Network (DBN) AE in [7]. A DBN is an ANN with a distinctive structure consisting of layers of ANs that can be considered a stack of Restricted Boltzmann Machines (RBMs). RBMs are ANNs consisting of two layers, namely the visual and the hidden layer, formed as a complete bipartite undirected graph as illustrated in Figure 10.

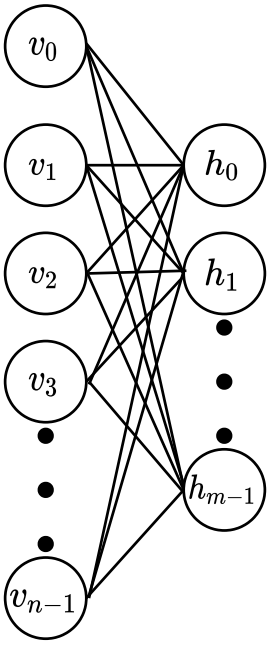


Figure 10: Illustration of an RBM, adapted from [7].

DBNs have been a popular choice when designing ANNs consisting of multiple layers since Hinton proposed a method for efficiently training each layer as a RBM before training the entire DBN together in [46]. This method is called Contrastive Divergence (CD) and lets the hidden neurons of the RBM learn the essential patterns in the data that is fed to its respective visual layer during this pre-training. The first RBM uses the input as its visible layer and has one hidden layer, as shown in the Figure above. The following RBM uses the hidden layer from the previous layer as its visible layer such that a N-layer DBN consists of $(N-1)$ RBMs where the hidden layer of the $(N-1)$-th RBM is the output of the DBN. Figure 11 illustrates how a DBN autoencoder with three layers is constructed using two RBMs.

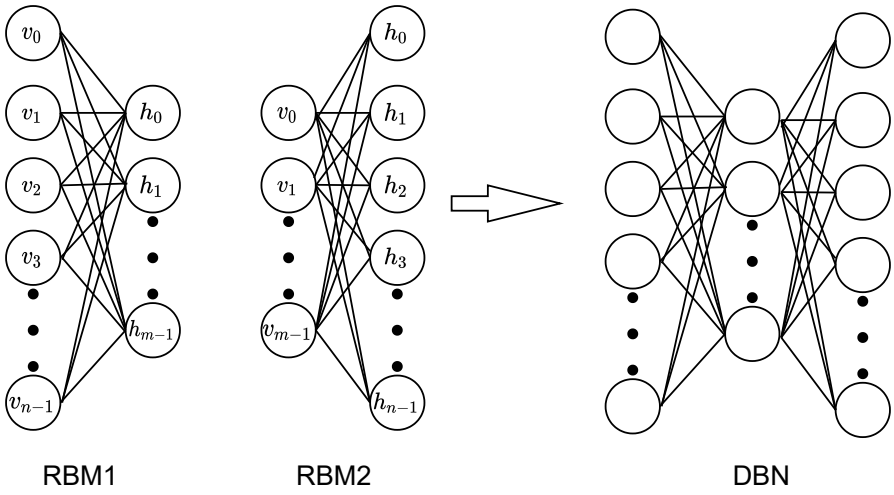


Figure 11: Illustration of an DBN as a stack of RBMs, adapted from [7].

The below theoretical description of RBMs and derivation of CD pre-training follows the descriptions in [47] and [48]. They are included here because the HAD algorithm implementation described in this thesis utilizes a DBN AE that is pre-trained using CD before it is fine-tuned with GD and BP. The theoretical description considers the case where the hidden and visual neurons of the RBM are binary values. However, the resulting training algorithm is shown to be efficient for continuous-valued inputs between 0, and 1 [47]. The energy function of a joint configuration of both layers $(\mathbf{v}, \mathbf{h}) \in \{0,1\}^{n+m}$, where $\mathbf{v}$ is a vector with the values of the n visible neurons and $\mathbf{h}$ is a vector with the values of the m hidden neurons, is described in [49] as

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i=0}^{n-1} a_i v_i - \sum_{j=0}^{m-1} b_j h_j - \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} v_i h_j w_{i,j}, \tag{36}$$

where parameter $w_{i,j}$ is the weight between v_i and h_j. The parameter a_i is the bias value for the i-th visible neurons and b_j for the j-th hidden neuron. All existing pairs of vectors for the two layers are assigned the probability

$$p(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}}, \tag{37}$$

from the network given by the Gibbs distribution [47]. Further, the probability assigned to a specific visual vector $\mathbf{v}$ is computed by taking the sum of the probabilities for all existing hidden vectors. The probability is

$$p(\mathbf{v}) = \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}}, \tag{38}$$

where the vector $\mathbf{v}$ is the training input. The probability $p(\mathbf{v})$ can be increased by altering the weights and biases of the network. A technique to determine the direction of change for the variables is to compute the gradient of the logarithm of the probability. The derivatives presented in [47] by Fischer and Igel is

$$\frac{\partial log(p(\mathbf{v}))}{\partial w_{i,j}} = P(h_j = 1|\mathbf{v})v_i - \sum_{\mathbf{v}} p(\mathbf{v})P(h_j = 1|\mathbf{v})v_i \tag{39}$$

$$\frac{\partial log(p(\mathbf{v}))}{\partial b_j} = P(h_j = 1|\mathbf{v}) - \sum_{\mathbf{v}} p(\mathbf{v})P(h_j = 1|\mathbf{v}), \tag{40}$$

and

$$\frac{\partial log(p(\mathbf{v}))}{\partial a_i} = v_i - \sum_{\mathbf{v}} p(\mathbf{v}) v_i. \tag{41}$$

The probabilities are defined as

$$P(h_j = 1|\mathbf{v}) = \phi(b_j + \sum_{i=1}^{N_{visible}} w_{i,j} v_i) \tag{42}$$

and

$$P(v_i = 1|\mathbf{h}) = \phi(a_i + \sum_{j=1}^{N_{hidden}} h_i w_{i,j}). \tag{43}$$

Contrastive Divergence algorithm

The first terms of the three gradients in equations 39, 40 and 41 are possible to obtain efficiently. The last terms of all three gradients of the logarithm of the probability given in equations 39, 40 and 41 are computationally expensive to calculate [47]. The CD algorithm efficiently updates the parameters by using an approximation of these gradients obtained by only performing Gibbs sampling (described in chapter 17.4 in [4]) k times instead of finding the actual gradients. For a training vector, $\mathbf{v^0}$ all the binary values for the hidden neurons are obtained in parallel by setting h_j^0 to 0 if the probability in equation 42 is smaller than a uniformly distributed random number and setting h_j^0 to 1 if it is larger. A reconstructed version of the visual vector $\mathbf{v^1}$ is obtained in a similar fashion using the probability in equation 43 and the calculated hidden vector. Lastly, the reconstructed $\mathbf{v^1}$ is used to compute a reconstructed $\mathbf{h^1}$ in the same way as before. This procedure can be repeated k times, so the CD algorithm is often denoted CD_k. The most used version is the CD_1 which is illustrated in Figure 12.

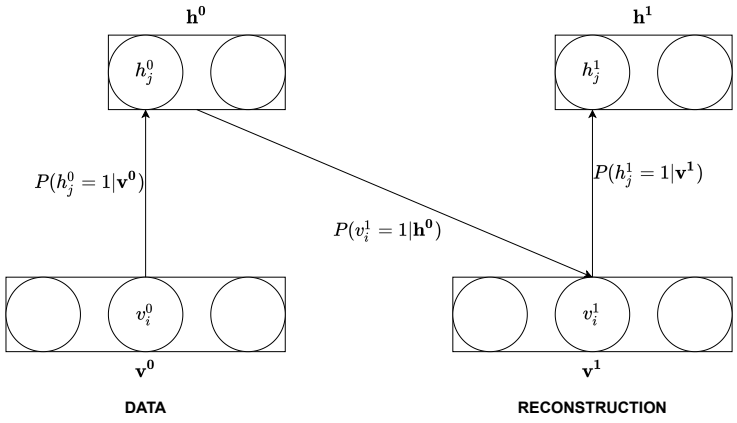


Figure 12: One step of Gibbs sampling to obtain reconstructed representation of visual and hidden vectors, adapted from [8].

With this approximation the learning rules used in CD_1 training as described in [47] are

$$\Delta w_{i,j} = \epsilon(P(h_j = 1|\mathbf{v^0}) v_i^0 - P(h_j = 1|\mathbf{v^1}) v_i^1), \tag{44}$$

$$\Delta a_i = \epsilon(v_i^0 - v_i^1) \tag{45}$$

and

$$\Delta b_j = \epsilon(P(h_j = 1|\mathbf{v^0}) - P(h_j = 1|\mathbf{v^1})). \tag{46}$$

The parameter ϵ is the step-ratio used to decide how much the parameters are changed per iteration.

Adaptive weights DBN AE

In [1] Ma et al. elaborated on their proposed DBN HAD algorithm by incorporating spatial information of the HSI with an adaptive weights strategy to improve the accuracy. The HAD algorithm is named Adaptive Weights Deep Belief Network Detector (AWDBN) and is the origin of the HAD algorithm proposed in this thesis. The following details from the AWDBN are based on the descriptions in [7] and [1]. For a HSI $\mathbf{X} \in \mathbb{R}^{h \times w \times n_b}$ with spatial height h, spatial width w and n_b spectral bands a N-layered DBN AE is created. After initialization, the (N-1) RBMs in the DBN are trained with the $(w \times h)$ pixels in $\mathbf{X}$ in a sequential manner using the highly efficient CD_1 algorithm to find a good starting point for fine-tuning the DBN AE. After the pre-training, the DBN is fine-tuned by regular training with GD and BP, also using the pixels of $\mathbf{X}$ as training data. The HSI is then encoded and decoded pixel-wise with the DBN to create both a code-layer representation of the HSI $\mathbf{C}$ of spectral dimension n_c and a reconstructed representation $\hat{\mathbf{X}}$ of spectral dimension n_b. The reconstruction error $\mathbf{R} \in \mathbb{R}^{h \times w}$ is calculated as the pixel-wise RMSE between $\mathbf{X}$ and $\hat{\mathbf{X}}$.

The adaptive weights strategy utilizes a DSW to locate k neighboring pixels in the code-layer representation $\{\mathbf{c_0^n}, \mathbf{c_1^n}, ..., \mathbf{c_{k-1}^n}\}$ for the PUT. The distance between each of these k neighboring pixels and the PUT $\mathbf{c^p}$ are computed in the code representation by

$$d_j = \sqrt{\sum_{i=0}^{n_c-1} |\mathbf{c_j^n}(i) - \mathbf{c^p}(i)|^2}. \tag{47}$$

The k distances are then weighted by

$$wt_j = \begin{cases} \frac{1}{r_j^n}, & \text{if } (r_j^n - \mu_r^n) < \sigma_r^n \\ \frac{pf}{r_j^n}, & \text{otherwise,} \end{cases} \tag{48}$$

where r_j^n is the reconstruction error for the j-th neighboring pixel. The parameter r_j^n is obtained from $\mathbf{R}$ with the same DSW that was utilized to find the neighboring pixels in the code-layer representation. The parameter pf is a penalty factor with a chosen value between 0 and 1, and μ_r^n is the mean, and σ_r^n is the standard deviation of the neighboring reconstruction errors. The weights and distances combine to provide the anomaly score β of the AWDBN as

$$\beta = \frac{1}{k} \sum_{j=0}^{k-1} wt_j d_j. \tag{49}$$

The algorithm is as follows:

1. Create the DBN and perform CD_1 pre-training and then fine-tuning with GD and BP.
2. Encode $\mathbf{X}$ to $\mathbf{C}$.
3. Decode $\mathbf{C}$ to $\hat{\mathbf{X}}$.
4. Compute $\mathbf{R}$ with $\mathbf{X}$ and $\hat{\mathbf{X}}$.

5. Use a DSW to select k neighboring pixels
6. Calculate d_j with $\mathbf{c^p}$ and $\mathbf{c_j^n}$ for the k neighbors
7. Calculate wt_j with r_j^n and pf for the k neighbors
8. Compute β for the PUT with $\{wt_0, wt_1, ..., wt_{k-1}\}$ and $\{d_0, d_1, ..., d_{k-1}\}$
9. Perform steps 5 - 8 for every pixel in the HSI.

2.3 Hardware acceleration

Modern computational applications are mostly implemented and tested initially using Software (SW) because it allows for rapid and flexible prototyping. SW implementations utilize programming languages with a high level of abstraction that can run on a wide range of different general-purpose processors. The most common general-purpose processor is the Central Processing Unit (CPU) which can be implemented in many different manners. When the application behaves as desired, the designer can improve aspects such as speed, efficiency, and power consumption by optimizing the SW, utilizing customized Hardware (HW), or combining the two (HW-SW codesign). When utilizing customized HW, a high level of abstraction and flexibility is traded for lower latency and power consumption. HW specialized for one specific task will outperform general-purpose HW due to reduced overhead, and the use of specialized HW to reduce latency is referred to as HW acceleration. Application-Specific Integrated Circuits (ASICs) are completely customized HW designed for one specific behavior. ASICs can execute computational tasks very efficiently but are complex to design and expensive to produce. Furthermore, if an error in the behavior is detected after the ASICs are produced, it is very hard or impossible to correct it without re-fabricating the chip. An option that offers similar performance opportunities and significantly higher flexibility is Field-Programmable Gate Arrays (FPGAs) [9].

Field-programmable gate array

FPGAs are integrated circuits that can be re-programmed many times to change their electrical behavior after being manufactured. This allows for more flexible and rapid implementation and testing than ASICs while maintaining most benefits concerning performance. FPGAs are programmed using Hardware Descriptive Languages (HDLs), where VHDL and Verilog are the most used. HDLs can describe the behavior of digital logic in three different manners, namely structural, behavioral, and Register-Transfer Level (RTL), which is the modeling of the flow of signals between registers. A FPGA is constructed of the five basic elements in the list below [9]:

- **Multiplexers**
  - Multiplexers are logical constructs that output one of 2^N input bits based on N separate selection input bits.
- **Flip-Flops (FFs)**
  - FFs are elementary memory elements that can store one bit of information per FF between clock cycles.
- **Look-Up Tables (LUTs)**
  - LUTs are simple electrical constructs where the desired output of a Boolean logic function corresponding to different inputs is stored. The LUTs use Muxes to select which stored values to access, meaning that for N input bits, 2^N elements stored in the LUTs can be accessed. This is an efficient implementation of a simple logical function where the solution is referenced rather than computed.
- **Interconnected wire matrix**

– When the FPGA is re-configured to have different behavior, the flow of signals between computational logic needs to change. Using a matrix of interconnected wires, the route of the signals in the FPGA can be changed to fit the new design.

- **Input/Output (I/O)**
  – Connections used to send data to and from the FPGA.

Multiplexers, LUTs and FFs are used to construct more complex blocks called Configurable Logic Blocks (CLBs). The CLBs are connected with the wire matrix to form complex logic circuits that can be configured to provide the programmed behavior. Block Random Access Memory (BRAM) is a memory construction commonly used in modern FPGAs to store data on the FPGA and transfer data to and from the FPGA. Figure 13 illustrates the internal architecture of a common FPGA.

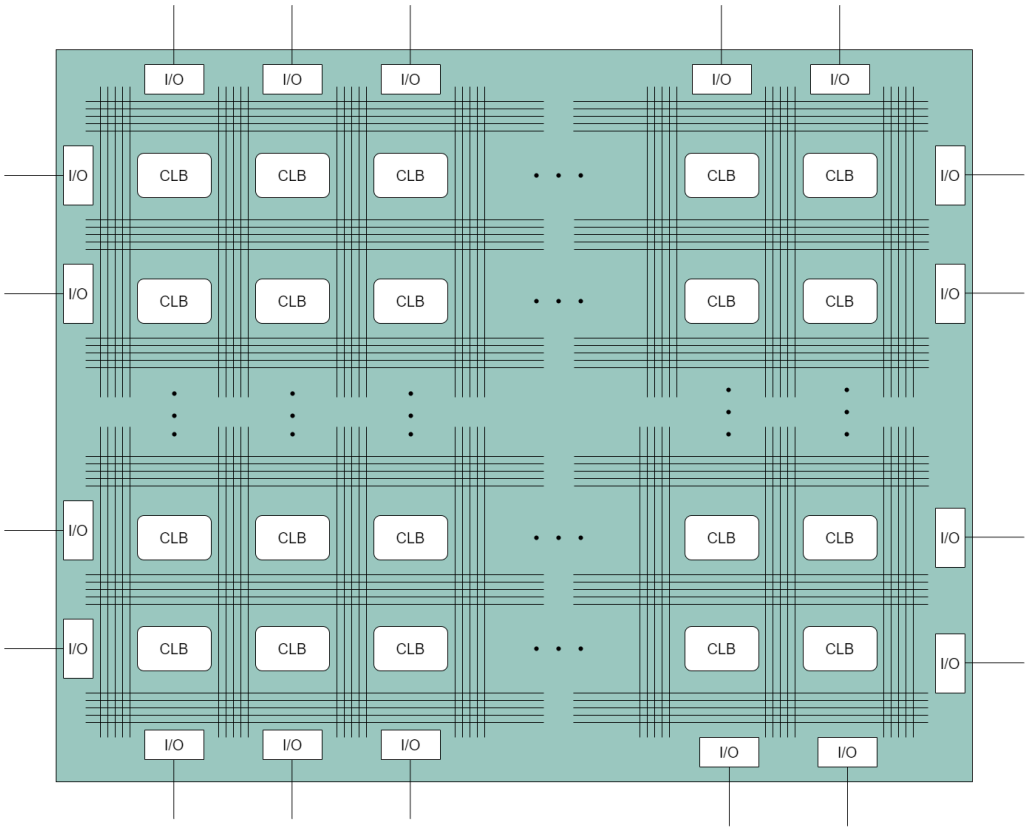


Figure 13: Illustration of the structure of a FPGA, adapted from [9].

2.3.1 Fixed-point representation

When using floating-point representation to represent variables, the decimal point can be placed everywhere within the number of bits used. This means that for large numbers, more bits would be used to represent the integer part, while for small numbers, more bits can be used to describe the fractional part. Fixed-point differ from floating-point by having a specified location to place the decimal point. This means that the number of bits representing the integer part and the fractional part is decided, resulting in less complex math operations. Finnerty and Ratigner states that the use of fixed-point representation in customized HW can reduce FPGA resource utilization, reduce power consumption, and improve the latency [50]. Another positive aspect of fixed-point is that the number of total bits used is flexible. Therefore, the number of bits can be reduced to reduce the utilization of HW resources.

2.3.2 High-level synthesis

High-Level Languages (HLLs) such as Python, C, C++, et cetera offer a high level of abstraction from binary machine language. HLLs are easier to understand because they use syntax close to human language and let the designer write algorithmic descriptions without regard to how the machine will execute it. These aspects allow for faster development of complex applications due to their more intuitive and understandable nature than low-level languages that require more implicit programming of the execution in the machine. Because of its low level of abstraction and high complexity, the design of large systems using HDLs is very time-consuming and requires advanced engineering skills. High-Level Synthesis (HLS) is a process that allows a designer to create an application in a HLL and automatically translate it to a RTL description in HDL. This combines the rapid and flexible development from HLL and the high efficiency, speed, and power consumption that can be achieved by customizing HW with HDLs.

Vitis HLS

Vitis HLS is a tool provided by Xilinx that allows designers to translate C or C++ code to RTL to be synthesized to Xilinx devices. The content of this section is derived from the Vitis HLS user guide [10] unless otherwise specified. The output from Vitis HLS is an RTL Intellectual Property (IP) that can be flashed to a specified Xilinx device. The main procedure the designer follows when using Vitis HLS to convert a C/C++ algorithm to a RTL IP is given in the list below:

1. SW Simulation
    - The Vitis HLS applications compile and run C/C++ code and output the results.
2. Synthesize
    - The Vitis HLS applications synthesize the C/C++ code to create all the necessary RTL files.
3. HW-SW co-simulation
    - The Vitis HLS applications allow the designer to verify the behavior of the generated RTL by running a co-simulation that simulates the RTL code together with the C/C++ code.
4. Export RTL IP
    - When the performance and behavior of the synthesis and co-simulation are satisfactory, the designer can export the generated RTL IP to be implemented on a Xilinx device using either the Vivado Design suite or Vitis.

The Vitis HLS tool requires a specific structure in the C/C++ code in order to function. The C/C++ code must include a testbench where the inputs to the algorithm that will be synthesized are created. The testbench should also test the behavior of the synthesized algorithm against the verified behavior in SW to validate that the RTL behaves as desired. In the testbench file, the tool further requires that the testbench function is named *main*() and that the code that will be synthesized is contained in a function called from *main*(). The function specified for synthesis is called the top-level function and can consist of any number of sub-functions if they are all called from the top-level function.

Vitis HLS also allows the designer to apply directives and constraints to the top-level function to optimize chosen aspects. The directives let the designer configure the synthesis in two manners: adding HLS PRAGMAS to the C/C++ code and adding optimization directives in a directive script. The designer can utilize the available directives to optimize the RTL for throughput, latency and area. Figure 14 illustrates the design-flow of Vitis HLS:

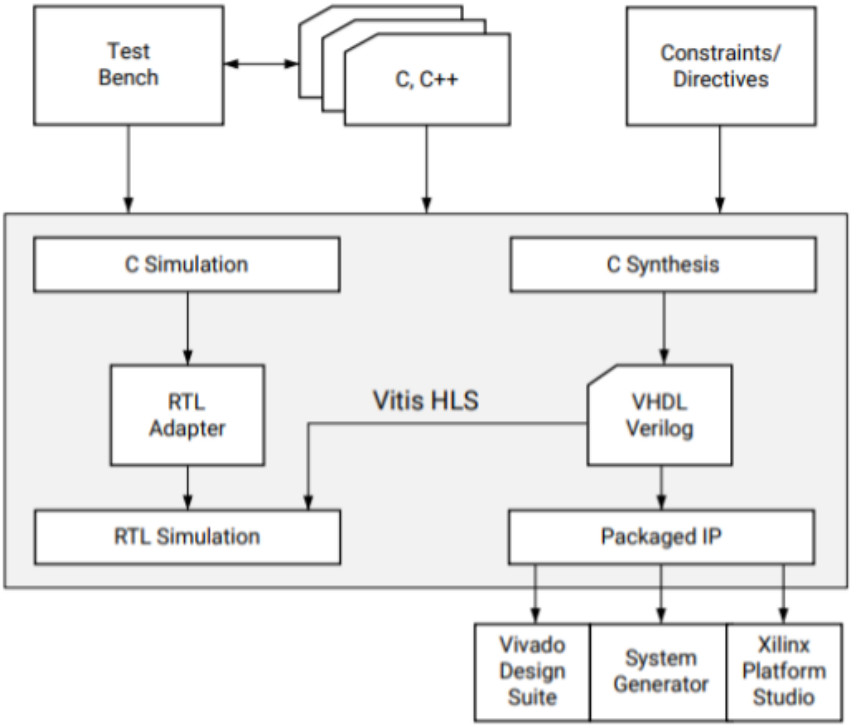


Figure 14: Design-flow of Vitis HLS, taken from [10].

2.3.3 Hardware accelerated hyperspectral anomaly detection

There exists only a limited amount of publications that describe HW acceleration of hyperspectral anomaly detection. The previously described CRD algorithm is accelerated in HW by Wu et al. in [51]. The most important aspect of their HW acceleration was implementing matrix inversion and matrix multiplication on the FPGA. Wu et al. reports about three times lower latency with the FPGA than on a 3.4GHz CPU.

Ma et al. describe an HW accelerated deep learning HAD algorithm in [5]. They use optimization techniques such as pruning and quantization to reduce the computational complexity of the HAD algorithm before accelerating it. A multi-objective optimization algorithm is utilized to optimize the neural network with a low loss of accuracy. The algorithm uses an AE as described in section 2.2. After the optimization process, the trained AE is implemented on a FPGA using HLS, and the anomaly detection is performed in HW.

Lei et al. describe a HAD algorithm that uses morphological and guided filters (F-MGD) accelerated in HW using HLS [52]. The results show that the HW accelerated implementation is 161 times faster or more than the Matlab implementation and 72 times faster or more than the C++ implementation on six real HSIs. The Matlab and C++ results are obtained using a 2.50 GHz Intel Core (TM) Quad CPU with 8GB RAM and, the HW results are obtained using a Virtex7 FPGA running at 200 MHz [52].

2.4 Datasets for hyperspectral anomaly detection

The Airport–Beach–Urban (ABU) dataset consists of 13 real HSIs tailored to assess HAD algorithms. The HSIs are published by Xudong Kang and can be downloaded without cost[1]. Each HSI has a corresponding ground-truth image where all the anomalies have been manually marked. The HSIs are fractions of larger HSIs captured by Airborne Visible/Infrared Imaging Spectrometer

[1] http://xudongkang.weebly.com/datasets.html

(AVIRIS) or Reflective Optics System Imaging Spectrometer (ROSIS-03). Below is a table from the author's specialization project [13] describing the HSIs in the ABU dataset:

Table 2: Details of ABU data set, from specialization project [13]

Scene	Image	Spatial resolution	Pixels	Sensor	Capture year	Capture place	Bands
Airport	(A)	7.1 m	100 x 100	AVIRIS	2011	Los Angeles	205
Airport	(B)	7.1 m	100 x 100	AVIRIS	2011	Los Angeles	205
Airport	(C)	7.1 m	100 x 100	AVIRIS	2011	Los Angeles	205
Airport	(D)	3.4 m	100 x 100	AVIRIS	2010	Gulfport	191
Beach	(A)	17.2 m	150 x 150	AVIRIS	2010	Cat island	188
Beach	(B)	7.5 m	100 x 100	AVIRIS	2011	San Diego	193
Beach	(C)	4.4 m	100 x 100	AVIRIS	2010	Bay Champagne	188
Beach	(D)	1.3 m	150 x 150	Rosis-03	Unknown	Pavia	102
Urban	(A)	17.2 m	100 x 100	AVIRIS	2010	Texas coast	204
Urban	(B)	17.2 m	100 x 100	AVIRIS	2010	Texas coast	207
Urban	(C)	3.5 m	100 x 100	AVIRIS	2010	Gainsville	191
Urban	(D)	7.1 m	100 x 100	AVIRIS	2011	Los Angeles	205
Urban	(E)	7.1 m	100 x 100	AVIRIS	2011	Los Angeles	205

In this chapter I have presented the theoretical background deemed relevant for the thesis. Recent advances in HAD including baseline and state-of-the-art algorithms was researched, and explained. The research shows that the most recent advances in HAD includes the use of DL, morphological attribute filters, variants of RX and variants of CRD among others. This is related to research question number one, regarding the advances in the field of HAD. Theory on HW acceleration was presented, and previous use in the field of HAD was researched. This is related to research question number 4, regarding HW acceleration of HAD algorithms.

3 Methodology

This chapter describes a HAD algorithm proposed in this thesis that is an altered version of the AWDBN proposed in [1]. The chapter describes work that was a part of the authors specialization project in the fall of 2020 [13], which was continued as a part of this thesis. A Matlab implementation of the proposed HAD algorithm is evaluated and compared with Matlab implementations of several baseline and state-of-the-art HAD algorithms. The motivation for this process is to ensure that the proposed HAD algorithm provides competitive results in accuracy and execution time before it is implemented using C and HLS. The notation used in this chapter is the same as in the previous chapter and is summarized in the table below:

Table 3: Notation used in this chapter

Symbol	**Description**
n_b	Number of spectral bands in the HSI
w	Number of pixels in one of the spatial dimensions of the HSI
h	Number of pixels in the second of the spatial dimensions of the HSI
n_c	Number of spectral bands in the code-layer representation of the HSI
$\mathbf{X} \in \mathbb{R}^{h \times w \times n_b}$	The HSI cube
$\hat{\mathbf{X}} \in \mathbb{R}^{h \times w \times n_b}$	The reconstructed HSI cube
$\mathbf{C} \in \mathbb{R}^{h \times w \times n_c}$	The code-layer representation of the HSI cube
$\mathbf{R} \in \mathbb{R}^{h \times w}$	The reconstruction error between $\mathbf{X}$ and $\hat{\mathbf{X}}$
k	Number of neighboring pixels extracted by the DSW
$\mathbf{c_i^n} \in \mathbb{R}^{n_c \times 1}$	The i-th of the k neighboring pixels in the code-layer representation
$\mathbf{c^P} \in \mathbb{R}^{n_c \times 1}$	The PUT in the code-layer representation
d_i	The i-th of the k distances between the neighboring pixels and the PUT in the code-layer representation
r_i^n	The i-th reconstruction error of the k neighboring pixels
r^p	The reconstruction error of the PUT
wt_i	The i-th of the k weights
pf	The penalty factor used to compute the weights
β	The anomaly score for the PUT

3.1 New weights strategy

Six different scenarios can occur when the distance between the PUT and one of the neighboring pixels is computed. If the PUT belongs to an anomaly class, the neighboring pixel can belong to the same anomaly class, a different anomaly class, or a background class. If the PUT belongs to a background class, the neighboring pixel can belong to an anomaly class, the same background class, or a different background class. Different classes of anomalies can, for example, be boats and algae blooms in the ocean, while different background classes can be grass and sea. The six scenarios are illustrated in Figure 15 where (0) shows how d_j (computed with equation 47) is the distance between the PUT and the j-th neighboring pixel in the code-layer representation.

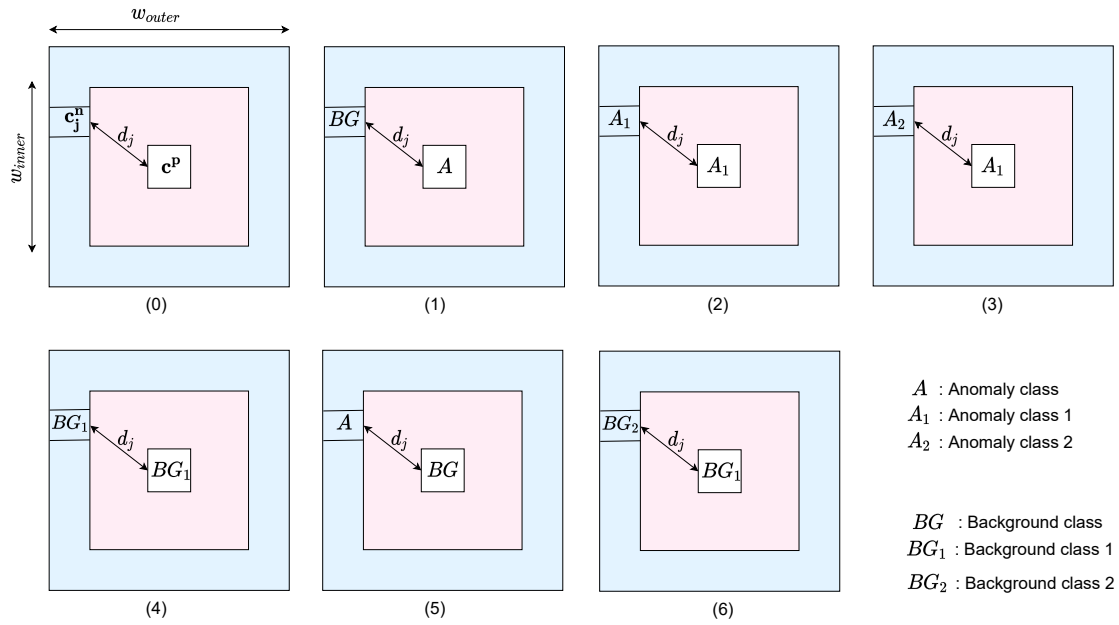


Figure 15: Illustration of the possible scenarios when computing the weights, adopted from [1].

The assumption is that the reconstruction error r for a pixel is high if it belongs to an anomaly class and low if it belongs to a background class. The distance d is assumed to be high between pixels that do not belong to the same class and low between pixels that belong to the same class. The matrix $\mathbf{R}$ contains the reconstruction errors for all pixels in the HSI. The reconstruction errors of the neighboring pixels are extracted from $\mathbf{R}$ with a DSW and represented as an array $\mathbf{r^n}$ containing the k neighboring reconstruction errors. r_j^n is the j-th of the k reconstruction errors in $\mathbf{r^n}$. The table below shows what parameters are expected to be high or low for the different scenarios.

Table 4: Expected values

Scenario	r^p	r_j^n	d_j	Desired β
(1)	High	Low	High	High
(2)	High	High	Low	High
(3)	High	High	High	High
(4)	Low	Low	Low	Low
(5)	Low	High	High	Low
(6)	Low	Low	High	Low

Since the objective is to detect anomalies, β should be high when the PUT is an anomaly and low for the other scenarios. The weights used in the AWDBN have the following characteristics based on the above assumptions:

Table 5: Values obtained with the weights proposed for AWDBN

Scenario	$wt_j = \frac{1}{r_j^n}$	d_j	β	Desired β
(1)	High	High	Very high	High
(2)	Low	Low	Very low	High
(3)	Low	High	~ 1	High
(4)	High	Low	~ 1	Low
(5)	Low	High	~ 1	Low
(6)	High	High	Very high	Low

From the above table, we observe that the adaptive weights wt_j used in AWDBN is high in scenario

(1) and low in scenario (5), which corresponds with the desired behaviour. Scenario (2) could be a problem, but the DSW is applied to make it less likely that anomalies from the same class are compared by having an inner window of a larger size than the anomalies. In scenario (4), the weights contribute in the opposite direction of what is desired. However, the distance between two pixels in the same BG class is expected to be very small, reducing the influence of the weights. Scenario (3) is an infrequent scenario due to the low probability of occurrence defining anomalies, but the weights also behave wrongly in this scenario. The most significant concern with this weight strategy is scenario (6), which is considered normal. In this scenario, the weights will assign high anomaly scores to the border between different background classes leading to higher false alarm rates. The penalty factor pf is multiplied with the weight when the difference between r_j^n and the mean of the other reconstruction errors in the neighboring pixels is larger than the standard deviation of the reconstruction errors in the neighboring pixels. The penalty factor will commonly penalize (decrease) weights where $\mathbf{c_i^n}$ belongs to an anomaly class (scenario (2), (3), and (5)). This will contribute positively to scenario (5) and poorly to the two others. Based on this, we can see that scenarios (1) and (5) are significantly improved by the weight strategy, and based on the results in [1] the overall contribution of the weights is increasing the accuracy for most HSIs.

In order to improve the accuracy, a different weight strategy is proposed. The weights wp_j are computed by

$$wp_j = \frac{r^p}{r_j^n}. \tag{50}$$

This leads to the following characteristics:

Table 6: Values obtained with the new weights

Scenario	$wt_j = \frac{r^p}{r_j^n}$	d_j	β	Desired β
(1)	Very high	High	Very high	High
(2)	~ 1	Low	Low	High
(3)	~ 1	High	High	High
(4)	~ 1	Low	Low	Low
(5)	Very low	High	~ 1	Low
(6)	~ 1	High	High	Low

The above table shows that the expected behavior of the new weights resonates well with the desired behavior. In scenario (1), the weights are increased since they are multiplied with the assumed high reconstruction error of the PUT. In scenarios (2) and (3), the multiplication with r^p is expected to increase the weight, moving the anomaly score in the correct direction. Scenarios (4), (5), and (6) will cause the weight to be multiplied by a low r^p leading to the desired lower weights. Reducing the weight in scenario (6) is expected to have the most significant effect since it will reduce the anomaly score assigned to borders between different background classes.

3.2 Matlab implementation

The proposed HAD algorithm consisting of a DBN AE inspired by [7, 1] and the novel weight-strategy described in section 3.1 is implemented in Matlab. The implementation is a proof of concept that is used to evaluate the HAD algorithm and to compare it with the results with other known HAD algorithms. The flow of the implementation is illustrated in Figure 16.

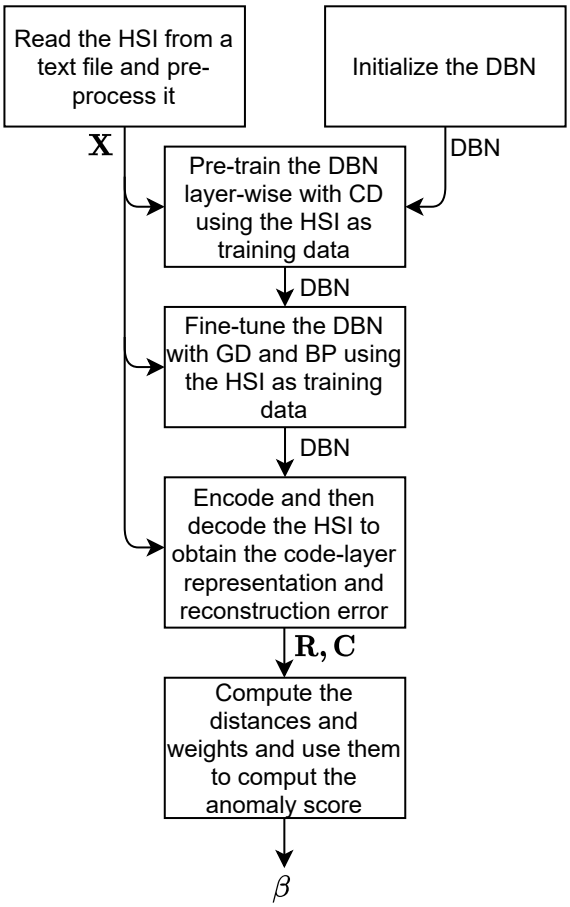


Figure 16: Algorithm flow.

3.2.1 Import and pre-process hyperspectral images

The datasets containing 13 different HSIs with belonging ground-truth maps are downloaded as described in section 2.4. A function named loadHSI.m was implemented and lets the user choose one of the HSIs before pre-processing it in the following manner:

```
1 if strcmp(HSI.dataset, 'air2')
2     load([path  'abu-airport-2.mat']);
3     HSI.an_map                 = map;
4     HSI.M_3D                   = hyperNormalize(data);
5     HSI.M_2D                   = hyperConvert2d(HSI.M_3D)';
6     [HSI.h, HSI.w, HSI.N_band] = size(HSI.M_3D);
7     return
```

The code above compares the value *dataset* in the struct HSI with the names of the different HSIs. When it finds the correct one, which in this case is $air2$, the data is stored in the HSI struct. The $hyperNormalize$ function normalizes all values in the HSI cube to values between 0 and 1. $hyperConvert2d$ takes in a 3-dimensional HSI cube and returns a 2-dimensional matrix with all pixels on one dimension and all bands in the second dimension.

Dimension reduction

Spectral dimension reduction is included as an optional pre-processing step. The Matlab-function $hyperpca(hcube, N)$ from the Hyperspectral Imaging Library in the Image Processing Toolbox uses PCA to find the N principal components in the spectral domain of the hypercube *hcube*. To motivate the choice of the parameter N, the cumulative sum of variances is plotted for all HSIs in the test data. Figure 17 shows that it is possible to contain over 98% of the variance between the bands in all the HSIs by using the 12 principal components.

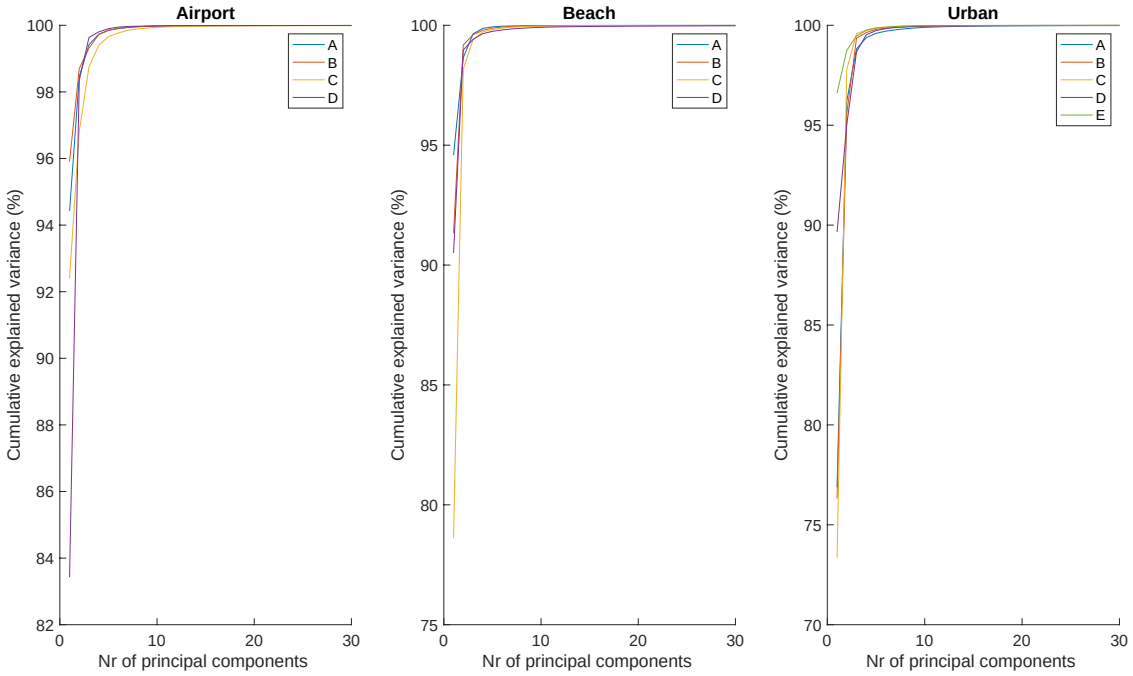


Figure 17: Cumulative sum of variances for PCA in the ABU dataset.

3.2.2 Deep belief network

The initialization, pre-training, and fine-tuning of the DBN AE are implemented by adapting the Deep Neural Network Toolbox[2] proposed in [53]. The main elements extracted from the toolbox are *CD* pre-training and standard ANN training using GD and BP. These elements were further simplified by removing unwanted features. The GD functionality supports the use of mini-batches, meaning that the change in the adjustable parameters is computed for a mini-batch of the training-set before the changes are applied to the network. A challenging task when using an ANN is to set good parameter values. The following list shows the most critical parameters and their chosen values.

- **Layer organization**
  - Decide the number of layers, and the number of ANs in each layer.
  - The number of layers is set to 3, with the first and last consisting of n_b ANs. The middle layer consists of 13 ANs as proposed for AWDBN in [1].
- **The number of iterations to perform training**
  - Both the pre-training and regular training require this value.
  - The value depend on the size and complexity of the HSI.
- **The step-ratio**

[2]https://www.mathworks.com/matlabcentral/fileexchange/42853-deep-neural-network

- This parameter affects how much each adjustable parameter in the ANN is changed for each iteration of training. The parameter value is set based on the suggestions in [48].
- The step-ratio is set to 0.01.

- **Momentum**
  - This value enforces acceleration in the speed of learning when the direction of change in the adjustable parameters is similar for longer periods.
  - Based on suggestions in [48] the momentum is initially 0.5 and changes to 0.9 after 5 iterations of training.

- **Size of mini-batch**
  - This size affects both the number of iterations in the innermost loop of training and the size of the matrix-multiplications that is the heaviest computational burden of the training algorithm. Hinton suggests using a value between 10 and 100 [48].
  - The value depend on the size and complexity of the HSI.

- **Weight decay**
  - This is a value that decreases large weights to enforce generalization during training.
  - Set to 0.0002 after empirical testing.

The parameters described above are collected in a struct *opts* that is passed to the pre-training and training function together with the HSI, and the DBN initialized with all zero biases and zero-mean normal distributed weights with a standard deviation of 0.01.

```
% Deep Belief Network
DBN   = initDBN( opts.layers );
DBN   = pretrainDBN(DBN, HSI, opts);
DBN   = trainDBN(DBN, HSI.M_2D, HSI.M_2D, opts);
```

The HSI is passed to *trainDBN* twice because it is used both as training input and the desired output during training.

Encode and Decode

With the fully trained DBN the next step is to extract **R** and **C** by encoding and decoding the entire HSI. For a DBN consisting of 3 layers this is performed in the following manner:

```
for i = 1:HSI.N_pix
    HSI.C(i, :)  = sigmoid( HSI.M_2D(i, :) * DBN.dbn.rbm{1}.W + DBN.dbn.rbm{1}.b );
    HSI.Y(i, :)  = sigmoid( HSI.C(i, :) * DBN.dbn.rbm{2}.W + DBN.dbn.rbm{2}.b );
    HSI.R(i)     = sqrt(sum((HSI.M_2D(i, :) - HSI.Y(i, :)).^2)/HSI.N_band);
end
```

3.2.3 Weights

To apply the proposed adaptive weights strategy the code-layer representation and the reconstruction error of the neighboring pixels must be extracted. This is done by calling the function:

```
NN = findneighbors(idx, M, win_i, win_o)
```

The parameter *idx* is the index of the PUT, and the parameters *win_i* and *win_o* are the sizes of the inner and outer windows of the DSW. The parameter NN is a vector consisting of all the neighboring pixels located between the inner and outer windows. $\mathbf{c^n}$ is obtained by setting M to $\mathbf{C}$ and $\mathbf{r^n}$ is obtained by setting M to $\mathbf{R}$. The anomaly score for the PUT is then computed in the following way:

```
wt    = HSI.R(idx)./Rn';                          % Compute weights
dist  = sum((sqrt((Cn - HSI.C(idx, :)).^2)), 2); % Compute distances
an_score(idx) = (1/length(dist))*(wt*dist); % Compute anomaly score
```

The adaptive weights procedure must be performed for all pixels in $\mathbf{X}$ creating an anomaly score gray-scale map which indicates the probability that each pixel is an anomaly.

3.3 Evaluation

3.3.1 Performance metrics

The most widely used performance metric for evaluating hyperspectral HAD algorithms is the Area Under the Receiver Operator Characteristics (ROC) curve (AUC). In this case, the AUC metric reflects the estimated probability that a random anomaly pixel is deemed more likely to be an anomaly than a random background pixel. The AUC does not consider a specific threshold, but in a sense, uses the average over every possible threshold [54]. The required elements for computing the AUC are a ground-truth map and an anomaly-score map. For a given threshold, anomalous pixels are called True Positives (TP) if classified as anomalies and False Negatives (FN) if classified as background. Background pixels are called TP if classified as background and FN if classified as anomalies. For a given threshold, the True Positive Rate (TPR) and False Positive Rate (FPR) are computed by

$$TPR = \frac{TP}{TP + FN} \tag{51}$$

and

$$FPR = \frac{FP}{FP + TN}. \tag{52}$$

The TPR and FPR are computed for a sufficiently high number of thresholds before the ROC curve is created. The ROC curve is created by plotting the TPR values on the Y-axis and the FPR on the X-axis. The AUC is computed with

$$AUC = \int_{-\infty}^{\infty} TPR(t) \cdot FPR'(t)dt. \tag{53}$$

The second metric used for evaluation is the execution time of the HAD algorithm. The execution time is important because many HAD algorithms are more time-consuming than what is acceptable for a real satellite mission.

3.3.2 Comparison with known HAD algorithms

The HAD algorithm proposed in this thesis is applied to the 13 real HSIs in the ABU-dataset described in section 2.4 with and without dimensionality reduction. The AUC results are compared with the 11 HAD algorithms listed in table 7. The rightmost column shows which paper has presented the AUC results of the corresponding HAD algorithm on the ABU dataset. The AUC results for AWDBN and DBN on the ABU-dataset were computed by the author of this thesis during the specialization project and presented in the specialization thesis [13]. The origin of LRX has not been found, but the algorithm is described in a large number of papers, including [34].

Table 7: The HAD algorithms that are used in the evaluation of the HAD algorithm proposed in this thesis.

Name of HAD algorithm	Abbreviation	Published	Paper presenting the AUC results
Global Reed-Xiaoli	GRX	[28]	[27]
Local Reed-Xiaoli	LRX	Unknown	[27]
Fractional Fourier Entropy Reed-Xiaoli	FrFE-RX	[37]	[43]
Collaborative Representation Detector	CRD	[38]	[27]
Attribute and Edge-preserving filters	AED	[27]	[43, 27]
Morphological Profile and Attribute Filters	MPAF	[43]	[43]
Deep Belief Network	DBN	[7]	[13]
Adaptive-weight DBN	AWDBN	[1]	[13]
Spatial Density Background Purification	SDBP	[39]	[39]
Support Vector Data Description	SVDD	[55]	[27]
Fast Morphological and guided filters	F-MGD	[52]	[52]

The HAD algorithm proposed in this thesis, the AWDBN, and DBN, have been implemented in Matlab 2019B, and the AUC values are computed on a 2015 Macbook Pro with a 2.9GHz dual-core i5 processor. The AUC results taken from [43] are computed on a computer with an i5 CPU [43]. The AUC results taken from [27] are computed using Matlab on a 2.8GHz CPU [27]. The AUC results of the AED are presented in both [27] and [43] with significant differences. The lower results presented in [43] are used in the comparison in this thesis. The results taken from [39] are obtained on a 2.50GHz Intel quad CPU with 8GB RAM.

The parameters such as thresholds, window sizes, et cetera have been optimally chosen for all HAD corresponding to the different HSIs [27, 43]. For the HAD algorithm proposed in this thesis, AWDBN and DBN, the AUC results are obtained with a script testing different combinations of the parameter values. The script iterates over a range of values for the parameters training-iterations, DSW size, and mini-batch size. The best AUC result for each of the three algorithms on each of the 13 HSIs is used in the table below. Table 8 shows the AUC results for all 13 HSIs in the ABU-dataset. The highest AUC in each row is highlighted with bold font, and the second-highest AUC is underlined. The parameter values used by the HAD algorithm proposed in this thesis to obtain the below results are presented in appendix A.

Table 8: AUC scores for the ABU-dataset

Scene	Image	AUC												
		Proposed	With PCA	AWDBN	DBN	MPAF	AED	FrFE-RX	LRX	GRX	SVDD	CRD	SDBP	F-MGD
Airport	(A)	0.9689	0.9479	0.9269	0.9113	0.9718	**0.9923**	0.9081	0.9458	0.8221	0.9501	0.9577	0.9615	–
Airport	(B)	0.9825	0.9375	0.9557	0.9563	0.9765	**0.9936**	0.9690	0.9874	0.8404	0.9900	0.9744	0.9842	–
Airport	(C)	0.9713	0.9531	0.9098	0.9632	**0.9785**	0.9756	0.9424	0.9467	0.9288	0.9080	0.9564	0.9662	–
Airport	(D)	0.9826	0.9900	0.9667	0.9783	**0.9997**	0.9953	0.9854	0.8740	0.9526	0.9848	0.9757	0.9930	–
Airport	Average	0.9763	0.9571	0.9397	0.9523	0.9816	**0.9892**	0.9512	0.9384	0.8859	0.9582	0.9660	0.9762	–
Beach	(A)	0.9970	0.9905	**0.9995**	0.9911	0.9992	0.9974	0.9862	0.9956	0.9807	0.9538	0.9916	0.9959	0.9977
Beach	(B)	0.9852	0.9722	0.9899	0.9302	**0.9910**	0.9550	0.9161	0.9777	0.9106	0.9620	0.9416	0.9876	–
Beach	(C)	**0.9999**	0.9995	**0.9999**	0.9993	0.9998	0.9997	0.9997	0.9997	**0.9999**	**0.9999**	0.9996	**0.9999**	–
Beach	(D)	0.9920	0.9863	0.9897	0.9808	**0.9985**	0.9916	0.9541	0.9391	0.9538	0.9561	0.9450	0.9763	–
Beach	Average	0.9935	0.9871	0.9947	0.9753	**0.9971**	0.9859	0.9640	0.9780	0.9613	0.9691	0.9695	0.9899	–
Urban	(A)	0.9950	0.9928	**0.9989**	0.9886	0.9986	0.9981	0.9918	0.9968	0.9907	0.9862	0.9948	**0.9989**	–
Urban	(B)	0.9974	0.9988	0.9724	0.9933	**0.9990**	0.7890	0.9962	0.9231	0.9946	0.9256	0.9410	0.9984	0.9843
Urban	(C)	0.9896	0.9894	0.9882	0.9591	0.9977	0.9976	0.9684	0.9800	0.9513	0.9658	0.9634	0.9946	**0.9994**
Urban	(D)	0.9949	0.9955	0.9955	0.9739	0.9959	0.9912	0.9809	0.9696	0.9887	0.9621	0.9816	0.9972	**0.9975**
Urban	(E)	0.9785	0.9658	0.9557	0.9039	**0.9900**	0.9845	0.9719	0.9538	0.9692	0.9449	0.9521	0.9877	–
Urban	Average	0.9911	0.9885	0.9821	0.9637	**0.9962**	0.9521	0.9818	0.9647	0.9789	0.9589	0.9666	0.9954	–
All scenes	Average	0.9870	0.9776	0.9721	0.9637	**0.9916**	0.9757	0.9657	0.9604	0.9420	0.9621	0.9673	0.9872	–

MPAF achieves the highest AUC scores on average and has the highest or second-highest AUC on 9 of 13 HSIs. The AED is the fifth most accurate HAD algorithm based on the above table, but the authors that propose AED reports significantly higher results for two of the HSIs (0.9985 for Urban (B) and 0.9825 for Beach (B)). Using the higher values, the average for all HSIs becomes 0.9924, which is the best of all the HAD algorithms used in this comparison. The HAD algorithm proposed in this thesis achieves the third or fourth highest AUC scores based on which results for AED are used. The difference between the HAD algorithm proposed in this thesis and MPAF is only 0.0046 on average for all 13 HSIs, and the proposed algorithm outperforms MPAF on 2 of the HSIs. The baseline HAD algorithms GRX and LRX achieve the lowest AUC values as expected. The state-of-the-art approaches SVDD, CRD, and FrFE-RX all achieve lower overall average AUC than the HAD algorithm proposed in this thesis while also achieving better AUC scores than the baseline HADs.

The execution times of the HAD algorithms are presented in the table below. The execution times are collected from the same papers as the AUC results. The execution times for MPAF, GRX, AED, SDBP, and FrFE-RX are measured as the average execution times of the HSIs in the Urban scene. The execution times for the HAD proposed in this thesis, AWDBN, DBN, CRD, and SVDD, are measured using a $100 \times 100 \times 189$ HSI, which is comparable to the average size of the Urban HSIs. The results for HAD algorithm proposed in this thesis and AWDBN are obtained using a 5×5 inner window and a 6×6 outer window. The training of the proposed algorithm, AWDBN, and DBN, consisted of 10 iterations of CD_1 pre-training and 15 iterations of fine-tuning using a mini-batch size of 15.

Table 9: Execution times results in seconds

MPAF	AED	FrFE-RX	LRX	GRX	SVDD	CRD	SDBP	F-MGD
0.17	0.41	22.89	57.59	0.14	377.55	39.25	7637.45	0.2

Table 10: Execution times results in seconds

Time	Proposed	With PCA	AWDBN	DBN
Total	3.89	1.66	3,86	3.64
Detection	0.53	0.40	0.50	0.28
Training	3.36	1.26	3.36	3.36

We observe that there are significant differences between the execution times of the different HAD algorithms. The SDBP, CRD, FrRE-RX, SVDD, and LRX all take over 22 seconds to finish in Matlab, making them less attractive for real remote sensing missions. The GRX is the fastest detector but also the one with the lowest AUC scores. The SDBP, which scored the second-highest AUC results, is significantly slower than the other algorithms, making it especially unsuitable for real satellite missions.

The performance of the HAD algorithm proposed in this thesis with PCA dimensionality reduction is, on average, below 1% lower than without PCA on the 13 HSIs in the ABU dataset. Furthermore, the execution time is significantly decreased when the HSI cube is reduced before performing the algorithm. This shows considerable flexibility where the algorithm can perform well both with and without dimensionality reduction.

3.3.3 Detection maps

The detection maps are plots of the anomaly score for each pixel arranged in the same order as the pixels are arranged in the original HSI. An issue with the visualization of the detection maps is that pixels of different anomalies can have different anomaly scores based on how much their spectral signature stands out from neighboring spectral signatures. A typical example could be if the background pixels have a mean anomaly score of 0.1 and there are two different anomalies with mean anomaly scores of 5 and 25, respectively. The anomaly with a lower anomaly score would

then hardly be visible in the normalized gray-scale detection maps. The AUC metric does not have this issue since it measures how well the algorithm separates anomalies from the background without regard to the separation size. In order to present detection maps that include all anomalies detected instead of only the ones with the highest score, all values above ten times the mean scores are being set to ten times the mean by the following line in Matlab.

```
1 det_map(det_map > 10*mean(det_map)) = 10*mean(det_map);
```

Airport scenes

Figure 18 shows the anomaly maps for all 4 HSIs in the Airport scene. We can observe that the AWDBN tends to assign high anomaly scores to borders between different background classes, which corresponds well with the assumptions in section 3.1. The difference in the AUC score between the proposed HAD algorithm and AWDBN is visible in the Figure as the airplanes are more distinctly separated from the background. The HAD algorithm proposed in this thesis also provides anomaly maps where the anomalies are easier to visualize than the DBN for all 4 HSIs.

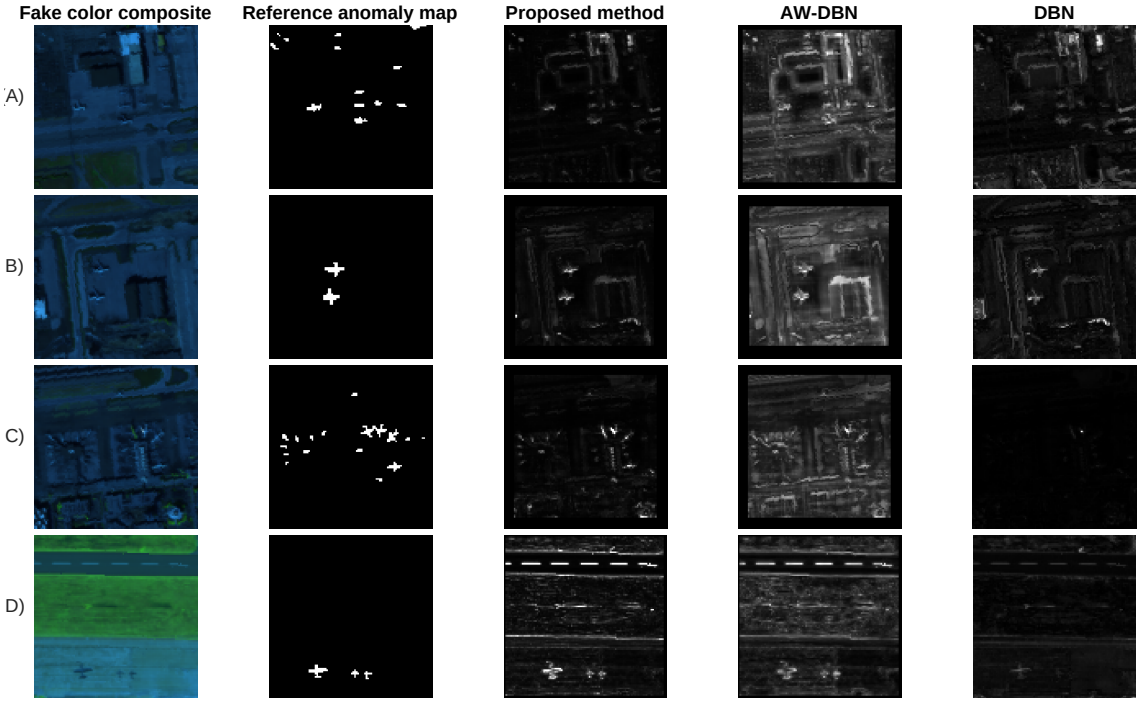


Figure 18: Detection maps for the four Airport scenes in the ABU dataset

Figure 19 confirms the results from the AUC scores where the HAD algorithm proposed in this thesis and the AWDBN have almost the same accuracy and outperform DBN significantly on Beach (B). The AWDBN performs better in these HSI where there are fewer different background classes. However, we can still observe that AWDBN assigns higher relative anomaly scores to the borders in Beach (D), causing lower AUC results.

Beach scenes

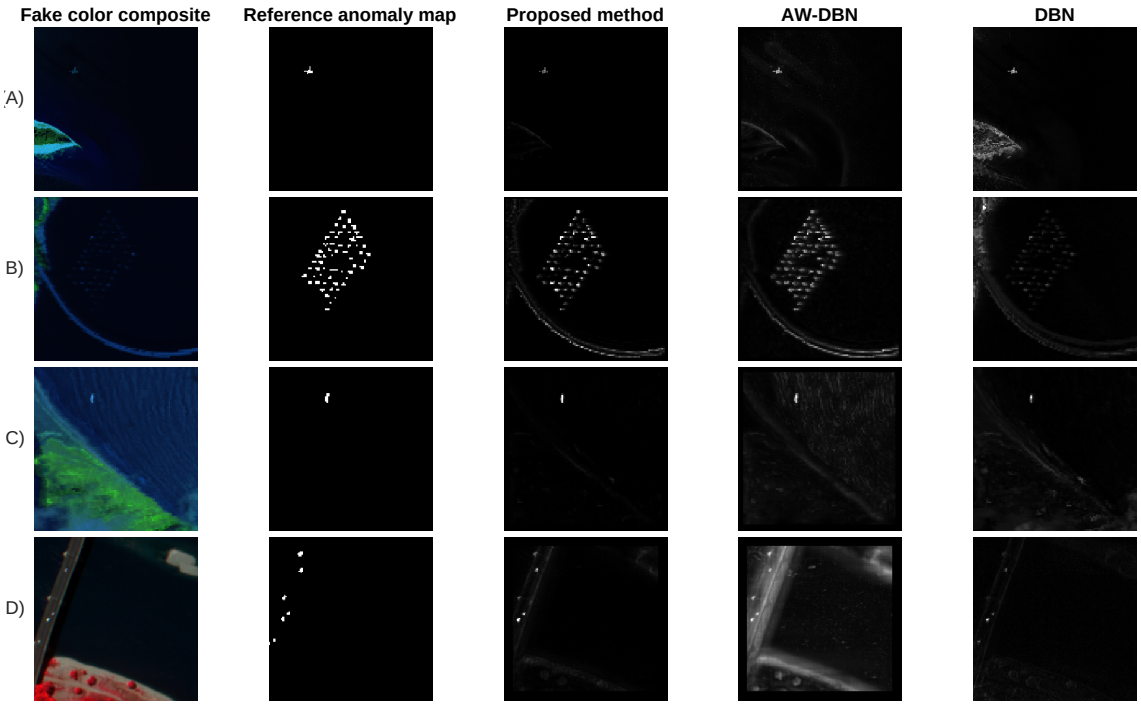


Figure 19: Detection maps for the four Beach scenes in the ABU dataset

Figure 20 supports the AUC results where the HAD algorithm proposed in this thesis has stable and high AUC, which outperforms AWDBN on (B), (C), and (E) and DBN on all five.

Urban scenes

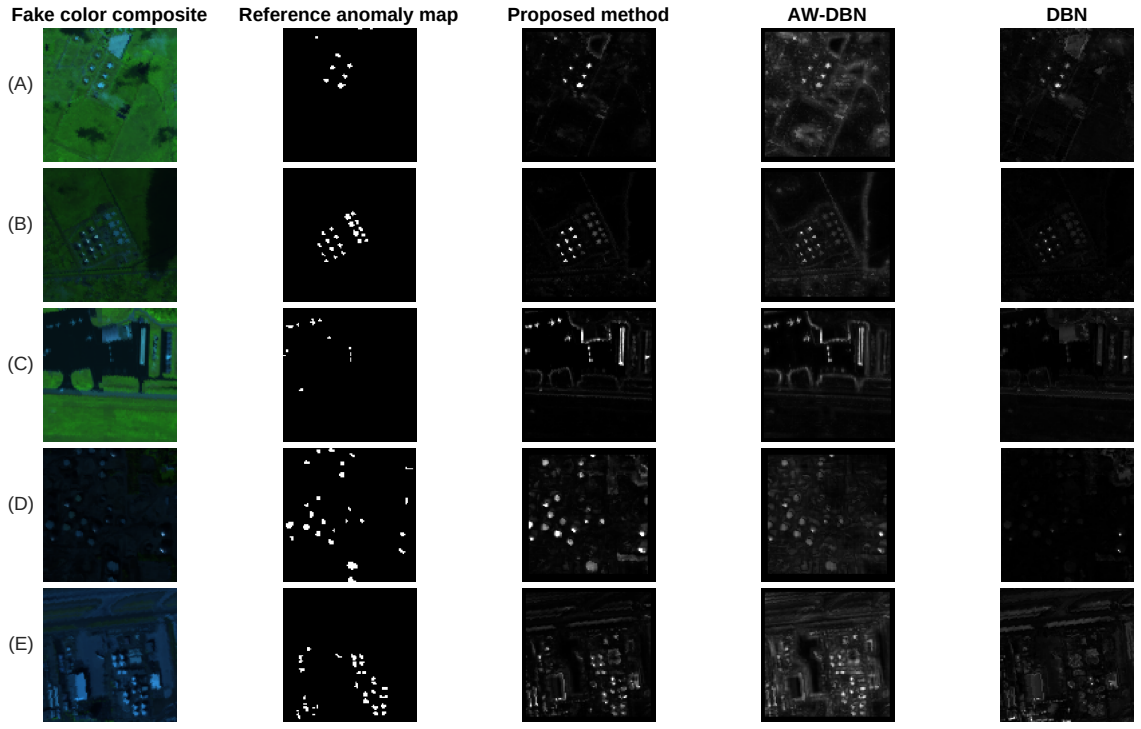


Figure 20: Detection maps for the four Urban scenes in the ABU dataset

One possible source of error when computing the AUC results with the AWDBN and the proposed HAD algorithm is that anomalies near the edge of the HSIs are ignored. When utilizing the DSW, the DSW must fit around the PUT, which means that the pixels too close to the edge of the HSI to use the DSW are ignored. This is not a problem for most HSIs, but it can have a slight effect on the AUC results as fewer pixels are part of the computation. The effects of this are visible in the detection maps, especially for **Beach (D)**. At the left side of the reference anomaly map there is an anomaly very close to the edge of the HSI. On the detection maps of AWDBN and the proposed HAD algorithm, this anomaly is ignored.

This chapter discussed how a HAD algorithm could be evaluated, related to research question number two. More specifically, the HAD algorithm proposed in this thesis is evaluated and compared to known algorithms. Based on the results, it is evident that the HAD proposed in this thesis can compete with state-of-the-art HADs. The execution time is within the boundaries for what is achievable in real missions, and the accuracy is, on average, below 0.5% lower than the accuracy achieved by MPAF. Since MPAF only uses one band, and AED only uses one principal component, they rely on anomalies that significantly stand out even when the HSI is reduced to a greyscale image, which the proposed does not. The dimensionality reduction feature is also unsuitable for some real missions onboard satellites due to strict resource requirements. The flexibility of the HAD proposed in this thesis to include a larger number of spectral components is positive. The proposed new weight strategy achieves higher AUC scores than the old on average and provides more consistent results.

4 Implementation

This chapter describes the implementation of a simplified version of the HAD algorithm proposed in this thesis using C-programming language. Furthermore, the chapter describes the acceleration of a part of the simplified HAD algorithm with HLS. The simplified HAD algorithm consists of a DBN AE and uses the reconstruction errors as anomaly scores directly. The simplified HAD algorithm is implemented as a proof of concept, and the pre-training and weight-strategy can be added later without modifying the existing implementation. The main reasons for implementing the HAD algorithm in C are the following:

- The software onboard the satellite used in the HYPSO mission is written in C-based languages. To add the HAD algorithm to the processing pipeline onboard it has to be implemented in C or C++.
- Vitis HLS synthesizes C/C++ code to RTL description, enabling rapid testing and implementation of customized HW to speed up parts of the HAD algorithm.
- C has a lower level of abstraction than Matlab and can speed up the HAD algorithm in SW.

The detection process of the simplified HAD algorithm is chosen as a target for HW acceleration. A HW kernel is created using Vitis HLS that takes in the trained parameters of the DBN and the HSI and returns the anomaly scores. A HW kernel is a part of the algorithm moved from SW to customized HW to be accelerated. Several optimization techniques is investigated using Vitis HLS before three different implementations are tested on a Zynq Ultrascale+ ZCU104 MPSoC Evaluation Kit.

4.1 Software

The flow and procedures of the C implementation is illustrated in the Figure below:

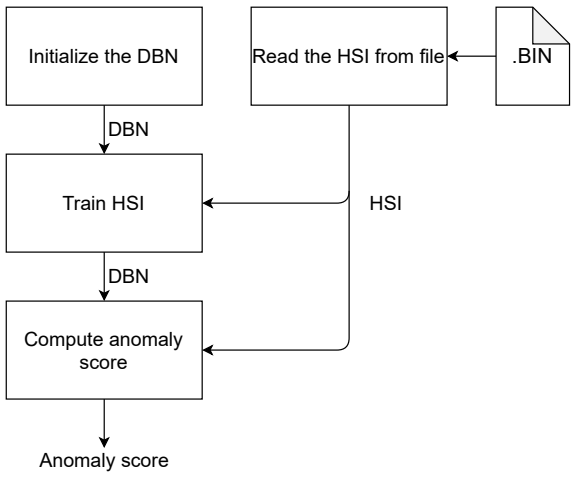


Figure 21: Illustration of the SW implementation.

Read HSI from file

The hyperspectral camera used in the HYPSO mission stores the captured HSIs using the Band Interleaved by Pixel (BIP) format. The values of all bands in each pixel are stored sequentially as illustrated in Figure 22 where x_n^p is the value of the n-th band in pixel p in HSI $\mathbf{X}$.

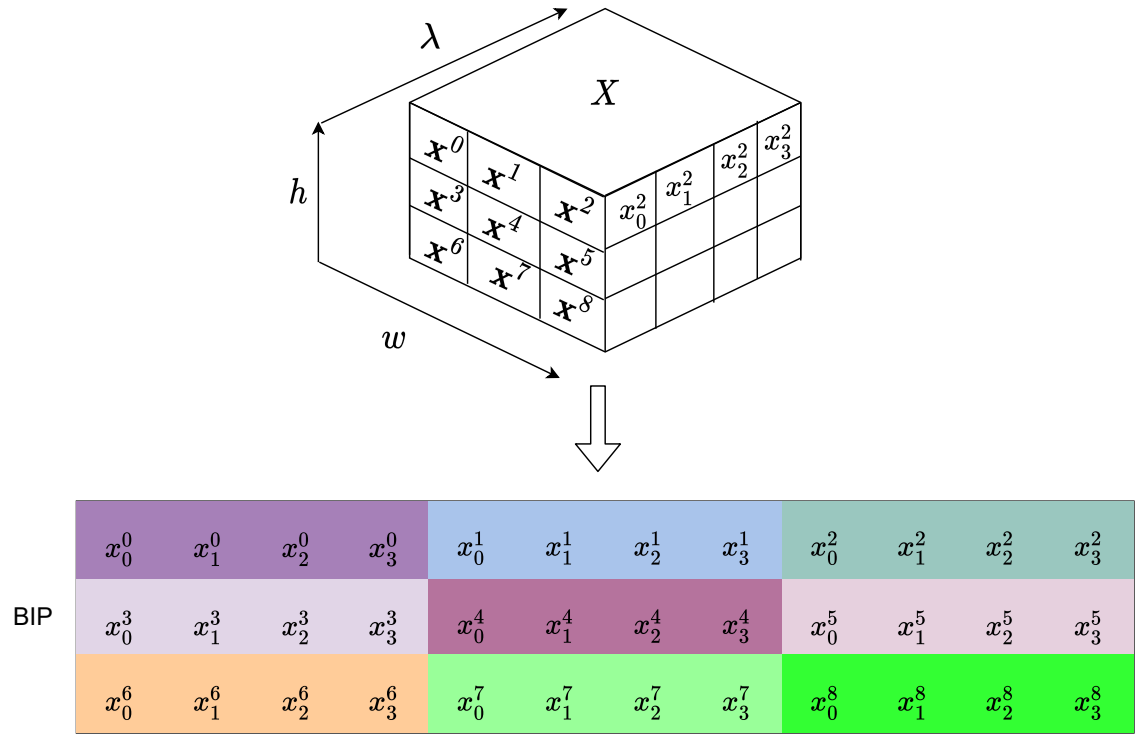


Figure 22: Illustration of BIP format for a $3 \times 3 \times 4$ HSI cube.

To ensure simple porting with the HYPSO onboard processing pipeline the implementation reads the HSI from a .bip file. The datatype `float`, which is a 32-bit floating point-representation, is used for the HSI. Two structs are declared to store and use the HSI in the implementation. The struct `matrix_float` stores the data as a one-dimensional array of floats, which can be used as a two-dimensional matrix by using the variables `width` and `height` of the `matrix_float` struct. The struct `HSI` stores the data from the .bip file and the most important parameters of the HSI. The actual HSI is then read and stored in the struct `HSI` with the built-in C-functions `fopen()` and `fread()` using the option `rb` for read binary. The data in the array in `matrix_float` is arranged as illustrated above with all bands of the first pixel, then all bands of the second pixel and so on.

Initializing the DBN

The DBN is initialized using a struct for the DBN containing two RBM structs. In the `RBM` struct the parameters `visual` and `hidden` refer to the number of visual and hidden ANs the RBM consists of respectively. In the `DBN` struct the parameter `bands_n` refer to the number of ANs in the input and output layers and `mid_layer_size` refer to the number of ANs in the middle layer. The biases of both RBMs are sat to zero and the weights are given normally distributed random numbers with a standard deviation of 0.1. The random values are generated with the `time.h` C library in the following manner:

```
time_t t;
srand((unsigned) time(&t));
for( int i = 0 ; i < bands_n ; i++ ){
    for( int j = 0 ; j < mid_layer ; j++ ){
        dbn->rbm1->weights->buf[i*mid_layer + j] = 0.1f*(rand()/RAND_MAX)-0.05f;
        dbn->rbm2->weights->buf[i*mid_layer + j] = 0.1f*(rand()/RAND_MAX)-0.05f;
    }
}
```

Training the DBN

The training algorithm consists of mini-batch GD and BP and is implemented with only basic C libraries. The implementation of the algorithm is adapted and translated from the Matlab Deep Neural Network Toolbox proposed in [53]. To implement the training algorithm, the following matrix functions were implemented to provide required matrix operations.

- `matrix_float* blank_matrix_float(size_t width, size_t height)`
  - Allocates memory dynamically and return a pointer to a blank `matrix_float`.
- `float mat_get(matrix_float* A, size_t h_idx, size_t w_idx)`
  - Returns the element (w_idx, h_idx) of `A`.
- `void free_matrix_float(matrix_float* A)`
  - Deallocate the dynamic memory allocated by `blank_matrix_float()` to avoid memory leaks.
- `void transpose(matrix_float* A)`
  - Transpose `A`.
- `void mat_mult(matrix_float* A, matrix_float* B, matrix_float* C)`
  - Performs matrix multiplication (`C` = `A` $\times$ `B`).
- `void mat_add(matrix_float* A, matrix_float* B, matrix_float* C)`
  - Performs matrix addition (`C` = `A` + `B`).
- `void mat_div_scalar(matrix_float* A, float scalar, matrix_float* C)`
  - Divide each element in `A` by the scalar and stores the result in `C`.
- `void mat_mul_scalar(matrix_float* A, float scalar, matrix_float* C)`
  - Multiply each element in `A` by the scalar and stores the result in `C`.
- `void mat_sub(matrix_float* A, matrix_float* B, matrix_float* C)`
  - Performs matrix subtraction (`C` = `A` - `B`).
- `void sigmoid(matrix_float* A, matrix_float* C)`
  - Element-wise logical Sigmoid function (`C` = ϕ(`A`)).
- `void mat_cpy_batch(int start_i, int batchSize, HSI* hsi, matrix_float* A, int* ind)`
  - Copy a mini-batch from the HSI and stores it in the `A`.

The training is initiated by calling the function `trainDBN()` which takes in the HSI, the parameters of training, and the initialized DBN AE. The parameters are the same as the parameters described in section 3.2 and have the same values. `trainDBN()` returns a `DBN` struct with the adjustable parameters trained on the HSI.

Computing the anomaly score

To compute the anomaly score, the HSI is encoded and decoded with the trained DBN AE before the reconstruction error between the original and reconstructed HSI is computed. The function `encodeDecode()`takes in the trained DBN and the HSI and feeds the HSI through the DBN to obtain the reconstructed HSI. The reconstruction error is computed as the RMSE between the original and the reconstructed HSI. `encodeDecode()` will be the target for HW acceleration with HLS, the C code is shown below:

```
void encodeDecode(DBN* dbn, HSI* X, matrix_float* R){
    matrix_float* H1    = blank_matrix_float(dbn->mid_layer_size, X->pixels);
    matrix_float* X_hat = blank_matrix_float(dbn->bands_n, X->pixels);

    mat_mult(X->data, dbn->rbm1->weights, H1);     %|
    mat_add_bias(H1, dbn->rbm1->bias_h, H1);       %|-> H = sigmoid(X*W1 + B1)
    sigmoid(H1, H1);                               %|

    mat_mult(H1, dbn->rbm2->weights, X_hat);       %|
    mat_add_bias(X_hat, dbn->rbm2->bias_h, X_hat); %|-> X_hat = sigmoid(H*W2 + B2)
    sigmoid(X_hat, X_hat);                         %|

    %Store the reconstruction error in X_hat
    mat_sub(hsi->data, X_hat, X_hat);              %|-> X_hat = X_hat - X
    for (size_t i = 0; i < hsi->pixels; i++)
    {
        float sum = 0.0f;
        for (size_t j = 0; j < dbn->bands_n; j++)
        {
            sum += powf(X_hat->buf[i*dbn->bands_n + j], 2);
        }
        R->buf[i] = sqrtf(sum/dbn->bands_n);%|-> R is reconstruction error
    }
}
```

4.2 High-level synthesis

The computation of the anomaly score is accelerated by implementing a HW kernel on a FPGA. The latency of the HW kernel is the time passed from the HW kernel is initiated to the results are provided. The HW kernel is written in C/C++ and synthesized to RTL by Vitis HLS. The basic requirements that apply when writing C/C++ targeted for HW synthesizing with Vitis HLS are the following:

- All arrays must be statically defined (Not allowed with `malloc()`/`calloc()`).
- No pointer to pointer.
- No CPU functions (`time()` etc).

The C/C++ code must follow a specific structure where the part of the code targeted for HW acceleration is contained inside a single top-layer function. The `main()` function is treated as a

testbench and the kernel represented by the top-layer function must be called from `main()`. The top-layer function and all sub-functions called from it are synthesized to RTL and made available for implementation on a selection of FPGAs. A flow diagram showing how the kernel is used to compute the anomaly score is shown in Figure 23.

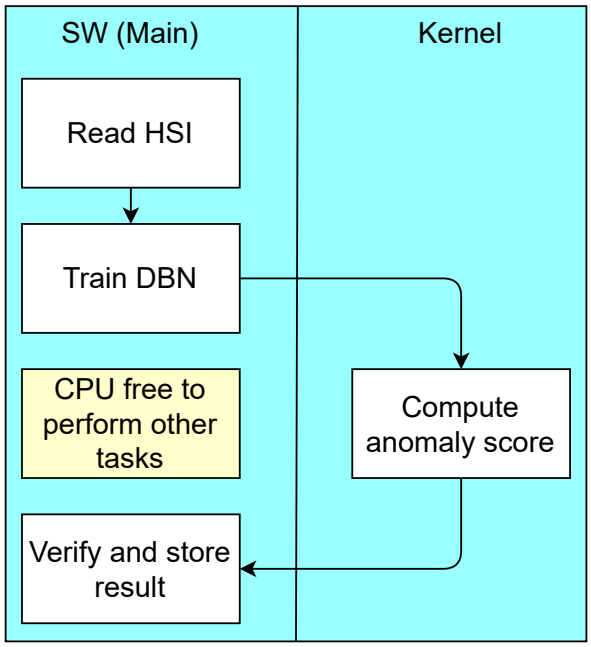


Figure 23: Overview of HLS code.

4.2.1 Baseline

Vitis HLS can translate both C and C++ to RTL, but the libraries `ap_fixed.h` and `hls_stream` are only available for C++. The two libraries are deemed necessary to create the intended behavior, so the code is translated from C to C++. The HW kernel is implemented with a "load - compute - store" structure as shown in Figure 24. The inputs to the HW kernel is the parameters of the trained DBN AE and the HSI.

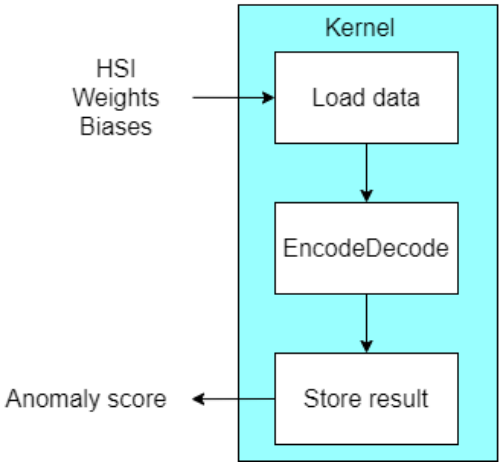


Figure 24: Load-compute-store kernel.

In Vitis HLS, the input data is passed as inputs to the top-layer function. The top-layer function for computing the anomaly score is declared as follows:

```
typedef float dat_t;

void top_layer(hls::vector<dat_t, BANDS>* hsi_f, dat_t out_r[WIDTH*HEIGHT],
    dat_t W1[MID_LAYER*BANDS], dat_t B1[MID_LAYER], dat_t W2[MID_LAYER*BANDS],
    dat_t B2[BANDS]);
```

The datatype `hls::vector` is a Single Instruction Multiple Data vector, which allows a single operation to be performed to all elements in parallel. The HSI is sent to the top-layer function as an array of `hls::vector` where each vector contains a whole pixel. The arrays are all declared with their sizes to satisfy the requirement of statically declared arrays.

Load input

The first stage loads the input to the HW kernel. The sub-function `load_input()` in the top-layer function takes in the array of `hls::vector` and transfers the data to the computational part of the kernel with the transfer type `hls::stream`. `hls::stream` transfers the data in a sequential order and does not require any further management of memory. The stream `hsi_stream` is declared in the top-layer function before the load function is called as follows:

```
static  stream< vector<dat_t, BANDS> > hsi_stream("input_stream_1");
load_input(hsi_f, hsi_stream);
```

Inside the `load_input()` function the pixels are added to the stream in the following way:

```
for (int i = 0; i < WIDTH*HEIGHT; i++)
{
    hsi_stream << hsi_f[i];
}
```

Compute

The computational functionality of the HW kernel is performed in the sub-function `EncodeDecode()` which takes in the stream of `hls::vector` all containing a full pixel of the HSI, the weights and the biases. The function loops over all pixels and computes the anomaly score pixel-wise contrary to the C implementation which computes the anomaly score for all pixels using large matrices. The computational part of the HW kernel performs the following logic for all pixels in the HSI:

```
pix = hsi_stream.read();                              %Buffer pixel vector
float sum_t = 0.0f;
float H1[MID_LAYER] ={0.0f};
float X_hat[BANDS] ={0.0f};

for (size_t j = 0; j < MID_LAYER; j++)
{
        for (size_t k = 0; k < BANDS; k++)
        {
           H1[j] += pix[k]*W1[k*MID_LAYER + j];     %Multiply weights and pixel
        }
        H1[j] = (H1[j] + B1[j]);                    %Add bias
        H1[j] = 1.0f/(1.0f + expf(-1.0f * H1[j] )); %Sigmoid
}

for (size_t m = 0; m < BANDS; m++)
{
        for (size_t n = 0; n < MID_LAYER; n++)
        {
                X_hat[m] += H1[n]*W2[n*BANDS + m];
        }
        X_hat = (X_hat[m] + B2[m]);                 %Add bias
        X_hat[m] = 1.0f/(1.0f + expf(-1.0f * tmp ));%Sigmoid
        X_hat[m] = X_hat[m] - pix[m];   %| Error     |
        sum_t += X_hat[m]*X_hat[m];     %| Square    |
}                                       %|           | -> RMSE
                                        %|           |
out_stream << sqrt(sum_t/BANDS);        %| Root mean |
```

Store result

The resulting RMSE for all pixels is written to an output stream. The sub-function `store_result()` takes in the output stream and stores the result in an array. The array is returned from the HW kernel to the testbench.

4.2.2 Optimization

As mentioned in the section 2.3.2, FPGAs can offer high levels of parallelism, low power consumption, and free up time for the CPU. The baseline HW kernel described in the previous chapter is functioning yet not efficient implementation. Vitis HLS uses optimization PRAGMAS to change the behavior of the HW kernel and improve the performance. The optimization of the HW kernel is an iterative process where different PRAGMAS are added, and the results are compared to find a solution that provides the desired performance.

Dataflow PRAGMA

The baseline kernel executes the loading, computing, and storing sequentially. An option to reduce the latency is to perform tasks simultaneously whenever possible. The dataflow pragma enables task-level parallelism by allowing the subsequent task to start before the current task is finished, thus increasing the HW kernel's concurrency. The dataflow PRAGMA is added to the top-layer function by inserting `#pragma HLS DATAFLOW` before calling the first sub-function. This allows the computational part to start executing when the first pixel vector has been added to the input stream instead of waiting for all pixels to be added to the stream. The results can be stored in the output array when computed instead of waiting for all the results before beginning to store them. The dataflow pragma only applies for the region it is specified in, meaning that it must be written inside the function if dataflow is desired inside a sub-function. Figure 25 illustrates the effect of the dataflow pragma on the implemented kernel.

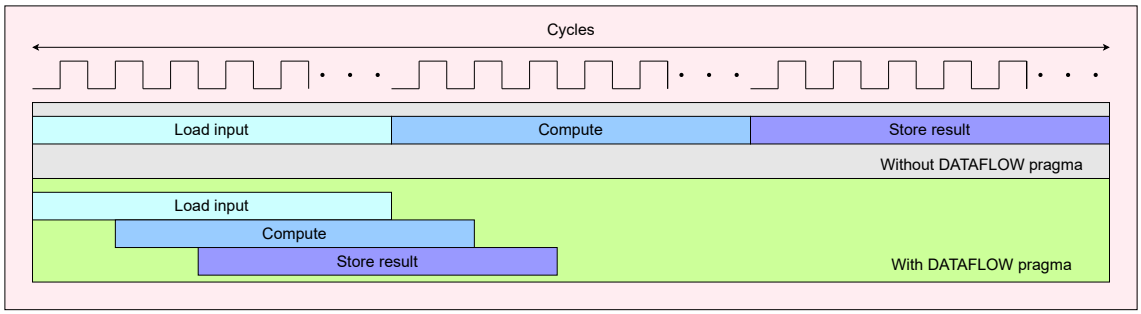


Figure 25: Illustration of kernel with and without dataflow pragma, adapted from the Vitis web-page [11].

In addition, to decrease the latency, the dataflow pragma also tends to increase the utilization of FPGA resources. New channels for transferring data between the tasks and complementary logic to communicate between tasks are added automatically by the synthesis in Vitis HLS.

Pipeline pragma

To further increase the concurrency of the HW kernel, the pipeline PRAGMA can be added to functions or loops by writing `#pragma HLS PIPELINE` inside the loop or function body. Vitis HLS inline small functions to the function they are called from and pipeline small loops unless explicitly specified not to do so. When not automatically performed, the addition of the pipeline PRAGMA can decrease the latency. In the same way that dataflow allows concurrent execution of tasks, the pipeline PRAGMA allows concurrent execution at the operation level. In the implemented HW kernel, the dataflow PRAGMA allows the compute and store sub-functions to start before the previous sub-functions are finished. The pipeline PRAGMA is applied inside the sub-functions. By adding the pipeline PRAGMA to the loops in the three sub-functions, the loop-bodies are executed concurrently instead of sequential. Usually, loops in FPGAs allow utilization of the same resources for each iteration, while pipelined loops perform multiple operations simultaneously, leading to higher resource utilization.

Array partition

Because the pipeline and dataflow pragmas increase the concurrency of the HW kernel, the kernel tries to access the different elements in the arrays simultaneously. It is possible to partition arrays, which provide more read/write ports for the memory, to enable multiple simultaneous accesses. The PRAGMA is applied to an array with the following syntax and changes the array set by the variable parameter from one memory into several smaller memories.

```
#Pragma HLS ARRAY_PARTITION variable=<array_name> <type> factor=<N>> dim=<M>
```

There are three types, namely block, cyclic and complete. In the HW kernel described in this thesis, the complete type is used. This type splits the chosen dimension of the array declared by `variable` in a manner that results in individual memory elements with designated read/write ports. This increases the utilization of the FPGA resources and decreases the latency.

The weights arrays are of significantly larger size than the other arrays used in the kernel implementation, and are exclusively used as the leftmost matrix in matrix multiplication. To reduce the pressure on resource utilization the two weights arrays are changed from one-dimensional to two-dimensional, so they can be partitioned on the second dimension instead of the entire array. This is done inside the kernel in the following manner:

```
dat_t W1_i[MID_LAYER][BANDS];
dat_t W2_i[BANDS][MID_LAYER];
#pragma HLS array_partition variable=W1_i complete dim=2
#pragma HLS array_partition variable=W2_i complete dim=2

for (size_t j = 0; j < BANDS; j++)
{
#pragma HLS PIPELINE
        for (size_t k = 0; k < MID_LAYER; k++)
        {
                W1_i[k][j] = W1[j*MID_LAYER + k];
                W2_i[j][k] = W2[k* BANDS + j];
        }
}
```

Effects of adding optimization Pragmas

The synthesis report from Vitis HLS is used to review the effects of adding the optimization PRAGMAS. Initially, the HW kernel is synthesized without any PRAGMAS. Then the pipeline PRAGMA is added to the loops inside the sub-functions. The PCA reduced **BEACH (C)** HSI from the ABU dataset is used to obtain the latency estimates. Table 11 shows how the latency of the different sub-functions changes when the pipeline PRAGMA is applied.

Table 11: Latency of the different sub-functions of the HW kernel with and without the pipeline PRAGMA.

Sub-function	Latency (Number of cycles)	
	Without pipeline PRAGMA	With pipeline PRAGMA
`load_input()`	120000	20009
`EncodeDecode()`	37350001	10587
`store_result`	20000	10001
Total	37490001	40597

From the table above, it is evident that the pipeline PRAGMA significantly reduces the latency. The effect is larger for `EncodeDecode()`, which is expected because it contains deeper and more complex loops. The HW resource utilization for `EncodeDecode()` with and without the

pipeline PRAGMA as estimated by Vitis HLS is shown in table 12. The resource utilization of the `load_input()` and `store_result` are negligible compared to `EncodeDecode()` and therefore not presented.

Table 12: HW resource utilization with and without pipeline PRAGMA.

PIPELINE	BRAM	DSP	FF	LUT
Yes	58	871	141475	95423
No	58	12	11433	20351

After adding the pipeline PRAGMA, the dataflow PRAGMA is added to the top-layer function. The latency results are shown in table 13.

Table 13: Latency of the different sub-functions of the HW kernel with and without the dataflow PRAGMA.

Sub-function	Latency (Number of cycles)	
	Without dataflow PRAGMA	With dataflow PRAGMA
Load_input()	20009	20011
EncDec()	10587	10589
Store_result()	10001	10007
Total	40597	20597

The results show that by adding the dataflow PRAGMA, the concurrency of the top-layer function is increased. The different sub-functions are allowed to start executing before the previous sub-function is finished, which leads to decreased total latency. Figure 26 illustrates how the latency of the entire top-layer function is related to the concurrent execution of the three sub-functions. The iteration latency of the main loop in `Encodedecode()` is reported by Vitis HLS to be 581 cycles, so the `store_results` function can start executing 582 cycles after the `Encodedecode()` started to execute.

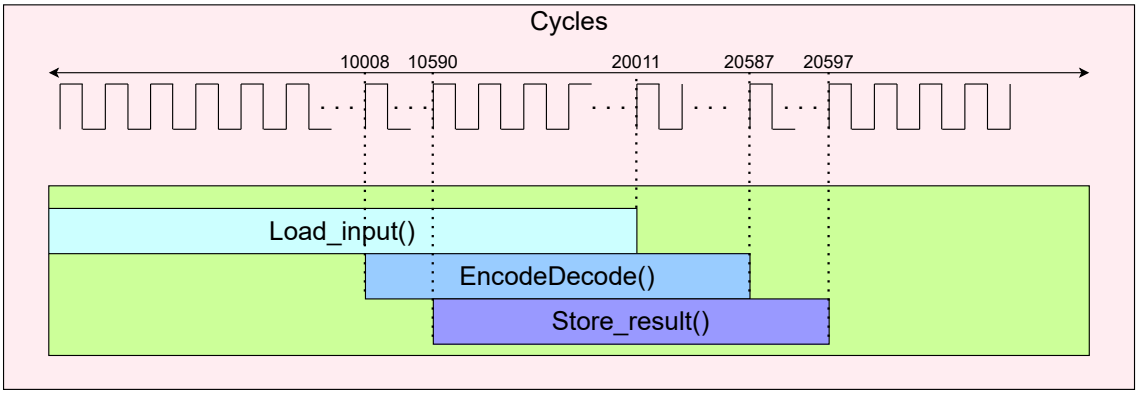


Figure 26: Illustration of the implemented HW kernel after applying dataflow and pipeline PRAGMAS.

4.2.3 Fixed point

The Xilinx library `ap_fixed.h` provides fixed-point datatypes in the format `ap_[u]fixed<W,I,Q,O,N>`. W is the total number of bits (Wordlength), I is the number of bits used to represent the integer part. Q allows the designer to decide the quantization mode. O controls the behavior on overflow, and N is the number of saturation bits in case of overflow.

Two `ap_fixed` data types are declared in the following manner.

```
#include "ap_fixed.h"

typedef ap_fixed<precision_total1, precision_int1> d_t1
typedef ap_fixed<precision_total2, precision_int2> d_t2
```

Using two types with different precision originates from the fact that most parameter values in the implementation tend to be between 0 and 1. At the same time, a few can become significantly higher/lower. It is not desired to use a fixed-point type with many bits reserved for the integer part if not needed, so the parameters that can require higher I are located. The parameters assumed to require higher I are set to `d_t2`, while the rest are set to `d_t1`. The HSI is changed from floating point to fixed point in the testbench before being sent to the HW kernel. The weights and biases are trained in the testbench as floating-point and then changed to fixed-point before being sent to the HW kernel.

An iterative test method is used to determine the precision parameters. The following procedure is for the dimensionality reduced HSI cube. Initially, all precision parameters are set so high that the difference between the floating-point and fixed-point implementations is negligible. The `precision_int1` value is then gradually decreased to find out where it overflows. The results are compared to the floating-point SW implementation to monitor the changes in precision. The table below shows the RMSE between the results and the most significant error for different precisions. `d_t2` is set to 32 bits, with 16 bits to represent the integer part. The results are obtained using C simulation in Vitis HLS.

Table 14: Loss of accuracy for different fixed-point precisions

d_t1	RMSE	Max error	Average SW result value	Max SW result value
ap_fixed<32, 16>	0.0003	0.0008 (1.8%)	0.08	0.60
ap_fixed<32, 8>	0.0003	0.0006 (1.3%)	0.09	0.064
ap_fixed<32, 4>	0.00025	0.0004 (0.7%)	0.09	0.061
ap_fixed<32, 3>	0.16	0.49 (305%)	0.09	0.066

As expected, the accuracy increases when `precision_int1` is decreased, since the number of bits used to represent the fractional part of the number increases. From the table we observe that when I is sat to 3, the results are unusable, due to overflow. The `precision_int1` is thus sat to 4, and the value `precision_tot1` is gradually decreased:

Table 15: Loss of accuracy for different fixed-point precisions

d_t1	RMSE	Max error	Average SW result value	Max SW result value
ap_fixed<32, 4>	0.00025	0.0004 (0.7%)	0.09	0.061
ap_fixed<20, 4>	0.00029	0.0008 (1.8%)	0.09	0.060
ap_fixed<14, 4>	0.00056	0.002 (3.25%)	0.085	0.064
ap_fixed<12, 4>	0.002	0.0077 (6.63%)	0.082	0.061

When reducing the total amount of bits, the precision is decreased. The `precision_tot1` is chosen to be 14 based on the desired precision. The fixed point type `d_t1` is now decided to be declared as `typedef ap_fixed<14, 4> d_t1`. By performing the process for `d_t2` with `d_t1` constant it is sat to `typedef ap_fixed<25, 8> d_t2`. The accuracy is then as follows:

Table 16: Loss of accuracy for the chosen fixed-point precision

RMSE	Max error	Average result value	Max result value
0.0006	0.01 (1.8%)	0.08	0.66

4.3 Hardware-software partitioned system

4.3.1 Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit

The entire HW-SW partitioned system is run on the Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit to verify and evaluate the HW kernel together with the SW implementation. The information in this sub-chapter is deducted from the official user guide for ZCU104 [56] and the Zynq UltraScale+ MPSoC Data Sheet [57].

The ZCU104 has the Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC, which consists of both a processing system (PS) and programmable logic (PL). The PS contains an Applications Processing Unit (APU) with a quad-core ARM Cortex-A53 processor. The PS and PL can be connected with various interfaces to communicate between configured HW in the PL and the SW program running on the APU. Both the PL and APU can access a DDR4 memory on the MPSoC. However, the PL can only access a specific part of the memory, called the global memory, and the remaining memory only accessible from the APU is called the local memory. Figure 27 shows how the MPSoC is built up of a PS and a PL and their different components.

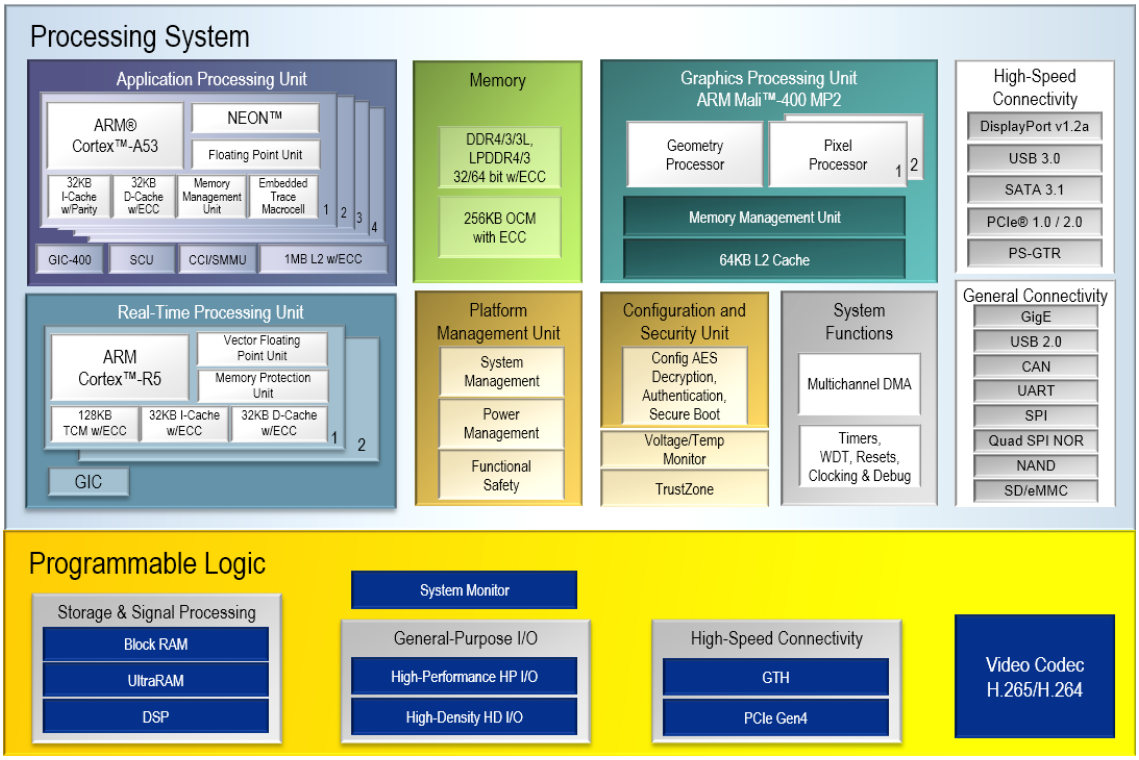


Figure 27: Block diagram of the Ultrascale+ XCZU7EV MPSoC, taken from [12].

The PS supports the operating system PetaLinux, enabling more complex applications to run on the APU. The different components of the PS and the PL are connected using the multi-layered Arm Advanced Microprocessor Bus Architecture (AMBA) (Advanced eXtensible Interface) AXI interconnect. AXI is an protocol designed by Arm, which is used for on-chip communication on the MPSoC.

The PL contains several 36 Kb BRAMs with two independent ports. In addition, the PL also contains several Digital Signal Processing (DSP) slices, consisting of a multiplier followed by an accumulator. The most relevant resources of the PL are given in table 17.

Table 17: FPGA resources on the ZCU104.

BRAM	DSP	FF	LUT	LUT-RAM
312	1728	460800	230400	101760

4.3.2 Hardware-software partitioned system

To import the HSI and train the DBN, installing the PetaLinux operating system on the board is necessary. A host program runs on the APU of the ZCU104, and the HW kernel is implemented in the PL. The HW-SW partitioned system that will run on the ZCU104 is built using the Vitis Unified software platform 2020.2, hereby called Vitis. A short guide that describes how to download and install the necessary items and run the HW-SW partitioned system on the ZCU104 is given in appendix B.

The program running on the APU is the `main()` function located in the file `host.cpp`. The functions from the SW implementation that read the HSI and train the DBN are used in the beginning of `main()`. The SW implementation of the computation of anomaly scores is then employed to obtain the results later used to verify the output of the HW kernel and to measure the execution time.

The HW kernel is implemented in C++, and synthesized to RTL as described in the previous sub-chapter. The only difference between Vitis and Vitis HLS is that the top-layer function must be declared as `extern "C"` in Vitis. The code used to program the HW kernel and control the memory transfers is adapted from a Vitis example project[3].

When the HW-SW partitioned system is built using Vitis, pre-compiled binary files describing the HW kernel and the SW program are generated. These binary files are stored on an SD card inserted into the ZCU104 before turning on the power. The testbench, i.e. the `main()` function, is executed by instructing the PetaLinux OS to run the pre-compiled testbench binary file. The program will then be loaded into DDR4 memory and executed by the APU. The `main()` function uses the HW kernel binary file to program the PL before using the industry standard openCL application programming interface [58] to handle the memory and employ the HW kernel in the following manner:

[3]https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/hello_world

```
// Allocate Buffer in Global Memory
cl::Buffer buf_in_hsi(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY ,
↪  size_hsi, in_hsi.data(), &err);
cl::Buffer buf_in_w1(context , CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY , size_w1
↪  , in_w1.data(), &err);
cl::Buffer buf_in_w2(context , CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY , size_w2
↪  , in_w2.data(), &err);
cl::Buffer buf_in_b1(context , CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY , size_b1
↪  , in_b1.data(), &err);
cl::Buffer buf_in_b2(context , CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY , size_b2
↪  , in_b2.data(), &err);
cl::Buffer buf_out_res(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
↪  size_out, res_anScore.data(), &err);

// Map the buffers to the inputs and outputs in the kernel
err = EncDec_kernel.setArg(0, buf_in_hsi);
err = EncDec_kernel.setArg(1, buf_in_w1);
err = EncDec_kernel.setArg(2, buf_in_w2);
err = EncDec_kernel.setArg(3, buf_in_b1);
err = EncDec_kernel.setArg(4, buf_in_b2);
err = EncDec_kernel.setArg(5, buf_out_re);

// Copy input data to device global memory
err = q.enqueueMigrateMemObjects({buf_in_hsi, buf_in_w1, buf_in_w2, buf_in_b1,
↪  buf_in_b2}, 0);

//Start kernel
err = q.enqueueTask(EncDec_kernel, nullptr, &event);

// Copy Result from global memory to local memory
err = q.enqueueMigrateMemObjects({buf_out_res}, CL_MIGRATE_MEM_OBJECT_HOST);
q.finish();
```

The anomaly score results from the HW kernel are now stored in `res_anScore` and can be compared with the results of the SW implementation. To measure the latency of the HW kernel, including the transferring of data to and from the global memory, the following code is added to the `main()` function after the kernel execution:

```
cl::Event event;
uint64_t nstimestart, nstimeend;

err = event.getProfilingInfo<uint64_t>(CL_PROFILING_COMMAND_START,
↪  &nstimestart);
err = event.getProfilingInfo<uint64_t>(CL_PROFILING_COMMAND_END, &nstimeend);

auto EncDec_time = nstimeend - nstimestart;
```

The HW-SW partitioned system is illustrated in Figure 28. The Figure shows how the system behaves when it is run on the APU and how the `main()` function programs the HW kernel with the binary file located in the SD-card. It is illustrated in which order the transfer of input and output data is moved to and from the global fraction of the DDR4 during execution. The inputs transferred to the global memory in step one are the HSI and the trained parameters of the network. In step two, the initiated HW kernel loads the inputs from the global memory. In step three, the anomaly scores computed by the HW kernel are stored in the global memory. In step four, the computed anomaly scores are transferred from the global to the local memory before the APU access it to evaluate the results.

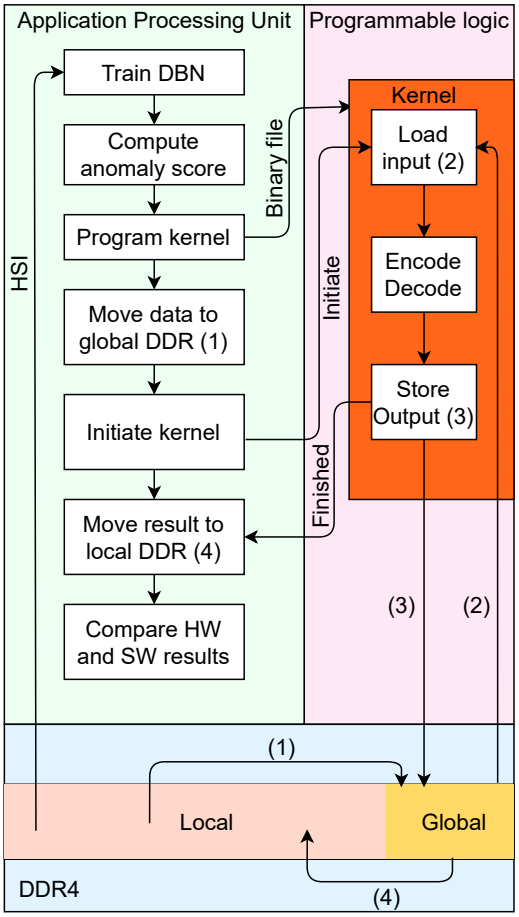


Figure 28: Illustration of HW-SW partitioned system on the ZCU104.

In this chapter, I have described the process of implementing the HAD algorithm in C and implementing a HW kernel using Vitis HLS. How to run the complete system together on a ZCU104 is also described. Implementation and optimizing a HW kernel is related to research question 4, regarding the acceleration of the HAD algorithms in HW. The implementation in C is related to the objective of the thesis, which is to provide the HYPSO mission with an implementation of a HAD algorithm suited for the mission.

5 Results

This chapter presents and discusses the results obtained with the SW implementation and the HW-SW partitioned implementation. Initially, the results of the SW implementation are compared to the Matlab implementation with respect to execution time and accuracy for verification. The HW-SW partitioned system is then tested, and the results of the HW kernel is verified with the results from the SW implementation. The resource utilization and latency of the HW kernel obtained from running on the PL on the ZCU104 is presented for three different implementations, namely, the baseline HW kernel, the optimized HW kernel and the optimized HW kernel with fixed point representation.

5.1 Software results

The results from the C implementation and the results from the Matlab implementation used for comparison are obtained using a Macbook Pro with a 2.9GHz dual-core i5 CPU and 8GB RAM. The C-code is compiled with the `-O2` C-flag, which instructs the compiler to optimize for speed. Matlab uses the standard data representation `Double` which is a 64-bit floating-point representation. The C-implementation uses the data representation `Float` which is a 32-bit floating-point representation. Figure 29 shows the detection result from the HSI **Beach (C)** in the ABU-dataset.

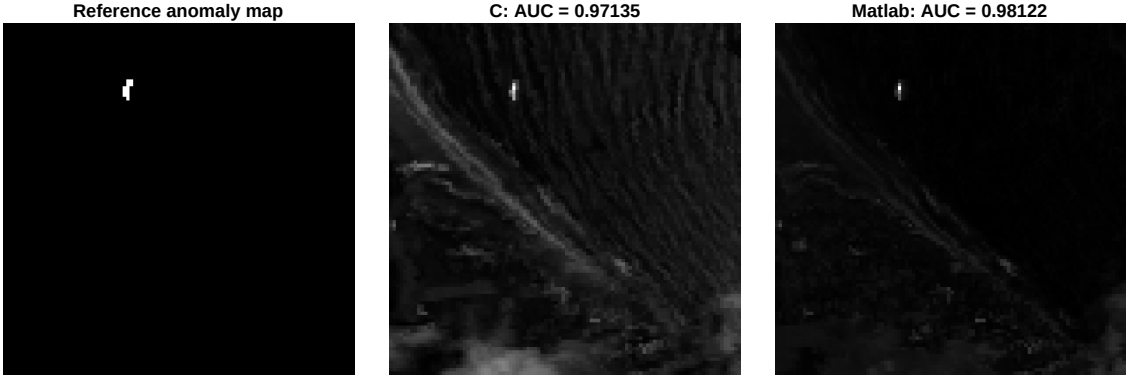


Figure 29: Result of C implementation compared with result from Matlab. The HSI has dimensions $100 \times 100 \times 188$.

We observe that the AUC results of the Matlab implementation is slightly lower than the results obtained with the DBN HAD algorithm in section 3.3. This is assumed to be because the simplified HAD does not use pre-training. The slight difference in AUC results between the C and Matlab implementations is assumed to be caused by the change of data representation and other minor differences between the implementations. The small differences in AUC results implies that the behaviour of the C implementation is as desired.

As in section 3.3, the AUC results after PCA dimensionality reduction differ insignificantly from the results without. The PCA is not implemented in C, so the PCA representation of the HSI cube is computed in Matlab and exported as a binary file read by the C implementation. When the reduced cube is used, the number of ANs in the input and output layers is the same as the number of principal spectral components, 12, and the number of ANs in the middle layer is 5.

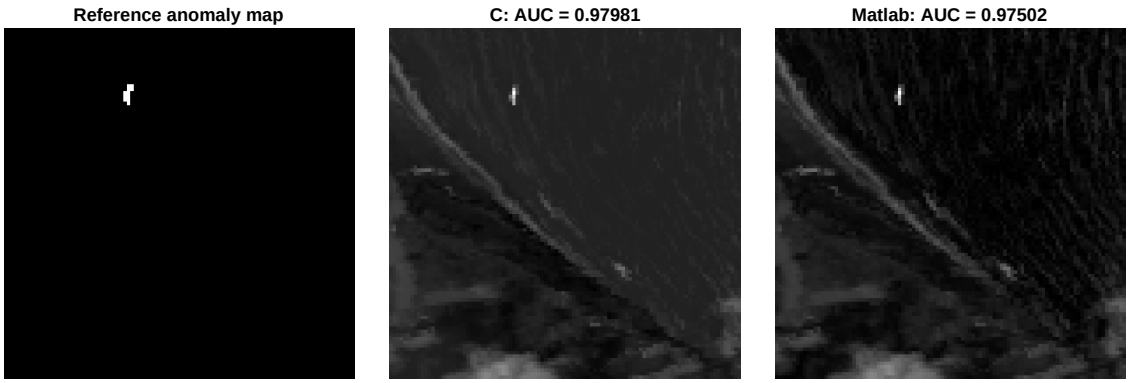


Figure 30: Result of C implementation compared with result from Matlab. The HSI has been pre-processed with PCA dimensionality reduction before testing. The HSI has dimensions $100 \times 100 \times 12$.

The differences in timing between the C-implementation and the Matlab-implementation when the DBN is trained for 30 iterations with a batch-size of 15 is as follows (pre-training is not performed):

Table 18: Execution times in seconds for the DBN HADs

Time	C	C with PCA	Matlab	Matlab with PCA
Total	9.612	0.331	6.686	1.795
Detection	0.139	0.006	0.316	0.146
Training	9.473	0.325	6.370	1.649

The built-in matrix operations and especially matrix multiplication in Matlab are highly optimized. Large matrices cause Matlab to outperform the simple matrix operations in the C implementation of the HAD. C is, in general, a faster language, and for smaller matrices and less complex functions, the C implementation is faster.

5.2 Hardware-software partitioned system results

The HW-SW partitioned system is tested on the HSI **Beach (C)** from the ABU-dataset with and without PCA dimensionality reduction. Without PCA the HSI is a $(100 \times 100 \times 188)$ cube, and with PCA the HSI is a $(100 \times 100 \times 12)$ cube. Three implementations of the HW kernel are tested on the ZCU104. The first is a basic implementation where only the dataflow PRAGMA is added. The Vitis framework performs some optimization automatically, so the basic implementation will pipeline small loops. The second implementation is optimized for speed with the PRAGMAS explored in section 4.2. The last implementation is the same as the optimized, but the floating-point representation of the data has been changed to fixed-point representation. The HSI cube without dimensionality reduction requires significantly more HW resources than the reduced HSI cube, and the possible degree of optimization is therefore lower. The main loop in the HW kernel iterates over the 10 000 pixels in the HSI to compute the anomaly score. This loop can be pipelined when the HSI is reduced, but it cannot when the HSI is not reduced due to limited resources. In order to verify the output of the HW kernel and compare the latency results, a SW version of the HW kernel is run on the APU of the ZCU104. For the baseline and optimized implementation using floating-point representation, the HW kernel and SW code output is a complete match.

The resource utilization results that are presented in this chapter are obtained from the post-synthesis reports generated by Vitis when building the system. The latency of the HW kernel is obtained as described in the previous chapter. The execution times for training and computing the anomaly score in SW is obtained with timers in the C++ code running on the APU.

5.2.1 Baseline implementation

Table 19 shows the post-synthesis utilization of the baseline HW kernel. The percent of the available resources utilized by the HW kernel is presented in parenthesis behind the number of utilized units.

Table 19: FPGA utilization of baseline HW kernel.

PCA	BRAM	DSP	FF	LUT	LUT-RAM
No	222 (71.00%)	21 (1.22%)	71911 (15.61%)	71250 (30.90%)	4698 (4.62%)
Yes	107 (34.21%)	12 (0.69%)	35207 (7.64%)	30251 (13.13%)	3237 (3.18%)

The latency of the HW kernel and the execution times of training the DBN and computing anomaly scores on the APU is shown in table 20. In this table, the latency of the detection with the HW kernel includes the transfer of input and output data to and from the global memory.

Table 20: Latency of baseline HW-SW partitioned system.

Process	Latency/Execution time without PCA	Latency/Execution time with PCA
Training	185.420 s	6.15 s
Detection SW	1.899 s	0.06 s
Detection HW	0.366 s	0.16 s

The baseline HW kernel is not an efficient implementation because it does not take advantage of the parallelization possibilities that PL provides. The baseline HW kernel is 5.2 times faster than the APU on the ZCU104 to compute the anomaly score for the full HSI. For the dimensionality reduced HSI cube, the overhead in loading the data and moving data to and from the global memory result in 2.6 times slower computation of the anomaly score with the HW kernel than on the APU. The utilization of resources shows that there is available space for increased parallelization. The baseline HW kernel for the full HSI utilizes 71 % of the available BRAM, which will restrict the possibilities of optimization.

5.2.2 Optimized implementation

As described in section 4.2 optimization PRAGMAS are applied to the baseline HW kernel to decrease the latency. Due to the number of available resources, the main loop, which iterates over all the pixels, can only be pipelined for the reduced HSI cube. Vitis automatically unrolls and pipelines all sub-loops when a loop is pipelined. Since the two double loops inside the main loop both iterate over the number of bands and ANs in the middle layer, the innermost loop is run 13×188 times for the full HSI. When pipelined and unrolled, this creates a HW kernel too large for the ZCU104. This results in a significant difference in relative improvement for the two HSI cubes. Tables 21 and 22 show the utilization and latency of the optimized HW kernel, and the execution times for the SW code running on the APU.

Table 21: FPGA utilization for optimized HW kernel.

PCA	BRAM	DSP	FF	LUT	LUT-RAM
No	221 (70.83%)	81 (4.69%)	91362 (19.83%)	106154 (46%)	28565 (28.07%)
Yes	107 (34.41%)	294 (17.1%)	87462 (18.98%)	69830 (18.98%)	7835 (7.70%)

Table 22: Latency for Optimized HW-SW partitioned system.

Process	Latency without PCA	Latency with PCA
Training	185.420 s	6.15 s
Detection SW	1.899 s	0.06 s
Detection HW	0.29 s	0.00057 s

We observe that the Optimized HW kernel is 25% faster than the baseline HW kernel for the full HSI. This is a result of partitioning arrays, and pipelining smaller loops. For the reduced HSI, the optimized HW kernel pipeline the main loop, which in this case is 10 000 iterations. The two double loops inside the main loop are then automatically pipelined and unrolled. This enables the optimized HW kernel to compute the anomaly score 280 times faster than the baseline HW kernel for the reduced HSI cube.

The optimized HW kernel for the full HSI is 6.5 times faster than the SW version of the kernel when run on the APU and two times slower than the SW version when ran on 2.9GHz dual-core I5 CPU (table 18). The optimized HW kernel for the reduced HSI is 105 times faster than the SW version ran on the APU and ten times faster than the SW version ran on 2.9GHz dual-core I5 CPU (table 18). The effect of pipelining the main loop is, as expected large, and results in more significant improvements for the reduced HSI relative to the full HSI. A critical remark is that the SW code ran on the 2.9GHz dual-core I5 CPU as previously stated compiled with the `-O2` compiler flag, while it is compiled without the `-O2` flag on the PS on the ZCU104. The HW kernel is connected to a 150MHz clock signal. This determines the speed of operation, even though the Vitis HLS estimates that the maximum possible frequency for the HW kernel is above 400MHz.

5.2.3 Fixed-point implementation

The data representation is changed from floating-point to fixed-point for the optimized HW kernel as described in section 4.3. The utilization is as follows:

Table 23: FPGA utilization for optimized HW kernel using fixed point representation.

PCA	BRAM	DSP	FF	LUT	LUT-RAM
No	215 (69%)	40 (2.30%)	64826 (14.07%)	69986 (30.38%)	15150 (14,89%)
Yes	67 (21.47%)	288 (16.67%)	49510 (10.74%)	40036 (17.38%)	3929 (3.86%)

The utilization is significantly reduced when changing from floating point to fixed point, as suggested in [50]. The HW kernel runs on 150 MHz, and since the number of cycles is the same for the two data representations, the latency is the same. The maximum possible estimated frequency is found in Vitis HLS. The estimated maximum frequency is 680 MHz when using fixed-point representation and 447 MHz when using floating-point. This means that in a system that offers a higher frequency the fixed-point implementation is estimated to be approximately 50% faster than the floating point implementation.

6 Discussion

The field of HAD is rapidly emerging, and the methods are growing in complexity compared to traditional statistical and geometrical methods. State-of-the-art methods utilize advanced pre-processing methods, morphological attribute filters, and deep learning techniques. The ten known state-of-the-art HAD algorithms used in evaluating the HAD algorithm proposed in this thesis all achieve better AUC results than the baseline GRX algorithm. However, most state-of-the-art algorithms have very high execution times compared to the baseline GRX and the faster state-of-the-art algorithms. Due to limited power, time, and memory onboard a satellite, these algorithms could be sub-optimal for remote sensing from space.

The two HAD algorithms utilizing morphological attribute filters (MPAF and AED) achieve very promising AUC and execution time results. The drawbacks of AED are that it is reliant on PCA and that it is susceptible to wrong size thresholds for anomalies. Dimensionality reduction with PCA is a feature that is not necessarily accessible onboard a satellite. MPAF uses a novel algorithm to select one single band from the HSI. Computing the anomaly score using only a single band leads to low execution times, but the algorithm relies on one single band where all anomalies stand out. The deep learning HAD algorithm AWDBN uses an AE combined with a weights strategy to perform the detection. The method has higher execution times than MPAF and AED, but it is also more flexible. There are no requirements for pre-processing or restrictions on the size of the HSI or number of bands. The weights strategy tends to assign higher anomaly scores to borders between different types of backgrounds in the HSIs than desired.

A novel weights strategy was proposed to improve the performance of AWDBN in this thesis. The new algorithm, consisting of a DBN AE and the new weights strategy, showed clear improvements compared to AWDBN on the 13 real HSIs in the ABU dataset. As expected, the anomaly scores assigned to the borders were decreased. The improvement was visible in the detection maps and understated by the increased average AUC results. The objective of this thesis was to provide the HYPSO mission with an implementation of a state-of-the-art HAD algorithm, which meets the requirements of the mission. The HAD algorithm proposed in this thesis, adapted and improved from AWDBN [1], was chosen as the HAD algorithm to be implemented. The choice was motivated by the high AUC results and the high flexibility of the algorithm.

The execution time of the proposed HAD algorithm is significantly higher than that of MPAF and AED but also significantly lower than the other state-of-the-art algorithms. However, there are challenges with the chosen algorithm. Several parameters need to be decided, where different HSIs require different parameter values to achieve maximum performance. As with most other algorithms, the size of the anomalies can be a challenge. The utilization of the DSW relies on the anomalies to be smaller than the inner window to function as desired. The DBN AE parameters that yield the most significant performance change are the number of iterations to perform training. Complex sceneries tend to require more iterations of training to achieve higher AUC results. For less complex sceneries, the AUC results converge for fewer iterations of training. Since the HYPSO mission aims at ocean monitoring, the scenery is assumed to be less complex than the Urban and Airport scenery, and comparable to the Beach scenery in the ABU dataset. The size of the inner window can be decided by the mission, dependent on what kind of objects or phenomena the mission aims to detect.

The proposed HAD algorithm was implemented in C-programming language. Implementing the algorithm in C was done to simplify adding the algorithm to the processing pipeline onboard the satellite used in the HYPSO mission and enabling HW acceleration using HLS. The HAD algorithm was simplified by not including CD_1 pre-training or the new weights strategy to allow rapid implementation and testing. The algorithm elements not included can be added to the implementation without changing the code that is written. The simplified algorithm is similar to the DBN HAD algorithm described in [7], and uses the reconstruction errors from the DBN AE as anomaly scores. The C implementation is implemented using 32-bits floating-point representations, while the Matlab implementation uses 64-bits floating-point representations. The AUC results and detection maps from the C implementation are compared with the results from Matlab, and the differences are negligible. The comparison between Matlab and C is considered a verification that the behavior of the C implementation is as desired. The execution time results show that the

anomaly detection is faster in C, while training is faster in Matlab. Matlab uses matrix operation algorithms that are highly optimized. During the training of the DBN AE, large matrices are used repeatedly, and Matlab's optimized use of the memory hierarchy explains why the training has lower execution time.

The part of the proposed HAD algorithm that uses the fully trained DBN AE to compute the reconstruction errors, which are used as anomaly scores, is accelerated in customized HW. Due to restrictions on available libraries when using the C-programming language, the code is translated from C to C++. Vitis HLS is used to synthesize the targeted C++ code to RTL. The targeted behavior can now be implemented as a HW kernel on programmable logic. The HW kernel is optimized with optimization PRAGMAS to achieve lower latency and resource utilization. Three versions of the HW kernel were implemented and tested as a part of a HW-SW partitioned system using the Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit. The baseline HW kernel was implemented in order to show the improvements made by adding the optimization PRAGMAS.

The results obtained from testing the design on the ZCU104 show the potential upside of accelerating part of the algorithm in HW. When applied on a PCA-reduced HSI cube, the optimized HW kernel is about 105 times faster than the SW version on the APU of the ZCU104 and ten times faster the SW version on a 2.9GHz dual-core I5 CPU. When applied to the full HSI cube, the HW kernel is about 6.5 times faster than the SW version on the APU of the ZCU104 and two times slower the SW version on a 2.9GHz dual-core I5 CPU. The HW kernel is run with a 150MHz clock, but it could run on a higher frequency to increase the speed further. The relative differences in acceleration shows that the utilization of a HW kernel is more efficient when using PCA dimensionality reduction than without. However, there is still a significant speedup for the full HSI compared to the SW version ran on the APU. The results show that customized HW can provide valuable speedup in a mission where timing is critical.

When the datatype was changed from floating-point to fixed-point, the resource utilization was reduced significantly. The change of datatype also changed the estimated maximum frequency, which shows the potential for further decrease of latency. The kernel could most likely be further optimized, seeing as there are still available resources.

The next iteration of the HYPSO satellite intends to have the same PS and PL as on the ZCU104 onboard and the inclusion of PCA dimensionality reduction in the onboard processing pipeline is planned. Based on this, the HAD algorithm is possible to implement onboard the HYPSO satellite.

6.1 Possible sources of error

There exist possible sources of error in the work related to this thesis. Three possible sources are identified and listed below. Most likely, there are more possible sources of error, however these three are the most important that are known to the author.

- The authors of the papers describing the DBN and AWDBN HAD algorithms have not used the ABU dataset for evaluation. The two algorithms were implemented by the author of this thesis during the specialization project, and the results of the two HAD algorithms presented in 3.3 are from the authors specialization thesis. The implementation follows the descriptions in [7] and [1] as closely as possible, however there may be differences.
- As described in section 3.3, the AWDBN and the HAD algorithm proposed in this thesis do not use the pixels at the edges of the HSIs when computing anomaly scores. This is a result of the DSW, where the PUT must be placed such that the DSW can fit within the HSI. The effects of this is assumed to be small, however it may have a slight impact on the AUC results.
- The loss of accuracy when converting from floating-point representation to fixed-point representation is obtained with simulating in Vitis HLS. There is a possibility that the corresponding loss of accuracy is different when the implementation is run in a real HW kernel.

6.2 Future work

The main objective of providing the HYPSO mission with an implementation of a state-of-the-art HAD algorithm. To improve the implemented system, and thus deliver a better product to the HYPSO mission, the following elements should be performed.

- Explore methods for optimizing and automating the choice of parameter values for the proposed HAD algorithm
- Implement the proposed weights strategy in C-programming language
- Implement CD pre-training in C-programming language
- Accelerate the proposed weights strategy in customized HW
- Further optimization of the implemented HW kernel
- Adapt and add the implemented system to the next iteration of HYPSO satellite onboard processing pipeline

7 Conclusion

A HAD algorithm has been developed, evaluated, implemented, tested, and accelerated with specialized HW in the work related to this thesis. The algorithm is adapted from the HAD algorithm proposed in [1] and modified to improve the accuracy. The changes made resulted in increased average accuracy, as presented in chapter 3.3. The proposed HAD algorithm achieved the third-best average accuracy on the ABU dataset compared to 11 other baseline and state-of-the-art HAD algorithms.

Research questions 1-3 regarding recent advances, evaluation methods, and state-of-the-art for HAD algorithms are explored in chapters 2 and 3. Recent advances in HAD include morphological attribute filters and machine learning methods such as autoencoders. The most used performance metrics in papers presenting HAD algorithms are AUC and execution time. The HAD with the state-of-the-art performance on the ABU-dataset used in this thesis is to the best of the author's knowledge, the MPAF detector [43].

The main objective of this thesis was to provide the HYPSO mission with an implementation of a state-of-the-art HAD algorithm suitable to the mission requirements. A simplified version of the proposed HAD algorithm was implemented in the C-programming language and adapted to the HSI storage format used in the HYPSO mission.

A part of the algorithm was accelerated in HW using Vitis HLS to create a HW kernel. This process is related to research question 4 regarding HW acceleration of the proposed HAD. A HW-SW partitioned system was implemented where the initial steps of the proposed HAD algorithm are performed in SW before the computing of anomaly scores is performed with a HW kernel. The HW-SW partitioned system is tested on the Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. The HW kernel achieved a maximum of 100 times faster computation of anomaly scores than C-code run at the ZCU104. The maximum speedup was achieved using a PCA-reduced HSI cube on an optimized HW kernel. The implemented HW-SW partitioned system can be added to the onboard processing pipeline without considerable effort based on the plans for the next iteration of the HYPSO satellite.

This thesis and the corresponding implementation is considered to satisfy the objective, even though there is still room for improvement as described in section 6.2.

Bibliography

[1] Ning Ma, Yu Peng, Shaojun Wang, and Philip H.W. Leong. An unsupervised deep hyperspectral anomaly detector. *Sensors (Switzerland)*, 18(3), 2018. ISSN 14248220. doi: 10.3390/s18030693.

[2] A.A. Gowen, C.P. O'Donnell, P.J. Cullen, G. Downey, and J.M. Frias. Hyperspectral imaging – an emerging process analytical tool for food quality and safety control. *Trends in Food Science and Technology*, 18(12):590–598, 2007. ISSN 0924-2244. doi: https://doi.org/10.1016/j.tifs.2007.06.001. URL `https://www.sciencedirect.com/science/article/pii/S0924224407002026`.

[3] Mariusz E Grøtte, F Fortuna, Roger Birkeland, Julian Veisdal, Milica Orlandic, Harald Martens, J Tommy Gravdahl, Fred Sigernes, Tor A Johansen, Jan Otto Reberg, et al. Hyperspectral imaging small satellite in multi-agent marine observation system. *Unpublished-Internal document*, 2018.

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

[5] Ning Ma, Ximing Yu, Yu Peng, and Shaojun Wang. A lightweight hyperspectral image anomaly detector for real-time mission. *Remote Sensing*, 11:1622, 07 2019. doi: 10.3390/rs11131622.

[6] M. Ning, P. Yu, W. Shaojun, and G. Wei. A weight sae based hyperspectral image anomaly targets detection. In *2017 13th IEEE International Conference on Electronic Measurement Instruments (ICEMI)*, pages 511–515, 2017. doi: 10.1109/ICEMI.2017.8265874.

[7] Ning Ma, Shaojun Wang, Jinxiang Yu, and Yu Peng. A DBN based anomaly targets detector for HSI. In Wolfgang Osten, Anand Krishna Asundi, and Huijie Zhao, editors, *AOPC 2017: 3D Measurement Technology for Intelligent Manufacturing*, volume 10458, pages 525 – 530. International Society for Optics and Photonics, SPIE, 2017. doi: 10.1117/12.2285766. URL `https://doi.org/10.1117/12.2285766`.

[8] Juan C Cuevas-Tello, Manuel Valenzuela-Rendón, and Juan A Nolazco-Flores. A tutorial on deep neural networks for intelligent systems. *arXiv preprint arXiv:1603.07249*, 2016.

[9] Introduction to fpga design with vivado high-level synthesis (ug998). `https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`, January 2019. (Accessed on 05/14/2021).

[10] Vitis high-level synthesis user guide. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf`, March 2021. (Accessed on 05/16/2021).

[11] HLS Pragmas. `https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_pragmas.html#sxx1504034358866`. (Accessed on 06/26/2021).

[12] Unleash the Unparalleled Power and Flexibility of Zynq UltraScale+ MPSoCs (WP470). `https://www.xilinx.com/support/documentation/white_papers/wp470-ultrascale-plus-power-flexibility.pdf`. (Accessed on 06/25/2021).

[13] Aksel L. Gundersen. Hyperspectral anomaly detection based on autoencoders. *Unpublished-Internal document*, 2020.

[14] Peg Shippert. Why use hyperspectral imagery?, 2004. ISSN 00991112.

[15] M T Eismann. *Hyperspectral remote sensing*. SPIE, 2012. doi: 10.1117/3.899758.

[16] M. Ben Salem, K. S. Ettabaa, and M. A. Hamdi. Anomaly detection in hyperspectral imagery: an overview. In *International Image Processing, Applications and Systems Conference*, pages 1–6, 2014. doi: 10.1109/IPAS.2014.7043320.

[17] S. Matteoli, M. Diani, and G. Corsini. A tutorial overview of anomaly detection in hyperspectral images. *IEEE Aerospace and Electronic Systems Magazine*, 25(7):5–28, 2010. doi: 10.1109/MAES.2010.5546306.

[18] Ryan Nugent, Riki Munakata, Alexander Chin, Roland Coelho, and Jordi Puig-Suari. The CubeSat: The picosatellite standard for research and education. In *Space 2008 Conference*, 2008. ISBN 9781563479465. doi: 10.2514/6.2008-7734.

[19] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 01 1998. ISBN 978-0-19-510270-3.

[20] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[21] Sagar Sharma and Simone Sharma. Activation functions in neural networks. *Towards Data Science*, 6(12):310–316, 2017.

[22] Sebastian Ruder. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016.

[23] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 1986. ISSN 00280836. doi: 10.1038/323533a0.

[24] Alireza Makhzani and Brendan Frey. k-Sparse Autoencoders. *arXiv e-prints*, art. arXiv:1312.5663, December 2013.

[25] Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[26] Chunhui Zhao, Xueyuan Li, and Haifeng Zhu. Hyperspectral anomaly detection based on stacked denoising autoencoders. *Journal of Applied Remote Sensing*, 11:1, 09 2017. doi: 10.1117/1.JRS.11.042605.

[27] Xudong Kang, Xiangping Zhang, Shutao Li, Kenli Li, Jun Li, and Jón Atli Benediktsson. Hyperspectral Anomaly Detection with Attribute and Edge-Preserving Filters. *IEEE Transactions on Geoscience and Remote Sensing*, 2017. ISSN 01962892. doi: 10.1109/TGRS.2017.2710145.

[28] Irving S. Reed and Xiaoli Yu. Adaptive Multiple-Band CFAR Detection of an Optical Pattern with Unknown Spectral Distribution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1990. ISSN 00963518. doi: 10.1109/29.60107.

[29] K. I. Ranney and M. Soumekh. Hyperspectral anomaly detection within the signal subspace. *IEEE Geoscience and Remote Sensing Letters*, 3(3):312–316, 2006. doi: 10.1109/LGRS.2006.870833.

[30] Michael E Wall, Andreas Rechtsteiner, and Luis M Rocha. *Singular Value Decomposition and Principal Component Analysis*, pages 91–109. Springer US, Boston, MA, 2003. ISBN 978-0-306-47815-4. doi: 10.1007/0-306-47815-3_5. URL `https://doi.org/10.1007/0-306-47815-3{_}5`.

[31] I T Jolliffe. *Principal Component Analysis and Factor Analysis*, pages 115–128. Springer New York, New York, NY, 1986. ISBN 978-1-4757-1904-8. doi: 10.1007/978-1-4757-1904-8_7. URL `https://doi.org/10.1007/978-1-4757-1904-8{_}7`.

[32] A.A. Green, M. Berman, P. Switzer, and M.D. Craig. A transformation for ordering multispectral data in terms of image quality with implications for noise removal. *IEEE Transactions on Geoscience and Remote Sensing*, 26(1):65–74, 1988. doi: 10.1109/36.3001.

[33] Prasanta Chandra Mahalanobis. On the generalized distance in statistics. National Institute of Science of India, 1936.

[34] S. Küçük and S. E. Yüksel. Comparison of rx-based anomaly detectors on synthetic and real hyperspectral data. In *2015 7th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, pages 1–4, 2015. doi: 10.1109/WHISPERS.2015.8075504.

[35] Chein-I Chang and Mingkai Hsueh. Characterization of anomaly detection in hyperspectral imagery. *Sensor Review - SENS REV*, 26:137–146, 04 2006. doi: 10.1108/02602280610652730.

[36] Heesung Kwon and N. M. Nasrabadi. Kernel rx-algorithm: a nonlinear anomaly detector for hyperspectral imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 43(2): 388–397, 2005. doi: 10.1109/TGRS.2004.841487.

[37] Chein-I Chang and Mingkai Hsueh. Characterization of anomaly detection in hyperspectral imagery. *Sensor Review - SENS REV*, 26:137–146, 04 2006. doi: 10.1108/02602280610652730.

[38] Wei Li and Qian Du. Collaborative Representation for Hyperspectral Anomaly Detection. *Geoscience and Remote Sensing, IEEE Transactions on*, 53:1463–1474, 2015. doi: 10.1109/TGRS.2014.2343955.

[39] Bing Tu, Nanying Li, Zhuolang Liao, Xianfeng Ou, and Guoyun Zhang. Hyperspectral anomaly detection via spatial density background purification. *Remote Sensing*, 11:2618, 11 2019. doi: 10.3390/rs11222618.

[40] Edmond J Breen and Ronald Jones. Attribute openings, thinnings, and granulometries. *Computer vision and image understanding*, 64(3):377–389, 1996.

[41] Mauro Dalla Mura, Jón Atli Benediktsson, Björn Waske, and Lorenzo Bruzzone. Morphological attribute profiles for the analysis of very high resolution images. *IEEE Transactions on Geoscience and Remote Sensing*, 48(10):3747–3762, 2010. doi: 10.1109/TGRS.2010.2048116.

[42] Eduardo Gastal and Manuel Oliveira. Domain transform for edge-aware image and video processing. *ACM Trans. Graph.*, 30:69, 07 2011. doi: 10.1145/2010324.1964964.

[43] Ferdi Andika, Mia Rizkinia, and Masahiro Okuda. A hyperspectral anomaly detection algorithm based on morphological profile and attribute filter with band selection and automatic determination of maximum area. *Remote Sensing*, 12(20), 2020. ISSN 2072-4292. doi: 10.3390/rs12203387. URL `https://www.mdpi.com/2072-4292/12/20/3387`.

[44] Emrecan Bati, Akın Çalışkan, Alper Koz, and A. Aydin Alatan. Hyperspectral anomaly detection method based on auto-encoder. In Lorenzo Bruzzone, editor, *Image and Signal Processing for Remote Sensing XXI*, volume 9643, pages 220 – 226. International Society for Optics and Photonics, SPIE, 2015. doi: 10.1117/12.2195180. URL `https://doi.org/10.1117/12.2195180`.

[45] Shizhen Chang, Bo Du, and Liangpei Zhang. A sparse autoencoder based hyperspectral anomaly detection algorihtm using residual of reconstruction error. In *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, pages 5488–5491, 2019. doi: 10.1109/IGARSS.2019.8898697.

[46] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 2002. ISSN 08997667. doi: 10.1162/089976602760128018.

[47] Asja Fischer and Christian Igel. Training restricted boltzmann machines: An introduction. *Pattern Recognition*, 47:25–39, 01 2014. doi: 10.1016/j.patcog.2013.05.025.

[48] Geoffrey E. Hinton. A practical guide to training restricted boltzmann machines. *Gatsby Computational neuroscience unit*, 2010.

[49] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. ISSN 0027-8424. doi: 10.1073/pnas.79.8.2554. URL `https://www.pnas.org/content/79/8/2554`.

[50] Ambrose Finnerty and Hervé Ratigner. Reduce power and cost by converting from floating point to fixed point. *WP491 (v1. 0)*, 2017.

[51] Jingjing Wu, Yu Jin, Wei Li, and Lianru Gao. Fpga implementation of collaborative representation algorithm for real-time hyperspectral target detection. *Journal of Real-Time Image Processing*, 15, 10 2018. doi: 10.1007/s11554-018-0823-7.

[52] Jie Lei, Geng Yang, Weiying Xie, Yunsong Li, and Xiuping Jia. A low-complexity hyperspectral anomaly detection algorithm and its fpga implementation. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 14:907–921, 2021. doi: 10.1109/JSTARS.2020.3034060.

[53] M. Tanaka and M. Okutomi. A novel inference of a restricted boltzmann machine. *2014 22nd International Conference on Pattern Recognition*, pages 1526–1531, 2014.

[54] Peter Flach, José Hernández-Orallo, and Cèsar Ferri. A coherent interpretation of AUC as a measure of aggregated classification performance. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, 2011. ISBN 9781450306195.

[55] A. Banerjee, P. Burlina, and C. Diehl. A support vector method for anomaly detection in hyperspectral imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 44(8):2282–2291, 2006. doi: 10.1109/TGRS.2006.873019.

[56] ZCU104 Evaluation Board User Guide (UG1267). `https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf`, October 2018. (Accessed on 06/19/2021).

[57] Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891). `https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf`, may 2021. (Accessed on 06/19/2021).

[58] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314, 2009. doi: 10.1109/HOTCHIPS.2009.7478342.

Appendix

A Parameter values used to obtain results in section 3.3

Scene	Image	Mini-batch size	Pre-training iterations	Training iterations	w_{inner}	w_{outer}
Airport	(A)	7	20	30	1	13
Airport	(B)	5	15	100	5	15
Airport	(C)	7	15	40	5	17
Airport	(D)	7	15	80	5	9
Beach	(A)	40	10	20	3	5
Beach	(B)	6	10	30	1	3
Beach	(C)	10	10	20	5	7
Beach	(D)	10	10	20	15	17
Urban	(A)	4	10	25	1	9
Urban	(B)	5	15	25	9	15
Urban	(C)	5	15	30	3	7
Urban	(D)	5	30	30	9	11
Urban	(E)	7	15	40	5	7

B Guide to run the system on the ZYNQ Ultrascale+ MPSoC ZCU104

The complete system is exported as a ZIP-file. To run the implemented HW-SW partitioned system on the Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit, please follow the below steps:

1. Download and install Vitis Unified Software Platform 2020.2[4]
2. Download the common image for petalinux from the xilinx website[5]
3. Download the program etcher from etcher.io.
4. Use etcher to flash the common image onto the SD-card.
5. Open a terminal in the Xilinx directory created when installing Vitis Unified Software Platform 2020.2
   - type "source ./settings64.sh"
   - type "sudo /bin/vitis"
   - The Vitis IDE should now open.
6. The project zip-file can be imported as described at the xilinx webpage[6]
7. Build the project for HW.
8. Copy the files in "workspace/sys1_system/hardware/package/sd_card to the SD-card.
9. Insert the SD-card in the ZCU104, set the board in "boot from SD-card mode" as described in [56] and turn the board on.
10. Use a serial terminal to communicate with the board using USB and run the sys1.

[4]https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/2020-2.html
[5]https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/2020-2.html
[6]https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/projectexportimport.html

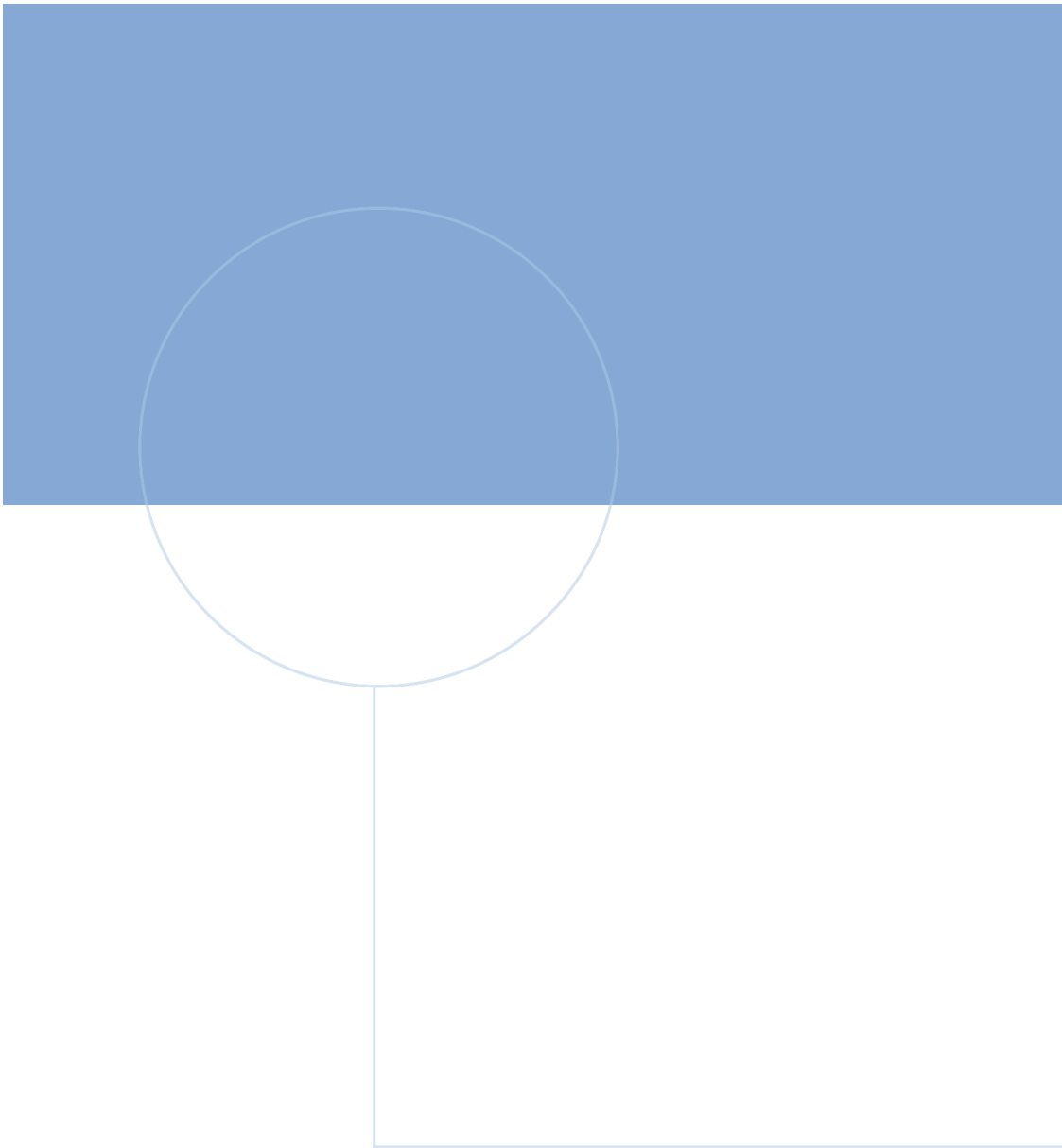
C Description of the code-base

The entire code-base for related to the thesis is contained in a private Github repository[7]. To access it please send an request on email to the author (gundersen1105@gmail.com).

The code-base is divided into the following folders:

1. Matlab
    - This folder contains all matlab files.
    - To run on a new computer the parameter PATH in startup.m must be changed to the correct path. After changing the Path, run the startup script. the PATH variable in load_HSI.m must also be changed.
    - The script "main.m" is used to run the implementation. Change the parameter opts.datasets to decide which HSI to use.
    - The DBN parameters can be changed in the script "HSI_TRAIN_DBN.m".
2. C
    - This folder contains all C files.
    - Change the global variable DATA in params.h to change which HSI is used.
    - Run the main-function in the main.c file to test the program.
3. HLS
    - This folder contains all C++ files used in the development of the three different kernels.
    - To run the program, please download Vitis HLS 2020.2, and compile the files from the IDE.
4. ZCU104
    - This folder contains all C++ files used in the HW-SW partitioned system that is ran on the ZCU104
    - To run the program, please download Vitis unified software platform as described in attachment B.

[7] https://github.com/akselgundersen/HSI_AD



NTNU

Norwegian University of
Science and Technology